LEHRVERANSTALTUNG – JAHR/SEMESTER

Aufgabe 1: **TITEL DER AUFGABE**

Entwurfsdokument

Hinweise zum Entwurfsdokument

Es handelt sich bei diesem Dokument um die **stark gekürzte** Fassung eines objektorientierten Entwurfs, wie in der Vorlesung vorgestellt und im vorangegangenen Projekt behandelt.

Der Entwurf beginnt mit Abschnitt 1 (Struktur der Komponenten). Sämtliche für TITEL DER AUFGABE nicht direkt benötigten Informationen wurden herausgelassen.

Einleitung

Es soll die Steuersoftware für Roboter in einer automatischen Paketsortieranlage entworfen werden. Diese Steuersoftware soll dabei den Transport von Paketen von Abfertigungsstationen (kurz Stationen), an denen die Roboter beladen werden, zu verschiedenen Abwurfschächten gewährleisten. Ebenfalls zu berücksichtigen sind hier die Rückfahrten der Roboter zu den Stationen sowie Fahrten innerhalb der Stationen.

Es werden über das Netzwerk Fahraufträge an die Roboter übermittelt. Die Aufträge beinhalten immer eine Zielkoordinate sowie die Art des Auftrags. Die Steuersoftware der Roboter realisiert daraufhin autonom (d.h. nur auf Basis der lokalen Sensorinformationen) die Anfahrt zur gegebenen Koordinate. Bei manchen Arten von Aufträgen muss zudem nach Ankunft noch eine Aktion durchgeführt werden (z.B. das Abladen eines Pakets).

Das Fahrverhalten der Roboter orientiert sich an der deutschen Straßenverkehrsordnung.

Umgebung

Die Struktur der Paketsortieranlage ist in Abbildung 1 dargestellt und im Folgenden beschrieben. Die (potentiell beliebig große) Fläche der Sortieranlage ist in quadratische Felder eingeteilt, die jeweils ganzzahlige Koordinaten haben und gezielt von den Robotern angefahren

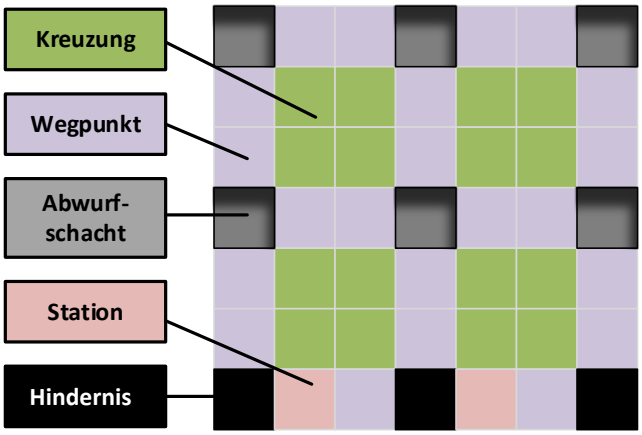


Abbildung 1: Übersicht über den Aufbau der Paketsortieranlage und die wichtigsten Begriffe. Die Farben der Felder kennzeichnen die Positionstypen, die von den Robotersensoren erkannt werden können.

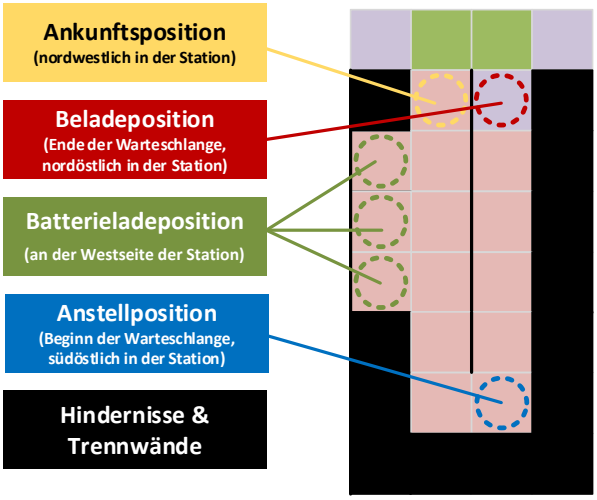


Abbildung 2: Übersicht über den Aufbau einer Station in der Paketsortieranlage, inklusive der wichtigsten Bezeichnungen für die einzelnen Positionen, zu denen ein Roboter disponiert werden kann.

werden können. Die Felder sind jeweils groß genug, so dass ein Roboter sich auf der Stelle drehen kann, ohne mit den Robotern auf den Nachbarfeldern zu kollidieren.

Auf der Fläche der Sortieranlage befinden sich in regelmäßigen Abständen *Abwurfschächte*. Zwischen je zwei Abwurfschächten sind immer zwei Felder frei. Die Streifen zwischen den Schächten bilden *Straßen*, die aus *Kreuzungen* und sogenannten *Wegpunkten* bestehen. Auf den Straßen herrscht **strikter Rechtsverkehr**. An Kreuzungen gilt die **Vorfahrtsregel „Rechts vor Links“**. Durch diese beiden Regeln können autonom fahrende Roboter Kollisionen und Deadlocks vermeiden.

An der Südseite der Sortieranlage befinden sich nebeneinander aufgereiht mehrere *Stationen*, die der in Abbildung 2 gezeigten Struktur entsprechen. Als Übergang zwischen einer Station und der restlichen Fläche der Sortieranlage dienen zwei Felder: die *Ankunftsposition*, auf der rechtsseitig fahrende Roboter aus dem freien Verkehr ankommen, und die *Beladeposition*, von der aus sich Roboter in den Verkehr einordnen können.

Innerhalb einer Station befindet sich links eine Strecke, an der die *Batterieladepositionen* liegen. Fährt ein Roboter rückwärts in die Batterieladeposition hinein, beginnt die Aufladung automatisch. Die serverseitige Roboterdisposition stellt sicher, dass diese Strecke links der Station immer nur von einem Roboter befahren wird, so dass beim Aus- und Einparken nicht auf Kollisionen geachtet werden muss.

Auf der rechten Seite der Station befindet sich eine *Warteschlange*, in der Roboter darauf warten, beladen zu werden. Die vorderste Position der Warteschlange ist die Beladeposition und zugleich der Eintrittspunkt in den regulären Verkehr. Da sich die Beladeposition vor einer Kreuzung befindet, erkennt ein darauf stehender Roboter sie als einen Wegpunkt. Die Roboter werden zuerst immer zur *Anstellposition* (Ende der Warteschlange) dirigiert und erst danach zur Beladeposition.

Aufgrund von Trennwänden zwischen der rechten und der linken Hälfte der Station, erkennen die Roboter die entsprechenden Nachbarfelder als blockiert. Gleiches gilt für *Hindernisse* und Abwurfschächte.

1 Innere Struktur der Komponenten

1.1 Struktur des Roboterteilsystems (Komponente `LogisticsRobot`)

Auf allen Robotern läuft die gleiche Software (beschrieben durch die gegebene Komponente `LogisticsRobot`), die sich aus vier Unterkomponenten zusammensetzt (siehe Abbildung 3).

Die Unterkomponente `RobotControl` ist die zentrale Unterkomponente, die den Roboter steuert. Sie enthält die aktive Klasse `DriveSystem`, welche Zugriff auf die Aktuatoren hat und dadurch das Fahrverhalten eines Roboters bestimmt. Außerdem beinhaltet die Unterkomponente `RobotControl` die Klasse `SensorDataManager` für das Abrufen von Sensordaten, die Klasse `DataSnapshot` für Speicherung der Sensordaten und die aktive Klasse `TaskProcessing` für das Empfangen von Aufträgen.

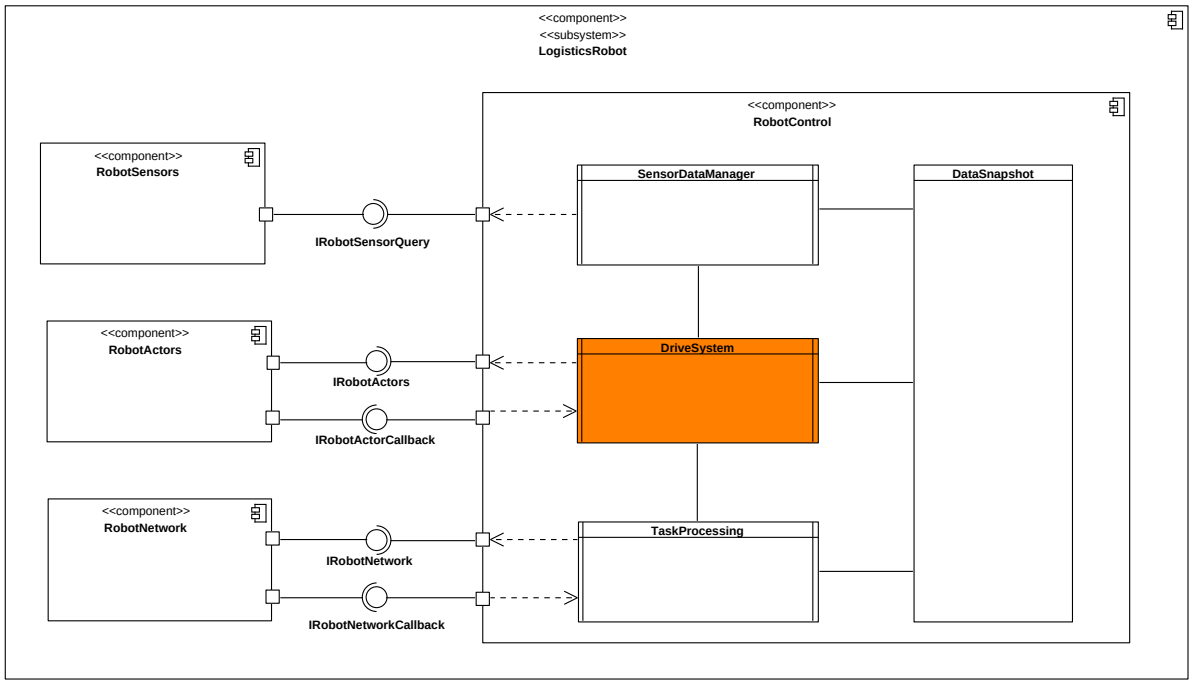


Abbildung 3: Architektur des Roboterteilsystems.

Zur eigentlichen Bewegung stehen physische Aktuatoren zur Verfügung, die von der Unterkomponente `RobotActors` kontrolliert werden. Neben Aktuatoren sind auch Sensoren verfügbar. Die Messwerte dieser Sensoren stellt die Unterkomponente `RobotSensors` zur Verfügung. Die Unterkomponente `RobotNetwork` wickelt die Kommunikation eines Roboters mit den anderen Teilsystemen ab.

2 Paketstruktur

Die Inhalte der Komponente `LogisticsRobot` werden in dem Paket `robot` implementiert. Dieses enthält für jede der vier Unterkomponenten `RobotControl`, `RobotActors`, `RobotSensors` und `RobotNetwork` ein Unterpaket. Die Unterpakete `actors`, `sensors` und `connection` sind nicht weiter unterteilt, während das Unterpaket `control` noch einmal in die Unterpakete `data`, `drivesystem` und `tasks` aufgeteilt ist.

Die sich ergebende Paketstruktur mit den Abhängigkeiten zwischen den einzelnen Paketen ist in Abbildung 4 dargestellt.

3 Paketdetails

In diesem Abschnitt werden für das Paket `robot.control.drivesystem` Details zur Implementierung gegeben.

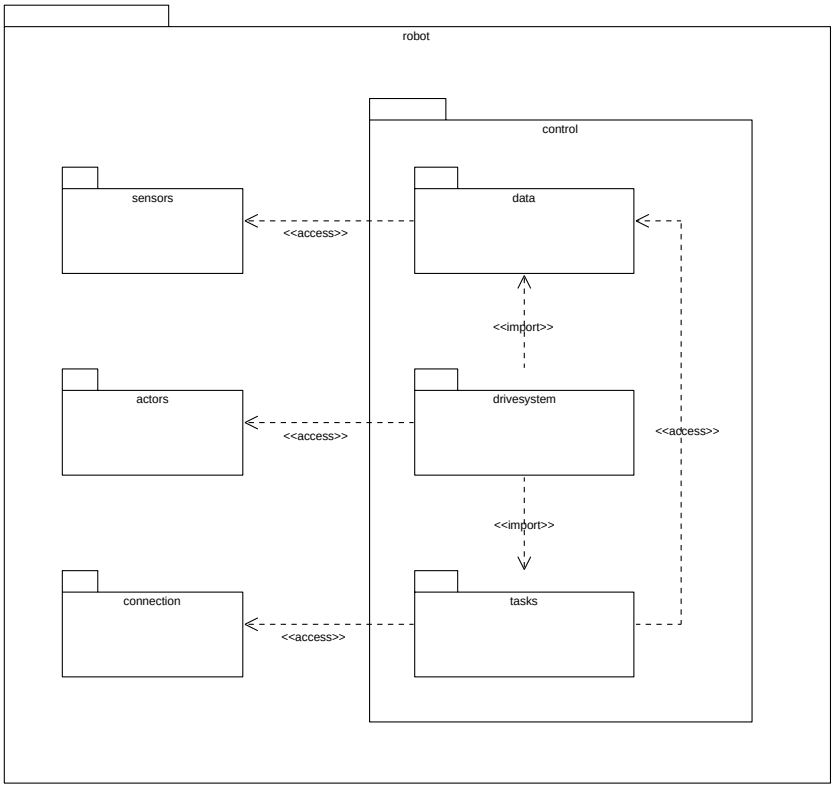


Abbildung 4: Paketstruktur des Roboterteilsystems.

Im Folgenden wird eine vollständige Spezifikation aller Klassen gegeben, die entweder direkt im Paket `drivesystem` implementiert werden oder aus anderen Paketen per Import- oder Zugriffsbeziehung eingebunden werden. Diese Klassen sind in Abbildung 5 gemeinsam dargestellt.

3.1 Klasse `DriveSystem`

Die aktive Klasse `DriveSystem` erhält Aufträge von der Klasse `TaskProcessing` und setzt diese (basierend auf den aktuellen Sensordaten) in Bewegungsbefehle um. Dieses Fahrverhalten soll durch ein Statechart beschrieben werden.

Die Klasse `DriveSystem` implementiert fünf Methoden mit öffentlicher Sichtbarkeit, die als Eingaben für das Statechart fungieren. Das jeweils beschriebene Verhalten ist mithilfe eines gemeinsamen Statecharts umzusetzen.

3.1.1 Methode `newTarget()`

Die Methode `newTarget()` wird aufgerufen, um der Klasse `DriveSystem` mitzuteilen, dass in der Klasse `DataSnapshot` neue Zielkoordinaten hinterlegt wurden. Die Klasse `TaskProcessing` ist verantwortlich für das Hinterlegen der Zielkoordinaten und das Aufrufen der Methode `newTarget()`.

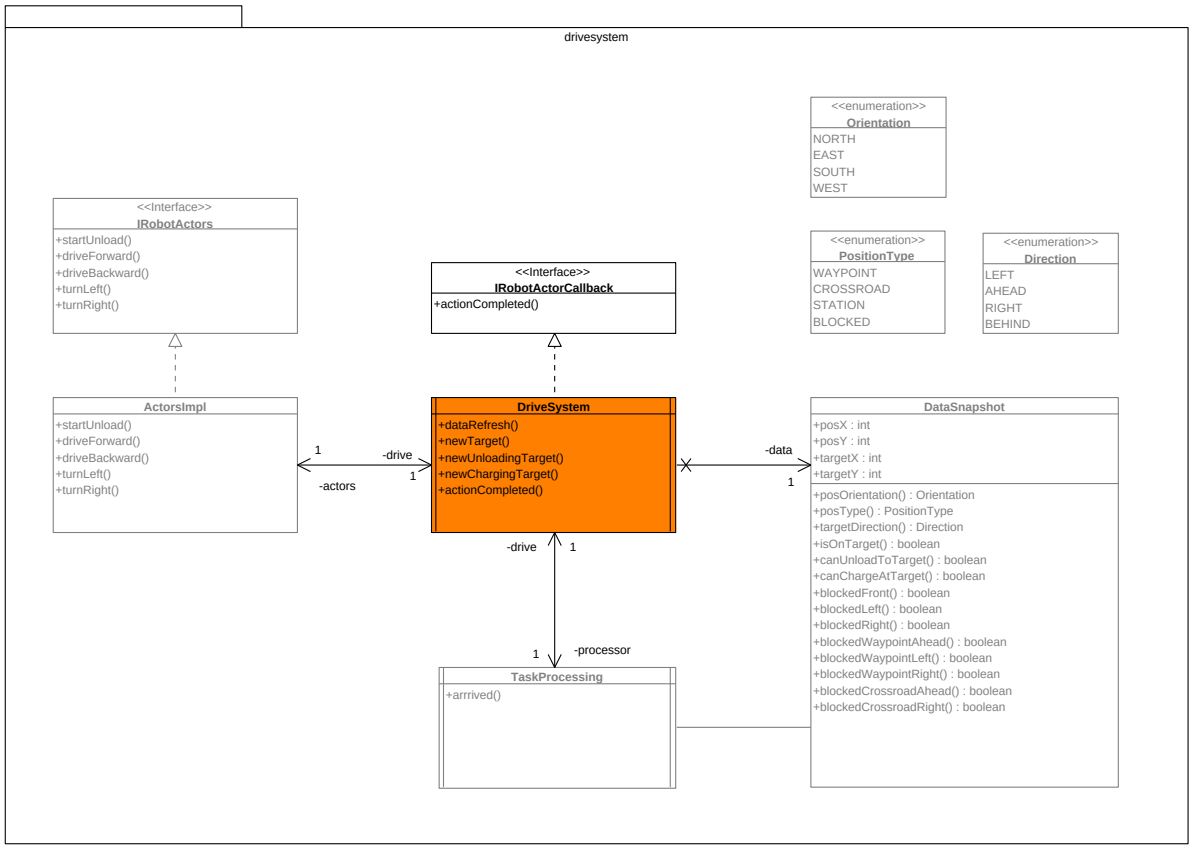


Abbildung 5: Inhalte des Pakets `robot.control.drivesystem`. Aus anderen Paketen per Import- oder Zugriffsbeziehung eingebundene Inhalte sind grau markiert.

Das Statechart soll als Reaktion auf diesen Methodenaufruf so lange Bewegungsbefehle an die Klasse `ActorsImpl` senden, bis das hinterlegte Ziel erreicht wurde. Das Erreichen des Ziels kann durch die Methode `isOnTarget()` der Klasse `DataSnapshot` überprüft werden.

Die Bewegungsbefehle sollen so erteilt werden, dass mithilfe der Sensordaten Verkehrsregeln eingehalten und Kollisionen und Deadlocks der Roboter vermieden werden.

Es ist davon auszugehen, dass die gegebenen Zielkoordinaten immer in einer Station oder auf einem Wegpunkt sind. Ist das Ziel wider Erwarten auf einer Kreuzung, ist diese nach Ankunft sofort zu verlassen.

Nach erfolgreicher Erfüllung des Fahrauftrages soll bei der Klasse `TaskProcessing` die Methode `arrived()` aufgerufen werden. Das Statechart soll keine weiteren Bewegungen veranlassen bis der Erhalt eines neuen Ziels gemeldet wird.

3.1.2 Methode `newUnloadingTarget()`

Ähnlich zu der Methode `newTarget()`, dient auch die Methode `newUnloadingTarget()` dazu, die Klasse `DriveSystem` auf neue Zielkoordinaten hinzuweisen.

Das Statechart soll als Reaktion auf diesen Methodenaufruf das hinterlegte Ziel anfahren, wie es auch bei der Methode `newTarget()` passiert. Allerdings handelt es sich hier bei den Zielkoordinaten um die Koordinaten eines Abwurfschachtes, so dass die Fahrt bereits *neben* diesem Schacht beendet werden soll. Ob der Roboter sich neben dem anzufahrenden Abwurfschacht befindet, kann mit der Methode `canUnloadToTarget()` der Klasse `DataSnapshot` überprüft werden.

Nach der Ankunft am Schacht und *vor* der Erfolgsmeldung durch die Methode `arrived()` muss das Paket mithilfe der Methode `startUnload()` in den Schacht abgeworfen werden.

3.1.3 Methode `newChargingTarget()`

Auch die Methode `newChargingTarget()` ist eine Sondervariante der Methode `newTarget()`, die die Klasse `DriveSystem` auf neue Zielkoordinaten hinweist.

Die Zielkoordinaten beschreiben in diesem Fall eine Batterieladeposition. In diese Batterieladeposition muss rückwärts eingeparkt werden, um die Batterieladevorrichtung nicht zu beschädigen. Um das Einparkmanöver zu beginnen, kann mit der Methode `canChargeAtTarget()` der Klasse `DataSnapshot` überprüft werden, ob der Roboter sich genau neben der anzufahrenden Batterieladeposition befindet.

3.1.4 Methode `actionCompleted()`

Die Methode `actionCompleted()` teilt der Klasse `DriveSystem` mit, dass eine laufende Bewegung beendet wurde. Der Aufruf dieser Methode signalisiert dem Statechart, dass die Klasse `ActorsImpl` für einen weiteren Bewegungsbefehl bereit ist.

3.1.5 Methode `dataRefresh()`

Die Methode `dataRefresh()` wird aufgerufen, um der Klasse `DriveSystem` mitzuteilen, dass die Sensordaten in der Klasse `DataSnapshot` aktualisiert wurden. Wenn der Roboter ein Ziel hat *und* aktuell keine Bewegung ausgeführt wird, soll das Statechart als Reaktion auf diesen Methodenaufruf die vorliegenden Sensordaten prüfen und, falls möglich, eine Bewegung der Aktuatoren veranlassen.

Die Methode `dataRefresh()` wird mit hoher Frequenz aufgerufen und kann daher zusätzlich genutzt werden, um innerhalb des Statecharts periodische Aktivitäten auszulösen.

3.2 Klasse `ActorsImpl`

Die Klasse `ActorsImpl` ist ein Teil der Komponente `RobotActors` und realisiert das von der Komponente nach außen angebotene Interface `IRobotActors`. Die Klasse `ActorsImpl` wird im Paket `robot.actors` implementiert.

Es stehen insgesamt fünf durch das Interface `IRobotActors` vorgegebene öffentliche Methoden zur Verfügung. Jede dieser Methoden wird zum Auslösen eines bestimmten Bewegungsbefehls genutzt. Wenn in einem Zustandsübergang mehrere Bewegungsbefehle aufgerufen werden, wird nur einer davon ausgelöst.

3.2.1 Methode `driveForward()`

Die Methode `driveForward()` lässt den Roboter ein einzelnes Feld nach vorne fahren. Nach der Ankunft wird die Methode `actionCompleted()` der Klasse `DriveSystem` aufgerufen. Kommt ein `driveForward()` Aufruf, während bereits eine Bewegung läuft, wird dieser verworfen.

Die Methode `driveForward()` nimmt keine Rücksicht auf eventuelle Hindernisse.

3.2.2 Methode `driveBackward()`

Die Methode `driveBackward()` lässt den Roboter ein einzelnes Feld nach hinten fahren. Nach der Ankunft wird die Methode `actionCompleted()` der Klasse `DriveSystem` aufgerufen. Kommt ein `driveBackward()` Aufruf, während bereits eine Bewegung läuft, wird dieser verworfen.

Die Methode `driveBackward()` nimmt keine Rücksicht auf eventuelle Hindernisse.

3.2.3 Methode `turnLeft()`

Die Methode `turnLeft()` lässt den Roboter auf einem Feld um 90° nach links rotieren. Nach dem Ende der Rotation wird die Methode `actionCompleted()` der Klasse `DriveSystem` aufgerufen. Kommt ein `turnLeft()` Aufruf, während bereits eine Bewegung läuft, wird dieser verworfen.

3.2.4 Methode `turnRight()`

Die Methode `turnRight()` lässt den Roboter auf einem Feld um 90° nach rechts rotieren. Nach dem Ende der Rotation wird die Methode `actionCompleted()` der Klasse `DriveSystem` aufgerufen. Kommt ein `turnRight()` Aufruf, während bereits eine Bewegung läuft, wird dieser verworfen.

3.2.5 Methode `startUnload()`

Die Methode `startUnload()` lässt den Roboter seine Ladefläche nach rechts kippen, so dass ein eventuell geladenes Paket abgeworfen wird. Nach dem Abwurf wird auch hier die Methode `actionCompleted()` aufgerufen. Kommt ein `startUnload()` Aufruf, während bereits eine Bewegung läuft, wird dieser verworfen.

Die Methode `startUnload()` kontrolliert nicht die Ausrichtung des Roboters zum Abwurfschacht.

3.3 Klasse `TaskProcessing`

Die aktive Klasse `TaskProcessing` ist ein Teil der Komponente `RobotControl` und wird im Paket `robot.control.tasks` implementiert. Diese Klasse verantwortet die Netzwerkkommunikation mit anderen Teilsystemen.

Die Klasse `TaskProcessing` erhält von einer externen Stelle die Aufträge. Beim Eingang eines Auftrages trägt die Klasse `TaskProcessing` die neuen Zielkoordinaten in der Klasse `DataSnapshot` ein und informiert die Klasse `DriveSystem` über den Eingang neuer Zielkoordinaten mit einem Aufruf der Methode `newTarget()`, der Methode `newUnloadingTarget()` oder der Methode `newChargingTarget()`.

Für Rückmeldungen der Klasse `DriveSystem` stellt die Klasse `TaskProcessing` zudem eine öffentlich sichtbare Methode `arrived()` zur Verfügung.

3.3.1 Methode `arrived()`

Die Methode `arrived()` informiert die Klasse `TaskProcessing` darüber, dass das hinterlegte Ziel von der Klasse `DriveSystem` erreicht wurde (inkl. der Fälle, bei denen der Roboter ein Paket abwirft bzw. zum Aufladen der Batterie einparkt). Der Aufruf der Methode `arrived()` löst aus, dass die Klasse `TaskProcessing` den Erfolg über das Netzwerk meldet und dann ggf. einen weiteren Auftrag für den Roboter erhalten kann.

3.4 Klasse `DataSnapshot`

Die Klasse `DataSnapshot` ist ein Teil der Komponente `RobotControl` und wird im Paket `robot.control.data` implementiert. Diese Klasse dient der Speicherung der jeweils aktuellen Sensordaten.

Um dem Statechart möglichst präzisen Zugriff auf die Sensordaten zu ermöglichen, implementiert die Klasse `DataSnapshot` eine Reihe von Methoden, die die Sensordaten präzise interpretieren. Diese Methoden lassen sich grob sortieren nach Zielüberprüfung, Orientierung im Raum und Hinderniserkennung.

Methoden zur Zielüberprüfung

3.4.1 Methode `isOnTarget()`

Die Methode `isOnTarget()` gibt einen `boolean` zurück. Die Rückgabe der Methode ist genau dann wahr, wenn die aktuelle Position des Roboters der Zielposition genau entspricht.

3.4.2 Methode `canUnloadToTarget()`

Die Methode `canUnloadToTarget()` gibt einen `boolean` zurück. Die Rückgabe der Methode ist genau dann wahr, wenn die Zielposition ein Abwurfschacht ist und der Roboter sich genau

ein Feld horizontal oder vertikal neben dieser Zielposition befindet, so dass er das Paket in den Abwurfschacht abwerfen kann.

3.4.3 Methode `canChargeAtTarget()`

Die Methode `canChargeAtTarget()` gibt einen `boolean` zurück. Die Rückgabe der Methode ist genau dann wahr, wenn die Zielposition eine Batterieladeposition ist und der Roboter sich genau ein Feld rechts von dieser Zielposition befindet, so dass er rückwärts einparken kann.

Methoden zur Orientierung im Raum

3.4.1 Methode `posType()`

Die Methode `posType()` gibt einen `PositionType` aus der entsprechenden Enumeration zurück. Die `PositionType`-Enumeration ist in Abbildung 5 gegeben und enthält als mögliche Werte `STATION`, `WAYPOINT`, `CROSSROAD` und `BLOCKED` (für Hindernisse). Mithilfe eines `PositionType`-Wertes kann festgestellt werden, auf welcher Art von Feld sich der Roboter derzeit befindet.

3.4.2 Methode `posOrientation()`

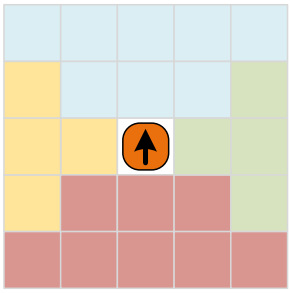
Die Methode `posOrientation()` gibt eine `Orientation` aus der entsprechenden Enumeration zurück. Die `Orientation`-Enumeration ist in Abbildung 5 gegeben und enthält als mögliche Werte `NORTH`, `SOUTH`, `EAST` und `WEST`. Mithilfe eines `Orientation`-Wertes kann festgestellt werden, in welche Himmelsrichtung ein Roboter ausgerichtet ist.

3.4.3 Methode `targetDirection()`

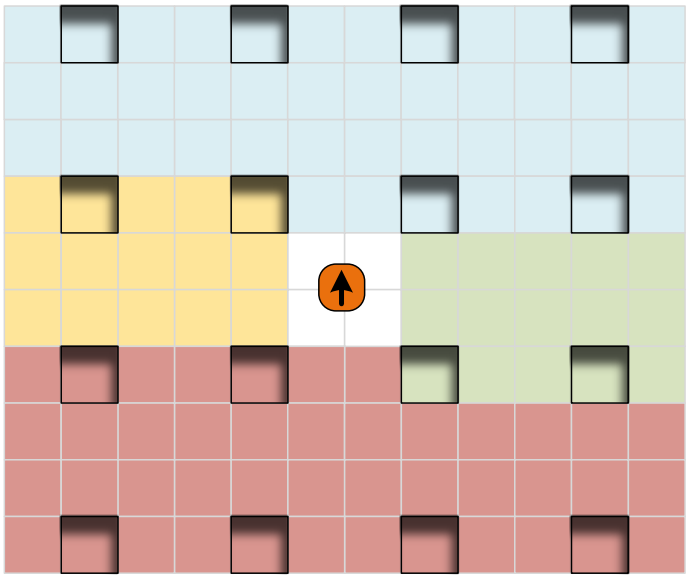
Die Methode `targetDirection()` gibt eine `Direction` aus der entsprechenden Enumeration zurück. Die `Direction`-Enumeration ist in Abbildung 5 gegeben und enthält als mögliche Werte `AHEAD`, `BEHIND`, `LEFT` und `RIGHT`. Mithilfe eines `Direction`-Wertes kann festgestellt werden, in welcher Richtung (ausgehend von der Fahrtrichtung des Roboters) sich das aktuelle Ziel befindet.

Die Methode `targetDirection()` nimmt bei ihrer Rückgabe Rücksicht auf das Rechtsfahrgebot, wenn Roboter auf einer Kreuzung oder einem Wegpunkt stehen. Bei den Zielen, die schräg vorne bzw. schräg hinten liegen, wird immer `AHEAD` bzw. `BEHIND` zurückgegeben. Die Rückgaben `LEFT` oder `RIGHT` kommen erst dann, wenn ein einmaliges Abbiegen genügt, um einen direkten Weg zum Ziel zu haben.

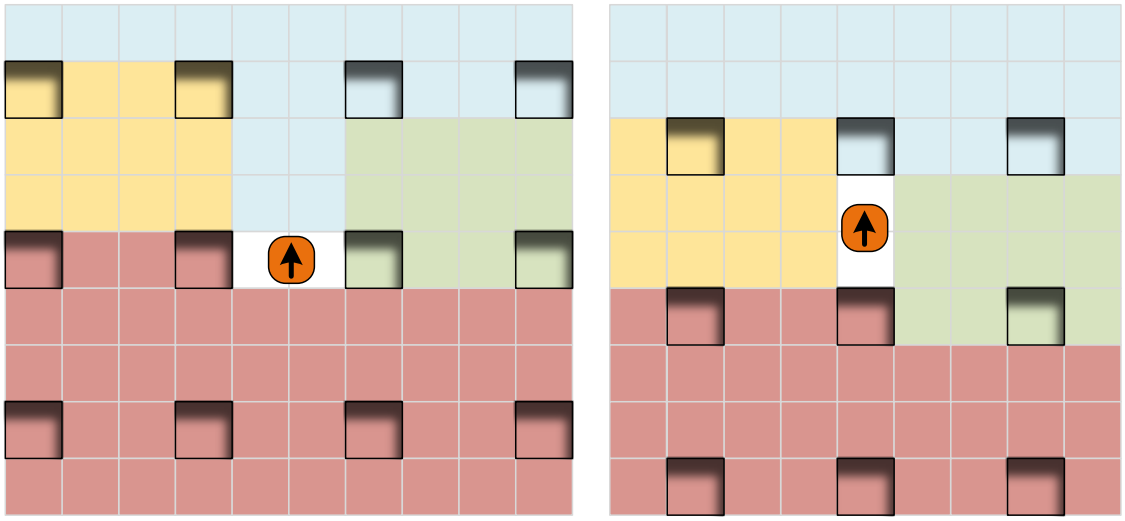
Die Rückgaben der Methode `targetDirection()` hängen von der Art von Feld ab, auf dem sich der Roboter befindet. Die schematische Darstellung dieser verschiedenen Rückgaben ist in Abbildung 6 visualisiert.



(a) Richtungsangaben für Roboter, die sich innerhalb einer Station befinden.



(b) Richtungsangaben für Roboter, die sich auf einer Kreuzung befinden.



(c) Richtungsangaben für Roboter, die sich auf einem Wegpunkt befinden.

Abbildung 6: Schematische Darstellung der Rückgaben von der Methode `targetDirection()` abhängig vom Standort des Roboters und des Ziels. In den Grafiken symbolisiert blau `AHEAD`, rot `BEHIND`, gelb `LEFT` und grün `RIGHT`.

Einfache Methoden zur Hinderniserkennung

Mithilfe der einfachen Methoden zur Hinderniserkennung der Klasse `DataSnapshot` ist es möglich abzufragen, ob eins der direkten Nachbarfelder des Roboters blockiert ist. Das Verhalten dieser Methoden ist in Abbildung 7a skizziert.

3.4.1 Methode `blockedFront()`

Die Methode `blockedFront()` gibt einen `boolean` zurück. Die Rückgabe der Methode ist genau dann wahr, wenn das in Fahrtrichtung voraus liegende Feld (ganz oder teilweise) blockiert ist (durch ein Hindernis, eine Trennwand, einen Abwurfschacht oder einen anderen Roboter). Siehe auch Abbildung 7a.

3.4.2 Methode `blockedLeft()`

Die Methode `blockedLeft()` gibt einen `boolean` zurück. Die Rückgabe der Methode ist genau dann wahr, wenn das in Fahrtrichtung links liegende Feld (ganz oder teilweise) blockiert ist (durch ein Hindernis, eine Trennwand, einen Abwurfschacht oder einen anderen Roboter). Siehe auch Abbildung 7a.

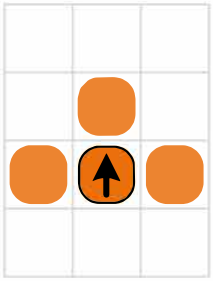
3.4.3 Methode `blockedRight()`

Die Methode `blockedRight()` gibt einen `boolean` zurück. Die Rückgabe der Methode ist genau dann wahr, wenn das in Fahrtrichtung rechts liegende Feld (ganz oder teilweise) blockiert ist (durch ein Hindernis, eine Trennwand, einen Abwurfschacht oder einen anderen Roboter). Siehe auch Abbildung 7a.

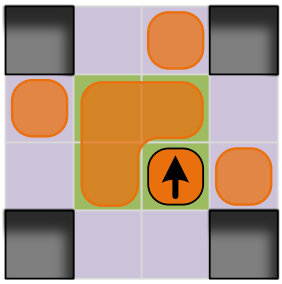
Komplexe Methoden zur Hinderniserkennung

Mithilfe der komplexen Methoden zur Hinderniserkennung der Klasse `DataSnapshot` ist es möglich das Fahrverhalten auf Kreuzungen und Wegpunkten zu definieren. Dabei wird die deutsche Straßenverkehrsordnung, konkret der Rechtsverkehr, besonders berücksichtigt.

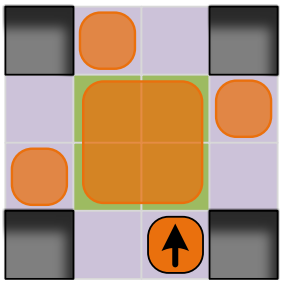
Die Methoden sind kontextsensitiv, d.h. je nach Position des Roboters werden verschiedene Rückgaben geliefert. Ist der Roboter auf einem Feld, bei dem eine bestimmte Abfragemethode unpassend ist, funktioniert diese Methode nach dem Prinzip „Garbage in, Garbage out“. So ist es z.B. nicht sinnvoll definiert, welche Rückgabe die Methode `blockedCrossroadAhead()` innerhalb einer Station liefert.



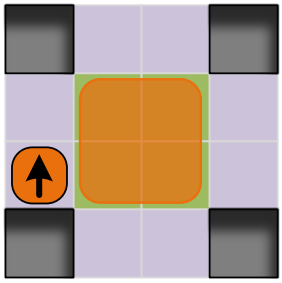
(a) Hindernisabfrage der direkt benachbarten Felder mit den Methoden `blocked*()`. Grund für eine Blockade können Hindernisse, Trennwände, Abwurfschächte und andere Roboter sein.



(b) Für Roboter auf einer Kreuzung: Hindernisabfrage der benachbarten Wegpunkte mit den Methoden `blockedWaypoint*()` und `blockedCrossroadAhead()` ausgehend von dem unten rechts liegenden Feld einer Kreuzung.



(c) Für Roboter auf einem Wegpunkt: Hindernisabfrage für eine voraus liegende Kreuzung mit der Methode `blockedCrossroadAhead()` sowie für die dahinter liegenden Wegpunkte mit den Methoden `blockedWaypoint*()`.



(d) Für Roboter auf einem Wegpunkt: Hindernisabfrage für eine rechts liegende Kreuzung mit der Methode `blockedCrossroadRight()`.

Abbildung 7: Funktionsweise der (einfachen und komplexen) Methoden zur Hinderniserkennung, die es der Klasse `DataSnapshot` ermöglichen zu prüfen, ob Felder in der Umgebung des Roboters blockiert sind.

3.4.1 Methode `blockedWaypointAhead()`

Die Methode `blockedWaypointAhead()` gibt einen `boolean` zurück.

Befindet sich der Roboter auf einem Wegpunkt mit Blick auf eine Kreuzung, gibt die Methode `blockedWaypointAhead()` genau dann wahr zurück, wenn ein anderer Roboter auf dem gegenüberliegenden Wegpunkt so steht, dass er (gemäß Rechtsverkehr) in die Kreuzung einfahren könnte (siehe Abbildung 7b).

Befindet sich der Roboter auf einer Kreuzung, wird dagegen geprüft, ob ein anderer Roboter so auf dem gegenüberliegenden Wegpunkt steht, dass der Roboter selbst die Kreuzung nicht verlassen kann (siehe Abbildung 7c).

3.4.2 Methode `blockedWaypointLeft()`

Die Methode `blockedWaypointLeft()` gibt einen `boolean` zurück.

Befindet sich der Roboter auf einem Wegpunkt mit Blick auf eine Kreuzung, gibt die Methode `blockedWaypointLeft()` genau dann wahr zurück, wenn ein anderer Roboter auf dem in Fahrtrichtung links liegenden Wegpunkt so steht, dass er (gemäß Rechtsverkehr) in die Kreuzung einfahren könnte (siehe Abbildung 7b).

Befindet sich der Roboter auf einer Kreuzung, wird dagegen geprüft, ob ein anderer Roboter so auf dem links liegenden Wegpunkt steht, dass der Roboter selbst die Kreuzung nicht verlassen kann (siehe Abbildung 7c).

3.4.3 Methode `blockedWaypointRight()`

Die Methode `blockedWaypointRight()` gibt einen `boolean` zurück.

Befindet sich der Roboter auf einem Wegpunkt mit Blick auf eine Kreuzung, gibt die Methode `blockedWaypointRight()` genau dann wahr zurück, wenn ein anderer Roboter auf dem in Fahrtrichtung rechts liegenden Wegpunkt so steht, dass er (gemäß Rechtsverkehr) in die Kreuzung einfahren könnte (siehe Abbildung 7b).

Befindet sich der Roboter auf einer Kreuzung, wird dagegen geprüft, ob ein anderer Roboter so auf dem rechts liegenden Wegpunkt steht, dass der Roboter selbst die Kreuzung nicht verlassen kann (siehe Abbildung 7c).

3.4.4 Methode `blockedCrossroadAhead()`

Die Methode `blockedCrossroadAhead()` gibt einen `boolean` zurück.

Befindet sich der Roboter auf einem Wegpunkt mit Blick auf eine Kreuzung, gibt die Methode `blockedCrossroadAhead()` genau dann wahr zurück, wenn die Kreuzung durch mindestens einen anderen Roboter blockiert ist (siehe Abbildung 7b).

Ist der Roboter bereits auf einer Kreuzung, gibt die Methode `blockedCrossroadAhead()` wahr zurück, wenn auf mindestens einem anderen Feld dieser Kreuzung bereits ein Roboter steht. Dabei darf der Roboter sich selbst nicht als Blockadegrund wahrnehmen (siehe Abbildung 7c).

3.4.5 Methode `blockedCrossroadRight()`

Die Methode `blockedCrossroadRight()` gibt einen `boolean` zurück. Die Rückgabe der Methode ist genau dann wahr, wenn die rechts vom Roboter liegende Kreuzung durch mindestens einen anderen Roboter blockiert ist (siehe Abbildung 7d).