Implementierungsbericht

Phase	Verantwortlicher	E-Mail
Pflichtenheft	Florian Lorenz	lorenz@fim.uni-passau.de
Entwurf	Andreas Wilhelm	wilhelma@fim.uni-passau.de
Spezifikation	Andreas Poxrucker	poxrucke@fim.uni-passau.de
Implementierung	Martin Freund	freund@fim.uni-passau.de
Validierung	Florian Bürchner	buerchne@fim.uni-passau.de
Präsentation	Max Binder	binder@fim.uni-passau.de

15. Dezember 2011

Inhaltsverzeichnis

Zusammenfassung

Dieses Dokument soll einen Überblick über den Ablauf der Implementierungsphase des Projekts geben. Insbesondere sollen die Änderungen und Erweiterungen, die im Vergleich zum während der Entwurfs- und Spezifikationsphase vorgeschlagenen Aufbau notwendig wurden hier zusammengefasst und erläutert werden. Allerdings wird dabei nur auf größere bzw. strukturelle Abweichungen vom Entwurf eingegangen, um den Rahmen dieses Dokuments nicht zu sprengen.

1 Android Anwendung

1.1 Paketaufbau

Während der Implementierung wurde die ursprünglich angedachte Paketstruktur erweitert, um eine bessere Übersichtlichkeit zu schaffen. Insbesondere wurde das Subpaket für Fahrverhalten des Logik-Threads (`sep.conquest.model.behaviour`) erstellt. Außerdem werden aufgrund des *Chain-Of-Responsibility*-Verhaltensmusters viele Nachrichtenklassen, sowie die entsprechenden Handlerklassen, benötigt. Diese wurden den beiden Subpaketen `sep.conquest.model.handler` und `sep.conquest.model.requests` zugeordnet.

Paket	**Beschreibung**
sep.conquest.activity	Aktivitätsklassen für Android Benutzeroberflächen
sep.conquest.controller	Controller-Klasse entsprechend dem MVC-Entwurfsmuster
sep.conquest.model	Klassen der Model-Schicht entsprechend dem MVC-Entwurfsmuster
sep.conquest.model.behaviour	Verhaltensklassen des Logik-Threads
sep.conquest.model.handler	Handlerklassen für Broadcast- und Bluetooth-Behandlung
sep.conquest.model.request	Nachrichtenklassen für Broadcast- und Bluetooth-Kommunikation
sep.conquest.util	Übergreifende Hilfsmittel

1.2 Update-Mechanismus der MVC-Architektur

Es wurde während der Implementierung festgelegt, dass die Anwendungsdialoge (*Observer*) keinen Status der e-pucks selbst halten dürfen. Dadurch musste die Struktur der Update-Nachrichten geändert werden. Neben der vollständigen Karte müssen auch von den Roboter-Statusinformationen tiefe Kopien erstellt und diese an die Dialoge gesendet werden. Um die Karte korrekt darstellen zu können, benötigt die Kartenansicht folgende Informationen:

- `borders` Die Grenzen des Spielfeldes, damit die Ausmaße erfasst werden können.
- `exploredNodes` Die von jedem einzelnen Roboter erkundeten Knoten. Diese Statistikdaten werden auf dem Statistik-Dialog angezeigt.
- `mapList` Eine Liste aller Knoten der bisher erkundeten Karte.

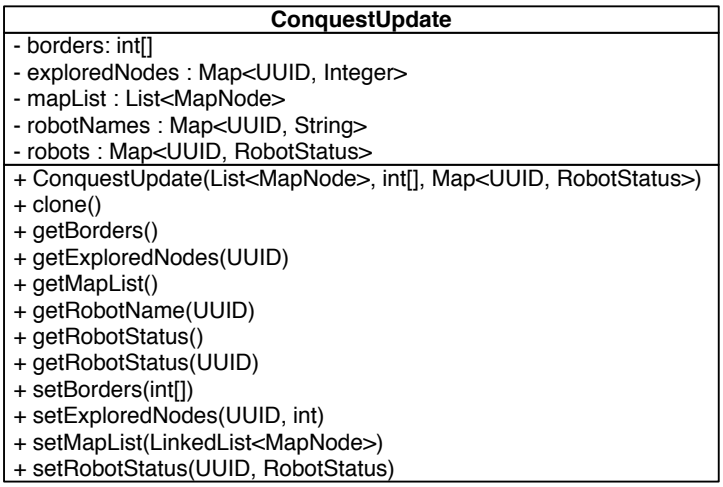

Abbildung 1: UML-Klassendiagramm für `ConquestUpdate`

- `robotNames` Eine *Map* aller Namen der Roboter, für die Identifizierung zur manuellen Steuerung.
- `robots` Die Statusinformationen aller Roboter inkl. Orientierung zur Darstellung.

1.3 Fabrik für Instanzen der Klassen `VirtualPuck` und `RealPuck`

Durch die verschiedenartigen Roboter-Instanzen für simulierte bzw. reale Erkundung wird ein Hilfsmittel benötigt, das die jeweilige Erzeugung erledigt. Die Klasse `PuckFactory` übernimmt diese Aufgabe, indem sie abhängig vom gewünschten Typ die Initialisierung durchführt. Dabei müssen die `Puck`-Ableitungen am Kommunikationsmanager registriert werden, die Startpositionen und Orientierungen gesetzt sein und schließlich das Handshaking initiiert werden.

1.4 Erweiterungen des Simulators

Um die Simulation der Roboter entsprechend den realen Vorbildern gestalten zu können, mussten gegenüber dem Entwurf einige Änderungen gemacht werden. Der Simulator hält je virtuellem Puck eine Instanz der Klasse `SimRobot`, welche die Zustände der Roboter speichert. Ein Timer steuert den Ablauf der Nachrichtenkommunikation, wobei sichergestellt wird, dass jeder Roboter gleichmäßig Nachrichten entgegennimmt und versendet. Damit auch Kollisionen vom Simulator berücksichtigt werden können, sind Fahranweisungen zum nächsten Knoten in 3 Schritten auszuwerten. Sollten sich zwei simulierte Roboter auf dem Weg zu einem Knoten auf der selben Position (in 1/3 Schritten) befinden, so wird eine Kollision gemeldet und zum zuletzt besuchten Knoten zurückgefahren.

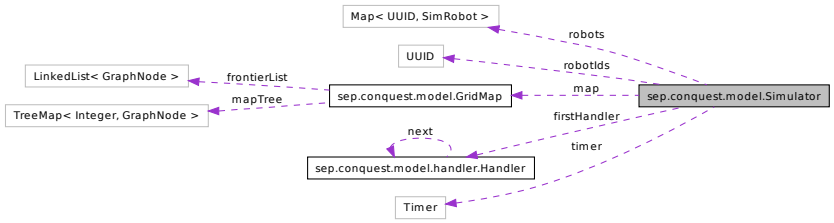

Abbildung 2: Verwendungen von Klasse `Simulator`

1.5 Implementierte Verhaltensklassen

Die Verhaltensklassen steuern das Fahrverhalten der Roboter und werden je nach deren Zustand (Initialisierung, Lokalisierung, Erkundung, Rückkehr) aktiviert. Die Klasse `LogicThread` hält den ersten Eintrag einer Liste von entsprechenden Verhaltensklassen. Beim Wechsel eines Zustands wird die Klasse `BehaviourFactory` verwendet, um eine neue Liste zu laden.
Es existieren folgende Verhaltensklassen:

Initialisierung

- `IdleBehaviour` In dem Verhalten wird für 5 Sekunden auf eingehende Handshaking-Nachrichten reagiert, um sämtliche Teilnehmer zu berücksichtigen.

Lokalisierung

- `LocalLocalizeBehaviour` Hier werden die Startpositionen und Orientierungen der Roboter gesetzt. Außerdem wird der Knotentyp des realen bzw. virtuellen Roboters angefordert und bei Erhalt gespeichert.

Erkundung

- `RemovePathlessBehaviour` Nicht erreichbare Frontier-Knoten werden entfernt.
- `DistanceBehaviour` Mit Hilfe des A*-Algorithmus werden Entfernungen zu den Frontier-Knoten ermittelt.
- `InnerBehaviour` Frontier-Knoten, welche vollständig von bereits erkundeten Knoten liegen, werden als weniger günstig eingestuft.
- `CooperativeBehaviour` Frontier-Knoten, welche bereits andere Teilnehmer erkunden, werden als weniger günstig eingestuft.

Rückkehr

- `ReturnBehaviour` Es wird mit Hilfe des A*-Algorithmus der kürzeste Rückweg ermittelt.

1.6 Implementierte Handlerklassen und Nachrichtenklassen

Die Nachrichtenkommunikation innerhalb der Anwendung und mit dem realen e-puck wurde vollständig durch das *Chain-Of-Responsibility* Entwurfsmuster realisiert. Es wurden folgende Handlerklassen (mit entsprechenden Nachrichtenklassen) verwendet:

Broadcast-Kommunikation der `Puck`-Ableitungen

- `CollisionRequestHandler (CollisionRequest)` Sobald eine Kollisionsmeldung eines anderen Teilnehmers empfangen wird, registriert der Puck diesen Zustand in seinen gespeicherten `RobotStatus` Instanzen.
- `ControlledRequestHandler (ControlledRequest)` Es wird am zugehörigen Puck registriert, dass eine manuelle Steuerung des Benutzers aktiv ist.
- `DriveRequestHandler (DriveRequest)` Eine Steueranweisung des Benutzers wird entgegengenommen und die entsprechenden Kommandos an den Puck weitergereicht.
- `FailureRequestHandler (FailureRequest)` Fehlermeldungen werden analysiert und die entsprechenden Zustandsvariablen an der Puck-Instanz gesetzt.
- `HelloRequestHandler (HelloRequest)` Handshaking-Nachrichten werden empfangen und ggf. an alle Teilnehmer weitergegeben.
- `IntentRequestHandler (IntenRequest)` Von einem anderen Teilnehmer wurde ein beabsichtigter Zielknoten empfangen, diese Information wird unter anderem bei der kooperativen Auswahl des nächsten Frontier-Knoten benötigt.
- `SpeedRequestHandler (SpeedRequest)` Eine Änderung der Fahrgeschwindigkeit wird vom Benutzer gesendet. Diese wird an die entsprechende Puck-Instanz weitergereicht und der Zustand an die anderen Teilnehmer gesendet.
- `StatusUpdateRequestHandler (StatusUpdateRequest)` Es wird eine Aktualisierungsnachricht eines anderen Teilnehmers registriert.

Bluetooth-Kommunikation der `RealPuck`-Instanzen

- `PuckAbyssHandler (PuckRequest)` Es wurde ein Abgrund gemeldet. Die Position wird auf den letzten Knoten und die Orientierung entgegengesetzt der letzten gesetzt.
- `PuckCollisionHandler (PuckRequest)` Es wurde eine Kollision gemeldet. Die Position wird auf den letzten Knoten und die Orientierung entgegengesetzt der letzten gesetzt.
- `PuckNodeHitHandler (PuckRequest)` Der e-puck meldet die Erkundung eines Knoten. Dieser wird in die lokale Karte eingetragen und an sämtliche Teilnehmer gesendet.

- `PuckOKHandler (PuckRequest)` Der e-puck meldet die Ausführung seiner letzten Anweisung, die keine spezielle Antwort benötigt. Ein Beispiel dafür ist eine Drehanweisung. Die OK-Nachricht bewirkt den nächsten Schritt des Logik-Threads.
- `PuckStatusHandler (PuckRequest)` Es wird der aktuelle Status (nach Anforderung) vom e-puck gesendet. Dieser Status wird lokal gespeichert und auch an die anderen Teilnehmer gesendet.

Interne Kommunikation von `SimRobot`-Instanzen

- `SimLEDHandler (VirtualPuckRequest)` Steuerungen der LEDs werden am Simulator in einfache Bestätigungsmeldungen (OK) umgesetzt und zurückgesendet.
- `SimResetHandler (VirtualPuckRequest)` Reset-Anforderungen werden vom `SimRobot` umgesetzt.
- `SimSpeedHandler (VirtualPuckRequest)` Änderungen der Geschwindigkeit werden am Simulator in einfache Bestätigungsmeldungen (OK) umgesetzt und zurückgesendet.
- `SimStatusHandler (VirtualPuckRequest)` Liefert den geforderten Zustand (Position, Orientierung, usw.).
- `SimTurnHandler (VirtualPuckRequest)` Initiiert eine Drehung der `SimRobot`-Instanz und gibt anschließend eine Bestätigungsmeldung (OK) zurück.

2 e-puck

3 Implementierungsaufwand

3.1 Android Anwendung

Tätigkeit	**Arbeitsaufwand** (Plan)	**Arbeitsaufwand** (Ist)
Controller	5 Std.	5 Std.
Environment	10 Std.	10 Std.
Pucks	35 Std.	20 Std.
GridMap	30 Std.	35 Std.
MapSurfaceView	30 Std.	40 Std.
Connect Activity	20 Std.	10 Std.
Bluetooth Suche	5 Std.	5 Std.
Benutzeroberfläche (sonst.)	12 Std.	15 Std.
Statistik Activity	6 Std.	10 Std.
Handler	10 Std.	20 Std.
Komm.-Manager	5 Std.	3 Std.
Nachrichten	5 Std.	3 Std.
Logik	20 Std.	30 Std.
Simulator	18 Std.	15 Std.
Simulator Activity	12 Std.	5 Std.
Anbindung (Simulator)	12 Std.	10 Std.
Abstimmung Broadcast	15 Std.	10 Std.
Abstimmung Clients	15 Std.	15 Std.
Abstimmung Handler	18 Std.	30 Std
Abstimmung Karte	12 Std.	15 Std.
Abstimmung Export	10 Std.	5 Std.
Abstimmung Steuerung	20 Std.	20 Std.
Summe	**325 Std.**	**331 Std.**

3.2 e-puck

Tätigkeit	**Arbeitsaufwand** (Plan)	**Arbeitsaufwand** (Ist)
Interrupt	10 Std.	15 Std.
Kommunikation	30 Std.	35 Std.
ADC	10 Std.	15 Std.
I2C-Bus	15 Std.	20 Std.
Sensoren Abstand	20 Std.	15 Std.
Sensoren Linien	15 Std.	20 Std.
Subsumption 1	25 Std.	30 Std.
Subsumption 2	25 Std.	35 Std.
Abstimmung	5 Std.	10 Std.
Echtzeituhr	5 Std.	5 Std.
Summe	**160 Std.**	**200 Std.**