PEGS

Private Environment Geofencing Solution

Entwurf

20. Dezember 2017

Betreuer: Marco Feuchter, Matthias Urschel (Fraunhofer IOSB)

Von: Stephan Bohr, Marco Esch, Moritz Laupichler, Niklas Meier, Marcel Mulorz, Kilian Wurm

Inhaltsverzeichnis

1 Einleitung

Die Entwurfsphase stellt den zweiten Teil des klassischen Wasserfallmodells dar. Sie befindet sich zwischen der Planungs-/Definitionsphase und der Implementierungsphase. In dieser Phase soll der Übergang des in der Planungs-/Definitionsphase erstellten Pflichtenhefts zu einer softwaretechnischen Implementierung in der Implementierungsphase geschaffen werden.
Genauer ist das Ziel der Entwurfsphase, aus den im Pflichtenheft spezifizierten Anforderungen einen Entwurf für ein Software-System zu erstellen, welcher wegweisende Grundlage für das Artefakt der Implementierungsphase ist. Dies geschieht durch Modellierung einer Pakethierarchie und dazugehörigen Klassen, welche mit Hilfe von Software-Architekturstilen und -Entwurfsmustern arrangiert werden. Des Weiteren werden durch Sequenzdiagramme bereits Abläufe des späteren Softwareprodukts dargestellt, und damit die Funktionsweise des zu entwerfenden Softwareprodukts anschaulich erklärt.

2 Sequenzdiagramme

Das Verhalten des PEGS Systems soll im Folgenden durch einige Sequenzdiagramme dargestellt werden. Die Sequenzdiagramme gehen aus den im Pflichtenheft definierten Anwendungsfällen hervor. Die Diagramme sind exemplarisch für Abläufe, die in verschiedenen Situationen auftreten, zu betrachten (siehe hierzu auch die detaillierten Beschreibungen der Diagramme). Daher kommen in den Diagrammen Querverweise vor, wenn solche häufiger vorkommenden Abläufe der Lesbarkeit halber nicht mehrfach definiert werden.

2.1 Starten des Systems

In diesem Sequenzdiagramm wird der Ablauf beim Starten bzw. der Initialisierung des Systems beschrieben.

In Abbildung 2.1.1 werden der zeitliche Ablauf sowie die Bekanntmachung der Instanziierungen geordnet dargestellt. Da diese Schritte für mehrere Komponenten wiederholt werden müssen, wird im Diagramm das Vorgehen nur exemplarisch für die Komponente `Permission` dargestellt. Für die anderen Komponenten erfolgt es analog.

Dieser Ablauf dient dazu, dass die Beziehungen der einzelnen Teile der MVC-Architektur korrekt gebildet werden.

Die einzelnen Pakete bauen ihre interne Struktur beim Konstruieren auf, im Diagramm sind die von außen zu bewirkenden Initialisierungen dargestellt.

Der Netzwerkadministrator weist die Initialisierung des Systemes an. Dies geschieht durch die Methode `initializeSystem` in der Klasse `SystemInitializer`. Diese Methode veranlasst nun verschiedene Instanziierungen:

1. `da : ConcreteDatabaseAccessor` mit Generic `Permission`.
2. `pdm : ConcretePersistentDataManager` mit Generic `Permission`, der wiederum `da` durch den Konstruktor kennt und einen `PersistentDataComponentObserver`

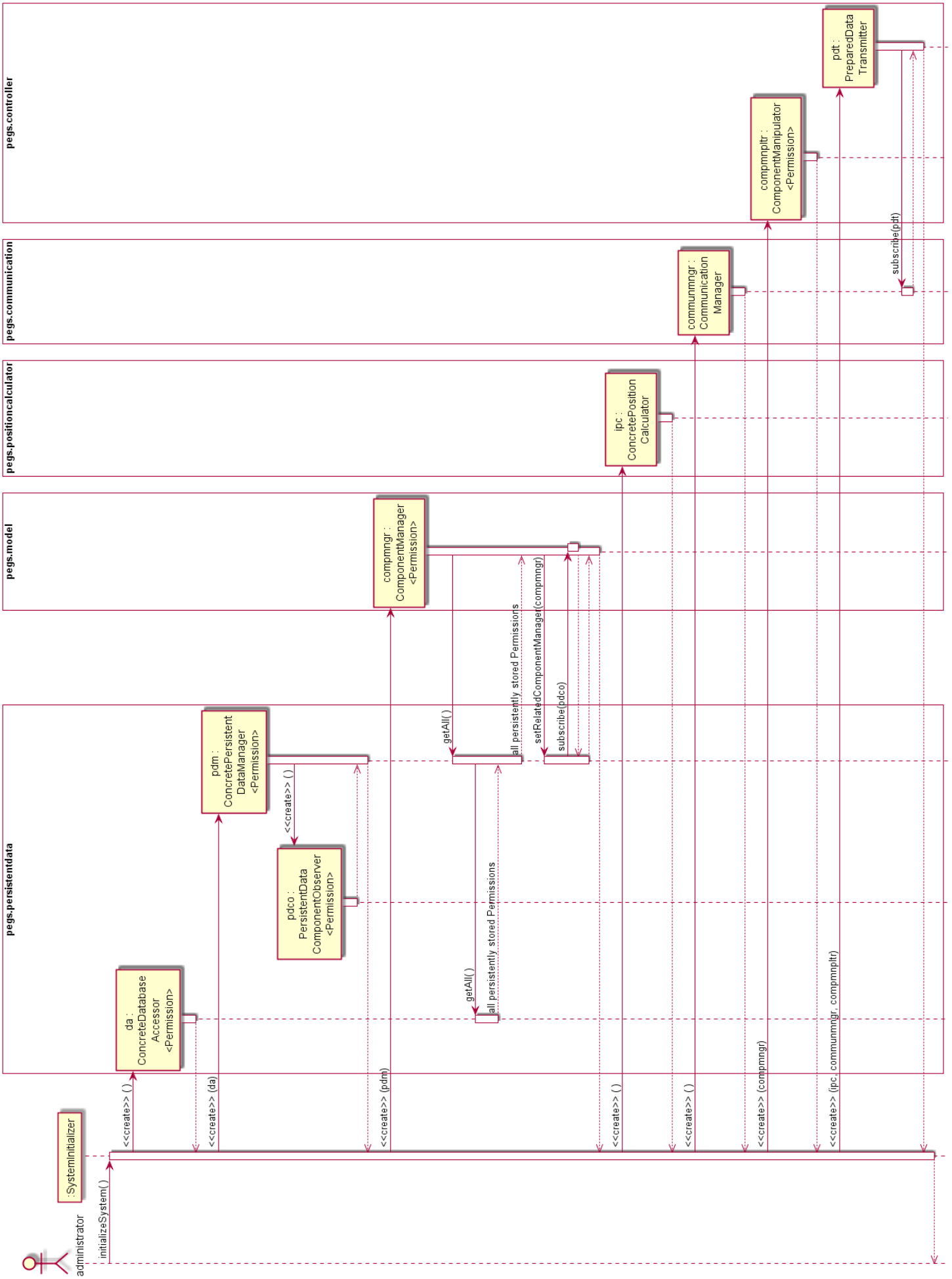

Abbildung 2.1.1: Sequenzdiagramm zur Initialisierung des Systems.

`pdco` mit Generic `Permission` konstruiert.

3. `compmngr: ComponentManager` mit Generic `Permission`, der den `PersistentDataManger pdm` kennt, um sich zuerst alle schon im Speicher vorhandenen Daten seines Typs (also `Permission`) anzufordern (mittels der `getAll()`-Methode) und danach mittels der `setRelatedComponentManager(compmngr)` zu veranlassen, dass der zuvor erzeugte Observer `pdco` sich bei ihm anmeldet.
4. `ipc : ConcretePositionCalculator`. Diese Instanz muss nicht pro Komponente erzeugt werden.
5. `communmngr : CommunicationManager`. Diese Instanz muss nicht pro Komponente erzeugt werden.
6. `compmnpltr : ComponentManipulator` mit Generic `Permission`, der `communmngr` durch den Konstruktor kennt.
7. `pdt : PreparedDataTransmitter`, der `ipc`, `communmngr` und `compmnpltr` durch den Konstruktor kennt und sich als Observer beim `communmngr` anmeldet.

Nun sind die Systembestandteile instantiiert und die notwendigen Beziehungen gebildet. Durch die MVC-Architektur und das Observer-Pattern wird nach der Initialisierung ein eigenständiger Betrieb ermöglicht.

Verwandte Anwendungsfälle: Keine. Der Start des Systems ist allerdings von großer Wichtigkeit und sollte trotz fehlendem Anwendungsfall behandelt werden.

2.2 Hinzufügen eines Bereichs aus Sicht des Controllers

Im folgenden Sequenzdiagramm wird der systeminterne Ablauf bei Hinzufügen eines Bereichs über die GUI zum System im Geltungsbereich des Controller-Pakets dargestellt.

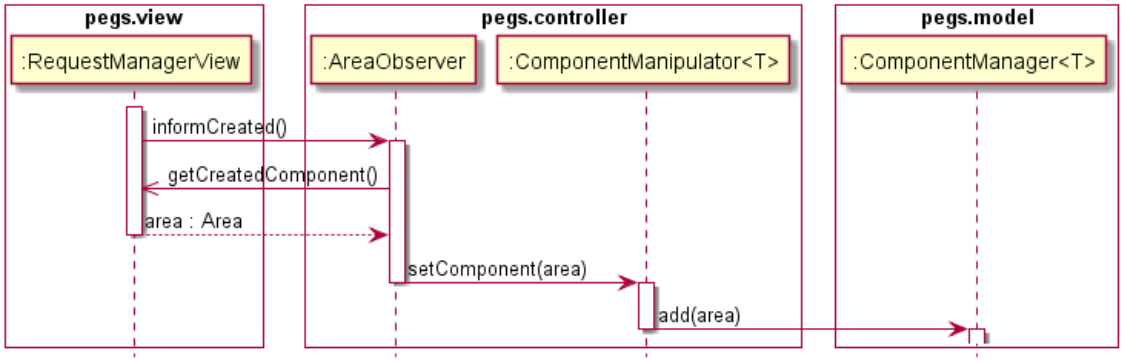

Abbildung 2.2.1: Hinzufügen eines Bereichs

Der Ablauf in Sequenzdiagramm 2.2.1 beginnt, nachdem durch den Netzwerkadministrator ein neuer Bereich in der GUI hinzugefügt wurde. Das Controller-Paket wird benachrichtigt, dass es Änderungen im View-Paket gab, indem aus der Klasse `RequestManagerView` des View-Pakets die Methode `informCreated()` der Klasse `AreaObserver` im Controller-Paket aufgerufen wird. Daraufhin holt sich die `AreaObserver`-Klasse die neu erzeugte

`Area` aus dem View-Paket über die Methode `getCreatedComponent()` der `RequestManagerView`-Klasse. Diese `Area` wird an die Klasse `ComponentManipulator` über die Methode `setComponent(area)` weitergegeben. Diese Methode veranlasst die Datenmanipulation des Model-Pakets durch den Aufruf der Methode `add(area)` der `ComponentManager`-Klasse.

Verwandte Anwendungsfälle: /ANW40/

2.3 Positionsberechnung für ein Endgerät auf dem Server

Das Sequenzdiagramm 2.3.1 stellt den systeminternen Verlauf von der Übergabe von vorsortierten Daten für die Positionsberechnung eines Endgeräts bis zum Erstellen dieses Endgeräts dar.

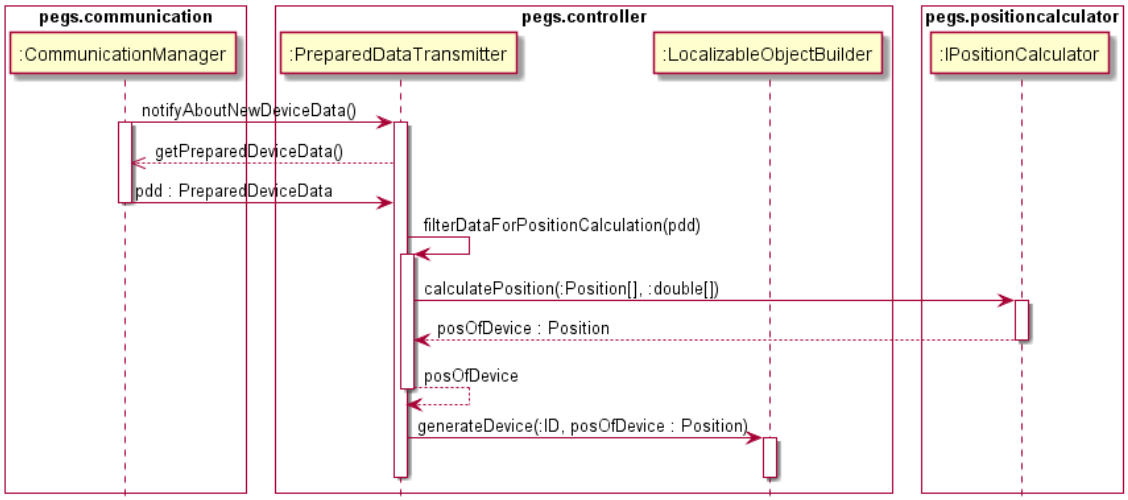

Abbildung 2.3.1: Erstellen eines Device-Objekts mit zugehöriger Positionsberechnung

Der Ablauf in Sequenzdiagramm 2.3.1 beginnt, nachdem im Communication-Paket die von den Access Points eintreffenden Daten für die Weiterverarbeitung umstrukturiert wurden. Die Klasse `CommunicationManager` benachrichtigt mit der Methode `notifyAboutNewDeviceData()` die `PreparedDataTransmitter`-Klasse des Controller-Pakets, welche sich daraufhin über die im `CommunicationManager` enthaltene Methode `getPreparedDeviceData()`, die von der `CommunicationManager`-Klasse vorsortierten Daten `pdd` holt.
Diese werden in der Methode `filterDataForPositionCalculation()` der Klasse `PreparedDataTransmitter` gefiltert, so dass ein Array aus `Position` und ein Array aus Gleitkommazahlen(double) entsteht. Die Arrays über die Schnittstelle `IPositionCalculator` vom Controller- zum PositionCalculator-Paket übergeben. Diese Filtrierung geschieht, damit das PositionCalculator-Paket nur die Daten erhält, die es zum Berechnen einer `Position` benötigt. So bleibt die `ID` des Endgeräts im Controller-Paket für das PositionCalculator-Paket verborgen. Die Rückgabe `posOfDevice` ist die `Position` des Endgeräts. Diese wird zusammen mit der `ID` des Endgeräts an die Klasse LocalizableObjectBuilder des Controller-Pakets weitergegeben, in dem dann das Endgerät-Objekt erzeugt und gespeichert wird.

Verwandte Anwendungsfälle: Keine. Verarbeitung von eingehenden Daten zu Endgeräten ist allerdings so elementar, dass es hierfür ein Sequenzdiagramm geben sollte.

2.4 Übertragen von Endgerätdaten innerhalb des Communication-Pakets

In Sequenzdiagramm 2.4.1 wird die Übertragung von „rohen “ Endgerätdaten innerhalb des Communication-Pakets, von dem AccessPointClient-Paket zum Controller-Paket, dargestellt.

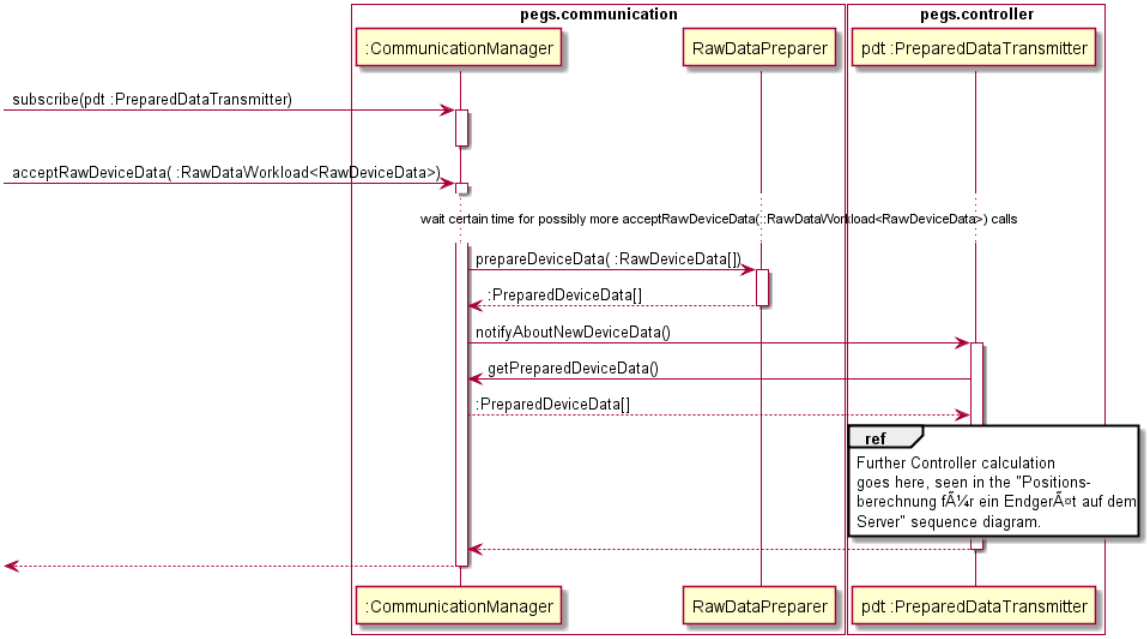

Abbildung 2.4.1: UML-Sequenzdiagramm zur Übertragung von Endgerätdaten im Communication-Paket

Beim Starten des Systems wird nach dem Erstellen einer Instanz des `CommunicationManagers`, dieser eine Instanz des `PreparedDataTransmitter`, wie im Sequenzdiagramm 2.4.1 zu sehen, mittels `subscribe( :PreparedDataTransmitter)` hinzugefügt, welche später die Funktion eines Beobachters auf neue Endgerätdaten realisiert.
Ab diesem Zeitpunkt kann der `CommunicationManager` von dem AccessPointClient-Paket durch die Methode `acceptRawDeviceData( :RawDataWorkload<RawDeviceData>)` (siehe Sequenzdiagramm 2.4.1) aufgefordert werden, neue Endgerätdaten zu akzeptieren. Diese Aufforderung findet über Wi-Fi statt. Durch die Aufforderung startet ein Countdown, währenddessen auf weitere Aufforderungen, neue Endgerätdaten zu akzeptieren, gewartet wird. Ist dieser Countdown abgelaufen, bereitet der `CommunicationManager` die „rohen “ Endgerätdaten mithilfe der Methode `prepareDeviceData( :RawDeviceData[])` der `RawDataPreparer`-Klasse auf, wie im Sequenzdiagramm 2.4.1 zu sehen ist.
Nachdem der `CommunicationManager` die aufbereiteten Endgerätdaten zurückerhält, benachrichtigt er den zuvor erwähnten `PreparedDataTransmitter` durch die Methode `notifyAboutNewDeviceData()` über veränderte Endgerätdaten. Der `PreparedDataTrans-`

mitter fordert die aufbereiteten Endgerätdaten durch die Methode getPreparedDeviceData() vom CommunicationManager an. Nachdem der PreparedDataTransmitter diese erhalten hat, werden sie innerhalb des Controller-Pakets weiterverarbeitet, zu sehen im Sequenzdiagramm 2.3.1.

Verwandte Anwendungsfälle: /ANW10/, /ANW20/

2.5 Hinzufügen eines Bereiches zu einer Kopplungsgruppe

Das folgende Sequenzdiagramm 2.5.1 beschreibt, wie, nachdem in der GUI ein Bereich zu einer Kopplungsgruppe hinzugefügt wurde, diese Änderung durch den Controller an das Model weitergegeben wird. Der in Diagramm 2.5.1 beschriebene Vorgang beschränkt sich auf die Pakete der MVC-Architektur, Model, View und Controller.

Diagramm 2.5.1 beschreibt zunächst eine nötige Ausgangssituation, indem ein Beobachter des Pakets Controller, der auf die Änderung von Kopplungsgruppen in der GUI hört (linkGroupObserver vom Typ LinkGroupObserver), durch die Klasse SystemInitializer im Zuge der Methode initializeGUI() am InstructionManagerView im View-Paket angemeldet wird.
Der beschriebene Ablauf beginnt im View-Paket, wenn in der GUI eine Nutzerinteraktion eine Verknüpfung zwischen einem Bereich und einer Kopplungsgruppe herstellt und linkGroup.add(:Area) aufruft (nicht dargestellt, vgl. Diagramm 2.6.1). Die veränderte Kopplungsgruppe linkGroup wird mit dem Aufruf setChangedLinkGroup(linkGroup) an das Objekt vom Typ GUIManager übergeben. Das Objekt vom Typ GUIManager macht nun einen Selbstaufruf mit der Methode notifyViewObserver(), damit alle angemeldeten Beobachter über die Veränderung benachrichtigt werden.
Hierzu wird getId() auf linkGroup aufgerufen, was die ID von linkGroup id zurückgibt. Der GUIManager ruft nun update(id) auf o.g. linkGroupObserver auf, womit der Ausführungsfokus jetzt im Paket Controller liegt. linkGroupObserver gibt die ID nun an den ComponentManipulator<LinkGroup> weiter, indem informChanged(id) aufgerufen wird.
Die ID der geänderten LinkGroup gelangt nun in das Paket Model, indem der ComponentManipulator<LinkGroup> auf dem ComponentManager<LinkGroup> notifyUpdated(id) aufruft. Damit liegt der Ausführungsfokus nun im Paket Model. Mit id wird sichergestellt, dass eine Kopplungsgruppe mit derselben ID schon im Model existiert: Der ComponentManager führt einen Selbstaufruf getById(id) auf, um den Datenbestand nach der ID zu befragen. Schließlich benachrichtigt der ComponentManager alle angemeldeten ComponentObserver über die Änderung von linkGroup. Es wird hier beispielhaft dargestellt, wie auf einer anonymen Instanz von ComponentObserver<LinkGroup> refreshByIds(id) aufgerufen wird. Der ComponentObserver holt sich dann die geänderte Kopplungsgruppe linkGroup durch den Aufruf getById(id) auf obigem ComponentManager<LinkGroup>. Die Veränderung, die von der GUI im Paket View ausgeht, wird also schließlich über ein Beobachtermuster durch das Paket Controller an das Paket Model weitergegeben, wo sie in die bestehende Datenhaltung integriert wird.

Diagramm 2.5.1 dient exemplarisch der Darstellung des Datenverkehrs beim Koppeln sowie Entkoppeln von Berechtigungen und Bereichen in Kopplungsgruppen, da diese

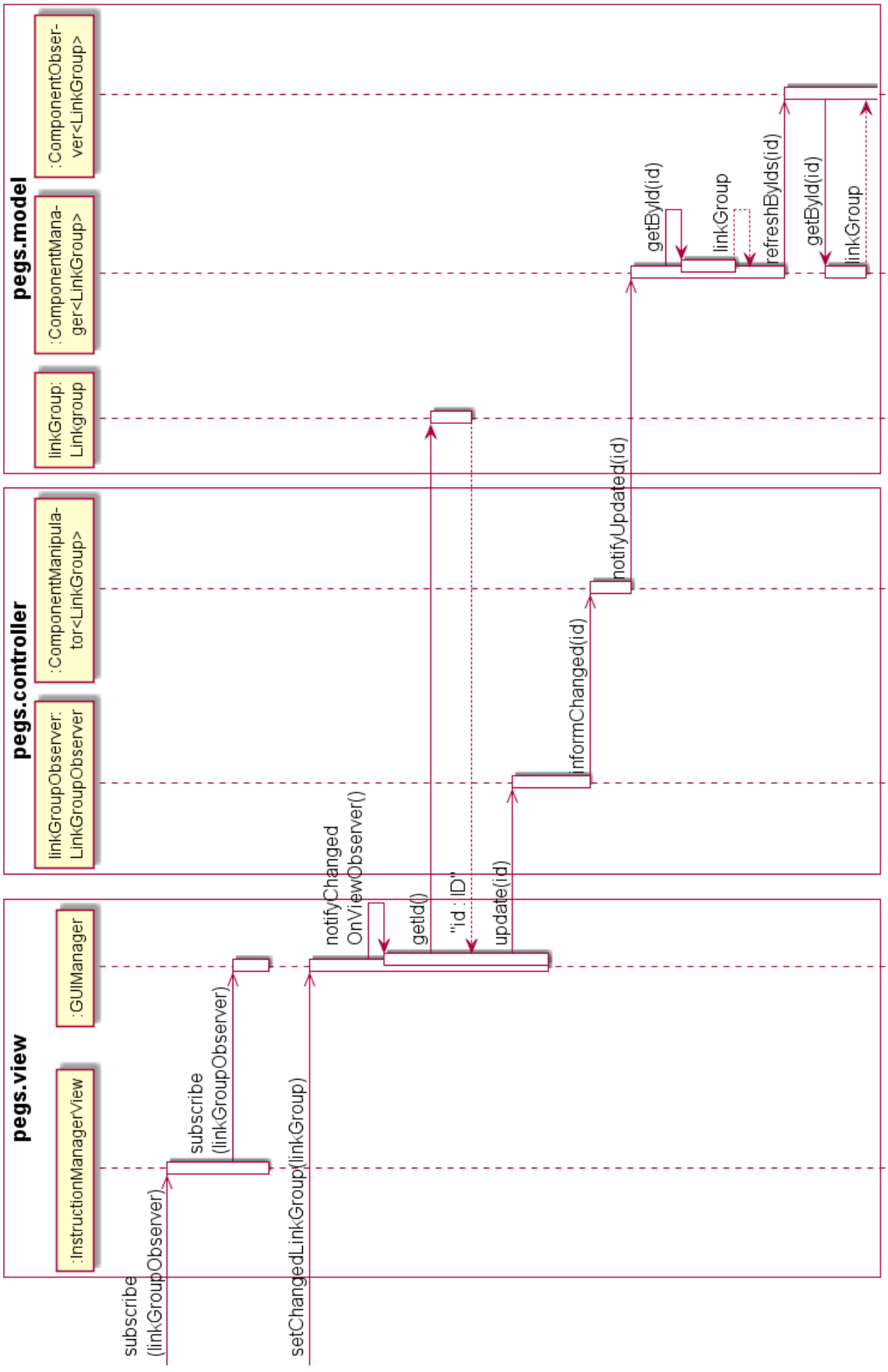

Abbildung 2.5.1: Hinzufügen eines Bereiches zu einer Kopplungsgruppe.

Anwendungsfälle einen sehr ähnlichen Ablauf und dasselbe Zusammenspiel zwischen den Paketen View, Controller und Model aufweisen.

Verwandte Anwendungsfälle: /ANW80/, /ANW90/, /ANW110/, /ANW120/

2.6 Hinzufügen eines Bereiches mit Fokus auf View

Im Sequenzdiagramm 2.6.1 wird die Kommunikation zwischen den einzelnen Klassen des View-Pakets dargestellt, wenn ein neuer Bereich in diesem erstellt wird. Wie die Verarbeitung des Bereichs im Controller abläuft finden sie hier: 2.2.1

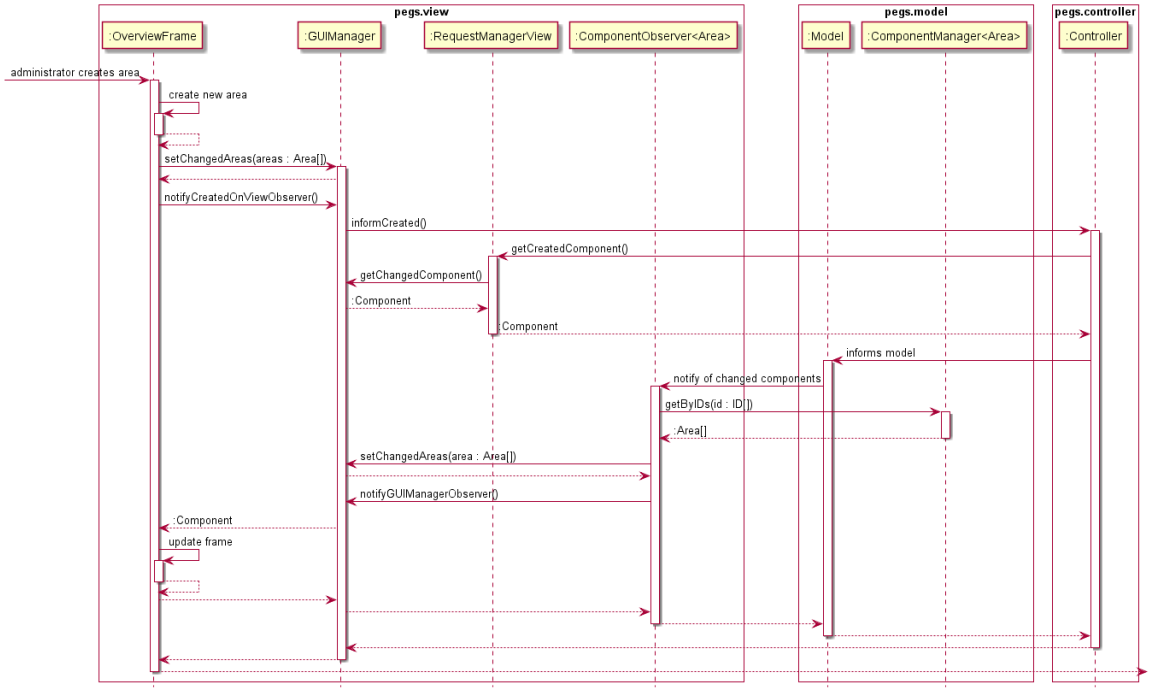

Abbildung 2.6.1: Hinzufügen eines Bereiches mit Fokus auf dem Paket View

Wird ein neuer Bereich durch den Administrator erstellt, so kann dies nur im Fenster `OverviewFrame` geschehen. Dabei wird ein neues Objekt der Klasse `Area` erstellt. Danach leitet das Fenster das neue Objekt an den `GUIManager` weiter und ruft auf diesem die Methode `notifyCreatedOnViewObserver()` auf. Daraufhin meldet der `GUIManager` den Observern, die das View-Paket beobachten, dass es eine Änderung der Business Data im View-Paket gab, woraufhin sich diese den neuen Bereich über die Klasse `RequestManagerView` holt. Ist die Änderung im Model-Paket eingetragen, wird der `ComponentObserver<Area>` über eine Änderung informiert. Dieser holt sich den veränderten Bereich, reicht ihn an den `GUIManager` weiter und ruft auf diesem die Methode `notifyGUIManagerObserver()` auf. Der `GUIManager` informiert alle `IGUIManagerObserver` über den geänderten Bereich. Diese holen sich den geänderten Bereich und, wenn nötig, aktualisieren ihre Anzeige.
Beim Hinzufügen von anderen Komponenten passiert dies analog mit der Änderung, dass der Observer des geänderten Komponententyps durch das Model-Paket informiert wird. Bei einem `AccessPoint` wird nur dessen Name und Beschreibung zurückgegeben.

Verwandte Anwendungsfälle: /ANW40/, /ANW60/, /ANW70/

2.7 Sammeln und Broadcasten von Rohdaten

Im Sequenzdiagramm 2.7.1 wird beschrieben, wie dieser Daten sammelt, ordnet und an benachbarte Access Points oder gegebenfalls den Server weiterleitet.

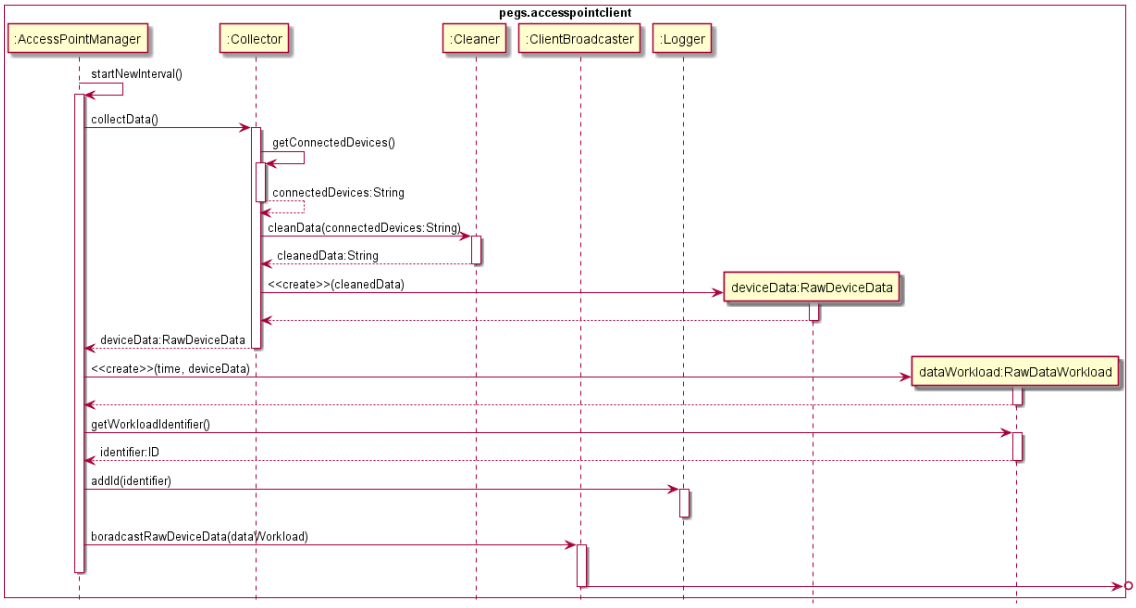

Abbildung 2.7.1: Broadcasten von Rohdaten nach Sammlung durch Access Point

Der `AccessPointManager` hält in seiner internen Struktur einen Timer, der nach Ablauf einer bestimmten Zeit die Methode `startNewInterval()` aufruft. Mit dem Aufrufen dieser Methode beginnt für den Access Point ein neuer Intervall, in dem dieser Daten sammelt, ordnet und weiterleitet. Durch das Aufrufen der `startNewIntervall()`-Methode wird zunächst im `Collector` die Methode `collectData()` aufgerufen. Dadurch werden vom Access Point die verbundenen Endgeräte, sowie deren Signalstärken durch die Methode `getConnectedDevices()` abgefragt. Da diese Informationen jedoch noch nicht so geordnet sind, dass sie weiter verarbeitet werden können, werden sie an den `Cleaner` weitergegeben. Hierfür bietet der `Cleaner` die Methode `cleanData(:String)` an. Dieser filtert nun die benötigten Daten heraus und ordnet sie so, dass sie vom `Collector` weiterverarbeitet werden können. Sobald der `Collector` Daten erhalten hat, erstellt er ein neues `RawDeviceData`-Objekt, welches er mit den vom `Cleaner` erhaltenen Daten füllt und an den `AccessPointManager` gibt. Dieser erstellt nun zum Versenden der Daten ein `RawDataWorkload`-Objekt.

Durch dieses Objekt können die Daten nun auch vom Access Point zum Server oder zu anderen Access Points versendet werden. Zuvor holt sich der `AccessPointManger` mit der Methode `getIdentifier()` noch die eindeutige Identifikation des `RawDataWorkload`-Objekts. Diese leitet er an den `Logger` weiter, damit sie in die Liste erhaltener IDs eingetragen werden kann. Sollte dieser Datensatz nun noch einmal von einem anderen

Access Point an diesen gesendet werden, leitet der Acess Point seine selbst gesammelten Daten nicht noch ein weiteres Mal weiter.
Zum Versenden gibt der `AccessPointManager` das Datenset direkt an den `ClientBroadcaster`. Dieser sendet es anschließend an benachbarte Access Points und gegenfalls den Server.

Verwandte Anwendungsfälle: Keine. Jedoch ist das Sammeln oben genannter Daten unabdingbar für die Funktionen des PEGS-Systems und muss somit behandelt werden.

2.8 Starten der GUI

Im Sequenzdiagramm 2.8.1 wird die Erstellung der einzelnen Objekte im Paket View beim Start der GUI dargestellt.

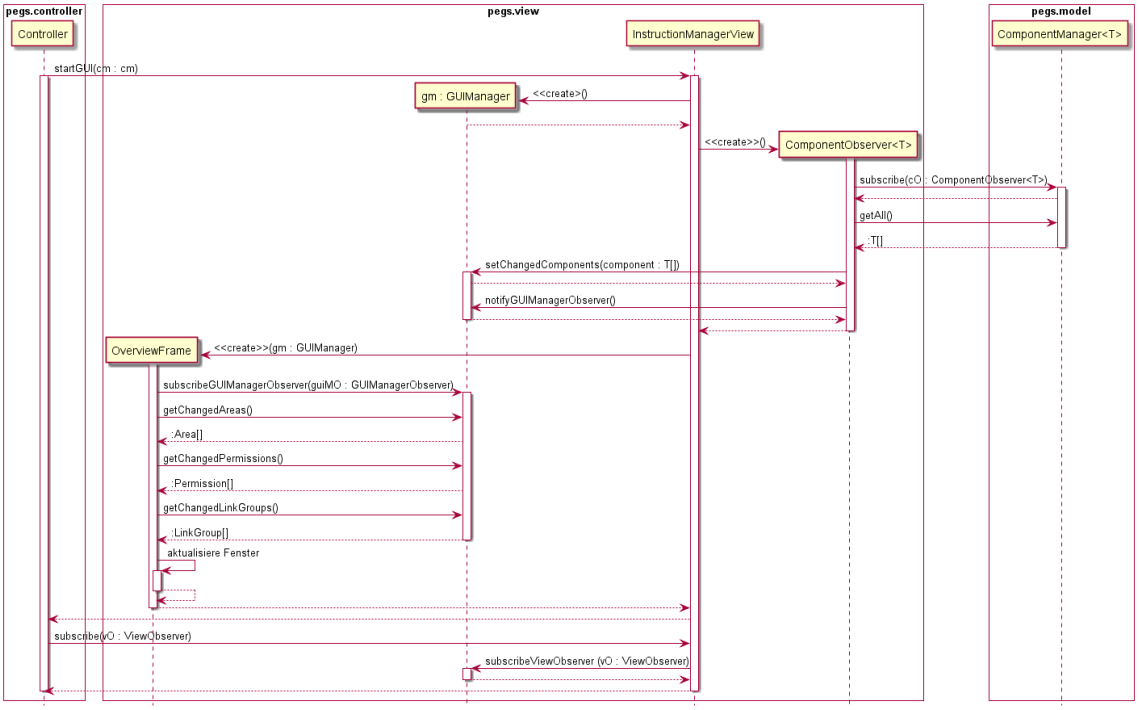

Abbildung 2.8.1: Starten der GUI

Der Start der GUI geht vom Paket Controller aus. Hierfür ruft dieses auf dem `InstructionManagerView` die Methode `startGUI(cm : cm)` mit allen ihm bekannten `ComponentManager<T>` auf. Der `InstructionManagerView` erstellt darauf hin den `GUIManager` und weist `ComponentObserver<T extends Component>` den entsprechenden `ComponentManager<T>` zu. Die `ComponentObserver` updaten dann den `GUIManager`. Der `InstructionManagerView` erstellt dann das `OverviewFrame`, welches sich beim `GUIManager` anmeldet und alle Business Data vom `GUIManager` holt. Darauf hin aktualisiert es sich. Ist dies geschehen meldet der Controller alle `ViewObserver` beim `InstructionManagerView` an.

Verwandte Anwendungsfälle: Keine. Der Start der GUI ist aber essentiell und sollte somit nicht in einem Sequenzdiagramm fehlen.

2.9 Verarbeiten von eingehenden Daten auf einem Access Point

Das Sequenzdiagramm 2.9.1 beschreibt das Verhalten eines Access Points beim Empfangen von Endgerätdaten eines anderen Access Points.

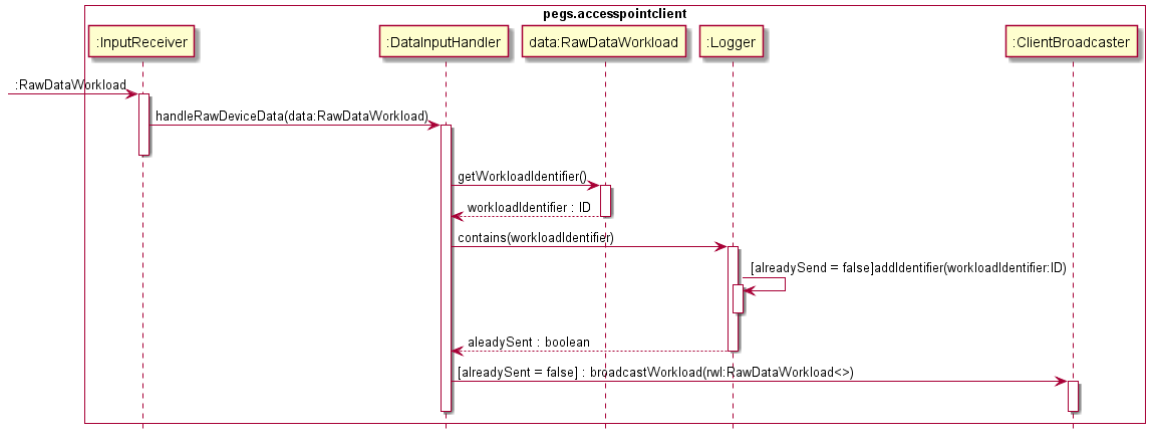

Abbildung 2.9.1: Verarbeiten von eingehenden Daten anderer Access Points auf einem Access Point

Zu Beginn erhält der Access Point von einem ihm benachbarten Access Point ein Datenset. Dieses beinhaltet Informationen über mit einem Access Point verbundene Endgeräte sowie deren Signalstärken zum Access Point. Über den `InputReceiver` erhält der Access Point diese Daten im Format `RawDeviceData`. Diese gibt er anschließend mit der Methode `handleRawDeviceData()` an den `DataInputHandler` weiter, der für die Verarbeitung der Daten zuständig ist.
Der `DataInputHandler` überprüft nun mithilfe des `Loggers`, ob dieses spezielle Datenpaket bereits in diesem Access Point verarbeitet wurde. Um dies zu überprüfen wird der `workloadIdentifier` der übermittelten `RawDataWorkload` verwendet. Dieser wird an den `Logger` übergeben, welcher in einer eigenen Liste `IDs` bereits erhaltener `Workloads` speichert. Der `Logger` überprüft, ob der übergebene Identifier sich bereits in der Liste befindet und somit schon vorher empfangen wurde. Ist dies der Fall, gibt er `true` zurück und der `DataInputHandler` ignoriert das Datenset. Ist es jedoch das erste Mal, dass dieses Datenset den Access Point erreicht und ist der Identifier somit noch nicht in der Liste des `Loggers`, so wird dem `DataInputHandler` ein `false` zurückgegeben. Des Weiteren wird die `ID` des Datensets auf die Liste geschrieben.
Der `DataInputHandler` gibt das `RawDataWorkload`-Datenset in diesem Fall an den `ClientBroadcaster` weiter, welcher es an die verbundenen Access Points, sowie gegebenenfalls den Server weiterleitet.

Verwandte Anwendungsfälle: Keine. Da der Fall jedoch eintreten kann, dass Access Points nicht direkt mit dem Server kommunizieren können muss das Weiterleiten von Daten über mehrere Access Points hinweg behandelt werden.

2.10 Synchronisierung der Systemzeit auf Access Points

Das Sequenzdiagramm 2.10.1 beschreibt den Vorgang, der in einem Access Point stattfindet, wenn dieser einen neuen `timeWorkload` erhält, durch den er seine Zeit an die des Servers anpasst.

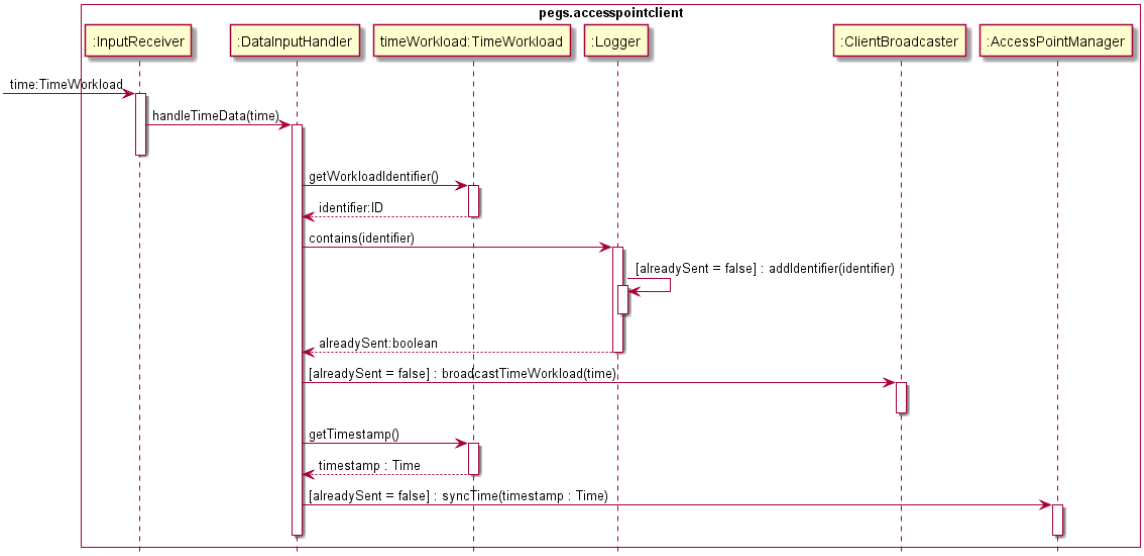

Abbildung 2.10.1: Synchronisierung der Systemzeit auf einem Access Point durch Signal von außen

Zu Beginn erhält der `InputReceiver` des Access Points ein `timeWorkload`-Objekt vom Server oder einem benachbarten Access Point. Das `Time`-Objekt der `timeWorkload` wird nach vollständigem Erhalten an den `DataInputHandler` weitergeleitet.
Um zu überprüfen, ob dieses `TimeWorkload`-Objekt schon zu einem früheren Zeitpunkt beim Access Point angekommen ist, wird dessen `ID` mithilfe des `Logger`s gegen eine Liste mit `ID`s abgeglichen, in welcher sich die Identifikationen vorheriger Datensets befinden. Hierfür holt sich der `DataInputHandler` vom `TimeWorkload`-Objekt dessen eindeutige Identifikation und überträgt sie an den `Logger`. Dieser gleicht sie wie oben beschrieben gegen seine Liste ab. Sollte er diese `ID` bereits in seine Liste haben, heißt das für den Access Point, dass er dieses Datenset bereits ein mal verarbeitet hat. Somit kann er es nun ignorieren. Dafür gibt der `Logger` dem `DataInputHandler` ein `false` zurück.
Ist es jedoch das erste mal, dass dieses Datenset den Access Point erreicht und der `Logger` die `ID` noch nicht kennt, so gibt er dem `DataInputHandler` ein false zurück und fügt die `ID` seiner Liste hinzu.
Daraufhin leitet der `DataInputHandler` das `TimeWorkload`-Objekt an den `ClientBroadcaster` weiter, der es an benachbarte Access Points oder gegebenfalls den Server sendet.

Außerdem holt er sich vom `TimeWorkload`-Objekt das `Time`-Attribut. Dieses gibt er über die `syncTime(:Time`-Methode des `AccessPointManager`s an diesen weiter. Der `AccessPointManager` gleicht seine interne Zeit der neu erhaltenen Zeit an.

Verwandte Anwendungsfälle: Keine. Um die Funktionsfähigkeit des PEGS-Systems

zu erhalten, ist das Abgleichen der Serverzeit mit den Zeiten der Access Points jedoch notwendig. Somit sollte dies trotz eines fehlenden Anwendungsfalles behandelt werden.

3 Pakete - Übersicht

Die folgenden Abschnitte des Entwurfsdokumentes beschreiben alle Pakete, aus denen sich PEGS zusammensetzt. Jedes Paket wird spezifiziert durch Titel, Klassendiagramm, Funktion, Schnittstellen, innere Struktur und die JavaDoc-Dokumentation aller enthaltenen Klassen und Interfaces. Die Pakete sind in alphabetischer Ordnung aufgeführt:

- 4 Access Point Client
- 5 Communication
- 6 Controller
- 7 Model
- 8 Persistent Data
- 9 Position Calculator
- 10 View

4 AccessPointClient

Das AccessPointClient-Paket ist für die Funktionalität des Access Points verantwortlich. Des Weiteren bietet es Schnittstellen zur Kommunikation mit anderen Access Points und dem Communication-Paket an.

4.1 Klassendiagramm

Das Diagramm 4.1.1 zeigt die Klassen des Pakets AccessPointClient.

4.2 Funktion

Das AccessPointClient-Paket holt sich nach einem bestimmten Zeitintervall Informationen über die mit dem Access Point verbunden Endgeräte und filtert die benötigten Daten, also Endgerätinformationen sowie Signalstärken, aus dem Datensatz heraus. Die gefilterten Daten werden anschließend an alle Nachbarn versendet. Außerdem werden auch Daten von anderen Access Points empfangen und verarbeitet sowie gegebenenfalls weitergeleitet.

4.3 Schnittstellen

Das Paket bietet eine Schnittstelle zu anderen Access Points sowie dem Server an. Mit dieser imitiert der Access Point die Schnittstelle, die der Server den Access Points anbietet.

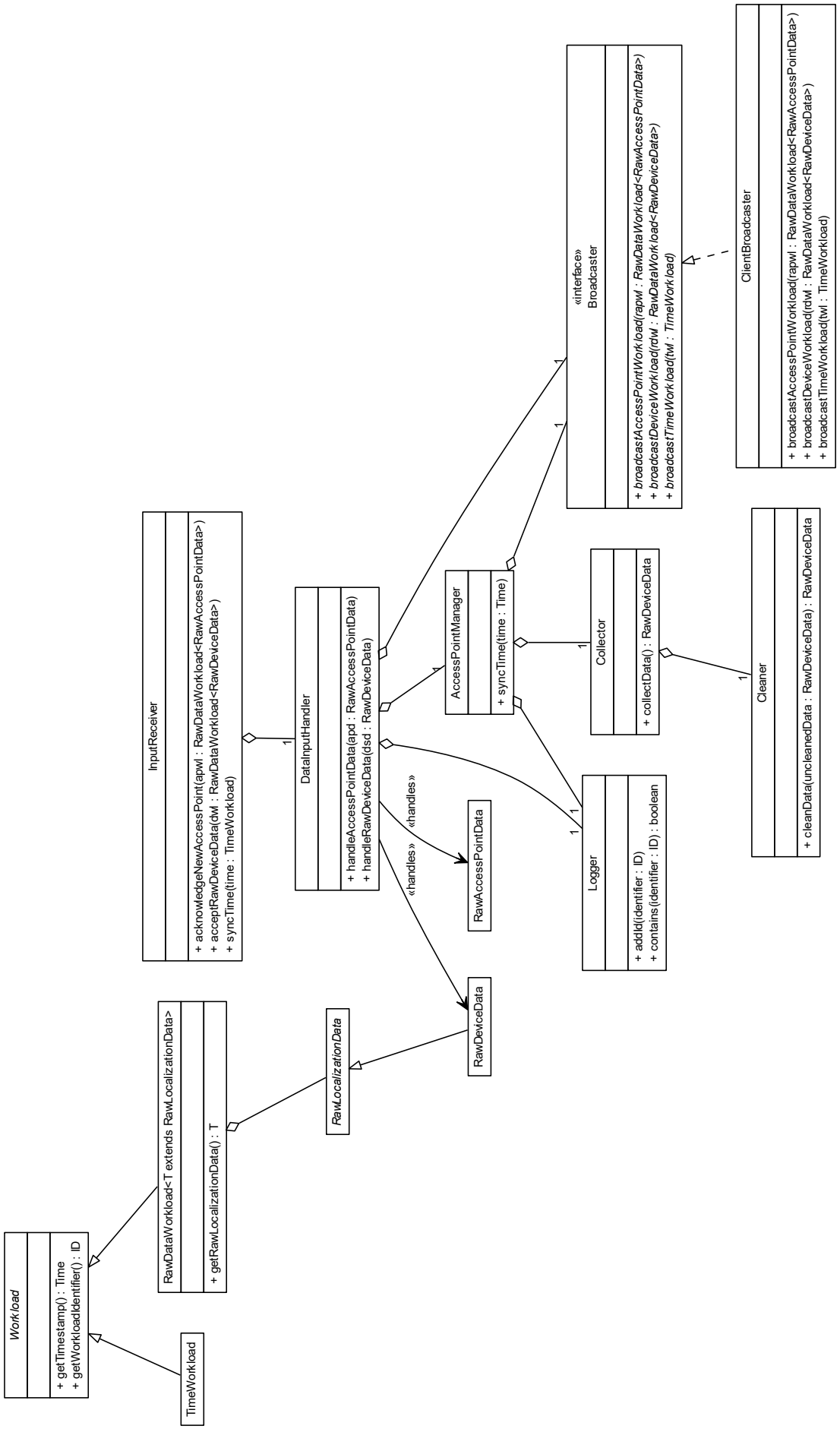

Abbildung 4.1.1: UML-Klassendiagramm zum Paket AccessPointClient.

Ein Access Point weiß somit nicht, ob er seine Daten an den Server oder einen anderen Access Point sendet. Man kann hier von einem adaptierten Proxy-Entwurfsmuster reden. Hinter der Schnittstelle wird in diesem Fall verborgen, ob es sich um den Server oder ein Access Point handelt. Dadurch wird das Weiterleiten von Daten eines Access Points, der keine direkte Verbindung zum Server vereinfacht. Er muss nicht differenzieren, ob er seine Daten an einen Access Point oder den Server sendet.

4.4 Interne Struktur

Den Kern des Pakets bildet die `AccessPointManager`-Klasse, die eine Schnittstelle zwischen den beiden Teilen des Pakets darstellt. Zum einen gibt es den Teil des Pakets, der sich um die Kommunikation mit anderen Access Points kümmert. Wie oben beschrieben bietet er die Schnittstelle an, mit der andere Access Points ihm Informationen senden können und über die er selbst Informationen versendet. Zur Annahme eingehender Daten bietet die Klasse `InputReceiever` die nötigen Methoden an, mit denen Endgerätedaten, Accesspointdaten oder die Serverzeit empfangen werden können. Die zum Versenden ausgehender Daten verwendete `ClientBroadcaster`-Klasse bietet Methoden zum Versenden eben jener Daten an. Zwischen diesen beiden befindet sich der `DataInputHandler`, der für die Verwaltung eingehender Daten verantwortlich ist. Dieser überprüft mit Hilfe des `Loggers`, der die IDs der bereits eingegangenen Datenpakete kennt, ob ein erhaltenes Datenpaket zu einem früheren Zeitpunkt bereits empfangen wurde oder nicht. Erhält ein Access Point einen solchen Datensatz zum ersten Mal, merkt sich der `Logger` dessen ID, die er vom `DataInputHanler` erhält. Falls es sich bei den Daten nicht um einen `TimeWorkload` handelt, gibt der `DataInputHandler` die Daten weiter an den `ClientBroadcaster`, der sie wiederum an benachbarte Access Points oder den Server sendet. Ist der erhaltene `Workload` jedoch ein `TimeWorkload`, so wird dieser zusätzlich verwendet, um die Zeit zwischen Server und Access Point zu synchronisieren. Der `DataInputHandler` gibt dann die `Time`-Instanz des `Workload`s an den `AccessPointManager` weiter, der diese dann übernimmt.
Zum anderen gibt es den Teil des AccessPointClient-Pakets, der dafür verantwortlich ist Daten innerhalb des Access Points zu erheben und diese zum Versenden vorzubereiten. Dieser Teil ist für andere Access Points und den Server nicht sichtbar. Der oben genannte Timer innerhalb des `AccessPointManager`s ist dafür verantwortlich nach Ablauf einer bestimmten Zeit die Daten neu zu erfassen und zu versenden. Hierfür sendet er dem `Collector` zunächst die Aufforderung, die Daten über mit dem Access Point verbundene Endgeräte zu sammeln. Dazu gehören eine eindeutige Identifikation des Endgerätes und die Signalstärke zwischen diesem Endgerät und dem Access Point. Diese Daten gibt der `Collector` anschließend an den `Cleaner` weiter, der aus diesen nur die Informationen herausfiltert, die benötigt werden und gibt sie dem `Collector` zurück. Dieser erstellt aus den Daten ein neues `RawLocalizationData`-Objekt, welches er an den `AccessPointManager` weitergibt. Der `AccessPointManager` kann aus diesem `RawLocalizationData`-Objekt und seiner Systemzeit ein neues `RawDeviceData`-Objekt erzeugen. Dessen `ID` holt sich der `AccessPointManager` um sie an den `Logger` weiter zu leiten. Diese `ID` wird somit in die Liste bereits verarbeiteter Datensets geschrieben. Dadurch wird vermieden, dass bereits verarbeitete Datensets erneut verarbeitet und weitergeleitet werden. Das `RawDeviceData`-Objekt leitet der `AccessPointManager` anschließend noch an den `ClientBroadcaster` weiter, welcher es an benachbarte Access Points und gegebenenfalls den Server sendet.

4.5 Package pegs.accesspointclient

Package Contents *Page*

Interfaces

Classes

Used to transmit time stamps between access points and between server and access points.

Workload 29
Represents a data package which can be sent between access points and between access points and the server.

Interface Broadcaster

Interface to send data out to nearby access points and to the server.

Declaration

```
public interface Broadcaster
```

All known subinterfaces

ClientBroadcaster (in 4.5, page 21)

All classes known to implement interface

ClientBroadcaster (in 4.5, page 21)

Methods

- **broadcastAccessPointWorkload**

  ```
  void broadcastAccessPointWorkload(RawDataWorkload rapwl)
  ```

  - **Description**

    Broadcasts a workload containing data for localization of Access Points to other Access Points and to the server.

  - **Parameters**

    * `rapwl` – The workload to broadcast.

- **broadcastDeviceWorkload**

  ```
  void broadcastDeviceWorkload(RawDataWorkload rdwl)
  ```

- **Description**

  Broadcasts a workload containing data for localization of devices to other access points and the server.

- **Parameters**

  * `rdwl` – The workload to broadcast.

- **broadcastTimeWorkload**

```
void broadcastTimeWorkload(TimeWorkload twl)
```

- **Description**

  Broadcasts a workload containing only a time stamp for time synchronization purposes.

- **Parameters**

  * `twl` – The workload to be broadcast.

Class AccessPointManager

Manages the communication between incoming data and created information within the access point.

Declaration

```
public class AccessPointManager
 extends java.lang.Object
```

Constructors

- **AccessPointManager**

```
public AccessPointManager()
```

Methods

- **syncTime**

```
public void syncTime(pegs.communication.Time time)
```

- **Description**

  Set the access point time to sync it with the server time.

- **Parameters**
  - `time` – The time coming form the server.

Class Cleaner

Cleans the data set for easier handling. It filters out the needed information from a larger data set.

Declaration

```
public class Cleaner
 extends java.lang.Object
```

Constructors

- **Cleaner**

  ```
  public Cleaner()
  ```

Methods

- **cleanData**

  ```
  public RawDeviceData cleanData(RawDeviceData uncleanedData)
  ```

  - **Description**

    Cleans the given data set. Extracts the needed data from a larger data set.
  - **Parameters**
    - `uncleanedData` – the data set before it was cleaned.
  - **Returns** – returns the data set which will be sent to the server.

Class ClientBroadcaster

Used by access points to communicate with its neighboring access points or the server. Implements the Broadcaster Interface.

Declaration

```
public class ClientBroadcaster
 extends java.lang.Object implements Broadcaster
```

Constructors

- **ClientBroadcaster**

  ```
  public ClientBroadcaster()
  ```

Methods

- **broadcastAccessPointWorkload**

  ```
  void broadcastAccessPointWorkload(RawDataWorkload rapwl)
  ```

  - **Description copied from Broadcaster** (**in 4.5, page 19**)

    Broadcasts a workload containing data for localization of Access Points to other Access Points and to the server.

  - **Parameters**

    * `rapwl` – The workload to broadcast.

- **broadcastDeviceWorkload**

  ```
  void broadcastDeviceWorkload(RawDataWorkload rdwl)
  ```

  - **Description copied from Broadcaster** (**in 4.5, page 19**)

    Broadcasts a workload containing data for localization of devices to other access points and the server.

  - **Parameters**

    * `rdwl` – The workload to broadcast.

- **broadcastTimeWorkload**

  ```
  void broadcastTimeWorkload(TimeWorkload twl)
  ```

  - **Description copied from Broadcaster** (**in 4.5, page 19**)

    Broadcasts a workload containing only a time stamp for time synchronization purposes.

  - **Parameters**

    * `twl` – The workload to be broadcast.

Class Collector

Collects information about mobile devices connected to the specific access point.

Declaration

```
public class Collector
 extends java.lang.Object
```

Constructors

- **Collector**

  ```
  public Collector()
  ```

Methods

- **collectData**

  ```
  public RawDeviceData collectData()
  ```

  - **Description**

    Collects the data from the access point before sending it to the cleaner to clean it up.

  - **Returns** – returns the cleaned up data set

Class DataInputHandler

Manages all the incoming data and distributes it to the intended receiving class. If an incoming data set was received for the first time it will be forwarded to all nearby access points or the server.

Declaration

```
public class DataInputHandler
 extends java.lang.Object
```

Constructors

- **DataInputHandler**

  ```
  public DataInputHandler()
  ```

Methods

- **handleAccessPointData**

  ```
  public void handleAccessPointData(RawAccessPointData apd)
  ```

  - **Description**

    Manages the incoming data containing a newly connected access point.

  - **Parameters**

    * `apd` – data about a newly connected access point

- **handleRawDeviceData**

  ```
  public void handleRawDeviceData(RawDeviceData dsd)
  ```

  - **Description**

    Manages the incoming data containing the devices from another access point.

  - **Parameters**

    * `dsd` – raw device data from an access point about its connected devices.

Class InputReceiver

Interface to receive data from nearby access points or form the server.

Declaration

```
public class InputReceiver
 extends java.lang.Object implements pegs.communication.
    IAccessPointCommunication
```

Constructors

- **InputReceiver**

  ```
  public InputReceiver()
  ```

Methods

- **acceptRawDeviceData**

  ```
  void acceptRawDeviceData(RawDataWorkload dwl)
  ```

  - **Description copied from pegs.communication.IAccessPointCommunication (in 5.5, page 33)**

    Accepts raw device data workload from an Access Point.

  - **Parameters**

    * `dwl` – The workload containing the Device localization data.

- **acknowledgeNewAccessPoint**

  ```
  void acknowledgeNewAccessPoint(RawDataWorkload apwl)
  ```

  - **Description copied from pegs.communication.IAccessPointCommunication (in 5.5, page 33)**

    Acknowledge notification and localization data of a newly `activated` Access Point.

  - **Parameters**

    * `apwl` – The workload containing the Access Point localization data.

- **syncTime**

  ```
  void syncTime(TimeWorkload time)
  ```

  - **Description copied from pegs.communication.IAccessPointCommunication (in 5.5, page 33)**

    Hands off the given time to synchronize the receiving end time with the senders time.

  - **Parameters**

    * `time` – The timestamp to synchronize to.

Class Logger

Keeps track of the data set which already reached this access point.

Declaration

```
public class Logger
 extends java.lang.Object
```

Constructors

- **Logger**

  ```
  public Logger()
  ```

Methods

- **addId**

  ```
  public void addId(pegs.model.ID identifier)
  ```

  - **Description**

    Adds the ID of a data set to its list of already received data sets.

  - **Parameters**

    * `identifer` – the ID of the data set to be added.

- **contains**

  ```
  public boolean contains(pegs.model.ID identifier)
  ```

  - **Description**

    Checks if the given ID is already in the list

  - **Parameters**

    * `accessPoint` – true, if the data set was already received before, else false.

Class RawAccessPointData

The data containing information about the location of an access point as collected by other access points. Format depends on protocol used for transmission between Server and clients.

Declaration

```
public class RawAccessPointData
 extends pegs.accesspointclient.RawLocalizationData
```

Constructors

- **RawAccessPointData**

```
public RawAccessPointData()
```

Class RawDataWorkload

Bundle of data to send from Access Point to Server. Includes information about localization data collected by Access Points as well as other information regarding transmission.

Declaration

```
public class RawDataWorkload
 extends pegs.accesspointclient.Workload
```

Constructors

- **RawDataWorkload**

```
public RawDataWorkload(pegs.communication.Time t,
    RawLocalizationData rld)
```

  - **Description**

    Constructor for the workload.

  - **Parameters**

    * `t` – the time stamp.
    * `rld` – the localization data.

Methods

- **getRawLocalizationData**

```
public RawLocalizationData getRawLocalizationData()
```

  - **Description**

    Getter for the localization data set.

  - **Returns** – the localization data set.

Members inherited from class Workload

`pegs.accesspointclient.Workload` (in 4.5, page 29)
- `public Time` **getTimestamp()**
- `public ID` **getWorkloadIdentifier()**

Class RawDeviceData

The data containing information about currently seen devices received directly from an Access Point that serves as a client. Format depends on protocol used for transmission between Server and clients.

Declaration

```
public class RawDeviceData
 extends pegs.accesspointclient.RawLocalizationData
```

Constructors

- **RawDeviceData**

  ```
  public RawDeviceData()
  ```

Class RawLocalizationData

Represents data collected by an Access Point regarding localization.

Declaration

```
public abstract class RawLocalizationData
 extends java.lang.Object
```

All known subclasses

RawDeviceData (in 4.5, page 28), RawAccessPointData (in 4.5, page 26)

Constructors

- **RawLocalizationData**

  ```
  public RawLocalizationData()
  ```

Class TimeWorkload

Used to transmit time stamps between access points and between server and access points.

Declaration

```
public class TimeWorkload
 extends pegs.accesspointclient.Workload
```

Constructors

- **TimeWorkload**

  ```
  public TimeWorkload()
  ```

Members inherited from class Workload

`pegs.accesspointclient.Workload` (in 4.5, page 29)
- `public Time` **getTimestamp**`()`
- `public ID` **getWorkloadIdentifier**`()`

Class Workload

Represents a data package which can be sent between access points and between access points and the server.

Declaration

```
public abstract class Workload
 extends java.lang.Object
```

All known subclasses

TimeWorkload (in 4.5, page 29), RawDataWorkload (in 4.5, page 27)

Constructors

- **Workload**

  ```
  public Workload()
  ```

Methods

- **getTimestamp**

  ```
  public pegs.communication.Time getTimestamp()
  ```

  - **Description**

    Getter for the time stamp of the bundle.

  - **Returns** – The time stamp of the workload.

- **getWorkloadIdentifier**

  ```
  public pegs.model.ID getWorkloadIdentifier()
  ```

  - **Description**

    Returns a unique identifier for this workload.

  - **Returns** – The ID of the workload.

5 Communication

Das `Communication`-Paket dient dem Datenaustausch zwischen dem `AccessPointClient`-Paket und dem `Controller`-Paket.

5.1 Klassendiagramm

Das Diagramm 5.1.1 zeigt die Klassen des Pakets Communication.

5.2 Funktion

Das Communication-Paket empfängt Lokalisationsdaten von den Access Points. Diese werden für den späteren Gebrauch aufbereitet, indem redundante Daten entfernt werden und alle restlichen Daten an den Controller weitergegeben werden.

5.3 Schnittstellen

Die Schnittstelle zu dem AccessPointClient-Paket ist durch das `Interface IAccessPoint-Communication` definiert. Sie nimmt Informationen über neu aktivierte Access Points und Lokalisationsdaten entgegen.

Die Schnittstelle zu dem Controller-Paket ist durch das `Interface IPreparedDataDistributor` definiert. Dieses Interface gibt die aufbereiteten Lokalisationsdaten der Access Points weiter.

Abbildung 5.1.1: UML-Klassendiagramm zum Paket Communication.

Das `Interface CommunicationObserver` stellt einen Beobachter auf neue Daten der Access Points dar.

5.4 Interne Struktur

Um den Datenaustausch von Access Points zum Controller-Paket zu ermöglichen, bietet das Paket Communication zwei Schnittstellen an. Das Interface `IAccessPointCommunication`, bietet mit der Methode `acceptRawDeviceData( :RawDataWorkload<RawDeviceData>)` die Funktion, Rohdaten von Endgeräten zu empfangen. Dies geschieht über eine Wi-Fi Verbindung. Außerdem kann sie durch die Methode `acknowledgeNewAccessPoint( :RawDataWorkload<RawAccessPointData)` aufgefordert werden, einen neuen Access Point zu akzeptieren, sodass dieser im PEGS-System genutzt werden kann.
Die zweite Schnittstelle, das Interface `IPreparedDataDistributor` bietet mit den Methoden `getNewAccessPointPreparedData()` und `getPreparedDeviceData()` die Funktion, aufbereitete Access Point-Daten bzw. Endgerätdaten zu erhalten.
Die Klasse `RawDataPreparer` stellt Methoden zur Aufbereitung empfangener Daten bereit. Die Methode `prepareDeviceData( :RawDeviceData[])` bereitet hierbei Endgerätdaten, die Methode `prepareAccessPointData( :RawAccessPointData[])` Access Point-Daten auf.
Das Zusammenspiel zwischen diesen Schnittstellen und der `RawDataPreparer`-Klasse stellt die `CommunicationManager`-Klasse dar. Sie implementiert beide Schnittstellen und nutzt die Methoden der `RawDataPreparer`-Klasse, um Daten, welche die `CommunicationManager`-Klasse mithilfe des Interface `IAccessPointCommunication` erhalten hat, weiter zu verarbeiten, um diese danach mit dem Interface `IPreparedDataDistributor` zur Verfügung zu stellen.
Die `CommunicationManager`-Klasse besitzt außerdem einen `CommunicationObserver`-Beobachter, welcher das Controller-Paket über geänderte aufbereitete Daten zu informieren, sodass das Controller-Paket diese dann über das Interface `IPreparedDataDistributor` anfordern kann.
Das Communication-Paket beinhaltet außerdem die Klasse `ServerBroadcaster` welche es dem Controller-Paket durch die Methode `broadcastTimeWorkload( :TimeWorkload)` ermöglicht, die Zeit aller Access Points gleich zu setzen. Dies wird für die Timer Funktionen der Klassen des AccessPointClient-Paket benötigt.

5.5 Package pegs.communication

Package Contents *Page*

Interfaces

Classes

Interface IAccessPointCommunication

Describes the receiving end of the communication with access points. Provides methods to accept raw device data and raw access point data. Needs to be used by an actual communication protocol.

Declaration

```
public interface IAccessPointCommunication
```

All known subinterfaces

CommunicationManager (in 5.5, page 35)

All classes known to implement interface

CommunicationManager (in 5.5, page 35)

Methods

- **acceptRawDeviceData**

  ```
  void acceptRawDeviceData(pegs.accesspointclient.
    RawDataWorkload dwl)
  ```

  - **Description**

    Accepts raw device data workload from an Access Point.

  - **Parameters**

    * `dwl` – The workload containing the Device localization data.

- **acknowledgeNewAccessPoint**

  ```
  void acknowledgeNewAccessPoint(pegs.accesspointclient.
    RawDataWorkload apwl)
  ```

  - **Description**

    Acknowledge notification and localization data of a newly `activated` Access Point.

  - **Parameters**

    * `apwl` – The workload containing the Access Point localization data.

- **syncTime**

  ```
  void syncTime(pegs.accesspointclient.TimeWorkload time)
  ```

  - **Description**

    Hands off the given time to synchronize the receiving end time with the senders time.

  - **Parameters**

    * `time` – The timestamp to synchronize to.

Interface IPreparedDataDistributor

Describes the distributing end of the communication package towards the controller package. Provides methods for distributing access point data and device data.

Declaration

```
public interface IPreparedDataDistributor
```

All known subinterfaces

CommunicationManager (in 5.5, page 35)

All classes known to implement interface

CommunicationManager (in 5.5, page 35)

Methods

- **getNewAccessPointPreparedData**

  ```
  PreparedAccessPointData getNewAccessPointPreparedData()
  ```

  - **Description**

    Gets previously calculated `PreparedAccessPointData` (in 5.5, page 38) so the access point may be added to the PEGS-System.

  - **Returns** – The `PreparedAccessPointData` (in 5.5, page 38) of the access point.

- **getPreparedDeviceData**

  ```
  PreparedDeviceData[] getPreparedDeviceData()
  ```

  - **Description**

    Gets the previously calculated `PreparedDeviceData` (in 5.5, page 39) to make localization possible.

  - **Returns** – The `PreparedDeviceData` (in 5.5, page 39) of the devices that have been reached by the access points.

Class CommunicationManager

Manages data passing between access points and Controller package. Buffers raw data received by the access points for preparation. Informs CommuncationObserver about changed data so the controller package can request the new data.

Declaration

```
public class CommunicationManager
 extends java.lang.Object implements IPreparedDataDistributor,
    IAccessPointCommunication, pegs.common.IObservable
```

Constructors

- **CommunicationManager**

  ```
  public CommunicationManager()
  ```

Methods

- **acceptRawDeviceData**

  ```
  void acceptRawDeviceData(pegs.accesspointclient.
     RawDataWorkload dwl)
  ```

  - **Description copied from IAccessPointCommunication** (**in 5.5, page 33**)

    Accepts raw device data workload from an Access Point.

  - **Parameters**

    * `dwl` – The workload containing the Device localization data.

- **acknowledgeNewAccessPoint**

  ```
  void acknowledgeNewAccessPoint(pegs.accesspointclient.
     RawDataWorkload apwl)
  ```

  - **Description copied from IAccessPointCommunication** (**in 5.5, page 33**)

    Acknowledge notification and localization data of a newly `activated` Access Point.

  - **Parameters**

    * `apwl` – The workload containing the Access Point localization data.

- **getNewAccessPointPreparedData**

  ```
  PreparedAccessPointData getNewAccessPointPreparedData()
  ```

– **Description copied from IPreparedDataDistributor (in 5.5, page 34)**

Gets previously calculated `PreparedAccessPointData` (in 5.5, page 38) so the access point may be added to the PEGS-System.

– **Returns** – The `PreparedAccessPointData` (in 5.5, page 38) of the access point.

- **getPreparedDeviceData**

```
PreparedDeviceData[] getPreparedDeviceData()
```

– **Description copied from IPreparedDataDistributor (in 5.5, page 34)**

Gets the previously calculated `PreparedDeviceData` (in 5.5, page 39) to make localization possible.

– **Returns** – The `PreparedDeviceData` (in 5.5, page 39) of the devices that have been reached by the access points.

- **subscribe**

```
public void subscribe(CommunicationObserver obs)
```

– **Description**

Subscribes an observer to notifications about prepared data previously received from the access points.

– **Parameters**

∗ `obs` – The Observer to subscribe.

- **syncTime**

```
void syncTime(pegs.accesspointclient.TimeWorkload time)
```

– **Description copied from IAccessPointCommunication (in 5.5, page 33)**

Hands off the given time to synchronize the receiving end time with the senders time.

– **Parameters**

∗ `time` – The timestamp to synchronize to.

- **unsubscribe**

```
public void unsubscribe(CommunicationObserver obs)
```

– **Description**

Unsubscribes an observer from notifications about prepared data.

– **Parameters**

* `obs` – The Observer to unsubscribe.

Class CommunicationObserver

Realizes an observer observing the communication package. Will get notified if new data has been prepared, i.e. `PreparedDeviceData` (in 5.5, page 39) or `PreparedAccessPointData` (in 5.5, page 38).

Declaration

```
public abstract class CommunicationObserver
 extends java.lang.Object implements pegs.common.IObserver
```

Constructors

- **CommunicationObserver**

  ```
  public CommunicationObserver()
  ```

Methods

- **notifyAboutNewAccessPointData**

  ```
  public void notifyAboutNewAccessPointData()
  ```

  - **Description**

    Informs the observer about an access point request to be added to the PEGS-System.

- **notifyAboutNewDeviceData**

  ```
  public void notifyAboutNewDeviceData()
  ```

  - **Description**

    Informs observer about new prepared user device data, so the controller package can then request it.

Class PreparedAccessPointData

Realizes data consisting of lists of access points paired with their signal strengths to a single given newly found access point.

Declaration

```
public class PreparedAccessPointData
 extends java.lang.Object
```

Constructors

- **PreparedAccessPointData**

  ```
  public PreparedAccessPointData()
  ```

Methods

- **getAccessPointPositions**

  ```
  public pegs.model.Position[] getAccessPointPositions()
  ```

  - **Description**

    Gets a list of all positions of access points that have received a signal from the newly found access point.

  - **Returns** – All access point positions.

- **getSignalStrengths**

  ```
  public double[] getSignalStrengths()
  ```

  - **Description**

    Gets a list of the signal strengths the access points have regarding the newly found access point.

  - **Returns** – All signal strengths.

Class PreparedDeviceData

Realizes data consisting of lists of access points paired with their signal strengths to a single given device.

Declaration

```
public class PreparedDeviceData
 extends java.lang.Object
```

Constructors

- **PreparedDeviceData**

```
public PreparedDeviceData()
```

Methods

- **getAccessPointPositions**

```
public pegs.model.Position[] getAccessPointPositions()
```

  - **Description**

    Gets a list of all positions of access points that have received a signal from the device.

  - **Returns** – All access point positions.

- **getSignalStrengths**

```
public double[] getSignalStrengths()
```

  - **Description**

    Gets a list of the signal strengths of the access points regarding the device.

  - **Returns** – All signal strengths.

Class RawDataPreparer

Combines and prepares data about scanned devices and access points from all access points for usage in controller package.

Declaration

```
public class RawDataPreparer
 extends java.lang.Object
```

Constructors

- **RawDataPreparer**

```
public RawDataPreparer()
```

Methods

- **prepareAccessPointData**

  ```
  public PreparedAccessPointData prepareAccessPointData(pegs.
      accesspointclient.RawAccessPointData[] rapd)
  ```

  - **Description**

    Extracts a list of access points and regarding signal strengths for the access point.

  - **Parameters**

    * `rapd` – The data to prepare.

  - **Returns** – The prepared access point data.

- **prepareDeviceData**

  ```
  public PreparedDeviceData[] prepareDeviceData(pegs.
      accesspointclient.RawDeviceData[] rdd)
  ```

  - **Description**

    Extracts a list of access points and regarding signal strengths for each device that can be seen by at least one access point from all RawDeviceData given.

  - **Parameters**

    * `rdd` – The data to prepare.

  - **Returns** – The prepared device data

Class ServerBroadcaster

Realizes the Broadcaster used by the Server to communicate unselectively with the access points.

Declaration

```
public class ServerBroadcaster
 extends java.lang.Object implements pegs.accesspointclient.
    Broadcaster
```

Constructors

- **ServerBroadcaster**

```
public ServerBroadcaster()
```

Methods

- **broadcastAccessPointWorkload**

  ```
  void broadcastAccessPointWorkload(pegs.accesspointclient.
    RawDataWorkload rapwl)
  ```

  - **Description copied from pegs.accesspointclient.Broadcaster** (**in 4.5, page 19**)

    Broadcasts a workload containing data for localization of Access Points to other Access Points and to the server.

  - **Parameters**

    * `rapwl` – The workload to broadcast.

- **broadcastDeviceWorkload**

  ```
  void broadcastDeviceWorkload(pegs.accesspointclient.
    RawDataWorkload rdwl)
  ```

  - **Description copied from pegs.accesspointclient.Broadcaster** (**in 4.5, page 19**)

    Broadcasts a workload containing data for localization of devices to other access points and the server.

  - **Parameters**

    * `rdwl` – The workload to broadcast.

- **broadcastTimeWorkload**

  ```
  void broadcastTimeWorkload(pegs.accesspointclient.
    TimeWorkload twl)
  ```

  - **Description copied from pegs.accesspointclient.Broadcaster** (**in 4.5, page 19**)

    Broadcasts a workload containing only a time stamp for time synchronization purposes.

  - **Parameters**

    * `twl` – The workload to be broadcast.

Class Time

Represents a timestamp.

Declaration

```
public class Time
 extends java.lang.Object
```

Constructors

- **Time**

  ```
  public Time()
  ```

6 Controller

Das Controller-Paket dient der Datenmanipulation des Model-Pakets bei Veränderung von Daten im View-Paket. Zudem kontrolliert es die Bearbeitung von eingegangenen Daten des Communication-Pakets mit Hilfe des PositionCalculator-Pakets und die anschließende Einbindung dieser Daten in den Datenbestand.

6.1 Klassendiagramm

Das Diagramm 6.1.1 zeigt die Klassen des Pakets Controller.

6.2 Funktion

Das Controller-Paket ist zuständig für die Kommunikation diverser Pakete innerhalb der Server-Software des PEGS-Systems. Das Paket kontrolliert zum einen die Analyse der Daten, die über das Communication-Paket in das Server-System gelangen und deren Verarbeitung, welche über das PositionCalculator-Paket betrieben wird. Zum anderen steuert das Controller-Paket die Kommunikation des View-Pakets zum Model-Paket. Daraus resultiert eine weitere Aufgabe dieses Pakets, welche darin liegt, den Datenbestand des Model-Pakets zu aktualisieren.

6.3 Schnittstellen

Das `Controller`-Paket bietet mehrere Schnittstellen an:

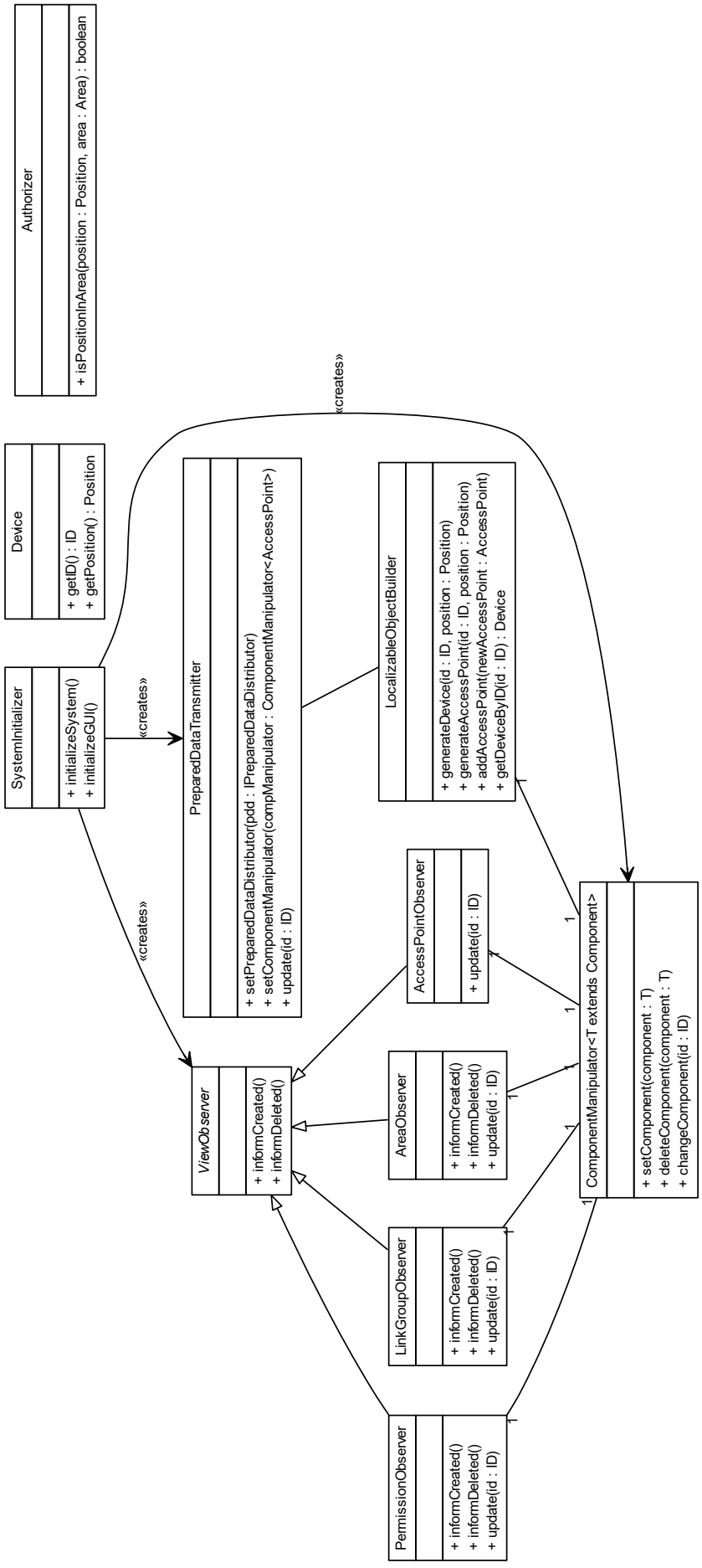

Abbildung 6.1.1: UML-Klassendiagramm zum Paket Controller.

- Eine Schnittstelle basiert auf der Klasse `PreparedDataTransmitter`. Die hier implementierte Methode `setDataDistributor` setzt die Schnittstelle zum Communication-Paket zur Übergabe von eingehenden Daten in das Server-System. Über die Methode `filterDataForPositionCalculation` der selben Klasse wird eine Schnittstelle zum PositionCalculator-Paket, genauer dem Interface `IPositionCalculator`, hergestellt.
- Es wird eine Schnittstelle für das View-Paket über die Klassen `AreaObserver`, `PermissionObserver` und `LinkGroupObserver` des Controller-Pakets angeboten, sodass das Controller-Paket über diese Beobachter bei der Änderung von Daten in der GUI benachrichtigt wird.
- Eine weitere Schnittstelle zum View-Paket besteht in der Klasse `AccessPointObserver`, welche aufgerufen wird, nachdem in der GUI ein Name und eine Beschreibung für einen neuen Access Point gegeben wurde.
- Es wird eine Schnittstelle für das Model-Paket zur Verfügung gestellt. Über die Klasse`ComponentManipulator` des Controller-Pakets werden Funktionen des Model-Pakets aufgerufen, um dessen Datenbestand zu aktualisieren.

6.4 Interne Struktur

Das Controller-Paket steuert das Zusammenspiel der Pakete Communication, PositionCalculator, View und Model. Die interne Struktur lässt sich daher aufteilen.
Eine interne Aufgabe des Controller-Pakets besteht darin, Änderungen in der GUI des View-Pakets über die Klassen`AreaObserver`, `PermissionObserver` und `LinkGroupObserver` wahrzunehmen und damit den Datenbestand des Model-Pakets zu aktualisieren. Die oben genannten Observer-Klassen werden bei Änderungen in der GUI von dieser benachrichtigt und holen sich über die `RequestManagerView`-Klasse des View-Pakets die geänderten Daten. Diese werden an die Klasse `ComponentManipulator` weitergegeben, welche die veränderten Daten an das Model weitergibt, damit dessen Datenbestand aktualisiert wird.
Des Weiteren werden im Controller-Paket die `ID`s aller gefundenen Endgeräte zusammen mit deren aktueller Position erzeugt und festgehalten. Dazu werden die vorsortierten Daten des Communication-Pakets in der Klasse `PreparedDataTransmitter` gefiltert und an das PositionCalculator-Paket weitergegeben, sodass dieses nur exakt die Informationen erhält, die es zum Berechnen einer `Position` benötigt. Die berechnete `Position` wird an die Klasse `PreparedDataTransmitter` zurückgegeben, welche diese an die `LocalizableObjectBuilder`-Klasse weitergibt, in der aus der `ID` und der `Position` des Endgeräts ein `Device` erzeugt wird.
Zudem werden so auch Access Points in das System eingefügt. Die `Position` wird wie oben beschrieben berechnet. Anschließend wird ein `AccessPoint` erstellt, indem die Klasse `LocalizableObjectBuilder` die `InstructionManagerView`-Klasse des View-Pakets aufruft, um einen Namen und eine Beschreibung für den zu erzeugenden `AccessPoint` anzufordern. Ist diese Eingabe erfolgt, wird die Klasse `AccessPointObserver` darüber in Kenntnis gesetzt und holt sich den Namen und die Beschreibung. Daraufhin wird der neu erzeugte `AccessPoint` an die Klasse `ComponentManipulator` weitergegeben, um den neuen `AccessPoint` in den Datenbestand des Model-Pakets zu integrieren.

6.5 Package pegs.controller

Package Contents *Page*

Classes

Class AccessPointObserver

Observer class for view, activates at changes concerning an AccessPoint in the GUI.

Declaration

```
public class AccessPointObserver
 extends pegs.controller.ViewObserver
```

Constructors

- **AccessPointObserver**

  ```
  public AccessPointObserver()
  ```

Methods

- **update**

  ```
  public void update(pegs.model.ID id)
  ```

Members inherited from class ViewObserver

`pegs.controller.ViewObserver` (in 6.5, page 57)
- `public void` **informCreated()**
- `public void` **informDeleted()**

Class AreaObserver

Observer class for view, activates at changes concerning an Area in the GUI.

Declaration

```
public class AreaObserver
 extends pegs.controller.ViewObserver
```

Constructors

- **AreaObserver**

  ```
  public AreaObserver()
  ```

Methods

- **informCreated**

  ```
  public void informCreated()
  ```

  - **Description**

    Method called by View after creation of an Area.

- **informDeleted**

  ```
  public void informDeleted()
  ```

  - **Description**

    Method called by View after deletion of an Area.

- **update**

  ```
  public void update(pegs.model.ID id)
  ```

Members inherited from class ViewObserver

`pegs.controller.ViewObserver` (in 6.5, page 57)
- `public void` **`informCreated()`**
- `public void` **`informDeleted()`**

Class Authorizer

Authorization request class for a Device on requesting a Permission.

Declaration

```
public class Authorizer
 extends java.lang.Object
```

Constructors

- **Authorizer**

  ```
  public Authorizer()
  ```

Methods

- **isPositionInArea**

  ```
  public boolean isPositionInArea(pegs.model.Position position
      ,pegs.model.Area area)
  ```

  - **Description**

    Tests if the given Position is in the specified Area.

  - **Parameters**

    * `position` – The given Position (of a Device).
    * `area` – Specified Area to be checked.

  - **Returns** – True if position is in area, false otherwise.

Class ComponentManipulator

Manipulates T in Model after there have been changes in the GUI of View.

Declaration

```
public class ComponentManipulator
 extends java.lang.Object
```

Constructors

- **ComponentManipulator**

  ```
  public ComponentManipulator(pegs.model.ComponentManager
      compManager)
  ```

  - **Description**

    Constructor to set an interface for Controller to Model, so the Controller can notify the Model on data changes.

  - **Parameters**

    * `compManager` – Interface to data storage manipulation in Model.

Methods

- **changeComponent**

  ```
  public void changeComponent(pegs.model.ID id)
  ```

  - **Description**

    Notifies the Model on the changed component.

  - **Parameters**

    * `component` – The changed component.

- **deleteComponent**

  ```
  public void deleteComponent(pegs.model.Component component)
  ```

  - **Description**

    Deletes a given `Component` (in 7.5, page 67) in the data storage of Model.

  - **Parameters**

    * `component` – The `Component` (in 7.5, page 67) to be deleted.

- **setComponent**

  ```
  public void setComponent(pegs.model.Component component)
  ```

  - **Description**

    Sets a given `Component` (in 7.5, page 67) in the data storage of Model.

  - **Parameters**

    * `component` – The `Component` (in 7.5, page 67) to be set.

Class Device

Represents a user device that is seen and located by PEGS.

Declaration

```
public class Device
 extends java.lang.Object
```

Constructors

- **Device**

  ```
  public Device(pegs.model.ID id, pegs.model.Position position)
  ```

  - **Description**

    Creates a new device with specified ID and position.

  - **Parameters**

    * `id` – The ID of the new device.
    * `position` – The position of the new device.

Methods

- **getID**

  ```
  public pegs.model.ID getID()
  ```

  - **Description**

    Getter-method for this.id.

  - **Returns** – The ID of this device.

- **getPosition**

  ```
  public pegs.model.Position getPosition()
  ```

  - **Description**

    Getter-method for device position.

  - **Returns** – The position of this device.

Class LinkGroupObserver

Observer class for view, activates at changes concerning a LinkGroup in the GUI.

Declaration

```
public class LinkGroupObserver
 extends pegs.controller.ViewObserver
```

Constructors

- **LinkGroupObserver**

  ```
  public LinkGroupObserver()
  ```

Methods

- **informCreated**

  ```
  public void informCreated()
  ```

  - **Description**

    Method called by View after creation of a LinkGroup.

- **informDeleted**

  ```
  public void informDeleted()
  ```

  - **Description**

    Method called by View after deletion of a LinkGroup.

- **update**

  ```
  public void update(pegs.model.ID id)
  ```

Members inherited from class ViewObserver

`pegs.controller.ViewObserver` (in 6.5, page 57)
- `public void` **`informCreated()`**
- `public void` **`informDeleted()`**

Class LocalizableObjectBuilder

Class for creating Device and AccessPoint objects and storing Device objects.

Declaration

```
public class LocalizableObjectBuilder
 extends java.lang.Object
```

Constructors

- **LocalizableObjectBuilder**

```
public LocalizableObjectBuilder()
```

Methods

- **addAccessPoint**

```
public void addAccessPoint(pegs.model.AccessPoint
    newAccessPoint)
```

  - **Description**

    Calls ComponentManipulator to add a new Access Point to the data storage.

  - **Parameters**

    * `newAccessPoint` – Access Point to be added.

- **generateAccessPoint**

```
public void generateAccessPoint(pegs.model.ID id, pegs.model.
    Position position)
```

  - **Description**

    Creates an Access Point with specified ID and Position.

  - **Parameters**

    * `id` – Identifier of the Access Point.

    * `position` – Position of the Access Point.

- **generateDevice**

```
public void generateDevice(pegs.model.ID id, pegs.model.
    Position position)
```

  - **Description**

    Creates a Device with specified ID and Position.

  - **Parameters**

    * `id` – Identifier of the Device

    * `position` – Position of the Device

- **getDeviceByID**

```
public Device getDeviceByID(pegs.model.ID id)
```

- **Description**

  Getter for the Device with the specified ID.

- **Parameters**

  * `id` – Identifier of the searched Device.

- **Returns** – The Device with the specified ID.

Class PermissionObserver

Observer class for view, activates at changes concerning a Permission in the GUI.

Declaration

```
public class PermissionObserver
 extends pegs.controller.ViewObserver
```

Constructors

- **PermissionObserver**

  ```
  public PermissionObserver()
  ```

Methods

- **informCreated**

  ```
  public void informCreated()
  ```

  - **Description**

    Method called by View after creation of a Permission.

- **informDeleted**

  ```
  public void informDeleted()
  ```

  - **Description**

    Method called by View after deletion of a Permission.

- **update**

  ```
  public void update(pegs.model.ID id)
  ```

Members inherited from class ViewObserver

pegs.controller.ViewObserver (in 6.5, page 57)
- public void **informCreated**()
- public void **informDeleted**()

Class PreparedDataTransmitter

Passes information needed to calculate a Position to IPositionCalculator.

Declaration

```
public class PreparedDataTransmitter
 extends pegs.communication.CommunicationObserver
```

Constructors

- **PreparedDataTransmitter**

  ```
  public PreparedDataTransmitter(pegs.positioncalculator.
      IPositionCalculator pc)
  ```

  - **Description**

    Constructor method to set an interface to the PositionCalculator, called at system initialization.

  - **Parameters**

    * pc – Interface between Controller and PositionCalculator.

Methods

- **setComponentManipulator**

  ```
  public void setComponentManipulator(ComponentManipulator
      compManipulator)
  ```

  - **Description**

    Build connection to ComponentManipulator class to save newly built Access Points in Model.

  - **Parameters**

    * compManipulator – The interface to Model.

- **setPreparedDataDistributor**

  ```
  public void setPreparedDataDistributor(pegs.communication.
      IPreparedDataDistributor pdd)
  ```

  - **Description**

    Sets interface IPreparedDataDistributor between Controller and Communication.

  - **Parameters**

    * `pdd` – Interface between Controller- and Communication-packages.

- **update**

  ```
  public void update(pegs.model.ID id)
  ```

Members inherited from class CommunicationObserver

`pegs.communication.CommunicationObserver` (in 5.5, page 38)

- `public void` **notifyAboutNewAccessPointData()**
- `public void` **notifyAboutNewDeviceData()**

Class SystemInitializer

Creator class, called at system initialization to create parts of Model, PositionCalculator and Controller-package. Also sets basic functionality of View-package when GUI is opened.

Declaration

```
public class SystemInitializer
 extends java.lang.Object
```

Constructors

- **SystemInitializer**

  ```
  public SystemInitializer()
  ```

Methods

- **initializeGUI**

  ```
  public void initializeGUI()
  ```

  - **Description**

    Starts the GUI.

- **initializeSystem**

  ```
  public void initializeSystem()
  ```

  - **Description**

    Starts the system.

Class ViewObserver

Represents an observer that is meant to observe changes in the View.

Declaration

```
public abstract class ViewObserver
 extends java.lang.Object implements pegs.common.IObserver
```

All known subclasses

PermissionObserver (in 6.5, page 53), LinkGroupObserver (in 6.5, page 51), AreaObserver (in 6.5, page 47), AccessPointObserver (in 6.5, page 46)

Constructors

- **ViewObserver**

  ```
  public ViewObserver()
  ```

Methods

- **informCreated**

  ```
  public void informCreated()
  ```

  - **Description**

    Method called by View after creation of a {Area, Link, LinkGroup, Permission}.

- **informDeleted**

```
public void informDeleted()
```

  - **Description**

    Method called by View after deletion of a {Area, Link, LinkGroup, Permission}.

7 Model

Das Model stellt eines der drei Pakete der grundlegenden *Model View Controller* Architektur da. Es ist verantwortlich für die Verwaltung, interne Änderung und Verbreitung der Business Data innerhalb des PEGS-Systems. Der Begriff Business Data bezeichnet alle Komponenten, die im Paket View visualisiert werden und/oder im Paket Controller und PositionCalculator der Positionsberechnung dienen.

7.1 Klassendiagramm

Das Diagramm 7.1.1 zeigt die Klassen des Pakets Model.

7.2 Funktion

Das Paket Model hat folgende Verantwortlichkeiten:

- Speichern der Business Data und Zugriff darauf bereitstellen.
- Aufträge zu Änderungen der Business Data entgegennehmen und intern Änderungen durchführen.
- Bei Änderungen in der Business Data geänderte Daten an alle Stellen in PEGS verteilen, die diese Daten verwenden.

Das Paket Model hält zur Systemlaufzeit die gesamte Business Data von PEGS. Konkret sind das alle Komponenten, die im Paket View visualisiert werden, sowie Daten zur Positionsbestimmung von Endgeräten und Access Points.

Das Paket Model nimmt von anderen Paketen Aufträge zur Änderung der Komponenten entgegen und führt die Änderungen selbst durch. So setzt das Paket Model auch eine schützende und kontrollierende Funktion für Änderungen der Business Data um. Im Sinne der MVC-Architektur werden Anfragen zur Änderung nur vom Paket Controller durchgeführt.

Wurde eine Änderung an einer Komponente im Paket Model ausgeführt, so nimmt Model eine weitere Rolle ein. Es benachrichtigt alle Pakete, die von der Business Data abhängig

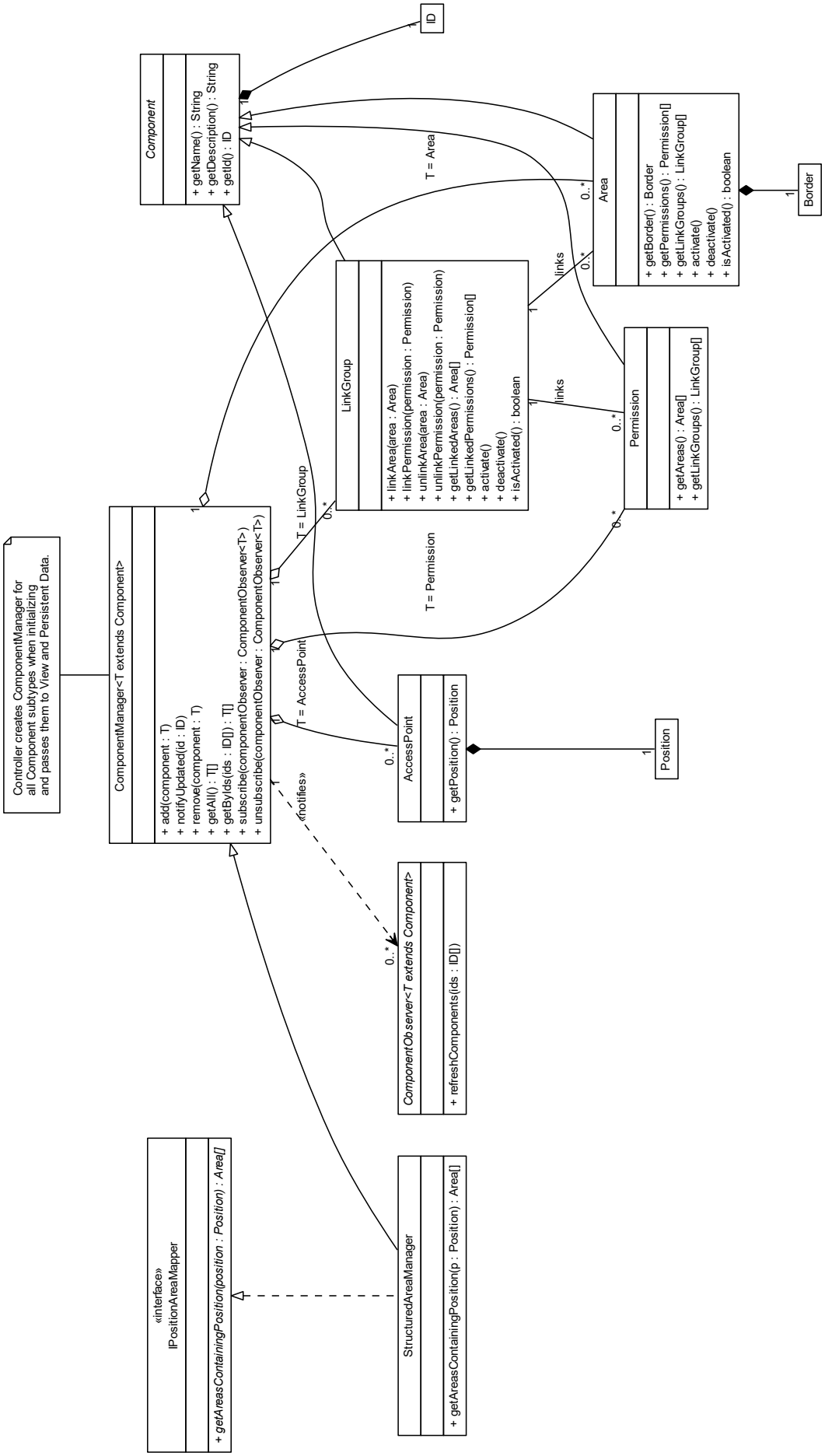

Abbildung 7.1.1: UML-Klassendiagramm zum Paket Model.

sind (das Paket View und das Paket PersistentData), über die Änderung, sodass diese erneut durch die Model Schnittstellen auf die geänderten Daten zugreifen können.

7.3 Schnittstellen

Zur Umsetzung seiner ersten Rolle, dem Ermöglichen des Zugriffs auf Komponenten, bietet das Paket Model die Klasse `ComponentManager<T extends Component>` mit einem generischen Untertypen von `Component` an. Diese Klasse ermöglicht einen gezielten Zugriff auf bestimmte Arten von Komponenten. Damit kann von außen auf genau die Daten der Art Komponente zugegriffen werden, die an der Stelle benötigt wird. Die Schnittstelle zum Zugreifen auf die Business Data kann so auch leicht um weitere Komponenten erweitert werden. Eine speziellere Art des Zugriffes auf Bereiche abhängig von Positionen ist über die Schnittstelle `IPositionAreaMapper` realisiert.

Auch die Aufträge zum Hinzufügen und Entfernen bestimmter Komponenten werden über die Schnitstelle `ComponentManager<T extends Component>`, realisiert. Hier stehen Methoden `add`, `update` und `remove` dazu zur Verfügung.

Benachrichtigungen über Änderungen in der Business Data werden an Beobachter gesendet, welche die angebotene Schnittstelle `ComponentObeserver<T extends Component>` implementieren. Dafür müssen diese Beobachter am entsprechenden `ComponentManager<T>` angemeldet sein. Die Benachrichtigung wird von einem `ComponentManager<T>` nur gesendet, wenn es Änderungen in den Komponenten des Typen `T` gegeben hat. Damit werden nur jene Klassen fremder Pakete benachrichtigt, die sich für eine Änderung einer Komponente dieses Typs interessieren. Genau diese können in der Folge über den `ComponentManager`, der Ursprung der Benachrichtigung gewesen ist, die aktualisierten Daten anfordern.

7.4 Interne Struktur

Wichtigster Bestandteil des Pakets Model sind die von der Klasse `Component` erbenden Typen, deren Objekte im Folgenden “Komponente” genannt werden. Es handelt sich um die Klassen `Area`, `Permission`, `LinkGroup` und `AccessPoint`.
Die Klasse `Area` modelliert einen geographischen Bereich, der durch eine Grenze definiert ist. Dementsprechend muss jedem Objekt des Typen `Area` ein Objekt des Typen `Border` zugeordnet sein, das diese Grenze darstellt. Auf dieses `Border` Objekt kann mit der Methode `getBorder` auf einem Objekt vom Typ `Area` zugegriffen werden. Weiterhin erlaubt die Klasse mit der Methode `getLinkGroups` Zugriff auf alle `LinkGroup`-Objekte, in denen dieses `Area`-Objekt enthalten ist. Außerdem gibt `Area` mit `getPermissions` Zugriff auf alle `Permission`-Objekte, die mit diesem `Area`-Objekt gekoppelt sind (s. hierzu LinkGroup). Diese Objekte stellen die Kopplungsgruppen bzw. Berechtigungen dar, mit denen der durch das `Area`-Objekt modellierte Bereich verbunden sein soll. Ein `Area`-Objekt kann mit der Methode `deactivate` deaktiviert werden. Die Methoden `getLinkGroups` und `getPermissions` werfen dann eine Exception. Mit der Methode `activate` kann ein `Area`-Objekt in den aktivierten Zustand versetzt werden. Die Methode `isActivated` gibt zurück, ob sich das Objekt im aktivierten Zustand befindet.
Die Klasse `Permission` modelliert eine Berechtigung, wie z.B. den Zugang zu WiFi oder einem anderen Dienst, die ein Endgerät haben kann. Die Klasse `Permission` erlaubt mit

der Methode `getLinkGroups` Zugriff auf alle `LinkGroup`-Objekte, in denen dieses `Permission`-Objekt enthalten ist. Außerdem gibt `Permission` mit `getAreas` Zugriff auf alle `Area`-Objekte, die mit diesem `Permission`-Objekt gekoppelt sind (s. hierzu LinkGroup). Diese Objekte stellen die Kopplungsgruppen bzw. Bereiche dar, mit denen die durch das `Permission`-Objekt modellierte Berechtigung verbunden sein soll.
Die Klasse `LinkGroup` modelliert eine Kopplungsgruppe, also eine bidirektionale Verknüpfung von Bereichen und Berechtigungen. Ein Objekt der Klasse `LinkGroup` kennt dementsprechend eine beliebige Anzahl von Objekten der Klassen `Area` und `Permission`. Diese stellen Bereiche und Berechtigungen dar, die durch eine Kopplungsgruppe verbunden werden sollen. Alle Objekte der Klassen `Area` und `Permission` in einer `LinkGroup` gelten für alle Zwecke im PEGS System als gekoppelt. Dies bedeutet in der Behandlung von Endgeräten in anderen Paketen folgendes:

1. Es wird festgestellt, in welchen Bereichen sich das Endgerät befindet.
2. Das Endgerät erhält alle Berechtigungen, die mit den Bereichen in einer `LinkGroup` gekoppelt sind.

Verknüpfte `Area`- und `Permission`-Objekte können mit den Methoden `getLinkedAreas` und `geLinkedPermissions` abgerufen werden. Hinzufügen und Entfernen dieser Objekte ist auf einem `LinkGroup`-Objekt möglich mittels der Methoden `linkArea`, `linkPermission` bzw. `unlinkArea` und `unlinkPermission`. Diese vier Methoden teilen jeweils dem neu verknüpften `Area`- oder `Permission`-Objekt das Objekt vom Typ `LinkGroup`, auf dem sie aufgerufen wurden, mit. Somit kennen diese im Zuge ihrer Methode `getLinkGroups` auch die `LinkGroup`-Objekte, in denen sie aktuell enthalten sind. Auch die Methoden `getPermissions` der Klasse `Area` und `getAreas` der Klasse `Permission` werden so möglich gemacht. Ein `LinkGroup`-Objekt kann mit der Methode `deactivate` deaktiviert werden. Die Methoden `getAreas` und `getPermissions` werfen dann eine Exception. Mit der Methode `activate` kann ein `LinkGroup`-Objekt in den aktivierten Zustand versetzt werden. Die Methode `isActivated` gibt zurück, ob sich das Objekt im aktivierten Zustand befindet.
Die Klasse `AccessPoint` modelliert einen Access Point Rechner, der WiFi sendet und Informationen über Endgeräte in seinem Bereich an den Server kommuniziert. Die Klasse `AccessPoint` darf nicht mit dem Paket `pegs.accesspointclient` verwechselt werden, welches die PEGS-Software auf einem solchen Access Point Rechner modelliert. Objekte der Klasse `AccessPoint` haben immer ein Objekt der Klasse `Position`, das die tatsächliche Position des Rechners innerhalb des Einzugsgebietes von PEGS darstellt. Diese Position wird vom Paket PositionCalculator verwendet, um Rückschlüsse auf Positionen von Endgeräten zu ermöglichen. Dafür ist die `Position` auf einem `AccessPoint`-Objekt mit der Methode `getPosition` zugänglich.
Gemeinsamkeiten der oben genannten vier Komponententypen sind in der abstrakten Generalisierung `Component` enthalten. Jedes Objekt einer Klasse, die von `Component` erbt muss ein Objekt der Klasse `ID`, einen Namen und eine Beschreibung haben, die über die Methoden `getId`, `getName` und `getDescription` zugänglich sind. Das `ID`-Objekt stellt einen systemweit eindeutigen Identifikator dieser Komponente da.

Ein `ComponentManager` spielt für jeden von `Component` erbenden Typ die zentrale Rolle im Model. Er stellt die Datenstruktur dar, die alle Objekte dieses Typen enthält, und fungiert gleichzeitig als vollständige Schnittstelle für diesen Untertyp von `Component`. Der `ComponentManager<T extends Component>` ermöglicht Zugriffe auf die vorhandenen Objekte des Typen T, sowie das Hinzufügen und Entfernen solcher. Um bei der

Initialisierung des Systems die Daten aus dem persistenten Speicher zu laden, erhält ein Objekt des Typen `ComponentManager` einen `IPersistentDataManager`, der mit demselben `Component`-Untertypen parametrisiert ist. Die vorhandenen Objekte dieses `Component`-Untertypen werden mittels der Methode `getAll` auf `IPersistentDataManager` erlangt. Die Klasse `StructuredAreaMapper` ist eine Spezialisierung der Klasse `ComponentManager` des Typs `Area`, die erweiternd das Interface `IPositionAreaMapper` implementiert. Die Klasse verwendet eine Datenstruktur, die `Area`-Objekte nach den Bereichsgrenzen ordnet. So werden Anfragen nach allen Bereichen, die eine gegebene Position enthalten, erleichtert. Diese Anfragen werden durch die Methode `getAreasContainingPosition` im Interface `IPositionAreaMapper` ermöglicht. Andere Pakete verwenden dieses Interface und diese Methode, um einer Position alle Bereiche, welche die Position enthalten, zuzuordnen.

Ein weiterer Kernbestandteil des Paketes Model ist die Klasse `ComponentObserver<T extends Component>`, die das Interface `IObserver` implementiert. Sie erlaubt die bidirektionale Kommunikation zu anderen Paketen, die auf Veränderungen der Business Data im Paket Model reagieren müssen. Ein anderes Paket, dem das Paket Model bekannt ist, hat hierzu eine Klasse, die von der abstrakten Klasse `ComponentObserver` erbt. Das Paket kann ein Objekt dieser Klasse am ihm bekannten `ComponentManager` mit der Methode `subscribe` anmelden, da `ComponentManager` das Interface `IObservable<ComponentObserver>` implementiert. Der `ComponentManager` benachrichtigt im Falle einer Änderung an der Business Data alle angemeldeten `ComponentObserver`. Sowohl die Klasse `ComponentManager<T extends Component>` als auch die Klasse `ComponentObserver<T extends Component>` sind mit einem Subtypen von `Component` parametrisiert. Damit kann von einem anderen Paket selbst über den Komponententyp, dessen Änderungen für das Paket relevant sind, entschieden werden. Nach einer Benachrichtigung der `ComponentObserver`-Objekte können die fremden Pakete die aktualisierten Daten über den ihnen bekannten `ComponentManager` des entsprechenden Typs beschaffen.

7.5 Package pegs.model

Package Contents *Page*

Interfaces

Classes

Interface IPositionAreaMapper

Used to map a `Position` (in 7.5, page 76) to all `Area(s)` (in 7.5, page 65) currently known in the data model that contain this `Position` (in 7.5, page 76) within their `Border` (in 7.5, page 67).

Declaration

```
public interface IPositionAreaMapper
```

All known subinterfaces

StructuredAreaManager (in 7.5, page 76)

All classes known to implement interface

StructuredAreaManager (in 7.5, page 76)

Methods

- **getAreasContainingPosition**

  ```
  Area[] getAreasContainingPosition(Position position)
  ```

  - **Description**

    Getter for all `Area(s)` (in 7.5, page 65) containing a given `Position` (in 7.5, page 76).

  - **Parameters**

    * `position` – The `Position` (in 7.5, page 76) given.

  - **Returns** – All `Area(s)` (in 7.5, page 65) containing the `Position` (in 7.5, page 76) position.

Class AccessPoint

Represents a physical access point with its `Position` (in 7.5, page 76). An object of this type includes all information the server needs about a physical access point including primarily its `Position` (in 7.5, page 76) and `ID` (in 7.5, page 71). This information can be used by the server in the process of calculating device positions (see IPositionCalculator).

Declaration

```
public class AccessPoint
 extends pegs.model.Component
```

Constructors

- **AccessPoint**

  ```
  public AccessPoint(Position position,java.lang.String name,
      java.lang.String description)
  ```

– **Description**

Creates a new `AccessPoint` (in 7.5, page 64) object with the given `Position` (in 7.5, page 76), name and description.

– **Parameters**

* `position` – The `Position` (in 7.5, page 76).
* `name` – The name.
* `description` – The description.

Methods

- **getPosition**

```
public Position getPosition()
```

– **Description**

Returns the `Position` (in 7.5, page 76) of the physical access point represented.

– **Returns** – The `Position` (in 7.5, page 76).

Members inherited from class Component

`pegs.model.Component` (in 7.5, page 67)
- `public String` **getDescription**`()`
- `public ID` **getId**`()`
- `public String` **getName**`()`

Class Area

Represents an area specified by a geofence, a `Border` (in 7.5, page 67). Devices located within the `Area` (in 7.5, page 65)s `Border` (in 7.5, page 67) are granted `Permission(s)` (in 7.5, page 75) linked to this `Area` (in 7.5, page 65). `Area` (in 7.5, page 65) objects are linked to `Permission` (in 7.5, page 75) objects via `LinkGroup` (in 7.5, page 72) objects. Can be in either `activated` or `deactivated` state (see activate, deactivate). Will only grant access to linked `Permission(s)` (in 7.5, page 75) and `LinkGroup(s)` (in 7.5, page 72) when in `activated` state.

Declaration

```
public class Area
 extends pegs.model.Component
```

Constructors

- **Area**

```
public Area(java.lang.String name,java.lang.String
    description,Border border)
```

  - **Description**

    Creates a new `Area` (in 7.5, page 65) object with the given `Border` (in 7.5, page 67), name and description. Initially the new `Area` (in 7.5, page 65) object is in `activated` state. The new `Area` (in 7.5, page 65) object initially is in `activated` state.

  - **Parameters**

    * `name` – The name to assign to the `Area` (in 7.5, page 65).
    * `description` – The description to assign to the `Area` (in 7.5, page 65).
    * `border` – The `Border` (in 7.5, page 67) that defines the edge of the `Area` (in 7.5, page 65).

Methods

- **activate**

```
public void activate()
```

  - **Description**

    Changes this `Area(s)` (in 7.5, page 65) state to `activated`.

- **deactivate**

```
public void deactivate()
```

  - **Description**

    Changes this `Area(s)` (in 7.5, page 65) state to `deactivated`.

- **getBorder**

```
public Border getBorder()
```

  - **Description**

    Returns the `Border` (in 7.5, page 67) of this `Area` (in 7.5, page 65).

  - **Returns** – The `Border` (in 7.5, page 67).

- **getLinkGroups**

```
public LinkGroup[] getLinkGroups()
```

- **Description**

  Returns all `LinkGroup(s)` (in 7.5, page 72) this `Area` (in 7.5, page 65) is part of if this `Area` (in 7.5, page 65) is in `activated` state.

- **Returns** – The `LinkGroup(s)` (in 7.5, page 72).

- **getPermissions**

```
public Permission[] getPermissions()
```

- **Description**

  Returns all `Permission(s)` (in 7.5, page 75) linked to this `Area` (in 7.5, page 65) if this `Area` (in 7.5, page 65) is in `activated` state.

- **Returns** – The `Permission(s)` (in 7.5, page 75).

- **isActivated**

```
public boolean isActivated()
```

- **Description**

  Returns true if this `Area` (in 7.5, page 65) is in `activated` state, false if this `Area` (in 7.5, page 65) is in `deactivated` state.

- **Returns** – true if in `activated` state, false else.

Members inherited from class Component

`pegs.model.Component` (in 7.5, page 67)
- `public String` **`getDescription()`**
- `public ID` **`getId()`**
- `public String` **`getName()`**

Class Border

Represents a rectangular border of an `Area` (in 7.5, page 65).

Declaration

```
public class Border
 extends java.lang.Object
```

Constructors

- **Border**

```
public Border()
```

Class Component

Represents a `Component` (in 7.5, page 67) of PEGS that is needed for position calculation (see pegs.positioncalculator.IPositionCalculator) or visual presentation (see pegs.view.InfoFrame).

Declaration

```
public abstract class Component
 extends java.lang.Object
```

All known subclasses

Permission (in 7.5, page 75), LinkGroup (in 7.5, page 72), Area (in 7.5, page 65), AccessPoint (in 7.5, page 64)

Constructors

- **Component**

```
public Component(java.lang.String name, java.lang.String
    description)
```

  - **Description**

    Creates a new `Component` (in 7.5, page 67) object with a given name and description. Creates `ID` (in 7.5, page 71) for the `Component` (in 7.5, page 67) object.

  - **Parameters**

    * `name` – The name to give the new `Component` (in 7.5, page 67).
    * `description` – The description to give the new `Component` (in 7.5, page 67).

Methods

- **getDescription**

```
public java.lang.String getDescription()
```

- **Description**

  Returns the description of the `Component` (in 7.5, page 67) object.

- **Returns** – The description.

- **getId**

```
public ID getId()
```

  - **Description**

    Returns the unique `ID` (in 7.5, page 71) of the `Component` (in 7.5, page 67) object.

  - **Returns** – The ID.

- **getName**

```
public java.lang.String getName()
```

  - **Description**

    Returns the name of the `Component` (in 7.5, page 67) object.

  - **Returns** – The name.

Class ComponentManager

Represents the full data model for a `Component` (in 7.5, page 67) subtype T. Offers interface for modifying the data model by adding, removing and updating `Component(s)` (in 7.5, page 67) of type T as well as an interface for retrieving currently known `Component(s)` (in 7.5, page 67) of type T.

Declaration

```
public class ComponentManager
 extends java.lang.Object implements pegs.common.IObservable
```

All known subclasses

StructuredAreaManager (in 7.5, page 76)

Constructors

- **ComponentManager**

```
public ComponentManager(pegs.persistentdata.
    IPersistentDataManager persistentDataManager)
```

- **Description**

  Creates a new `ComponentManager` (in 7.5, page 69). Gets known `Component(s)` (in 7.5, page 67) of type T from persistent data storage via a given `IPersistentDataManager` (in 8.5, page 82).

Methods

- **add**

  ```
  public void add(Component component)
  ```

  - **Description**

    Adds a `Component` (in 7.5, page 67) of generic type T.

  - **Parameters**

    * `component` – The `Component` (in 7.5, page 67) to add.

- **getAll**

  ```
  public Component[] getAll()
  ```

  - **Description**

    Returns all `Component(s)` (in 7.5, page 67) of generic type T.

  - **Returns** – All `Component(s)` (in 7.5, page 67) of generic type T.

- **getByIds**

  ```
  public Component[] getByIds(ID[] ids)
  ```

  - **Description**

    Returns all `Component(s)` (in 7.5, page 67) of generic type T corresponding to given `ID(s)` (in 7.5, page 71).

  - **Parameters**

    * `ids` – The `ID(s)` (in 7.5, page 71) of `Component(s)` (in 7.5, page 67) to retrieve.

  - **Returns** – The `Component(s)` (in 7.5, page 67) of all matching `ID(s)` (in 7.5, page 71), empty array if `ID(s)` (in 7.5, page 71) do not match known `Component(s)` (in 7.5, page 67).

- **notifyUpdated**

  ```
  public void notifyUpdated(ID id)
  ```

- **Description**

  Notifies all `ComponentObserver(s)` (in 7.5, page 71) that the `Component` (in 7.5, page 67) of generic type T with the `ID` (in 7.5, page 71) `id` has been updated.

- **Parameters**

  * `id` – the `ID` (in 7.5, page 71) of the updated `Component` (in 7.5, page 67).

- **remove**

```
public void remove(Component component)
```

  - **Description**

    Removes a `Component` (in 7.5, page 67) of generic type T.

  - **Parameters**

    * `component` – The `Component` (in 7.5, page 67) to remove.

- **subscribe**

```
public void subscribe(ComponentObserver componentObserver)
```

- **unsubscribe**

```
public void unsubscribe(ComponentObserver componentObserver)
```

Class ComponentObserver

Used to subscribe to notifications about changes regarding `Component(s)` (in 7.5, page 67) of type T in the data model.

Declaration

```
public abstract class ComponentObserver
 extends java.lang.Object implements pegs.common.IObserver
```

Constructors

- **ComponentObserver**

```
public ComponentObserver()
```

Methods

- **refreshComponents**

```
public void refreshComponents(ID[] ids)
```

  - **Description**

    Tells observer to update all `Component(s)` (in 7.5, page 67) with the given `ID(s)` (in 7.5, page 71) of `Component(s)` (in 7.5, page 67) that have changed.

  - **Parameters**

    * `ids` – The `ID(s)` (in 7.5, page 71) of `Component(s)` (in 7.5, page 67) that have changed.

Class ID

Represents an identifier that is unique across all packages.

Declaration

```
public class ID
 extends java.lang.Object
```

Constructors

- **ID**

```
public ID()
```

Class LinkGroup

Represents a logical group of `Area(s)` (in 7.5, page 65) and `Permission(s)` (in 7.5, page 75) that are linked (n to n association between `Area(s)` (in 7.5, page 65) and `Permission(s)` (in 7.5, page 75)). Any device located in an `Area` (in 7.5, page 65) will hold all `Permission(s)` (in 7.5, page 75) in the same `LinkGroup` (in 7.5, page 72) as the `Area` (in 7.5, page 65). Can be in either `activated` or `deactivated` state (see activate, deactivate). Will only grant access to linked `Area(s)` (in 7.5, page 65) and `Permission(s)` (in 7.5, page 75) when in `activated` state.

Declaration

```
public class LinkGroup
 extends pegs.model.Component
```

Constructors

- **LinkGroup**

  ```
  public LinkGroup(java.lang.String name,java.lang.String
      description)
  ```

  - **Description**

    Creates a new `LinkGroup` (in 7.5, page 72) object with the given name and description. The new `LinkGroup` (in 7.5, page 72) will not link any `Area(s)` (in 7.5, page 65)/`Permission(s)` (in 7.5, page 75) until at least one `Area` (in 7.5, page 65) and one `Permission` (in 7.5, page 75) is added with linkArea and linkPermission.

  - **Parameters**

    * `name` – the name to assign to the new `LinkGroup` (in 7.5, page 72) object.
    * `description` – the description to assign to the new `LinkGroup` (in 7.5, page 72) object.

Methods

- **activate**

  ```
  public void activate()
  ```

  - **Description**

    Changes this `LinkGroup(s)` (in 7.5, page 72) state to `activated`.

- **deactivate**

  ```
  public void deactivate()
  ```

  - **Description**

    Changes this `LinkGroup(s)` (in 7.5, page 72) state to `deactivated`.

- **getLinkedAreas**

  ```
  public Area[] getLinkedAreas()
  ```

- **Description**

  Returns all `Area(s)` (in 7.5, page 65) currently linked in the `LinkGroup` (in 7.5, page 72) if this `LinkGroup` (in 7.5, page 72) is in `activated` state.

- **Returns** – All `Area(s)` (in 7.5, page 65) linked.

- **getLinkedPermissions**

```
public Permission[] getLinkedPermissions()
```

  - **Description**

    Returns all `Permission(s)` (in 7.5, page 75) currently linked in the `LinkGroup` (in 7.5, page 72) if this `LinkGroup` (in 7.5, page 72) is in `activated` state.

  - **Returns** – All `Permission(s)` (in 7.5, page 75) linked.

- **isActivated**

```
public boolean isActivated()
```

  - **Description**

    Returns true if this `LinkGroup` (in 7.5, page 72) is in `activated` state, false if this `LinkGroup` (in 7.5, page 72) is in `deactivated` state.

  - **Returns** – true if in `activated` state, false else.

- **linkArea**

```
public void linkArea(Area area)
```

  - **Description**

    Adds a given `Area` (in 7.5, page 65) to the `LinkGroup` (in 7.5, page 72) and thus links it to all `Permission(s)` (in 7.5, page 75) in the `LinkGroup` (in 7.5, page 72).

  - **Parameters**

    * `area` – The `Area` (in 7.5, page 65) to link.

- **linkPermission**

```
public void linkPermission(Permission permission)
```

  - **Description**

    Adds a given `Permission` (in 7.5, page 75) to the `LinkGroup` (in 7.5, page 72) and thus links it to all `Area(s)` (in 7.5, page 65) in the `LinkGroup` (in 7.5, page 72).

  - **Parameters**

    * `permission` – The `Permission` (in 7.5, page 75) to link.

- **unlinkArea**

  ```
  public void unlinkArea(Area area)
  ```

  - **Description**

    Removes a given `Area` (in 7.5, page 65) from the `LinkGroup` (in 7.5, page 72) and thus unlinks it from all `Permission(s)` (in 7.5, page 75) in the `LinkGroup` (in 7.5, page 72).

  - **Parameters**

    * `area` – The `Area` (in 7.5, page 65) to unlink.

- **unlinkPermission**

  ```
  public void unlinkPermission(Permission permission)
  ```

  - **Description**

    Removes a given `Permission` (in 7.5, page 75) from the `LinkGroup` (in 7.5, page 72) and thus unlinks it from all `Area(s)` (in 7.5, page 65) in the `LinkGroup` (in 7.5, page 72).

  - **Parameters**

    * `permission` – The `Permission` (in 7.5, page 75) to unlink.

Members inherited from class Component

`pegs.model.Component` (in 7.5, page 67)
- `public String` **`getDescription()`**
- `public ID` **`getId()`**
- `public String` **`getName()`**

Class Permission

Represents a permission to access a certain real world service (e.g. WiFi-Access or access to a data server). `Permission(s)` (in 7.5, page 75) are linked to `Area(s)` (in 7.5, page 65) via `LinkGroup(s)` (in 7.5, page 72).

Declaration

```
public class Permission
 extends pegs.model.Component
```

Constructors

- **Permission**

  ```
  public Permission(java.lang.String name,java.lang.String
      description)
  ```

  - **Description**

    Creates a new `Permission` (in 7.5, page 75) object with the given name and description.

  - **Parameters**

    * `name` – The name to assign to the `Permission` (in 7.5, page 75).
    * `description` – The description to assign to the `Permission` (in 7.5, page 75).

Methods

- **getAreas**

  ```
  public Area[] getAreas()
  ```

  - **Description**

    Returns all `Area(s)` (in 7.5, page 65) linked to this `Permission` (in 7.5, page 75).

  - **Returns** – The Areas.

- **getLinkGroups**

  ```
  public LinkGroup[] getLinkGroups()
  ```

  - **Description**

    Returns all `LinkGroup(s)` (in 7.5, page 72) this `Permission` (in 7.5, page 75) is currently part of.

  - **Returns** – The `LinkGroup(s)` (in 7.5, page 72) this `Permission` (in 7.5, page 75) is part of.

Members inherited from class Component

`pegs.model.Component` (in 7.5, page 67)
- `public String` **`getDescription()`**
- `public ID` **`getId()`**
- `public String` **`getName()`**

Class Position

Represents a position or rather a two-dimensional vector with its coordinates.

Declaration

```
public class Position
 extends java.lang.Object
```

Constructors

- **Position**

  ```
  public Position()
  ```

Class StructuredAreaManager

A specialized `ComponentManager` (in 7.5, page 69) of `Area` (in 7.5, page 65) that holds `Area(s)` (in 7.5, page 65) in a structured fashion to allow for methods for mapping `StructuredAreaManager` (in 7.5, page 76) to `Area(s)` (in 7.5, page 65) containing them.

Declaration

```
public class StructuredAreaManager
 extends pegs.model.ComponentManager implements
    IPositionAreaMapper
```

Constructors

- **StructuredAreaManager**

  ```
  public StructuredAreaManager(pegs.persistentdata.
     IPersistentDataManager pdm)
  ```

  – **Description**

Methods

- **getAreasContainingPosition**

  ```
  Area[] getAreasContainingPosition(Position position)
  ```

– **Description copied from IPositionAreaMapper (in 7.5, page 63)**

Getter for all `Area(s)` (in 7.5, page 65) containing a given `Position` (in 7.5, page 76).

– **Parameters**

* `position` – The `Position` (in 7.5, page 76) given.

– **Returns** – All `Area(s)` (in 7.5, page 65) containing the `Position` (in 7.5, page 76) position.

Members inherited from class ComponentManager

`pegs.model.ComponentManager` (in 7.5, page 69)

- `public void` **add**`(Component` **component**`)`
- `public Component` **getAll**`()`
- `public Component` **getByIds**`(ID[]` **ids**`)`
- `public void` **notifyUpdated**`(ID` **id**`)`
- `public void` **remove**`(Component` **component**`)`
- `public void` **subscribe**`(ComponentObserver` **componentObserver**`)`
- `public void` **unsubscribe**`(ComponentObserver` **componentObserver**`)`

8 PersistentData

Das `PersistentData`-Paket dient der Verwaltung eines Speichers, der für die dauerhafte Speicherung verantwortlich ist.
Das Ziel ist, dass der Netzwerkadministrator bei einem Neustart des Systems nicht alle Daten erneut integrieren muss, sondern auf bereits gespeicherte Daten zurückgreifen kann. Beim Starten (Initialisieren) des Systems muss der Speicherinhalt ausgelesen werden. Während dem Systembetrieb müssen Daten in den Speicher gespeichert werden.

8.1 Klassendiagramm

Das Klassendiagramm in Abbildung 8.1.1 zeigt die Klassen des Pakets PersistentData.

8.2 Funktion

Das `PersistentData`-Paket wird während dem Systembetrieb mittels eines `PersistentDataComponentObserver`s benachrichtigt, falls Daten hinzugefügt, gelöscht oder geändert werden. Es lässt sich die betreffenden Daten geben und integriert diese in eine Datenstruktur, die für die langfristige Speicherung geeignet ist.

Außerdem kann das System alle Daten eines Typs (`AccessPoint`, `Area`, `Permission` und `LinkGroup`) ausgeben, die im Speicher aktuell gespeichert sind. Dies kommt vor allem beim Starten (Initialisieren) des Systems zum Einsatz.

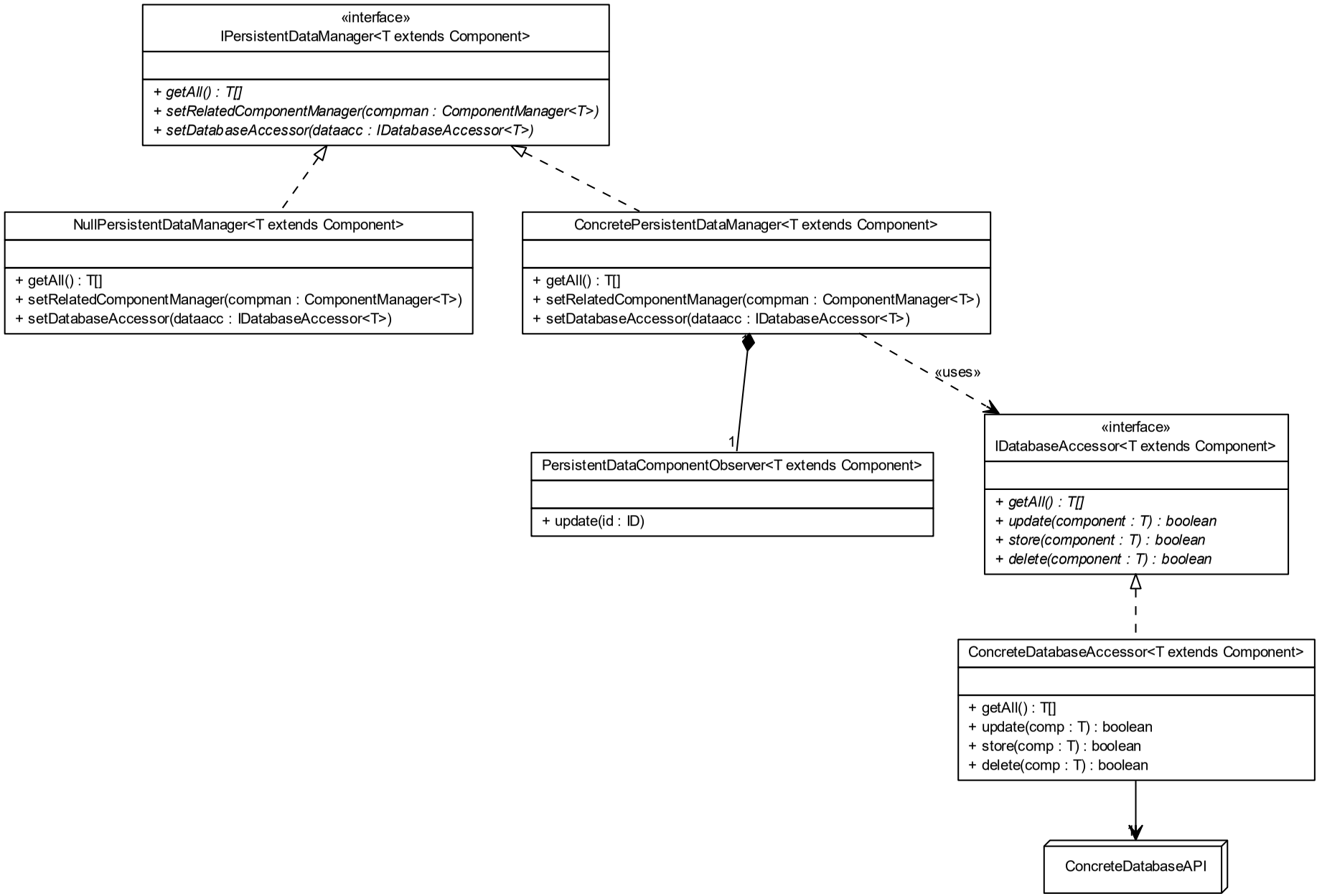

Abbildung 8.1.1: UML-Klassendiagramm zum Paket PersistentData.

8.3 Schnittstellen

Die Schnittstelle nach Außen, die einzige Schnittstelle, ist definiert durch das «`interface IPersistentDataManager<T extends Component>`». Sie bietet die Möglichkeiten, sämtliche gespeicherte Daten aus dem Speicher auszugeben: Mit der Methode `getAll()` werden alle Komponenten des Typs `T` ausgelesen, die sich aktuell in der Datenbank befinden.

8.4 Interne Struktur

Das Paket besteht aus zwei Teilen, dem Speicher und dessen Zugriffsverwaltung.

Der Speicher ist eine konkrete Datenbank (`ConcreteDatabaseAPI`), die durch eine auf sie zugeschnittene Schnittstelle (`ConcreteDatabaseAccessor`) direkt adressiert werden kann. Um modular und unabhängig von einer konkreten Datenbank-Implementierung zu bleiben, wird die zugeschnittene Schnittstelle im Interface `IDatabaseAccessor` beschrieben. Diese Schnittstelle muss pro Komponente (`AccessPoint`, `Area`, `Permission` und `LinkGroup`) konstruiert werden. Deswegen wird wie in Diagramm 8.1.1 ersichtlich ein Generic verwendet. Folgende Speicheroperationen sind möglich:

- eine `getAllStored`-Methode, die alle gespeicherten Daten dieser Komponente ausgibt
- eine `update`-Methode, die eine gegebene konkrete Komponente mit veränderten Daten updatet
- eine `store`-Methode, die eine neue Komponente in die Datenbank einspeichert und
- eine `delete`-Methode, die eine gegebene konkrete Komponente aus dem Speicher löscht

Die Zugriffsverwaltung (`PersistenDataManager`) koordiniert den Zugriff auf den Speicher. Auch sie muss pro Komponente initialisiert werden. Sie implentiert die bereits beschriebene Schnittstelle nach außen, sodass sie bei Systemstart aufgerufen werden kann. Die automatische Benachrichtigung bei Hinzufügen, Ändern und Löschen von zu speichernden Komponenten geschieht durch das Observer-Pattern. Die Zugriffsverwaltung hat für ihren Komponenten-Typ einen Observer, der bei Benachrichtigung eine entsprechende Aktion auslöst. Dieser Observer (`PersistentDataComponentObserver`) implementiert die Schnittstelle `ComponentObserver` (aus dem Model) und kennt den jeweiligen `ComponentManager` (ebenfalls aus dem Model), um sich notwendige Daten geben zu lassen.

Ist keine langfristige Speicherung erwünscht, so kann bei Systemstart anstatt eines `ConcretePersistentDataManager` auch ein Nullobjekt `NullPersistentDataManager` konstruiert werden. Dieses gibt bei entsprechender Anfrage durch das Model keine Daten aus, da keine Daten dauerhaft schon gespeichert sind. Außerdem meldet es keine Observer an, sodass bei Änderung der Komponenten im Model keine Aktion ausgelöst wird. Für das Model bleibt die Funktionsweise jedoch gleich.

8.5 Package pegs.persistentdata

Package Contents *Page*

Interfaces

Classes

Interface IDatabaseAccessor

Interface to access the database storage.

Declaration

```
public interface IDatabaseAccessor
```

All known subinterfaces

ConcreteDatabaseAccessor (in 8.5, page 83)

All classes known to implement interface

ConcreteDatabaseAccessor (in 8.5, page 83)

Methods

- **delete**

  ```
  boolean delete(pegs.model.Component component)
  ```

  - **Description**

    Deletes a `Component` (in 7.5, page 67) in the data base.

  - **Parameters**

    * `component` – The `Component` (in 7.5, page 67) to delete.

  - **Returns** – True if deleting the `Component` (in 7.5, page 67) was successful, false else.

- **getAll**

  ```
  pegs.model.Component[] getAll()
  ```

  - **Description**

    Retrieves all known `Component` (in 7.5, page 67) objects of type T stored in the data base.

  - **Returns** – The `Component` (in 7.5, page 67) objects from the persistent data storage.

- **store**

  ```
  boolean store(pegs.model.Component component)
  ```

  - **Description**

    Stores a `Component` (in 7.5, page 67) in the data base.

  - **Parameters**

    * `component` – The `Component` (in 7.5, page 67) to store.

  - **Returns** – True if storing the `Component` (in 7.5, page 67) was successful, false else.

- **update**

  ```
  boolean update(pegs.model.Component component)
  ```

- **Description**

  Updates an existing `Component` (in 7.5, page 67) in the data base.

- **Parameters**

  * `component` – The updated `Component` (in 7.5, page 67) to make changes known in the data base.

- **Returns** – True if updating the `Component` (in 7.5, page 67) was successful, false else.

Interface IPersistentDataManager

Interface for managing the connection between the temporary data stored in the Model and a persistent data storage such as a database. Provides methods mainly needed through system initialization.

Declaration

```
public interface IPersistentDataManager
```

All known subinterfaces

NullPersistentDataManager (in 8.5, page 86), ConcretePersistentDataManager (in 8.5, page 85)

All classes known to implement interface

NullPersistentDataManager (in 8.5, page 86), ConcretePersistentDataManager (in 8.5, page 85)

Methods

- **getAll**

  ```
  pegs.model.Component[] getAll()
  ```

  - **Description**

    Retrieves all known `Component` (in 7.5, page 67) objects of type T from the persistent data storage.

  - **Returns** – the `Component(s)` (in 7.5, page 67) from the persistent data storage.

- **setDatabaseAccessor**

```
void setDatabaseAccessor(IDatabaseAccessor dataacc)
```

- **Description**

  Sets the `DatabaseAccessor` (in 8.5, page 80) the PersistentDataManager accesses the database with

- **Parameters**

  * `dataacc` – the `DatabaseAccessor` (in 8.5, page 80) to access the database with

- **setRelatedComponentManager**

```
void setRelatedComponentManager(pegs.model.ComponentManager
    compman)
```

- **Description**

  Sets the `ComponentManager` (in 7.5, page 69) whose data this PersistentDataManager stores persistently.

- **Parameters**

  * `compman` – the `ComponentManager` (in 7.5, page 69) to relate this PersistentDataManager to.

Class ConcreteDatabaseAccessor

A concrete `IDatabaseAccessor` (in 8.5, page 80).

Declaration

```
public class ConcreteDatabaseAccessor
 extends java.lang.Object implements IDatabaseAccessor
```

Constructors

- **ConcreteDatabaseAccessor**

```
public ConcreteDatabaseAccessor()
```

Methods

- **delete**

```
boolean delete(pegs.model.Component component)
```

- **Description copied from IDatabaseAccessor** (**in 8.5, page 80**)

  Deletes a `Component` (in 7.5, page 67) in the data base.

- **Parameters**

  * `component` – The `Component` (in 7.5, page 67) to delete.

- **Returns** – True if deleting the `Component` (in 7.5, page 67) was successful, false else.

- **getAll**

  ```
  pegs.model.Component[] getAll()
  ```

  - **Description copied from IDatabaseAccessor** (**in 8.5, page 80**)

    Retrieves all known `Component` (in 7.5, page 67) objects of type T stored in the data base.

  - **Returns** – The `Component` (in 7.5, page 67) objects from the persistent data storage.

- **store**

  ```
  boolean store(pegs.model.Component component)
  ```

  - **Description copied from IDatabaseAccessor** (**in 8.5, page 80**)

    Stores a `Component` (in 7.5, page 67) in the data base.

  - **Parameters**

    * `component` – The `Component` (in 7.5, page 67) to store.

  - **Returns** – True if storing the `Component` (in 7.5, page 67) was successful, false else.

- **update**

  ```
  boolean update(pegs.model.Component component)
  ```

  - **Description copied from IDatabaseAccessor** (**in 8.5, page 80**)

    Updates an existing `Component` (in 7.5, page 67) in the data base.

  - **Parameters**

    * `component` – The updated `Component` (in 7.5, page 67) to make changes known in the data base.

  - **Returns** – True if updating the `Component` (in 7.5, page 67) was successful, false else.

Class ConcreteDatabaseAPI

A concrete database API.

Declaration

```
public class ConcreteDatabaseAPI
 extends java.lang.Object
```

Constructors

- **ConcreteDatabaseAPI**

  ```
  public ConcreteDatabaseAPI()
  ```

Class ConcretePersistentDataManager

Manages the loading and storing of one type of `Component` (in 7.5, page 67).

Declaration

```
public class ConcretePersistentDataManager
 extends java.lang.Object implements IPersistentDataManager
```

Constructors

- **ConcretePersistentDataManager**

  ```
  public ConcretePersistentDataManager()
  ```

Methods

- **getAll**

  ```
  pegs.model.Component[] getAll()
  ```

  - **Description copied from IPersistentDataManager (in 8.5, page 82)**

    Retrieves all known `Component` (in 7.5, page 67) objects of type T from the persistent data storage.

  - **Returns** – the `Component(s)` (in 7.5, page 67) from the persistent data storage.

- **setDatabaseAccessor**

  ```
  void setDatabaseAccessor(IDatabaseAccessor dataacc)
  ```

  - **Description copied from IPersistentDataManager (in 8.5, page 82)**

    Sets the `DatabaseAccessor` (in 8.5, page 80) the PersistentDataManager accesses the database with

  - **Parameters**

    * `dataacc` – the `DatabaseAccessor` (in 8.5, page 80) to access the database with

- **setRelatedComponentManager**

  ```
  void setRelatedComponentManager(pegs.model.ComponentManager
    compman)
  ```

  - **Description copied from IPersistentDataManager (in 8.5, page 82)**

    Sets the `ComponentManager` (in 7.5, page 69) whose data this PersistentDataManager stores persistently.

  - **Parameters**

    * `compman` – the `ComponentManager` (in 7.5, page 69) to relate this PersistentDataManager to.

Class NullPersistentDataManager

Realizes the null object pattern for `ConcretePersistentDataManager` (in 8.5, page 85) if no storage of persistent data is supposed to exist In this case an object of `NullPersistentDataManager` (in 8.5, page 86) works as a dummy object for `ComponentManager` (in 7.5, page 69).

Declaration

```
public class NullPersistentDataManager
 extends java.lang.Object implements IPersistentDataManager
```

Constructors

- **NullPersistentDataManager**

  ```
  public NullPersistentDataManager()
  ```

Methods

- **getAll**

  ```
  pegs.model.Component[] getAll()
  ```

  - **Description copied from IPersistentDataManager** (**in 8.5, page 82**)

    Retrieves all known `Component` (in 7.5, page 67) objects of type T from the persistent data storage.

  - **Returns** – the `Component(s)` (in 7.5, page 67) from the persistent data storage.

- **setDatabaseAccessor**

  ```
  void setDatabaseAccessor(IDatabaseAccessor dataacc)
  ```

  - **Description copied from IPersistentDataManager** (**in 8.5, page 82**)

    Sets the `DatabaseAccessor` (in 8.5, page 80) the PersistentDataManager accesses the database with

  - **Parameters**

    * `dataacc` – the `DatabaseAccessor` (in 8.5, page 80) to access the database with

- **setRelatedComponentManager**

  ```
  void setRelatedComponentManager(pegs.model.ComponentManager
    compman)
  ```

  - **Description copied from IPersistentDataManager** (**in 8.5, page 82**)

    Sets the `ComponentManager` (in 7.5, page 69) whose data this PersistentDataManager stores persistently.

  - **Parameters**

    * `compman` – the `ComponentManager` (in 7.5, page 69) to relate this PersistentDataManager to.

Class PersistentDataComponentObserver

An Observer for a `ComponentManager` (in 7.5, page 69) as part of the Observer pattern. It is used to get all data on change in the model to store it persistently.

Declaration

```
public class PersistentDataComponentObserver
 extends pegs.model.ComponentObserver
```

Constructors

- **PersistentDataComponentObserver**

  ```
  public PersistentDataComponentObserver()
  ```

Methods

- **update**

  ```
  public void update(pegs.model.ID id)
  ```

Members inherited from class ComponentObserver

`pegs.model.ComponentObserver` (in 7.5, page 71)
- `public void` **refreshComponents**`(ID[] ids)`

9 PositionCalculator

Das `PositionCalculator`-Paket dient der Berechnung einer Position.

9.1 Klassendiagramm

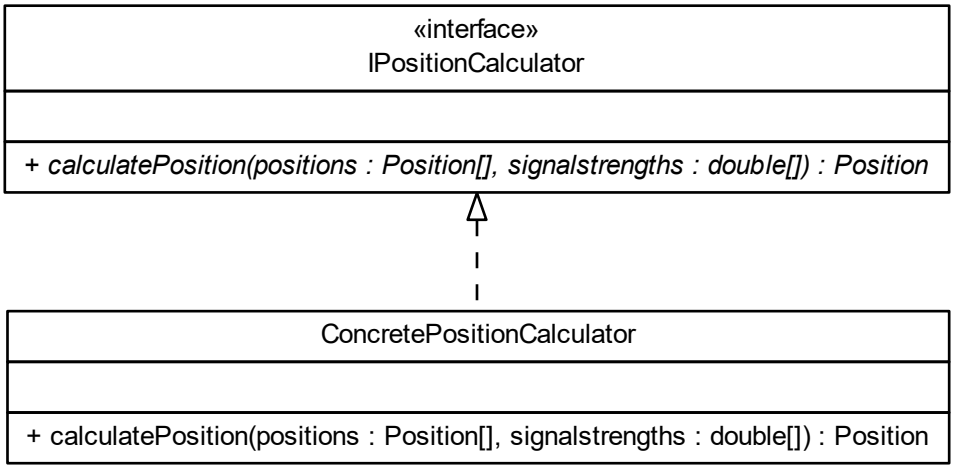

Abbildung 9.1.1: UML-Klassendiagramm zum Paket PositionCalculator

Das Diagramm 9.1.1 zeigt die Klassen des Pakets PositionCalculator.

9.2 Funktion

Das Paket hat genau eine Funktion: Wird ein neues ortenbaren Gerät (`AccessPoint` oder `Device` gefunden, so muss ihm eine Position zugewiesen werden. Das Berechnen dieser Position mit empfangenden/ortenden Geräten bzw. deren Positionen und den jeweiligen Signalstärken ist die Funktion dieses Pakets.

9.3 Schnittstellen

Die Schnittstelle nach außen, die einzige Schnittstelle, ist definiert durch das «`interface PositionCalculator`». Sie bietet die beschriebene Funktion, indem sie mit gegebenen `Positions` und den jeweiligen Signalstärken die Methode `calculatePosition` anbietet, die die berechnete Position zurückgibt. Diese ist Abbildung 9.1.1 zu erkennen.

9.4 Interne Struktur

Um Unabhängig von der jeweiligen Lokalisierungstechnik zu bleiben, gibt es eine Schnittstelle, die die Funktionalität abdeckt. Diese kann von konkreten Klassen implementiert werden. Die konkrete Technik (z.B. Lateration bzw. Multilateration) ist ein Implementierungsdetaill und kann daher hinter einer Schnittstelle verborgen bleiben.

9.5 Klassendokumentation

9.6 Package pegs.positioncalculator

Package Contents *Page*

Interfaces

Classes

Interface IPositionCalculator

Interface for the functionality of calculating a `Position` (in 7.5, page 76). This interface provides a single method for that: `calculatePosition(Position[], double[])` .

Declaration

```
public interface IPositionCalculator
```

All known subinterfaces

ConcretePositionCalculator (in 9.6, page 90)

All classes known to implement interface

ConcretePositionCalculator (in 9.6, page 90)

Methods

- **calculatePosition**

  ```
  pegs.model.Position calculatePosition(pegs.model.Position[]
      positions,double[] signalstrengths)
  ```

  - **Description**

    Returns the calculated `Position` (in 7.5, page 76) of a certain object. For the calculation the `Position` (in 7.5, page 76) of the locating objects is used as well as the measured signal strengths when receiving the certain object.

  - **Parameters**

    * `positions` – The `Positions` (in 7.5, page 76) of the objects localizing the certain object.
    * `signalstrengths` – The signal strengths the localizing objects measure.

  - **Returns** – The calculated `Position` (in 7.5, page 76).

Class ConcretePositionCalculator

An example class implementing the `IPositionCalculator` (in 9.6, page 89) interface.

Declaration

```
public class ConcretePositionCalculator
 extends java.lang.Object implements IPositionCalculator
```

Constructors

- **ConcretePositionCalculator**

  ```
  public ConcretePositionCalculator()
  ```

Methods

- **calculatePosition**

  ```
  pegs.model.Position calculatePosition(pegs.model.Position[]
      positions,double[] signalstrengths)
  ```

  - **Description copied from IPositionCalculator** (**in 9.6, page 89**)

    Returns the calculated `Position` (in 7.5, page 76) of a certain object. For the calculation the `Position` (in 7.5, page 76) of the locating objects is used as well as the measured signal strengths when receiving the certain object.

  - **Parameters**
    - `positions` – The `Positions` (in 7.5, page 76) of the objects localizing the certain object.
    - `signalstrengths` – The signal strengths the localizing objects measure.
  - **Returns** – The calculated `Position` (in 7.5, page 76).

10 View

Das Paket View stellt eines der drei Pakete der grundlegenden *Model View Controller* Architektur dar. Es ist sowohl verantwortlich für die Darstellung der Business Data, als auch für das Entgegennehmen von Änderungswünschen des Administrators an diesen.

10.1 Klassendiagramm

Das Diagramm 10.1.1 zeigt die Klassen des Pakets View.

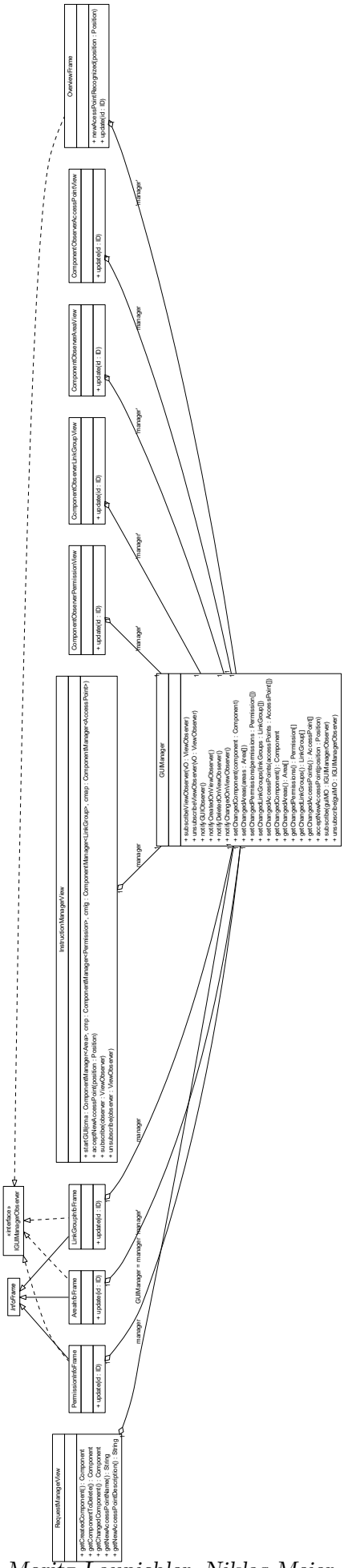

Abbildung 10.1.1: UML-Klassendiagramm zum Paket View.

10.2 Funktion

Das Paket View stellt die graphische Schnittstelle zum Administrator dar, um die Business Data anzuzeigen. Sie besitzt folgende Aufgaben:

- Darstellen der Business Data.
- Entgegennehmen von Änderungswünschen an Komponenten der Business Data.
- Bei Änderungen der Business Data außerhalb des Pakets diese umgehend anzuzeigen.

Das Paket View stellt zur jeder Zeit die Business Data des Pakets Model graphisch dar, um dem Administrator eine einfache Übersicht über das momentan laufende System zu geben. Hierbei ist es ihm möglich, detailliertere Informationen zu einzelnen Komponenten einzusehen und Komponenten wie Bereiche, Kopplungsgruppen und Berechtigungen zu erstellen. Sollte eine Änderungen der Business Data ohne Mitwirken des Administrators geschehen, so wird dieses Paket informiert und stellt die neuen Daten unverzüglich visuell zur Verfügung.
Das Paket besitzt die Möglichkeit, neue Komponenten zu erstellen und diese zu modifizieren. Das Entfernen einer Komponente hingegen ist nicht dessen Aufgabe.

10.3 Schnittstellen

Um die Business Data darzustellen implementiert View jeweils für die Komponenten `Area`, `Permission` und `LinkGroup` einen `ComponentObserver<T extends Component>` mit entsprechender Komponente. Diese ermöglichen es über Änderungen im Model informiert zu werden, sich diese zu holen und letztendlich anzuzeigen.

Für Änderungswünsche hingegen besitzt das Paket eine Schnittstelle zum Paket Controller. Diese Schnittstelle ist dazu da, um Anfragen des Controllers einfach zu beartbeiten und um einen direkten Zugriff auf den `GUIManager`, welcher das Herzstück des Pakets ist, zu verhindern. Konkret besteht diese Schnittstelle aus zwei Klassen.
Die Klasse `InstructionManagerView` dient dazu, Befehle des Controllers entgegenzunehmen, wie zum Beispiel die GUI starten oder einen Observer an- oder abzumelden.
Die Klasse `RequestManagerView` hingegen verwaltet Anfragen des Pakets Controller, wenn dieser Daten aus dem Paket haben möchte. Hierfür besitzt die Klasse verschiedene getter-Methoden, um dies zu ermöglichen.

10.4 Interne Struktur

Die interne Struktur des Packets View lässt sich in drei Teile unterteilen. Genauer sind dies die Schnittstelle zum Paket Controller, einem `Beobachter-Entwurfsmuster` zur Kommunikation zwischen den einzelnen offenen Fenstern und der Implementierung der Beobachter, deren Superklasse `ComponentObserver<T extends Component>` in dem Paket Model lokalisiert ist.

Schnittstelle zum Paket Controller Die Schnittstelle zum Paket Controller besteht aus zwei Klassen. Die Klasse `InstructionManagerView` delegiert Befehle weiter, welche

die GUI aktiv sichtbar beeinflussen. Hierbei ruft sie durch die Methode `startGUI()` den Constructor des Startfensters `OverviewFrame` auf. Des weiteren meldet sie dem Startfenster, sollte ein neuer `AccessPoint` erkannt worden sein, diesen. Auch können hier durch die Methoden subscribe und unsubscribe Beobachter des Controllers an- beziehungsweise abgemeldet werden. Dies kapselt die Implementierungsdetails im View vom Controller ab, wodurch dieser diese nicht erkennen kann. Die Klasse `RequestManagerView` bietet dem Controller Methoden an, damit dieser nicht direkt auf die Daten im View zugreift und sich um die Datenhaltung im View keine Gedanken machen muss. Hierzu beinhaltet sie getter-Methoden, welche wahlweise ein erstellte, geänderte oder zu löschende `Component` zurück gibt.

Beobachter-Entwurfsmuster Das Beobachter-Entwurfsmuster innerhalb des Pakets umfasst den `GUIManager`, welcher die Business Data, die in der GUI dargestellt wird, sowie alle offenen Fenster der Benutzeroberfläche, hält. Hierbei nimmt die Klasse `GUIManager` die Position des Subjekts und des Publishers in einem `Beobachter-Entwurfsmusters` ein. Er informiert die Beobachter und wird gleichzeitig von diesen beobachtet. Die Fenster hingegen stellen die Beobachter dar. Sie beobachten den `GUIManager` und werden von diesem bei Änderungen benachrichtigt und können geänderte `Components` an diesen weiterleiten.

Beobachter des Models Die Implementierung der Klasse `ComponentObserver<T extends Component>` des Pakets Model erfolgt, in dem vier Klassen mit einem der Komponenten `Area`, `Permission`, `LinkGroup` und `AccessPoint` von diesem erben. Die so entstehenden Beobachter werden bei Veränderungen der Business Data im Model angesprochen. Hierbei wird nur der Observer der zugehörigen geänderten Komponente benachrichtigt. Dieser Observer kann sich dann die Daten aus dem Model mittels des jeweiligen `ComponentManager` holen und diese an den GUIManager weitergeben.

10.5 Package pegs.view

Package Contents *Page*

Interfaces

Classes

Interface IGUIManagerObserver

The interface IGUIManagerObserver extends the interface IObserver.

Declaration

```
public interface IGUIManagerObserver
 extends pegs.common.IObserver
```

All known subinterfaces

PermissionInfoFrame (in 10.5, page 107), OverviewFrame (in 10.5, page 106), LinkGroupInfoFrame (in 10.5, page 106), AreaInfoFrame (in 10.5, page 96)

All classes known to implement interface

PermissionInfoFrame (in 10.5, page 107), OverviewFrame (in 10.5, page 106), LinkGroupInfoFrame (in 10.5, page 106), AreaInfoFrame (in 10.5, page 96)

Class AreaInfoFrame

AreaInfoFrame is a window in which an `Area` (in 7.5, page 65) with its `AreaInfoFrame` (in 10.5, page 96) and `AreaInfoFrame` (in 10.5, page 96) is visualized.

Declaration

```
public class AreaInfoFrame
 extends pegs.view.InfoFrame implements IGUIManagerObserver
```

Constructors

- **AreaInfoFrame**

  ```
  public AreaInfoFrame(pegs.model.Area area,GUIManager manager
      )
  ```

  - **Description**

    Initializes an AreaInfo window with its `Area` (in 7.5, page 65) and the `GUIManager` (in 10.5, page 99), to access its data.

  - **Parameters**

    * `area` – An `Area` (in 7.5, page 65) which is to visualize.
    * `manager` – The `GUIManager` (in 10.5, page 99), to access its data.

Methods

- **update**

  ```
  public void update(pegs.model.ID id)
  ```

Class ComponentObserverAccessPointView

ComponentObserverAreaView observes the ComponentManager.

Declaration

```
public class ComponentObserverAccessPointView
 extends pegs.model.ComponentObserver
```

Constructors

- **ComponentObserverAccessPointView**

  ```
  public ComponentObserverAccessPointView(GUIManager manager)
  ```

  - **Description**

    Creates a new ComponentObserverAccessPointView.

  - **Parameters**

    * `manager` – The `GUIManager` (in 10.5, page 99).

Methods

- **update**

  ```
  public void update(pegs.model.ID id)
  ```

Members inherited from class ComponentObserver

`pegs.model.ComponentObserver` (in 7.5, page 71)
- `public void` **refreshComponents**`(ID[] ids)`

Class ComponentObserverAreaView

ComponentObserverAreaView observes the ComponentManager.

Declaration

```
public class ComponentObserverAreaView
 extends pegs.model.ComponentObserver
```

Constructors

- **ComponentObserverAreaView**

  ```
  public ComponentObserverAreaView(GUIManager manager)
  ```

  - **Description**

    Creates a new ComponentObserverAreaView.

  - **Parameters**

    * `manager` – The `GUIManager` (in 10.5, page 99).

Methods

- **update**

  ```
  public void update(pegs.model.ID id)
  ```

Members inherited from class ComponentObserver

`pegs.model.ComponentObserver` (in 7.5, page 71)
- `public void` **refreshComponents**`(ID[] ids)`

Class ComponentObserverLinkGroupView

ComponentObserverLinkGroupView observes the ComponentManager.

Declaration

```
public class ComponentObserverLinkGroupView
 extends pegs.model.ComponentObserver
```

Constructors

- **ComponentObserverLinkGroupView**

  ```
  public ComponentObserverLinkGroupView(GUIManager manager)
  ```

  - **Description**

    Creates a new ComponentObserverLinkGroupView.

  - **Parameters**

* `manager` – The `GUIManager` (in 10.5, page 99).

Methods

- **update**

```
public void update(pegs.model.ID id)
```

Members inherited from class ComponentObserver

`pegs.model.ComponentObserver` (in 7.5, page 71)
- `public void` **refreshComponents**`(ID[] ids)`

Class ComponentObserverPermissionView

ComponentObserverPermissionView observes the ComponentManager.

Declaration

```
public class ComponentObserverPermissionView
 extends pegs.model.ComponentObserver
```

Constructors

- **ComponentObserverPermissionView**

```
public ComponentObserverPermissionView(GUIManager manager)
```

  - **Description**

    Creates a new ComponentPermissionView.

  - **Parameters**

    * `manager` – The `GUIManager` (in 10.5, page 99).

Methods

- **update**

```
public void update(pegs.model.ID id)
```

Members inherited from class ComponentObserver

`pegs.model.ComponentObserver` (in 7.5, page 71)
- `public void` **refreshComponents**`(ID[] ids)`

Class GUIManager

The GUIManager contains every data known to the GUI.

Declaration

```
public class GUIManager
 extends java.lang.Object implements pegs.common.IObservable
```

Constructors

- **GUIManager**

  ```
  public GUIManager()
  ```

Methods

- **acceptNewAccessPoint**

  ```
  public void acceptNewAccessPoint(pegs.model.Position
      position)
  ```

  - **Description**

    Notifies the `OverviewFrame` (in 10.5, page 106), that a new `AccessPoint` (in 7.5, page 64) is available by giving him its `Position` (in 7.5, page 76).

  - **Parameters**

    * `position` – The `Position` (in 7.5, page 76) of the `AccessPoint` (in 7.5, page 64).

- **getChangedAccessPoints**

  ```
  public pegs.model.AccessPoint[] getChangedAccessPoints()
  ```

  - **Description**

    Returns an AccessPoint[] of changed `GUIManager` (in 10.5, page 99).

  - **Returns** – An AccessPoint[].

- **getChangedAreas**

  ```
  public pegs.model.Area[] getChangedAreas()
  ```

  - **Description**

    Returns an Area[] of changed `GUIManager` (in 10.5, page 99).

  - **Returns** – An Area[].

- **getChangedComponent**

  ```
  public pegs.model.Component getChangedComponent()
  ```

  - **Description**

    Returns a changed `Component` (in 7.5, page 67).

  - **Returns** – A `Component` (in 7.5, page 67).

- **getChangedLinkGroups**

  ```
  public pegs.model.LinkGroup[] getChangedLinkGroups()
  ```

  - **Description**

    Returns a LinkGroup[] of changed `GUIManager` (in 10.5, page 99).

  - **Returns** – A LinkGroup[].

- **getChangedPermissions**

  ```
  public pegs.model.Permission[] getChangedPermissions()
  ```

  - **Description**

    Returns a Permission[] of changed `GUIManager` (in 10.5, page 99).

  - **Returns** – A Permission[].

- **notifyChangedOnViewObserver**

  ```
  public void notifyChangedOnViewObserver()
  ```

  - **Description**

    Notifies every `ViewObserver` (in 6.5, page 57) known to the GUIManager, that a `Component` (in 7.5, page 67) got changed.

- **notifyCreatedOnViewObserver**

  ```
  public void notifyCreatedOnViewObserver()
  ```

- **Description**

  Notifies every `ViewObserver` (in 6.5, page 57) known to the GUIManager, that a `Component` (in 7.5, page 67) got created.

- **notifyDeletedOnViewObserver**

  ```
  public void notifyDeletedOnViewObserver()
  ```

  - **Description**

    Notifies every `ViewObserver` (in 6.5, page 57) known to the GUIManager, that a `Component` (in 7.5, page 67) should get deleted.

- **notifyGUIObserver**

  ```
  public void notifyGUIObserver()
  ```

  - **Description**

    Notifies every `IGUIManagerObserver` (in 10.5, page 95) known to the GUIManager.

- **setChangedAccessPoints**

  ```
  public void setChangedAccessPoints(pegs.model.AccessPoint[]
      accessPoints)
  ```

  - **Description**

    Sets the AccessPoint[] changedAccessPoints variable in this class to accessPoint.

  - **Parameters**

    * `accessPoint` – An AccessPoint[] to change.

- **setChangedAreas**

  ```
  public void setChangedAreas(pegs.model.Area[] areas)
  ```

  - **Description**

    Sets the Area[] changedAreas variable in this class to areas.

  - **Parameters**

    * `areas` – The Area[] to change.

- **setChangedComponent**

  ```
  public void setChangedComponent(pegs.model.Component
      component)
  ```

- **Description**

  Sets the ChangedComponent variable in this class to `Component` (in 7.5, page 67).

- **Parameters**

  * `component` – The `Component` (in 7.5, page 67) to set.

- **setChangedLinkGroups**

  ```
  public void setChangedLinkGroups(pegs.model.LinkGroup[]
      linkGroups)
  ```

  - **Description**

    Sets the LinkGroup[] changedLinkGroups variable in this class to linkGroups.

  - **Parameters**

    * `linkGroups` – A LinkGroup[] to change.

- **setChangedPermissions**

  ```
  public void setChangedPermissions(pegs.model.Permission[]
      permissions)
  ```

  - **Description**

    Sets the Permission[] changedPermission variable in this class to permissions.

  - **Parameters**

    * `permissions` – The Permission[] to change.

- **subscribe**

  ```
  public void subscribe(IGUIManagerObserver guiMO)
  ```

- **subscribeViewObserver**

  ```
  public void subscribeViewObserver(pegs.controller.
      ViewObserver vO)
  ```

  - **Description**

    Subscribes a `ViewObserver` (in 6.5, page 57).

  - **Parameters**

    * `vo` – A `ViewObserver` (in 6.5, page 57).

- **unsubscribe**

  ```
  public void unsubscribe(IGUIManagerObserver guiMO)
  ```

- **unsubscribeViewObserver**

  ```
  public void unsubscribeViewObserver(pegs.controller.
      ViewObserver vO)
  ```

  - **Description**

    Unsubscribes a `ViewObserver` (in 6.5, page 57).

  - **Parameters**

    * `vo` – A `ViewObserver` (in 6.5, page 57).

Class InfoFrame

Implements basic methods for an InfoFrame used in PEGS.

Declaration

```
public abstract class InfoFrame
 extends java.lang.Object
```

All known subclasses

PermissionInfoFrame (in 10.5, page 107), LinkGroupInfoFrame (in 10.5, page 106), AreaInfoFrame (in 10.5, page 96)

Constructors

- **InfoFrame**

  ```
  public InfoFrame()
  ```

Class InstructionManagerView

InstructionManagerView is a class which provides all essential methods a different package would like to access, to give instructions to this package.

Declaration

```
public class InstructionManagerView
 extends java.lang.Object implements pegs.common.IObservable
```

Constructors

- **InstructionManagerView**

  ```
  public InstructionManagerView()
  ```

Methods

- **acceptNewAccessPoint**

  ```
  public void acceptNewAccessPoint(pegs.model.Position
      position)
  ```

  - **Description**

    Notifies the `GUIManager` (in 10.5, page 99), that a new `AccessPoint` (in 7.5, page 64) is available by giving him its `Position` (in 7.5, page 76).

  - **Parameters**

    * `position` – The `Position` (in 7.5, page 76) of the `AccessPoint` (in 7.5, page 64).

- **startGUI**

  ```
  public void startGUI(pegs.model.ComponentManager cma, pegs.
      model.ComponentManager cmp, pegs.model.ComponentManager
      cmlg, pegs.model.ComponentManager cmap)
  ```

  - **Description**

    Starts the GUI and adds the associated `ViewObserver` (in 6.5, page 57) to every `ComponentObserver` (in 7.5, page 71).

  - **Parameters**

    * `cma` – The `ComponentManager` (in 7.5, page 69) for `Area` (in 7.5, page 65).
    * `cmp` – The `ComponentManager` (in 7.5, page 69) for `Permission` (in 7.5, page 75).
    * `cmlg` – The `ComponentManager` (in 7.5, page 69) for `LinkGroup` (in 7.5, page 72).
    * `cmap` – The `ComponentManager` (in 7.5, page 69) for `AccessPoint` (in 7.5, page 64).

- **subscribe**

  ```
  public void subscribe(pegs.controller.ViewObserver observer)
  ```

- **Description**

  Adds a `ViewObserver` (in 6.5, page 57) to the observer list.

- **Parameters**

  * `observer` – A `ViewObserver` (in 6.5, page 57) observing this package.

- **unsubscribe**

```
public void unsubscribe(pegs.controller.ViewObserver
    observer)
```

- **Description**

  Removes a `ViewObserver` (in 6.5, page 57) from the observer list.

- **Parameters**

  * `observer` – A `ViewObserver` (in 6.5, page 57) observing this package.

Class LinkGroupInfoFrame

LinkGroupInfoFrame is a window in which a `LinkGroup` (in 7.5, page 72) with its `LinkGroupInfoFrame` (in 10.5, page 106) and `LinkGroupInfoFrame` (in 10.5, page 106) is visualized.

Declaration

```
public class LinkGroupInfoFrame
 extends pegs.view.InfoFrame implements IGUIManagerObserver
```

Constructors

- **LinkGroupInfoFrame**

```
public LinkGroupInfoFrame(pegs.model.LinkGroup linkGroup,
    GUIManager manager)
```

- **Description**

  Initializes a LinkGroupInfoFrame window with its `LinkGroup` (in 7.5, page 72) and the `GUIManager` (in 10.5, page 99), to access its data.

- **Parameters**

  * `linkGroup` – The `LinkGroup` (in 7.5, page 72) which is to visualize.

  * `manager` – The `GUIManager` (in 10.5, page 99), to access its data.

Methods

- **update**

```
public void update(pegs.model.ID id)
```

Class OverviewFrame

OverviewFrame is the window shown when the GUI is initalized. Closing it will close the whole GUI.

Declaration

```
public class OverviewFrame
 extends java.lang.Object implements IGUIManagerObserver
```

Constructors

- **OverviewFrame**

```
public OverviewFrame(GUIManager manager)
```

  - **Description**

    Creates the window with the existing `GUIManager` (in 10.5, page 99) to be able to access the data held by the `GUIManager` (in 10.5, page 99). Only gets created when the GUI gets initialized, normally called by the `InstructionManagerView` (in 10.5, page 104).

  - **Parameters**

    * `manager` – A GUIManager.

Methods

- **newAcessPointRecognized**

```
public void newAcessPointRecognized(pegs.model.Position
    position)
```

  - **Description**

    Should be used only if a new `AccessPoint` (in 7.5, page 64) wants to join PEGS.

  - **Parameters**

    * `position` – The `Position` (in 7.5, page 76) of the new `AccessPoint` (in 7.5, page 64).

- **update**

```
public void update(pegs.model.ID id)
```

Class PermissionInfoFrame

PermissionInfoFrame is a window in which a `Permission` (in 7.5, page 75) with its `PermissionInfoFrame` (in 10.5, page 107) and `PermissionInfoFrame` (in 10.5, page 107) is visualized.

Declaration

```
public class PermissionInfoFrame
 extends pegs.view.InfoFrame implements IGUIManagerObserver
```

Constructors

- **PermissionInfoFrame**

```
public PermissionInfoFrame(pegs.model.Permission permission,
    GUIManager manager)
```

  - **Description**

    Initializes a PermissionInfo window with its `Permission` (in 7.5, page 75) and the `GUIManager` (in 10.5, page 99), to access its data.

  - **Parameters**

    * `permission` – A `Permission` (in 7.5, page 75) which is to visualize.
    * `manager` – The `GUIManager` (in 10.5, page 99), to access its data.

Methods

- **update**

```
public void update(pegs.model.ID id)
```

Class RequestManagerView

RequestManagerView is a class which provides various getter methods to access data, which have to get updated elsewhere.

Declaration

```
public class RequestManagerView
 extends java.lang.Object
```

Constructors

- **RequestManagerView**

  ```
  public RequestManagerView()
  ```

Methods

- **getChangedComponent**

  ```
  public pegs.model.Component getChangedComponent()
  ```

  - **Description**

    Returns a `Component` (in 7.5, page 67), which got changed in this package.

  - **Returns** – The new `Component` (in 7.5, page 67).

- **getComponentToDelete**

  ```
  public pegs.model.Component getComponentToDelete()
  ```

  - **Description**

    Returns a `Component` (in 7.5, page 67), which should get deleted.

  - **Returns** – The new `Component` (in 7.5, page 67).

- **getCreatedComponent**

  ```
  public pegs.model.Component getCreatedComponent()
  ```

  - **Description**

    Returns a `Component` (in 7.5, page 67), which got created in this package.

  - **Returns** – The new `Component` (in 7.5, page 67).

- **getNewAccessPointDescription**

  ```
  public java.lang.String getNewAccessPointDescription()
  ```

  - **Description**

    Returns the description of a new `AccessPoint` (in 7.5, page 64).

  - **Returns** – String containing the description.

- **getNewAccessPointName**

  ```
  public java.lang.String getNewAccessPointName()
  ```

  - **Description**

    Returns the name of a new `AccessPoint` (in 7.5, page 64).

  - **Returns** – String containing the name.

11 Änderungen zum Pflichtenheft

Es gibt einige entwurfsbedingte Änderungen, die sich von den Anforderungen des Pflichtenhefts unterscheiden. Aufgrund des Wasserfallmodells sind diese Änderungen nun im Entwurfsdokument beschrieben.

Die folgenden Anforderungen mussten verändert werden:

- **OFA30**: Bearbeiten der Positionen von bereits gesetzten Access Points und Bereichen
  Änderung: Nur Positionen von Bereichen können verändert werden, nicht aber der Access Points
  Grund: Das manuelle Verschieben eines Access Points, also die Veränderung einer Position, führt zu Inkonsistenz von manuellen Positionen und den durch Signalstärke berechenbaren Positionen.

Die folgenden Anforderungen mussten verworfen werden:

- **SFA190**: Verwalten von Whitelists über alle Endgeräte, die sich aktuell in den mit einer Berechtigung gekoppelten Bereichen befinden
  Grund: Aufgrund einer Entwurfsentscheidung wird das Erteilen einer Berechtigung nicht an eine Whitelist, also eine Datenhaltung, geknüpft, sondern kann berechnet werden.

- **OFA10**: Visualisieren der Position aller Access Points, Bereichen und Kopplungsgruppen auf einem Grundriss
  Grund: Das Anzeigen eines Grundrisses ist aufgrund von Inkonsistenz von realen Positionen und den durch Signalstärke berechenbaren Position verworfen worden.

- **OFA11**: Laden/Hinzufügen eines Grundrisses
  Grund: Obsolet wegen Verwerfen von OFA11.

- **OFA12**: Anzeigen der Access Points auf einem Grundriss
  Grund: Obsolet wegen Verwerfen von OFA11.
- **OFA13**: Zoomen eines Grundrisses an beliebiger Stelle
  Grund: Obsolet wegen Verwerfen von OFA11.
- **OFA40**: Definieren eines kreisförmigen Bereichs um einen Access Point
  Grund: Diese Anforderung entstammt dem Wunsch, einen Bereich um nur einen Access Point herum zu definieren. Dies zieht jedoch Ungenauigkeit bei der Lokalisierung mit sich und bringt keine wichtige Funktionialität für den Netzwerkadministrator.
- **OFA50**: Entfernen eines Access Points
  Grund: Access Points werden im System zur möglichst genauen Lokalisierung und für die Realisierung der Datenweitergabe über mehrere Sprünge gebraucht. Existiert ein Access Point, sollte er auch verwendet werden können.
- **OFA80**: Berechnen einer Heatmap für einen Bereich
  Grund: Aufgrund der Entwurfsentscheidung zu SFA190 kann eine Heatmap nicht mehr ohne großen Mehraufwand berrechnet werden.
- **OFA90**: Anzeigen einer Heatmap
  Grund: Obsolet wegen Verwerfen von OFA90
- **OFA100**: Manuelles Eintragen für dauerhafte Berechtigung von bestimmten Endgeräten
  Grund: Siehe SFA190
- **OFA110**: Manuelles Eintragen für dauerhaftes Blockieren von bestimmten Endgeräten
  Grund: Siehe SFA190
- **OFA120**: Speichern von manuell veränderbaren Blacklists und Whitelists
  Grund: Siehe SFA190
- **OFA130**: Erweitern von Geofences durch einen Komplementärbereich in einem Bereich, in dem die Berechtigungen nicht zugänglich sind
  Grund: Aufgrund einer Entwurfsentscheidung bestehen `Bereiche` aus genau einem Geofence. Ein Komplementärbereich kann also nicht ausgedrückt werden.

Bis auf diese aufgelisteten Anforderungen wurden alle funktionalen Anforderungen und optionalen funktionalen Anforderungen, die im Pflichtenheft genannt wurden, im Entwurf berücksichtigt.