Lego hunts the Wumpus

– Team 0 –

Software-Entwicklungspraktikum (SEP)
Sommersemester 2012

F e i n e n t w u r f

Auftraggeber:
Technische Universität Braunschweig
Institut für Programmierung und Reaktive Systeme
Prof. Dr. Ursula Goltz
Mühlenpfordtstraße 23
38106 Braunschweig

Betreuer: Benjamin Mensing

Auftragnehmer:

Name	E-Mail-Adresse
Lars Luthmann	l.luthmann@tu-bs.de
Marcel Mast	m.mast@tu-bs.de
Christian Pek	c.pek@tu-bs.de
Yevgen Pikus	y.pikus@tu-bs.de
Tobias Maximilian Vogt	t.vogt@tu-braunschweig.de

Braunschweig, 27.06.2012

Versionsübersicht

Version	Datum	Autor	Status	Kommentar
0.1	13.06.2012	Lars Luthmann	in Bearbeitung	erste Version der Kapitel 4 und 5
0.2	13.06.2012	Christian Pek	in Bearbeitung	Abschnitt 3.1
0.3	18.06.2012	Lars Luthmann, Marcel Mast, Christian Pek, Yevgen Pikus, Tobias Vogt	in Bearbeitung	erste vollständige Version von Kapitel 3
0.4	20.06.2012	Lars Luthmann, Marcel Mast, Yevgen Pikus, Tobias Vogt	in Bearbeitung	erste vollständige Version von Kapitel 2
0.5	21.06.2012	Christian Pek	in Bearbeitung	Kapitel 1 und Einleitung von Kapitel 2 und 3
1.0	22.06.2012	Lars Luthmann, Marcel Mast, Christian Pek, Yevgen Pikus, Tobias Vogt	in Bearbeitung	erster Release Candidate
1.1	25.06.2012	Lars Luthmann, Marcel Mast, Christian Pek, Yevgen Pikus, Tobias Vogt	in Bearbeitung	Verbesserungen in allen Bereichen
1.2	26.06.2012	Lars Luthmann, Marcel Mast, Christian Pek, Yevgen Pikus, Tobias Vogt	in Bearbeitung	Fertigstellung des Dokuments
2.0	27.06.2012	Lars Luthmann, Marcel Mast, Christian Pek, Yevgen Pikus, Tobias Vogt	abgenommen	fertige Version

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Im Grobentwurf wurde die Architektur des Systems vorgestellt und ihre Komponenten erläutert. Anhand dieser Spezifikationen muss die Implementierung vorgenommen werden, sodass das fertige System den Anforderungen aus dem Pflichtenheft entspricht.
Ziel dieses Dokumentes ist es, die Vorgehensweise bei der Implementierung dieser Anforderungen aufzuzeigen. Dabei wird zuerst auf die Aufgabe der im Pflichtenheft definierten Muss-, Wunsch- und Abgrenzungskriterien eingegangen. Im darauffolgenden Kapitel werden dafür erste Implementierungsrichtlinien wie Pakete, Klassen, Attribute sowie Methoden definiert. Zum Abschluss wird das für das gesamte System gültige Datenmodell und die Konfiguration des Servers vorgestellt.

2 Erfüllung der Kriterien

Im folgenden werden die im Pflichtenheft definierten Muss-, Wunsch- und Abgrezungskriterien konkretisiert. Dabei wird ihre Aufgabe detaillierter angeben, sodass das Ziel der Implementierung des jeweiligen Kriterium eindeutig ist. Hierbei ist es wichtig , die Details der Kriterien auf deren Spezifikation abzustimmen.

2.1 Musskriterien

Die folgenden Kriterien entsprechen den Musskriterien des Pflichtenhefts. Ihre Erfüllung ist für den Erfolg des Produkts unabdingbar.

/M10/ Bewegen im Labyrinth:
Der Agent erkennt mit Hilfe der Ultraschallsensoren, wo sich im aktuellen Raum Wände befinden und weiß dadurch, wo er sich hinbewegen kann. Es ist darauf zu achten, dass die Bewegungen und Drehungen des Roboters ungenau sind. Deshalb muss er seine aktuelle Position im Raum durch die Ultraschallsensoren feststellen und ggf. korrigieren.

/M20/ Einhaltung der Regeln:
Der Agent sendet dem Server nach jedem Zug, welche Aktion er ausgeführt hat. Dadurch kann der Server die Einhaltung der Regeln überprüfen.

/M30/ Eigenständiges Lösen des Spiels:
Der Agent löst das Rätsel mit Hilfe der Komponente *Künstliche Intelligenz (/C40/)*.

/M40/ Einlesen der Karte aus einer XML-Datei:
Das Einlesen der Karte geschieht durch die Komponente *Kartentool (/C80/)*. Beim Erstellen der Karte muss darauf geachtet werden, dass die erstellte Datei korrekt ist. Eine Überprüfung der Datei durch das Programm findet beim Wunschkriterium */W40/* statt.

/M50/ Form des Labyrinths:
Beim Erstellen des Labyrinths muss darauf geachtet werden, dass es rechteckig ist. Außerdem darf es keine abgeschlossenen Räume geben (also Räume, die in alle vier Richtungen eine Wand haben).

/M60/ Startposition des Agenten:
Der Agent wird durch den Spieler an die korrekte Position gesetzt. Dabei ist darauf zu achten, dass der Agent im Labyrinth unten links mit Blickrichtung rechts startet.

/M70/ Spiel in absehbarer Zeit beenden:
Das Beenden in absehbarer Zeit ist dadurch gewährleistet, dass der Agent für jeden Zug maximal 15 Sekunden benötigt. Außerdem speichert der Agent, auf welchen Feldern er bereits war. Dadurch bewegt er sich in die Richtung von neuen Felder und findet schließlich das Gold (bzw. stirbt durch den Wumpus oder eine Falle).

/M80/ abwechselndes Ziehen von Agent und Wumpus:
Das abwechselnde Ziehen von Agent und Wumpus ist durch die Komponente *Spielkoordination* (siehe /C50/) gegeben. Diese lässt Agent und Wumpus abwechselnd ziehen.

/M90/ Kommunikation über Bluetooth:
Die Kommunikation findet über Bluetooth statt. Die Nachrichten werden dabei mit Integers kodiert. Beteiligt sind die Komponenten *Kommunikation (Client)* (siehe /C30/) und *Kommunikation (Server)* (siehe /C60/).

/M100/ Überprüfung der Karte:
Die Überprüfung der eingegebenen Karte durch das Programm findet beim Wunschkriterium */W40/* statt. Dies geschiet in der Komponente *Kartentool* (siehe /C80/).

/M110/ Austauschbarkeit des KI-Moduls:
Dieses Kriterium wird dadurch erfüllt, dass das KI-Modul (in Komponente *Künstliche Intelligenz* (siehe/C40/)) nur aus der Klasse *AgentPilot* besteht. Durch Überschreiben der Klasse kann das Modul ausgetauscht werden. Zu beachten ist dabei die Schnittstelle */I30/*, welche zur Kommunikation mit der Komponente *Steuerung* (siehe /C10/) dient und die von *AgentPilot* implementiert wird.

/M120/ Pfeil des Agenten:
Der Roboter des Agenten verfügt über einen Pfeil, der mit Hilfe eines Motors abgefeuert werden kann. Diese Vorrichtung ist fest in Blickrichtung angebaut und kann nicht vom Roboter selbst nachgeladen werden. Während der reale Pfeil nicht unendlich weit fliegen kann, berechnet der Server die Flugbahn des Pfeiles weiter und überprüft so auf mögliche Treffer.

/M130/ Schrei des Wumpus:
Der Roboter des Wumpus wird vom Server benachrichtigt und gibt danach einen Ton aus. Somit schreit er.

/M140/ Erweiterbarkeit für beweglichen Wumpus Das Produkt soll sich leicht um einen beweglichen Wumpus erweitern lassen. Das wird vor allem dadurch sichergestellt, dass fast die gesamte Implementierung des Agenten auch für den Wumpus verwendet werden kann.

/M150/ Wahrnehmung der Wände:
Der Agent nimmt die Wände in jedem Raum des Spielfeldes durch Ultraschallsensoren wahr.

/M160/ Reize erkennen:
Der Agent kann Reize wie Luftzüge, Gestank oder Glitzern nicht physikalisch wahrnehmen *(vgl. /A10/, /A30/ und /A40/)*. Deshalb bekommt der Agent beim Betreten eines neuen Feldes die Information von auf dem Spielfeld befindlichen Reizen vom Server übermittelt.

/M170/ Wahrnehmungseinschränkungen:
Der Agent kann die in */M160/* erwähnten Reize nicht durch Wände hindurch wahrnehmen. Außerdem ist es dem Agenten nicht möglich sämtliche Informationen von diagonal angrenzenden Feldern zu gewinnen.

/M180/ Zielpunktestand:
Der Agent hat das Ziel eine möglichst hohe Punktzahl zu erreichen. Dies erreicht er dadurch, dass der Agent in möglichst wenigen Schritten das Gold findet, ohne dabei in eine Falle oder die Fänge des Wumpus zu geraten, und unversehrt wieder zurück zum Eingang zurückkehrt.

2.2 Wunschkriterien

Die folgenden Kriterien entsprechen den Wunschkriterien des Pflichtenhefts.

/W10/ Wumpus:
Der Wumpus soll sich ebenfalls im Labyrinth bewegen können und erhält am Anfang des Spiels Kenntnis über das gesamte Spielfeld. Durch diese Option wird das Lösen der Aufgabe für den Agenten erschwert.

/W20/ GUI:
Die Erstellung der Karte ist durch eine GUI möglich.

/W30/ Spielergebnisse speichern:
Das Speichern von Spielergebnissen, wie dem Punktestand oder der Karte, ist möglich. Die Informationen werden in extra Dateien gespeichert um zu späteren Zeitpunkten darauf zugreifen zu können. Dies ermöglicht das Erstellen von Spielstatistiken sowie das Laden von bereits erstellten Karten.

/W40/ Korrektheit der Karten:
Die eingelesene Karte wird auf Korrektheit überprüft, d.h. es wird überprüft, ob es einen möglichen Weg für den Agenten vom Startpunkt zum Gold gibt.

2.3 Abgrenzungskriterien:

Die folgenden Kriterien entsprechen den Abgrenzungskriterien des Pflichtenhefts.

/A10/ Wahrnehmung des Gestanks:
Mit vorhandenen Sensoren ist diese Wahrnehmung nicht möglich. Der Agent erhält nach jedem Zug die Feldeininformationen vom Server. Nach der Auswertung dieser Information weiß er, dass auf seinem aktuellen Feld den Gestank gibt *(vgl. /M160/)*.

/A20/ Durch Falltür fallen:
Aufgrund des verfügbares Raumes kann keine physikalischen Falltür aufgebaut werden. Das Fallen durch eine Falltür wird vom Server simuliert. Sollte der Agent den Raum mit einer Falltür befahren, dann wird das Spiel vom Server beendet.

/A30/ Glitzern erkennen:
Mit vorhandenen Sensoren ist die Simulation dieser Wahrnehmung nicht möglich. Das Glitzern erhält der Agent mit der Feldinformation vom Server *(vgl. /M160/)*.

/A40/ Wahrnehmung des Luftzug:
Mit vorhandenen Sensoren ist die Simulation dieser Wahrnehmung nicht möglich. Der Agent erhält die Feldeininformationen vom Server. Nach der Auswertung der Feldinformationen weiß der Agent, dass auf seinem aktuellen Feld ein Luftzug ist *(vgl. /M160/)*.

/A50/ Spieldauer:
Bei großen Spielfeldern wird die Menge der gespeicherten Information sehr groß, dies führt zu längeren Rechenzeiten. Die Fahrzeit vergrößert sich auch, die Anzahl der Räume, die besucht werden müssen, steigt. Dies kann zu unangemessenen Zeiten für die Lösung des Labyrinths führen.

3 Implementierungsentwurf

Im Folgenden werden die ersten Implementierungsdetails der Komponenten des Systems in Hinsicht auf deren Spezifikation, unter Einhaltung der Kriterien aus dem Pflichtenheft, dargestellt. Dabei werden die zu implementierenden Klassen, deren Paketstruktur, sowie deren Attribute und Methoden und deren Bedeutung vorgestellt.

3.1 Gesamtsystem

Im Folgenden werden die Komponenten des Systems spezifiziert. Zuerst wird der Server betrachtet und daraufhin der Client, der sowohl für den Agenten als auch für einen beweglichen Wumpus (siehe */W10/*) gedacht ist.

3.1.1 Komponenten des Servers

Der Server besteht aus vier verschiedenen Komponenten (siehe Abbildung 3.1). Die Komponente *Spielkoordination* ist für die Koordination des Spielablaufs da. Sie soll mit Hilfe der Komponente *Kommunikation* die Verbindungen zu den Clients Agent und Wumpus aufbauen und die Aktionen des Agenten verfolgen, um ihm die nicht von der Sensorik wahrnehmbaren Merkmale, wie das Glitzern des Goldes oder den Gestank des Wumpus, mitteilen zu können. Weiterhin soll sie die Karte laden und bearbeitet Nutzereingaben wie den Start des Spiels.

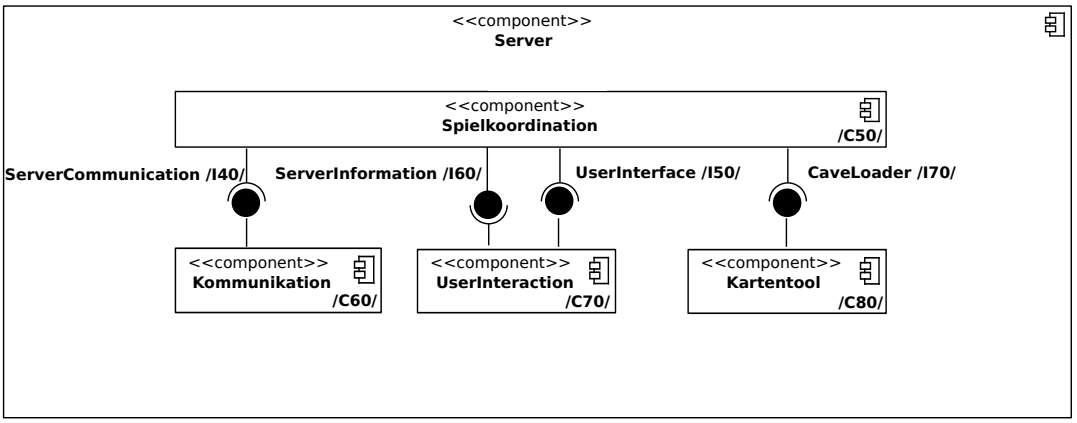

Abbildung 3.1: Komponentendiagramm des Servers

Die Komponente *Kartentool* beinhaltet einen XML-Parser mit dem eine zuvor im XML-Format gespeicherte Karte geladen werden kann. Nach dem Laden soll diese Komponente zudem überprüfen, ob es sich um eine Karte (durch ein XSD-Schema) bzw. eine valide Karte handelt, indem zum Beispiel geprüft wird, ob alle Räume zu erreichen sind (keine abgeschlossenen Räume).
Wie weiter oben beschrieben, soll die Komponente *Kommunikation* für einen Kommunikationsaufbau zwischen dem Server und den Clients zuständig sein. Die Verbindung wird über Bluetooth realisiert. Mit ihr können ganze Zahlen im Bereich des Java-Typs *int* versendet und empfangen werden.
Die vierte Komponente, *UserInteraction*, ist für Eingaben seitens des Benutzers zuständig. Sie wartet auf Tastatureingaben und wandelt diese in Aktionen, wie zum Beispiel ein LOAD für das Laden einer Karte, für die Komponente *Spielkoordination* um und teilt ihr diese mit. Weiterhin können mit Hilfe der *UserInteraction* beliebige Textausgaben für den Benutzer erzeugt werden.

3.1.2 Komponenten des Clients

Wie die Architektur des Servers, besteht auch die Architektur des Clients aus vier Komponenten (siehe Abbildung 3.2). Die Komponente *Steuerung* soll den Ablauf des Spiels auf der Seite des Clients koordinieren. Mit Hilfe anderer Komponenten baut sie so zum Beispiel eine Verbindung zum Server auf (Komponente */I10/*), aktualisiert die intern geführte Karte mit Wänden, die auf dem aktuellen Feld gemessen werden, und lässt damit den nächsten Zug berechnen (Komponente */I30/*).

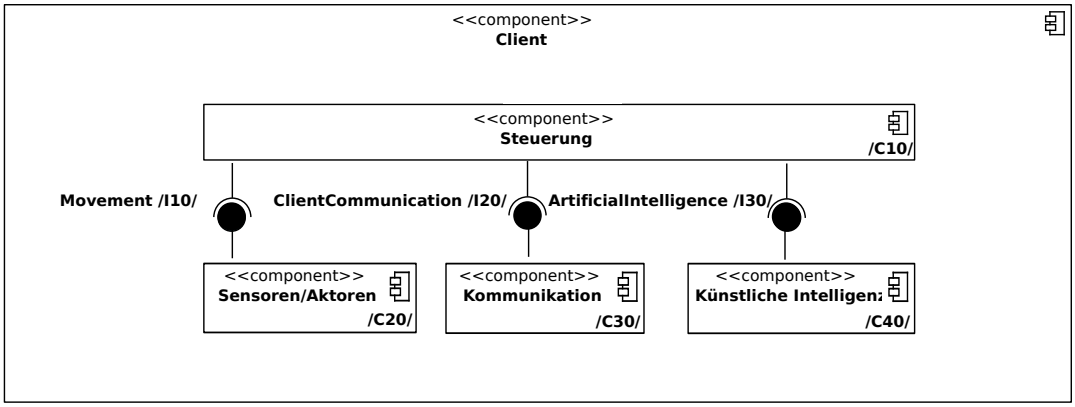

Abbildung 3.2: Komponentendiagramm des Clients

Die Komponente *Sensoren/Aktoren* ist für die Bewegung des Roboters zuständig. Die Kombination von Sensorik und Aktorik hat den Vorteil, dass jede Bewegung über die Aktorik mittels der Sensorik unmittelbar korrigiert werden kann, sodass der Roboter nicht gegen die Wände des Labyrinths fährt. Weiterhin prüft diese Komponente zu welchen Seiten auf dem aktuellen Spielfeld des Roboters sich Wände befinden.
Eine der wichtigsten Komponenten ist die *Künstliche Intelligenz*. Sie berechnet mit der von der *Steuerung* geführten Karte den nächsten Zug im Labyrinth. Sie behandelt die eingelesene Karte als einen Graph und markiert Knoten (Spielfelder) nach bestimmten Kriterien, zum Beispiel, ob an diesem Knoten eine Falltür erwartet wird. Dadurch, dass jede Fehlentscheidung das Spiel beenden oder sogar das Spielfeld durch das Anfahren der Wände beschädigen kann, ist die Korrektheit dieser Komponente von entscheidender Bedeutung.
Die letzte Komponente des Clients ist die *Kommunikation*. Wie schon beim Server sie für den Verbindungsaufbau beziehungsweise -abbau sowie das Versenden und Empfangen von ganzen Zahlen des Java-Typs *int* zuständig sein. Die Verbindung wird über Bluetooth realisiert.
Falls Wunschkriterium */W10/* nicht umgesetzt werden kann, entfällt für den Wumpus die Komponente *Künstliche Intelligenz* und die Komponente *Sensoren/Aktoren* beschränkt sich auf die Funktion Schreien.

3.2 Implementierung von Komponente /C10/: Steuerung

Die Komponente *Steuerung* kontrolliert den Roboter. Sie interagiert dabei mit den drei anderen Komponenten des Roboters über die Schnittstellen */I10/*, */I20/* und */I30/* . Die Klasse *Agent-Movement* gehört zur Komponente *Sensoren/Aktoren* und wird in Kapitel 3.2 erläutert. Die Klasse *Communication* gehört zur Komponente *Kommunikation (Client)* und wird in Kapitel 3.4 erläutert. Die Klasse *AgentPilot* gehört zur Komponente *Künstliche Intelligenz* und wird in Kapitel 3.5 erläutert.

3.2.1 Paketdiagramm

Im folgenden Abschnitt ist das Klassendiagramm zu der Klasse AgentControl abgebildet. Agent-Control implementiert die Komponente *Steuerung*.

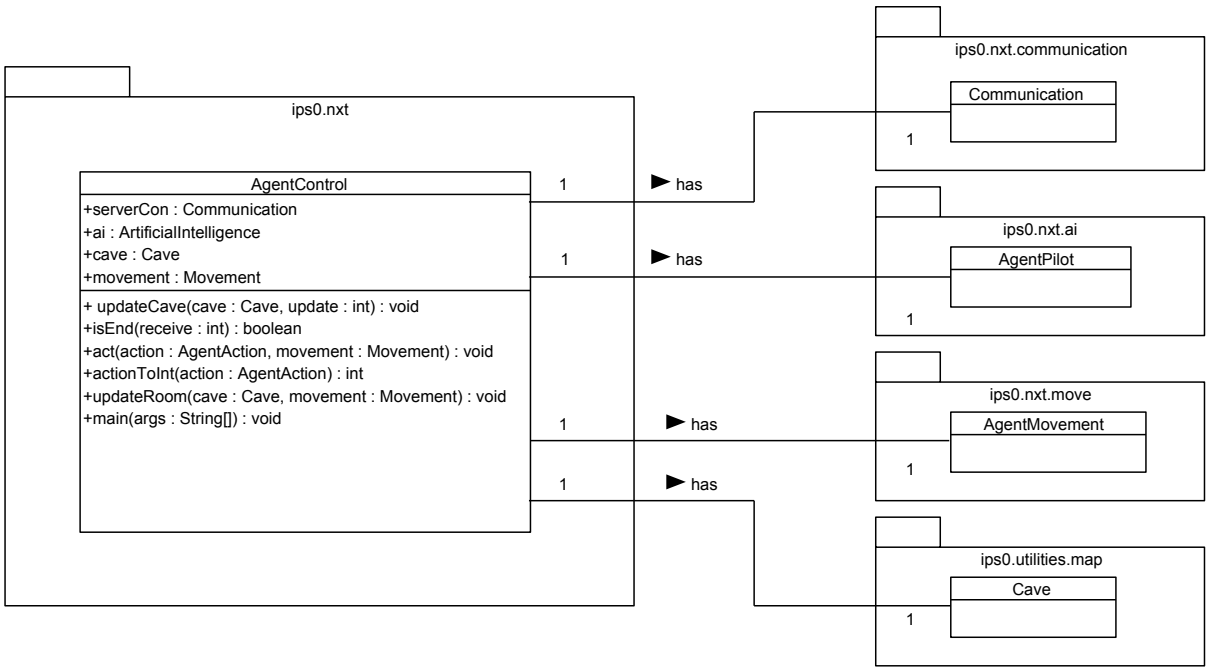

Abbildung 3.3: Paketdiagramm der Steuerung /C10/

3.2.2 Erläuterung

Im folgenden wird die Implementierung der Komponente *Steuerung* genauer erläutert.

Klasse AgentControl:

Die Klasse *AgentControl* gehört zum Paket *ips0.nxt*. Die Klasse *AgentControl* steuert den Roboter.

Attribute:

- *Communication serverCon* ist das Kommunikations-Modul des Agenten.
- *ArtificialIntelligence ai* ist die KI-Komponente des Agenten.
- *Cave cave* speichert die Karte der Höhle.
- *Movement movement* ist die Sensoren/Aktoren-Komponente des Agenten.

Operationen:

- *updateCave(cave : Cave, update : int)* aktualisiert die Höhle.*cave* ist dabei die Höhle, in welcher sich der Agent befindet, und *update* die als Integer kodierte Information über das Feld (z.B. Luftzug etc. ...).
- *isEnd(receive : int)* stellt fest, ob das Spiel noch läuft. Dazu überprüft die Methode *receive*. Diesen Integer hat das Kommunikationsmodul geliefert.
- *act(action : AgentAction, movement : Movement)* lässt den Agenten eine Aktion ausführen. *action* ist hierbei die Aktion, welche der Agent ausführen will. Diese wird dann über *movement* (siehe /C20/) realisiert.
- *actionToInt(action : AgentAction)* übersetzt das erhaltene Enum *action* in einen Integer.
- *updateRoom(cave : Cave, movement : Movement)* aktualisiert den Raum, in dem der Agent sich befindet. *cave* ist die Höhle, in der sich der Agent befindet. Über *movement* (siehe /C20/) lassen sich die Sensoren benutzen, mit denen die Wände erkannt werden.

Kommunikationspartner:

- Die Klasse *ips0.nxt.ai.AgentPilot*,welche den Weg, welchen der Agent nehmen möchte, errechnet.
- Die Klasse *ips0.nxt.communication.Communication*, welche zur Kommunikation mit dem Server dient.
- Die Klasse *ips0.nxt.move.AgentMovement* steuert die Motoren und Sensoren des Roboter des Agenten.
- Die Klasse *ips0.utilities.AgentAction* ist ein Enum und legt Bezeichnungen für die Aktionen des Agenten fest.
- Die Klasse *ips0.utilities.map.Cave* ist die Datenstruktur, die die Karte repräsentiert.
- Die Klasse *ips0.utilities.map.Room* ist die Datenstruktur, die einen Raum in der Karte repräsentiert.
- Die Klasse *ips0.utilities.ServerCommConstants* definiert Konstanten für die Kommunikation zwischen Server und Roboter.

3.3 Implementierung von Komponente /C20/: Sensoren/Aktoren

Im folgenden Abschnitt wird die Implementierung der Komponente *Sensoren/Aktoren* erläutert (siehe Abbildung 3.4). Die Kommunikation der Komponente mit dem Restsystem erfolgt über die vordefinierte Schnittstelle *Movement /I10/*. Dieses Interface wird von der Klasse *Movement* im Paket *ips0.nxt.move* implementiert. Die Hauptaufgabe ist, die Bewegungen des Roboters zu steuern, sowie das Erkennen der Umgebung. In der Klasse *ips0.nxt.move.Movement* wird die Bibliothek *ips0.utilities.map.Direction (siehe Kartentool 3.9)* zur Orientierung im Labyrinth verwendet.

3.3.1 Paketdiagramm

Das folgende Paketdiagramm gibt einen Überblick über die Struktur der Komponente *Sensoren/Aktoren*:

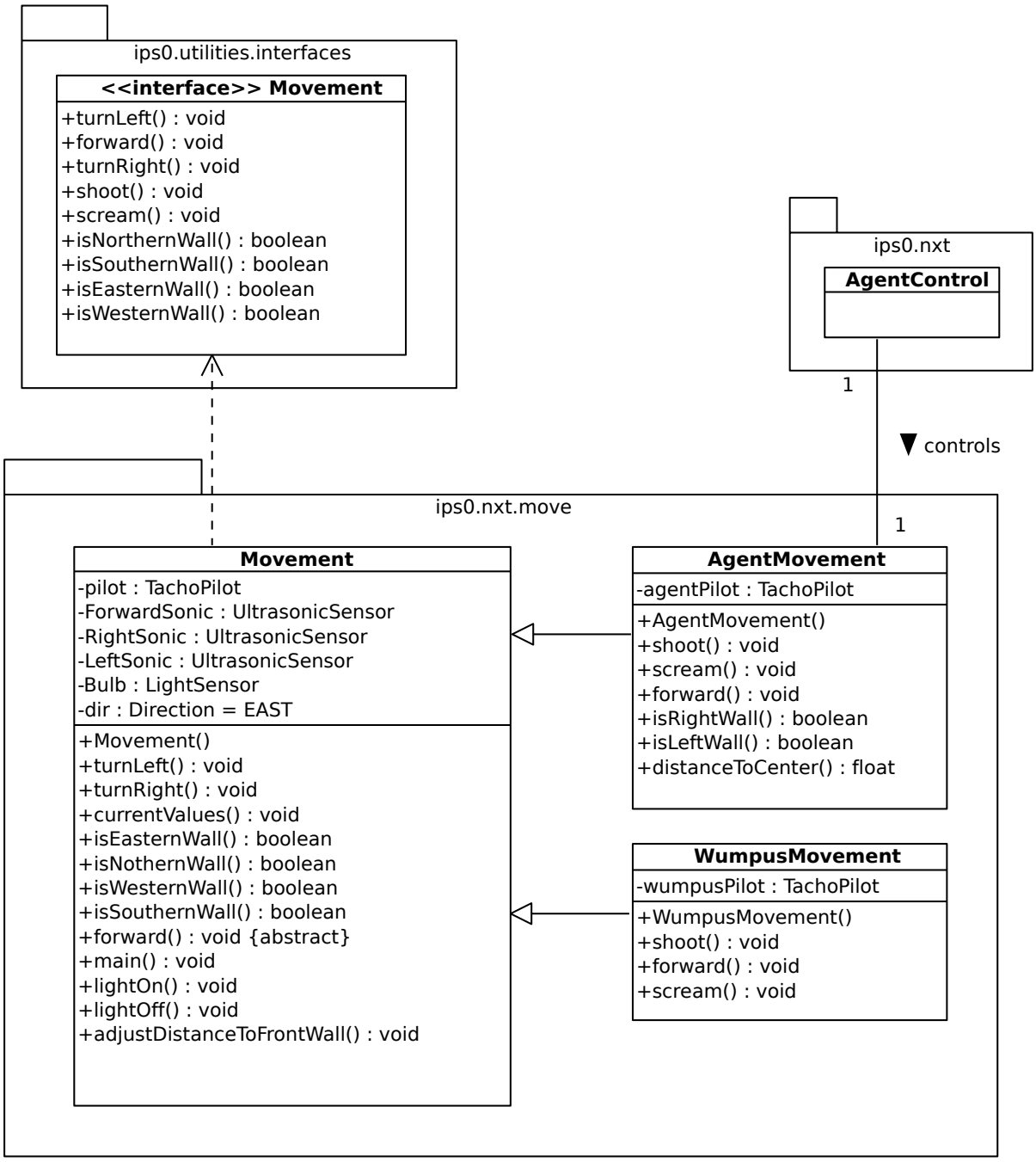

Abbildung 3.4: Paketdiagramm der Sensoren/Aktoren /C20/

3.3.2 Erläuterung

Movement:

Die Klasse *Movement* liegt im Paket *ips0.nxt.move*.

Attribute:

- *TachoPilot pilot* ist der TachoPilot der Klasse.
- *UltrasonicSensor forwardSonic* ist der vordere UltrasonicSensor.
- *UltrasonicSensor rightSonic* ist der rechte UltrasonicSensor.
- *UltrasonicSensor leftSonic* ist der linke UltrasonicSensor.
- *LightSensor bulb* ist der LightSensor. Dieser wird allerdings nicht als LightSensort genutzt, es wird nur die Lampe des Sonsors benutzt, um zu Signalisieren, dass der Roboter eingeschaltet ist.
- *Direction dir* speichert die aktuelle Blickrichtung des Agenten. Zu Anfang blick der Agent immer nach *E*AST.

Operationen:

- *Movement()* ist der Konstruktor der Klasse.
- *turnLeft()* dreht den Roboter nach links.
- *turnRight()* dreht den Roboter nach rechts.
- *currentValues()* gibt die aktuellen UltraSonic Werte aus.
- *isEasternWall()* überprüft ob östlich des Roboters eine Wand ist.
- *isNothernWall()* überprüft ob nördlich des Roboters eine Wand ist.
- *isWesternWall()* überprüft ob westlich des Roboters eine Wand ist.
- *isSouthernWall()* überprüft ob südlich des Roboters eine Wand ist.
- *forward()* lässt den Roboter vorwärts fahren.
- *main()* ist die Main-Methode der Klasse.
- *lightOn()* schaltet die Lampe des LightSensors an.
- *lightOff()* schaltet die Lampe des LightSensors aus.
- *adjustDistanceToFrontWall()* ermittelt den optimalen Abstand zur vorderen Wand.

Kommunikationspartner:

- keine

WumpusMovement:

Die Klasse *WumpusMovement* liegt im Paket *ips0.nxt.move*. Die Klasse ist für die spezielle Sensorik und Aktorik des Wumpus verantwortlich.

Attribute:

- *TachoPilot wumpusPilot* ist der TachoPilot des Wumpus.

Operationen:

- *WumpusMovement()* ist der Konstruktor der Klasse.
- *shoot()* wirft eine UnsupportedOperationException(), da der Wumpus keinen Pfeil abschießen kann.
- *forward()* lässt den Wumpus vorwärts fahren.
- *scream()* lässt den Wumpus schreien.

Kommunikationspartner:

- keine

AgentMovement:

Die Klasse *TachoPilot AgentMovement* liegt im Paket *ips0.nxt.move*. Die Klasse ist für die spezielle Sensorik und Aktorik des Agenten verantwortlich.

Attribute:

- *agentPilot* ist der TachoPilot des Agenten.

Operationen:

- *AgentMovement()* ist der Konstruktor der Klasse.
- *shoot()* schießt den Pfeil des Agenten ab.
- *forward()* lässt den Wumpus vorwärts fahren.
- *scream()* wirft eine UnsupportedOperationException(), da der Agent nicht vom Pfeil getroffen werden kann und somit nicht schreien kann.
- *isRightWall()* überprüft, ob es rechts vom Agenten eine Wand gibt.
- *isLeftWall()* überprüft, ob es links vom Agenten eine Wand gibt.
- *distanceToCenter()* ermittelt die perfekte Position auf dem aktuellen Feld.

Kommunikationspartner:

- Die Klasse *ips0.nxt.ai.AgentControl* kontrolliert den Roboter.

3.4 Implementierung von Komponente /C30/: Kommunikation(Client)

Im folgenden Abschnitt wird die Implementierung der Komponente *Kommunikation(Client)* erläutert(siehe Abbildung 3.5). Die Kommunikation der Komponente mit dem Restsystem erfolgt über die vordefinierte Schnittstelle *AgentCommunication /I20/*. Dieses Interface wird von der Klasse *Communication* im Paket *ips0.nxt.communication* implementiert. Die Hauptaufgabe der Klasse ist die Sicherstellung der Kommunikation zwischen Client und Server. In der Klasse werden die Bibliotheken *java.io.DataInputStream* und *java.io.DataOutputStream* verwendet um die Nachrichten aus dem Stream zu lesen bzw. in den Stream zu schreiben. Die Klassen *lejos.nxt.comm.BTConnection* und *lejos.nxt.comm.Bluetooth* ermöglichen eine streambasierte Kommunikation über Bluetooth mit dem Server.

3.4.1 Paketdiagramm

Das folgende Paketdiagramm gibt einen Überblick über die Struktur der Komponente *Kommunikation (Client)*:

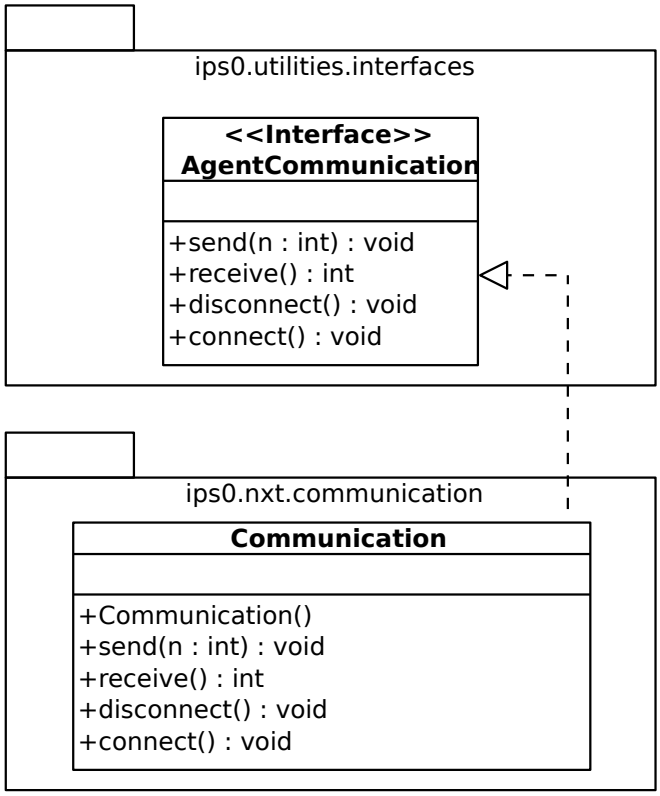

Abbildung 3.5: Paketdiagramm der Kommunikation (Client) /C30/

3.4.2 Erläuterung

Communication:

Die Klasse *Communication* liegt im Paket *ips0.nxt.communication*. Die Sichtbarkeit des Konstruktors ist auf *public* gesetzt. Die Hauptaufgabe der Klasse ist die Sicherstellung der Kommunikation zwischen Client und Server.

Attribute:

- keine

Operationen:

- *send(n : int)*: Diese Operation sendet den übergebenen Integer an den Server.
- *receive()*: Diese Operation empfängt eine Nachricht vom Server.
- *disconnect()*: Diese Operation schließt die Verbindung.
- *connect()*: Diese Operation baut ein Verbindung zum Server auf .

Kommunikationspartner:

- keine

3.5 Implementierung von Komponente /C40/: Künstliche Intelligenz

Im folgenden Abschnitt wird die Implementierung der Komponente *Künstliche Intelligenz* erläutert (siehe Abbildung 3.6). Die Kommunikation der Komponente mit dem Restsystem erfolgt über die vordefinierte Schnittstelle *ArtificialIntelligence /I30/*. Dieses Interface wird von der Klasse *AgentPilot* im Paket *ips0.nxt.ai* implementiert. Die Klasse besitzt genau eine Instanz der Klasse *ips0.nxt.ai.MoveConstants*, der Klasse *ips0.nxt.ai.converter.Converter* und der Klasse *ips0.nxt.ai.bfs.Search*. Diese Klasse besitzt einen bestimmten Zustand, deren Konsistenz durch das Singletonmuster sichergestellt wird. Die Hauptaufgabe ist das Lösen des Labyrinths und die Berechnung der Zügen des Agenten.

In der Klasse *ips0.nxt.ai.AgentPilot* werden folgende Bibliotheken verwendet:

- *ips0.nxt.ai.bfs.Search*: Mit Hilfe dieser Klasse erfolgt die Suche nach kürzesten Wegen im Labyrinth.
- *ips0.utilities.AgentAction*: Das ist ein vordefiniertes Enum, das für die Rückgabe verwendet wird.
- *ips0.utilities.LinkedList*: Das ist die Implementierung von einer Liste (in LeJOS-Bibliothek nicht vorhanden).
- *ips0.utilities.map.Cave*: Das ist die Datenstruktur, die die Karte repräsentiert.
- *ips0.utilities.map.Room*: Das ist die Datenstruktur, die einen Raum in der Karte repräsentiert.

Die Klasse *ips0.nxt.ai.MoveConstants* ist ebenfalls als Singleton implementiert. Hiermit wird der unnötiger Speicherverbrauch verhindert. In der Klasse werden alle mögliche Ketten der Agentaktionen gespeichert.

In der Klasse *ips0.nxt.ai.MoveConstants* werden folgende Bibliotheken verwendet:

- *ips0.utilities.AgentAction*: Das ist ein vordefiniertes Enum, in dem die Agentaktionen spezifiziert werden.
- *ips0.utilities.LinkedList*: Das ist die Implementierung von einer Liste (in LeJOS-Bibliothek nicht vorhanden).

Die Klasse *ips0.nxt.ai.converter.Converter* besitzt die Aufgabe eine Liste von Räumen in eine Liste mit Agentaktionenen umzuwandeln.

In der Klasse *ips0.nxt.ai.converter.Converter* werden folgende Bibliotheken verwendet:

- *ips0.utilities.AgentAction*: Das ist ein vordefiniertes Enum, in dem die Agentaktionen spezifiziert werden.
- *ips0.utilities.LinkedList*: Das ist die Implementierung von einer Liste (in LeJOS-Bibliothek nicht vorhanden).

Die Klasse *ips0.nxt.ai.bfs.Search* hat die Aufgabe die kürzeste Wege zwischen zwei Räumen oder zu einem unbesuchten Raum zu berechnen.

In der Klasse *ips0.nxt.ai.bfs.Search* werden folgende Bibliotheken verwendet:

- *ips0.nxt.ai.converter.Converter*: Eine Instanz dieser Klasse konvertiert den Weg zwischen zwei Räumen in Agentaktionen.
- *ips0.utilities.AgentAction*: Das ist ein vordefiniertes Enum, in dem die Agentaktionen spezifiziert werden.
- *ips0.utilities.ArrayQueue*: Das ist die Implementierung einer Warteschlange.
- *ips0.utilities.LinkedList*: Das ist die Implementierung von einer Liste (in LeJOS-Bibliothek nicht vorhanden).
- *ips0.utilities.map.Cave*: Das ist die Datenstruktur, die die Karte repräsentiert.
- *ips0.utilities.map.Room*: Das ist die Datenstruktur, die ein Raum in der Karte repräsentiert.

Die Klasse *ips0.nxt.ai.bfs.Node* ist ein komplexer Datentyp. Die Objekte dieser Klasse sind Hilfsobjekte für die Suche der Klasse *ips0.nxt.ai.bfs.Search*.

In der Klasse *ips0.nxt.ai.bfs.Node* werden folgende Bibliotheken verwendet:

- *ips0.utilities.LinkedList*: Das ist die Implementierung von einer Liste.
- *ips0.utilities.map.Room*: Das die Datenstruktur, die ein Raum in der Karte repräsentiert.

3.5.1 Paketdiagramm

Das folgende Paketdiagramm gibt einen Überblick über die Struktur der Komponente *Künstliche Intelligenz*:

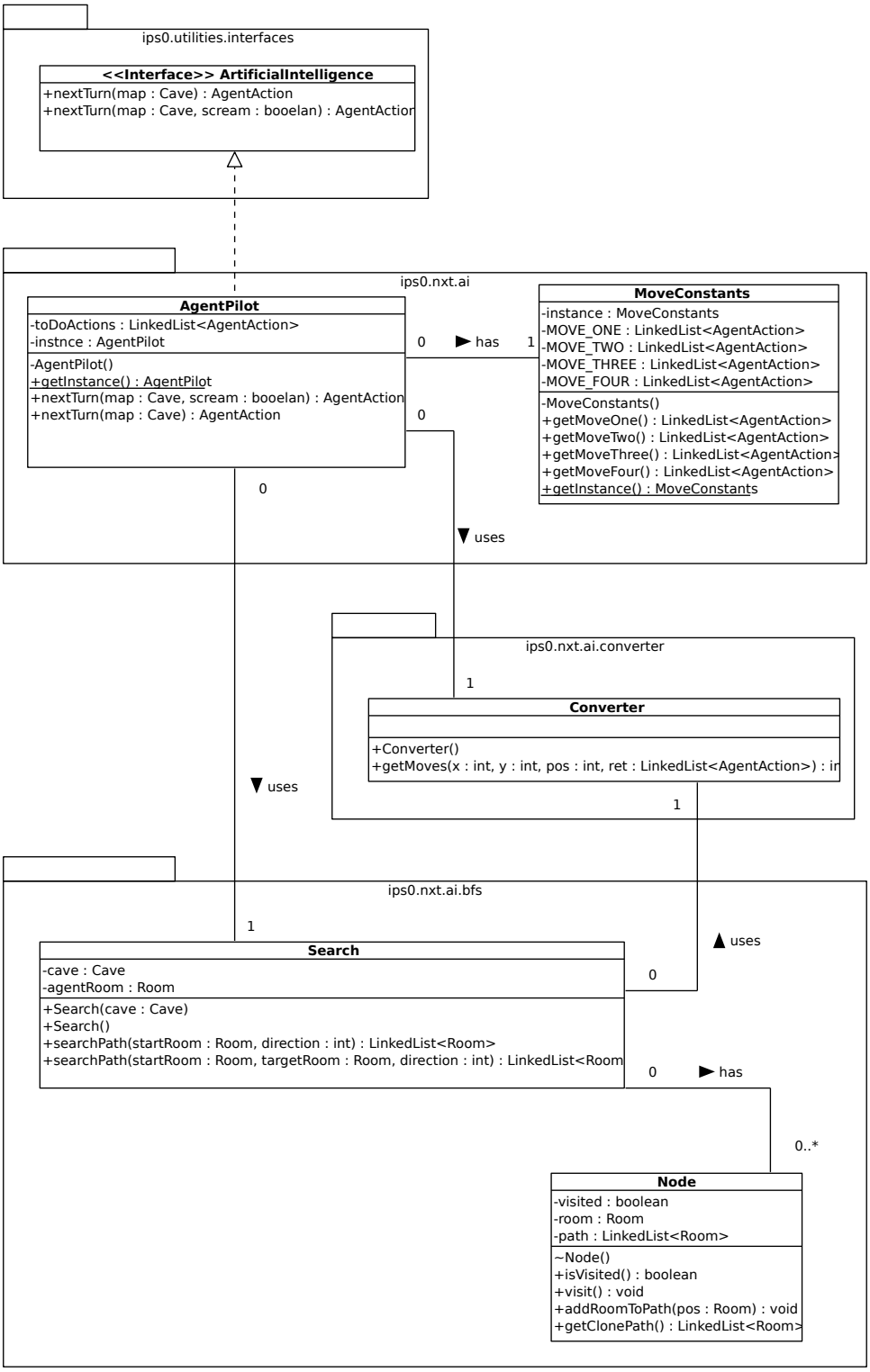

Abbildung 3.6: Paketdiagramm der Künstlichen Intelligenz /C40/

3.5.2 Erläuterung

AgentPilot:

Die Klasse *AgentPilot* steht im Mittelpunkt der Komponente *Künstliche Intelligenz* und liegt im Paket *ips0.nxt.ai*. Die Klasse besitzt einen *private* Konstruktor. Auf eine Instanz dieser Klasse greift man mit der Operation *getInstance()* zu. Die Hauptaufgabe ist das Lösen des Labyrinths und die Berechnung der Zügen des Agenten.

Attribute:

- *LinkedList<AgentAction> toDoActions*: Dies ist eine Liste mit Aktionen (z.B. vorwärts fahren und schießen), die vom Agent mit nächsten Zügen ausgeführt werden.
- *AgentPilot instance*: Das ist ein Objekt der Klasse *AgentPilot* für das Singletonmuster.

Operationen:

- *getInstance()*: Diese Operation liefert eine Instanz dieser Klasse (Singleton).
- *nextTurn(map : Cave, scream : boolean)*: Die Information über die Karte und ob der Wumpus getroffen wurde (Scream), wird übergeben. Die Methode berechnet den nächsten Zug des Agenten.
- *nextTurn(cream : boolean)*: Die Karte wird übergeben. Die Methode berechnet den nächsten Zug des Agenten.

Kommunikationspartner:

- Die Klasse *ips0.nxt.ai.MoveConstants*, in der alle mögliche Ketten der Agentaktionen vor gespeichert sind.
- Die Klasse *ips0.nxt.ai.converter.Converter* wandelt eine Liste von Räumen in eine Liste mit Agentaktionenen um.
- Die Klasse *ips0.nxt.ai.bfs.Search* berechnet die kürzeste Wege zwischen zwei Räumen oder zu einem unbesuchten Raum.
- Die Klasse *ips0.utilities.LinkedList* ist eine eigene Implementierung einer *LinkedList* aus der Java-Standard-Bibliothek, da es diese Klasse in leJOS nicht gibt.
- Die Klasse *ips0.utilities.map.Cave* ist die Datenstruktur, die die Karte repräsentiert.
- Die Klasse *ips0.utilities.map.Room* ist die Datenstruktur, die einen Raum in der Karte repräsentiert.

MoveConstants:

Die Klasse *MoveConstants* liegt im Paket *ips0.nxt.ai*. Die Klasse besitzt einen *private* Konstruktor. Auf eine Instanz dieser Klasse greift man mit der Operation *getInstance()* zu.

Attribute:

- *MoveConstants instance*: Das ist ein Objekt der Klasse *MoveConstants* für das Singletonmuster.
- *LinkedList<AgentAction> MOVE_ONE*: Dieses Attribut ist die erste Kette der Aktionen (vorwärts).
- *LinkedList<AgentAction> MOVE_TWO*: Dieses Attribut ist die zweite Kette der Aktionen (links, vorwärts).
- *LinkedList<AgentAction> MOVE_THREE*: Dieses Attribut ist die dritte Kette der Aktionen (links, links, vorwärts).
- *LinkedList<AgentAction> MOVE_FOUR*: Dieses Attribut ist die vierte Kette der Aktionen (rechts, vorwärts).

Operationen:

- *getInstance()*: Die Operation liefert eine Instanz dieser Klasse(Singleton).
- *getMoveOne()*: Die Operation liefert die erste Liste mit Aktionen.
- *getMoveTwo()*: Die Methode liefert die zweite Liste mit Aktionen.
- *getMoveThree()*: Diese Operation liefert die dritte Liste mit Aktionen.
- *getMoveFour()*: Die Operation liefert die vierte Liste mit Aktionen.

Kommunikationspartner:

- Die Klasse *ips0.utilities.LinkedList* ist eine eigene Implementierung einer *LinkedList* aus der Java-Standard-Bibliothek, da es diese Klasse in leJOS nicht gibt.

Converter:

Die Klasse *Converter* liegt im Paket *ips0.nxt.ai.converter*. Die Klasse besitzt die Aufgabe eine Liste von Räumen in eine Liste mit Agentaktionenen umzuwandeln.

Attribute:

- keine

Operationen:

- *getMoves(x : int, y : int, direction : int, ret : LinkedList<AgentAction>)*: Die Werte x,y sind die Koordinaten des Agenten im Labyrinth. Die Operation fügt die Aktionen in die *ret* Liste ein und als Rückgabe liefert die Methode die neue Richtung des Agenten im Labyrinth.

Kommunikationspartner:

- Die Klasse *ips0.utilities.LinkedList* ist eine eigene Implementierung einer *LinkedList* aus der Java-Standard-Bibliothek, da es diese Klasse in leJOS nicht gibt.

Search:

Die Klasse *Search* liegt im Paket *ips0.nxt.ai.bfs*. Die Hauptaufgabe ist das Berechnen der kürzesten Wege zwischen zwei Räumen oder zu einem unbesuchten Raum. Die Klasse besitzt genau eine Instanz der Klasse *ips0.nxt.ai.converter.Converter*. Für die Suche werden mehrere Instanzen der Klasse *ips0.nxt.ai.bfs.Node* erstellt. Die Klasse besitzt einen Konstruktor ohne Parameter und einen mit einem Parameter der Klasse *ips0.utilities.map.Cave*. Im zweiten Fall wird die Karte übergeben in der die Suche stattfinden soll.

Attribute:

- *Cave cave*: Dieses Attribut ist die Karte auf der nach Wegen gesucht werden soll.
- *Room agentRoom*: Das ist der Raum, auf dem sich der Agent befindet.

Operationen:

- *searchPath(startRoom : Room, direction : int)*: Als Übergabeparameter bekommt die Methode den Startraum und die Richtung des Agenten. Die Methode liefert einen Weg zum nächsten Raum.
- *searchPath(startRoom : Room, targetRoom : Room, direction : int)*: Als Übergabeparameter bekommt die Methode den Startraum, den Zielraum und die Richtung des Agenten. Diese Operation liefert den kürzesten Weg zum Zielraum.

Kommunikationspartner:

- Die Klasse *ips0.nxt.ai.converter.Converter* wandelt eine Liste von Räumen in eine Liste mit Agentaktionenen um.
- Die Klasse *ips0.nxt.ai.bfs.Node* ist ein komplexer Datentyp. Die Objekte dieser Klasse sind Hilfsobjekte für die Suche.
- Die Klasse *ips0.utilities.ArrayQueue* ist eine Implementierung einer Warteschlange.

- Die Klasse *ips0.utilities.LinkedList* ist eine Implementierung von einer Liste (in LeJOS-Bibliothek nicht vorhanden).
- Die Klasse *ips0.utilities.map.Cave* ist die Datenstruktur, die die Karte repräsentiert.
- Die Klasse *ips0.utilities.map.Room* ist die Datenstruktur, die ein Raum in der Karte repräsentiert.

Node:

Die Klasse *Node* liegt im Paket *ips0.nxt.ai.bfs*. Die Objekte dieser Klasse sind Hilfsobjekte für die Suche der Klasse *ips0.nxt.ai.bfs.Search*. Die Sichtbarkeit des Konstruktors ist *default*, somit dient die Klasse als Hilfsdatenstruktur für die Suche.

Attribute:

- *boolean visited*: Dieses Attribut speichert die Information, ob der Knoten schon besucht wurde.
- *Room room*: Dieses Attribut bildet den Knoten auf einen Raum ab.
- *LinkedList<Room> path*: Das ist der Weg von dem Agentenraum zu diesem Knoten.

Operationen:

- *isVisited()*: Diese Operation liefert die Information, ob der Knoten schon besucht wurde.
- *visit()*: Mit dieser Methode wird der Knoten als 'Besucht' markiert.
- *addRoomToPath(pos : Room)*: Die Methode fügt den übergebenen Raum zum Weg hinzu.
- *getClonePath()*: Die Operation liefert die geklonte Liste von *path*.

Kommunikationspartner:

- Die Klasse *ips0.utilities.LinkedList* ist eine eigene Implementierung einer *LinkedList* aus der Java-Standard-Bibliothek, da es diese Klasse in leJOS nicht gibt.
- Die Klasse *ips0.utilities.map.Room* ist die Datenstruktur, die einen Raum in der Karte repräsentiert.

3.6 Implementierung von Komponente /C50/: Spielkoordination

Die Komponente *Spielkoordination* befindet sich auf der Seite des Servers. Sie sorgt für einen korrekten Ablauf der Spielsimulation. Dafür muss sie mit drei weiteren Komponenten des Servers interagieren. Durch geeignete Schnittstellen (siehe Schnittstellen im Grobentwurf) für diese Komponenten ist die Austauschbarkeit und Wartbarkeit der *Spielkoordination* gewährleistet. Sie selbst implementiert die Schnittstelle *ServerInformation* (siehe */I60/*) um neue Befehle seitens der Komponente *UserInterface* entgegenzunehmen und erbt von der Klasse Thread um Parallelität bereitszustellen.

3.6.1 Paketdiagramm

Im Folgenden ist das Klassendiagramm (siehe Abbildung 3.7) zu der Komponente *Spielkoordination* (hier *GameCoordinator* genannt) zu sehen. Die Implementierung der assoziierten Klassen ist den entsprechenden Abschnitten 3.7, 3.8 und 3.9 zu entnehmen.

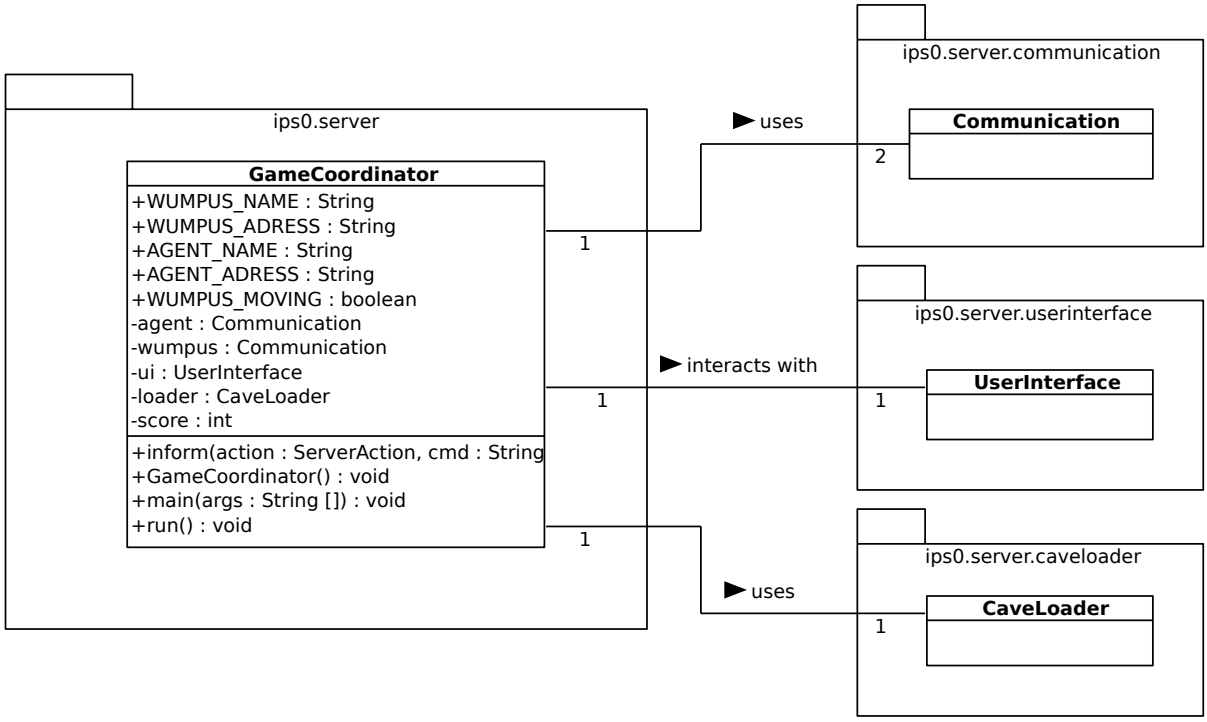

Abbildung 3.7: Paketdiagramm der Spielkoordination /C50/

3.6.2 Erläuterung

Im Folgenden werden die Implementierungsentscheidungen der Komponente *Spielkoordination* näher erläutert. Zuerst wird auf die Attribute und danach auf die Operationen der Klasse (siehe Abbildung 3.7)eingegangen.

Klasse GameCoordinator:

Attribute:

- *String WUMPUS_NAME*: Diese Konstante enthält den Namen des NXT-Roboters, der den Wumpus simuliert.
- *String AGENT_NAME*: Diese Konstante enthält den Namen des NXT-Roboters, der den Agenten simuliert.
- *boolean WUMPUS_MOVING*: Diese Konstante gibt an, ob sich der NXT-Roboter des Wumpus bewegen kann.
- *Communication agent*: Diese Variable dient der Kommunikation des Servers mit dem Agenten.
- *Communication wumpus*: Diese Variable dient der Kommunikation des Servers mit dem Wumpus.
- *UserInterface ui*: Diese Variable ist für die Interaktion mit der Komponente *User Interface* zuständig. Die Interaktion besteht im Wesentlichen aus Ausgaben und dem Empfangen von Eingaben.
- *CaveLoader loader*: Diese Variable ist für die Interaktion mit der Komponente *Kartentool* zuständig. Die Interaktion besteht aus dem Laden einer Karte aus einer XML-Datei.
- *int score*: Diese Variable ist für das Speichern des Spielpunktestandes zuständig.

Operationen:

- *inform(action : ServerAction, cmd : String)*: Diese Operation ist im Interface */I60/* definiert. Mit ihrer Hilfe kann die Klasse *GameCoordinator* Steuerbefehle, wie LOAD für das Laden einer Karte, von der Komponente *User Interface* empfangen.
- *main(args : String[])*: Diese Operation ist für das Starten der Komponente *Spielkoordination* bzw. der Klasse *GameCoordinator* in einer Java VM nötig. Sie erzeugt lediglich eine neue Instanz der Klasse und startet ihren Thread.

- *run()*: Diese Operation ist im Interface der Java Schnittstelle *Runnable* definiert. Ihr Codeinhalt wird beim Starten des Threads ausgeführt. Diese Operation enthält den gesamten Code für den Ablauf des Spiels. Im Groben besteht dieser aus drei Phasen, die jeweils nacheinander ausgeführt werden:
  - **PreGame**: Zuerst wird das UserInterface registriert. Daraufhin werden die Verbindungen zu den Clients Wumpus und Agent aufgebaut und sie erhalten ihre Startinformationen.
  - **Game**: Der Agent erhält Informationen zu seinem aktuellen Feld und dem Spielzustand. Der Server empfängt die Aktion des Agenten und verarbeitet sie für den neuen Spielzustand. Informationen werden über die Komponente *User Interface* ausgegeben. Diese Phase wird so lange wiederholt bis das Spiel zu Ende ist.
  - **PostGame**: Die Resultate des Spiels (Gewinn/Verlust und Punktestand) werden über die Komponente *User Interface* ausgegeben. Danach werden die Verbindungen zu den Clients und die Komponente *Spielkoordination* bzw. der Thread der Klasse *GameCoordinator* beendet.

Kommunikationspartner:

- Die Klasse *ips0.server.communication.Communication* entspricht einem Teil der Komponente */C60/* (siehe Kapitel 3.7).
- Die Klasse *ips0.server.userinterface.UserInterface* entspricht einem Teil der Komponente */C70/* (siehe Kapitel 3.8).
- Die Klasse *ips0.server.caveloader.CaveLoader* entspricht einem Teil der Komponente */C80/* (siehe Kapitel 3.9).

3.7 Implementierung von Komponente /C60/: Kommunikation (Server)

Im folgenden Abschnitt wird die Implementierung der Komponente *Kommunikation(Server)* erläutert (siehe Abbildung 3.8). Die Kommunikation der Komponente mit dem Restsystem erfolgt über die vordefinierte Schnittstelle *ServerCommunication /I40/*. Dieses Interface wird von der Klasse *Communication* im Paket *ips0.server.communication* implementiert. Die Hauptaufgabe der Klasse ist die Sicherstellung der Kommunikation zwischen Server und Client.

3.7.1 Paketdiagramm

Das folgende Paketdiagramm gibt einen Überblick über die Struktur der Komponente *Kommunikation (Server)*:

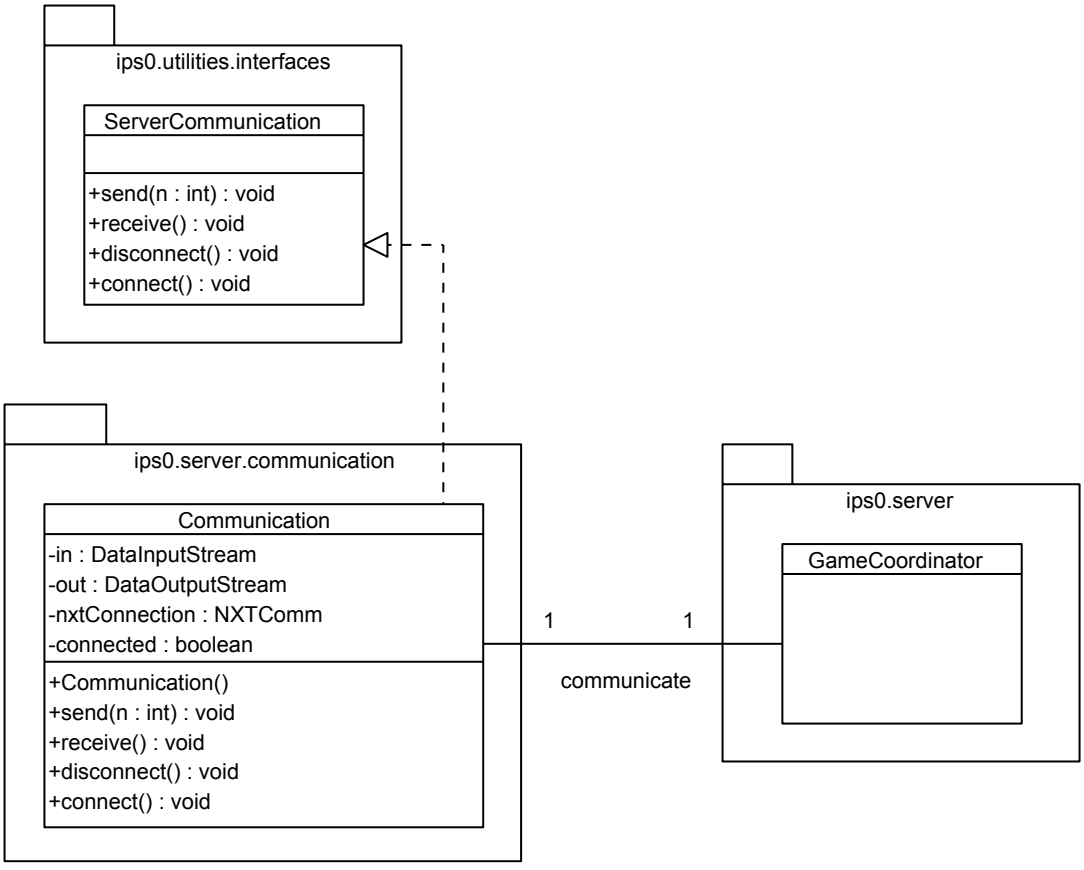

Abbildung 3.8: Paketdiagramm der Kommunikation (Server) /C60/

3.7.2 Erläuterung

Communication:

Die Klasse *Communication* liegt im Paket *ips0.server.communication*.

Attribute:

- *DataInputStream in* ist ein *DataInputStream* vom Agenten.
- *DataOutputStream out* ist ein *DataOutputStream* zum Agenten.
- *NXTComm nxtConnection* ist die Bluetoothverbindung zum Agenten.
- *boolean connected* wird auf *true* gesetzt, wenn die Verbindung zum Agenten aufgebaut ist.

Operationen:

- *Communication()* ist der Konstruktor der Klasse.
- *send(n : int)* sendet einen Integer an den Agenten.
- *receive()* empfängt einen Integer vom Agenten.
- *disconnect()* schließt die Verbindung.
- *connect()* baut eine Verbindung zum Agenten auf .

Kommunikationspartner:

- Die Klasse *ips0.server.GameCoordinator* sorgt für einen korrekten Ablauf der Spielsimulation.

3.8 Implementierung von Komponente /C70/: User Interface

Im folgenden Abschnitt wird die Komponente *UserInterface* erläutert. Diese Komponente ist in der Klasse *UserInterface* verwirklicht. Diese Klasse erbt von der Klasse *java.lang.Thread* und implementiert die Schnittstelle *UserInterface*. Die Schnittstelle *UserInterface* (/I60/) ermöglicht die Kommunikation mit der Komponente *Spielkoordination*.

3.8.1 Paketdiagramm

Das folgende Paketdiagramm gibt einen Überblick über die Struktur der Komponente *User Interface*:

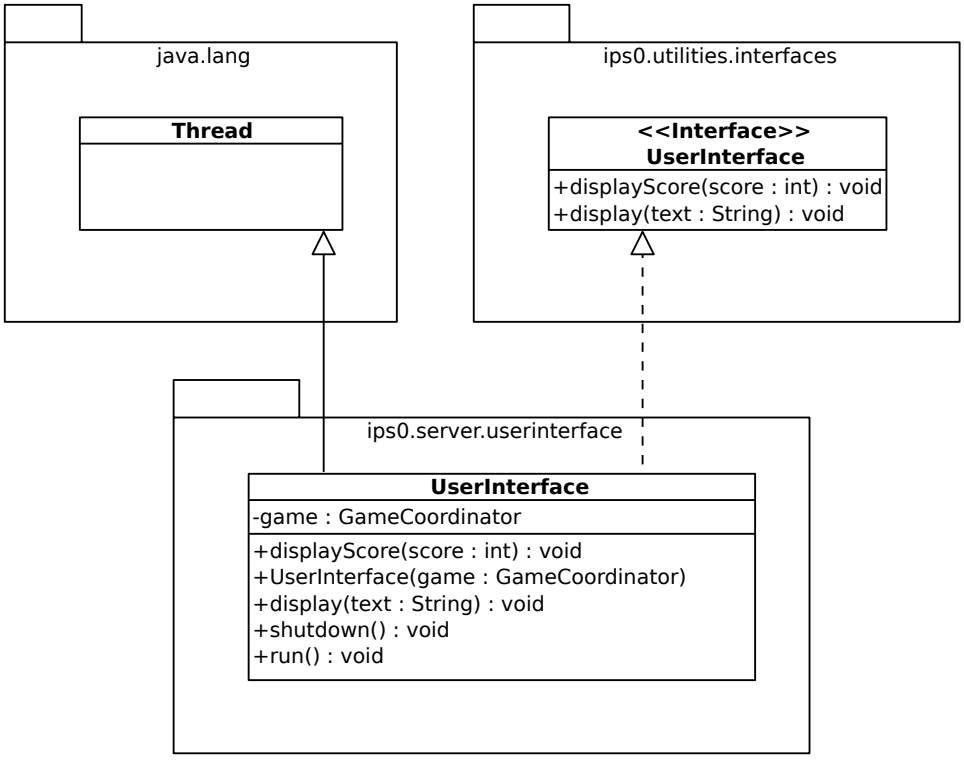

Abbildung 3.9: Paketdiagramm des UserInterface /C70/

3.8.2 Erläuterung

Klasse UserInterface:

Die Klasse *UserInterface* hat die Aufgabe Eingaben vom Benutzer entgegenzunehmen und diese an die Klasse *GameCoordinator* (siehe */C50/* weiterzugeben.

Attribute:

- *GameCoordinator game* ist die aktuelle Instanz von GameCoordinator.

Operationen:

- *displayScore(score : int)* zeigt den übergebenen Punktestand an.
- *display(text : String)* zeigt den übergebenen String an.
- *shutdown()* schließt das UserInterface.

Kommunikationspartner:

- Die Klasse *ips0.server.GameCoordinator*,welche die Komponente Spielkoordination */C50/* repräsentiert.
- Die Klasse *ips0.utilities.ServerAction* legt die möglichen Aktionen des Servers fest.
- Die Klasse *java.io.BufferedReader* dient der erleichterten Benutzereingabe.

3.9 Implementierung von Komponente /C80/: Kartentool

In diesem Abschnitt wird die Implementierung des *Kartentools* (siehe Abbildung 3.10) erläutert. Die Kommunikation mit dem restlichen System findet über die Schnittstelle */I70/ (CaveLoader)* statt. Diese wird von der Klasse *CaveLoader* im Package *ips0.server.caveloader* implementiert. Die Aufgabe der Klasse ist das Laden eines Labyrinths aus einer XML-Datei.

3.9.1 Paketdiagramm

Das folgende Paketdiagramm gibt einen Überblick über die Struktur der Komponente *Kartentool*:

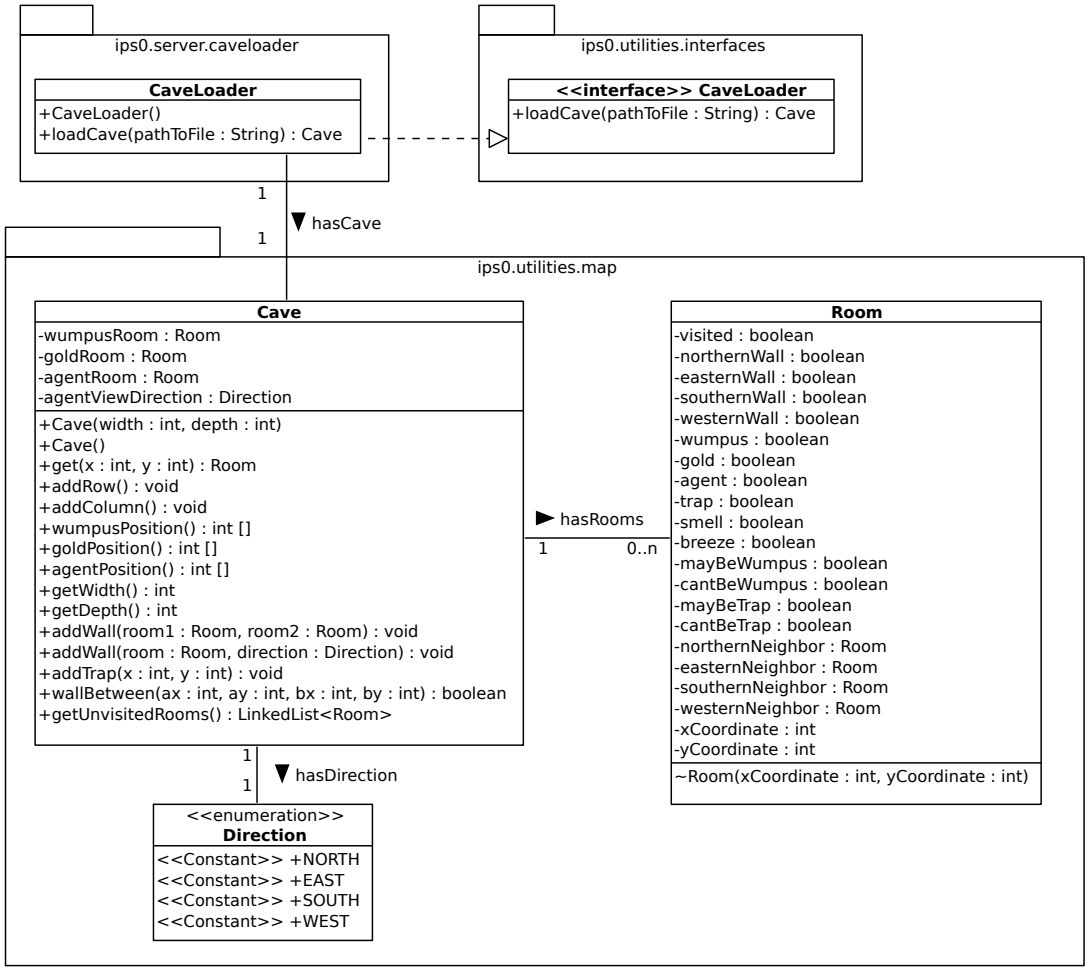

Abbildung 3.10: Paketdiagramm des Kartentools /C80/

3.9.2 Erläuterung

Klasse CaveLoader:

Die Klasse *ips0.server.CaveLoader* steht im Mittelpunkt des Kartentools.
Attribute:

- keine

Operationen:

- *CaveLoader()*: Dies ist der default-Konstruktor.
- *loadCave(pathToFile : String)*: Als Übergabeparameter erhält die Methode den Pfad zur XML-Datei, in der das Labyrinth gespeichert ist. Als Rückgabe erhält der Aufrufer ein Labyrinth-Objekt vom Typ *Cave* (Form des Labyrinths, mit dem das Programm arbeiten kann). Außerdem überprüft die Methode, ob die XML-Datei den Vorgaben entspricht. Desweiteren wird getestet, ob das Labyrinth überhaupt lösbar ist.

Kommunikationspartner:

- *Cave*: Diese Klasse repräsentiert ein Labyrinth.

Klasse Cave:

Die Klasse *Cave*, also der Rückgabetyp der Methode *loadCave()*, befindet sich im Package *ips0.utilities*, da sie auch anderen Stellen noch Verwendung findet. Es sollte zwei Konstruktoren geben. Den ersten Konstruktor ist für den Fall, das die Größe des Labyrinths bekannt ist (die übergebenen Werte stellen die Größe dar). Den zweiten Konstruktor verwendet man, wenn die Größe nicht bekannt ist und das Labyrinth dynamisch erweitert werden soll.
Attribute:

- *Room wumpusRoom*: Dieses Attribut ist ist eine Referenz auf den Raum, in dem sich der Wumpus befindet.
- *Room goldRoom*: Dieses Attribut ist eine Referenz auf den Raum, in dem sich das Gold befindet.
- *Room agentRoom*: Dieses Attribut ist eine Referenz auf den Raum, in dem sich der Agent befindet.
- *Direction agentViewDirection*: Mit diesem Attribut wird die aktuelle Blickrichtung des Agenten gespeichert.

Operationen:

- *Cave()*: Dieser Konstruktor sollte verwendet werden, wenn die Größe des Labyrinths nicht bekannt ist.
- *Cave(width : int, depth : int)*: Dieser Konstruktor sollte verwendet werden, wenn die Größe des Labyrinths bekannt ist.
- *get(x : int, y : int)*: Diese Methode gibt den Raum mit den Koordinaten (x,y) zurück.
- *addRow()*: Diese Methode fügt im Norden des Labyrinths eine Reihe hinzu.
- *addColumn()*: Diese Methode fügt im Osten des Labyrinths eine Spalte hinzu.
- *wumpusPosition()*: Diese Methode liefert ein Array mit der x- und y-Koordinate des Wumpus.
- *agentPosition()*: Diese Methode liefert ein Array mit der x- und y-Koordinate des Agenten.
- *goldPosition()*: Diese Methode liefert ein Array mit der x- und y-Koordinate des Goldes.
- *getWidth()*: Diese Methode gibt die Breite des Labyrinths zurück.
- *getDepth()*: Diese Methode gibt die Tiefe des Labyrinths zurück.
- *addWall(room1 : Room, room2 : Room)*: Diese Methode fügt eine Wand zwischen dem ersten und dem zweiten Raum ein.
- *addWall(room : Room, direction : Direction)*: Diese Methode fügt im Raum eine Wand in der angegebenen Richtung hinzu.
- *addTrap(int x, int y)*: Mit dieser Methode lässt sich im Raum mit den Koordinaten (x,y) eine Falle hinzufügen.
- *wallBetween(ax : int, ay : int, bx : int, by : int)*: Hiermit wird überprüft, ob sich eine Wand zwischen den Räumen mit den Koordinaten (ax,ay) und (bx,by) befindet. Die Räume müssen dafür nicht nebeneinander sein. Es reicht, dass sie entweder in der gleichen Reihe oder in der gleichen Spalte sind.
- *getUnvisitedRooms()*: Diese Methode liefert eine Liste aller Räume, bei denen das Attribut *visited* auf *false* gesetzt ist.

Kommunikationspartner:

- *ips0.utilities.LinkedList*: Dies ist eine eigene Implementierung einer *LinkedList* aus der Java-Standard-Bibliothek, da es diese Klasse in leJOS nicht gibt.
- *Room*: Hiermit werden einzelne Räume des Labyrinths dargestellt.
- *Direction*: Diese Klasse beschreibt Richtungen im Labyrinth.

Klasse Room:

Die Klasse *Room*, die Informationen über die einzelnen Räume einer *Cave* speichert, befindet sich ebenfalls im Package *ips0.utilities.map*. Die Sichtbarkeit des Konstruktors wurde hierbei auf *default* gesetzt, damit Räume nur vom *Cave*-Objekt erzeugt werden können. Dadurch bleibt die Kontrolle des Labyrinths bei eben diesem Objekt.
Attribute:

- *boolean visited*: Hiermit wird gespeichert, ob der Agent den Raum bereits besucht hat.
- *boolean northernWall*: Dies speichert, ob es eine Wand im Norden des Raumes gibt.
- *boolean easternWall*: Dies speichert, ob es eine Wand im Osten des Raumes gibt.
- *boolean southernWall*: Dies speichert, ob es eine Wand im Süden des Raumes gibt.
- *boolean westernWall* :Dies speichert, ob es eine Wand im Westen des Raumes gibt.
- *boolean wumpus*: Mit diesem Attribut wird gespeichert, ob der Wumpus sich in diesem Raum befindet. Die set-Methode hat als Sichtbarkeit *default*, damit dies nur über das *Cave*-Objekt geändert werden kann. Damit werden Inkonsistenzen vermieden.
- *boolean gold*: Hier wird gespeichert, ob das Gold in diesem Raum ist. Die set-Methode hat als Sichtbarkeit *default*, damit dies nur über das *Cave*-Objekt geändert werden kann. Damit werden Inkonsistenzen vermieden.
- *boolean agent*: Dieses Attribut speichert, ob der Agent sich in diesem Raum befindet. Die set-Methode hat als Sichtbarkeit *default*, damit dies nur über das *Cave*-Objekt geändert werden kann. Damit werden Inkonsistenzen vermieden.
- *boolean trap*: Dieses Attribut speichert, ob sich im Raum eine Falle gefindet.
- *boolean smell*: Dieses Attribut speichert, ob es in dem Raum stinkt.
- *boolean breeze*: Mit diesem Attribut wird der Luftzug des Raums gespeichert.
- *boolean mayBeWumpus*: Mit diesem Attribut kann der Agent speichern, ob sich ein Wumpus in dem Raum befinden könnte.
- *boolean cantBeWumpus*: Mit diesem Attribut kann der Agent speichern, dass sich kein Wumpus in dem Raum befinden kann.
- *boolean mayBeTrap*: Mit diesem Attribut kann der Agent speichern, ob sich in dem Raum eine Falle befinden könnte.
- *boolean cantBeTrap*: Mit diesem Attribut kann der Agent speichern, dass in diesem Raum keine Falle sein kann.
- *Room northernNeighbor*: Hiermit wird der nördliche Nachbarraum gespeichert.

- *Room easternNeighbor*: Hiermit wird der östliche Nachbarraum gespeichert.
- *Room southernNeighbor*: Dies speichert den südlichen Nachbarraum.
- *Room westernNeighbor*: Hiermit wird der Nachbar im Westen gespeichert.
- *int xCoordinate*: Dieses Attribut speichert die x-Koordinate des Raums.
- *int yCoordinate*: Mit diesem Attribut wird die y-Koordinate des Raums gespeichert.

Operationen:

- keine (außer get- und set-Methoden für die oben genannten Attribute)

Kommunikationspartner:

- keine

Klasse Direction:

Die Enumeration-Klasse *Direction* befindet sich ebenfalls im Package *ips0.utilities.map*. Mit Hilfe dieser Klasse können die Richtungen *NORTH, EAST, SOUTH* und *WEST* gespeichert werden (z.B. für die Blickrichtung des Agenten).
Attribute:

- keine

Operationen:

- keine

Kommunikationspartner:

- keine

4 Datenmodell

In diesem Kapitel werden die dauerhaft zu speichernden Daten dargestellt und erläutert. Dabei handelt es sich um die Informationen des Labyrinths, damit dasselbe Labyrinth mehrfach gespielt werden kann, ohne dass man es bei jedem Spiel neu eingeben muss. Die entsprechenden Daten werden in einer XML-Datei gespeichert.

4.1 Diagramm

In diesem Abschnitt wird das Datenmodell der XML-Datei mit Hilfe eines Diagramms beschrieben.

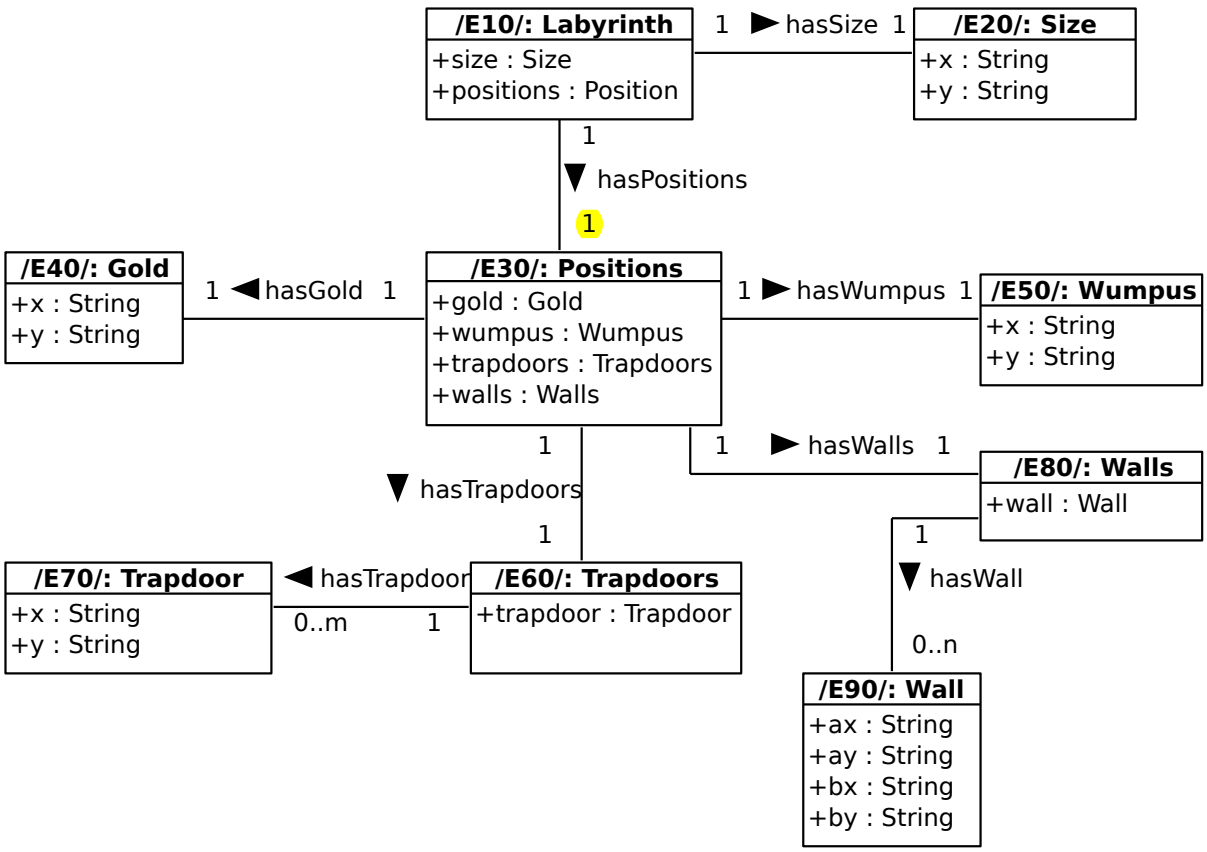

Abbildung 4.1: Datenmodell des Labyrinths

Wie in Abbildung 4.1 zu sehen ist, hat das Labyrinth (*/E10/*) eine feste Größe (*/E20/*). Außerdem müssen die Positionen (*/E30/*) von Gold (*/E40/*), Wumpus (*/E50/*), Falltüren (*/E60/, /E70/*) und Wänden (*/E80/, /E90/*) gespeichert werden.
Das Labyrinth speichert jeweils eine feste Größe (x ist die Breite und y die Tiefe) und es gibt mehrere Positionen. Bei den Positionen werden Gold, Wumpus, Falltüren und Wände gespeichert. Bei Gold, Wumpus und den Falltüren stehen x und y jeweils für die Koordinaten im Labyrinth, wobei das Feld unten links als x- und y-Wert 0 hat. Bei den Wänden stehen ax und ay für die Koordinaten des einen angrenzenden Raumes und bx und by für die Koordinaten des anderen angrenzenden Raumes.

4.2 Erläuterung

Im Folgenden werden die Beziehungen der einzelnen Entitäten aus dem Klassendiagramm (Abbildung 4.1) erläutert.

Entität	**Beziehungen**	
/E10/: Labyrinth	**Name der Beziehung**	**Kardinalität**
	hasSize	1
	hasPositions	1
/E20/: Size	**Name der Beziehung**	**Kardinalität**
	hasSize	1
/E30/: Position	**Name der Beziehung**	**Kardinalität**
	hasPositions	1
	hasGold	1
	hasWumpus	1
	hasTrapdoors	1
	hasWalls	1
/E40/: Gold	**Name der Beziehung**	**Kardinalität**
	hasGold	1
/E50/: Wumpus	**Name der Beziehung**	**Kardinalität**
	hasWumpus	1
/E60/: Trapdoors	**Name der Beziehung**	**Kardinalität**
	hasTrapdoors	1
	hasTrapdoor	0..m
/E70/: Trapdoor	**Name der Beziehung**	**Kardinalität**
	hasTrapdoor	1
/E80/: Walls	**Name der Beziehung**	**Kardinalität**
	hasWalls	1
	hasWall	0..n
/E90/: Wall	**Name der Beziehung**	**Kardinalität**
	hasWall	1

Größe, Gold und Wumpus kann es jeweils nur einmal geben. Falltüren kann es 0 bis m geben, da es nicht unbedingt eine Falle geben muss, aber auch mehrere vorkommen können. Wenn x und y die Breite bzw. die Tiefe des Labyrinths sind, dann gilt $m \leq x * y - 1$ (theoretisch könnten in allen Räumen außer dem Eingang Falltüren sein, falls das Gold im Eingang liegt). Bei den Wänden kann es 0 bis n geben, da die Außenwände nicht mitgezählt werden und es im Labyrinth eine beliebige Anzahl von Wänden geben kann (solange alle Räume erreichbar sind).

5 Serverkonfiguration

Bei dem Server dieses Projekts handelt es sich nicht um einen Server im klassischen Sinne. Vielmehr existiert ein serverähnliches System, welches das Spiel koordiniert und für das Zählen der Punkte zuständig ist. Auf diesem System wird das Programm auf einer Java-VM ausgeführt. Die Kommunikation mit den handelnden NXTs (also Agent und Wumpus) findet über Bluetooth statt. Für die Konfiguration wird eine XML-Datei benötigt, in der der Aufbau des Labyrinths und der enthaltenen Objekte beschrieben sind (siehe Kapitel 4). Folgende Informationen sind in der XML-Datei enthalten:

- Größe des Labyrinths
- Positionen von verschiedenen Objekten
  - Position des Goldes
  - Position des Wumpus
  - Postionen von Falltüren (falls es welche gibt)
  - Positionen von Wänden innerhalb des Labyrinths (falls es welche gibt)

Außer der genannten XML-Datei gibt es keine weiteren Konfigurationsmöglichkeiten.

6 Glossar

Agent - Sucht das versteckte Gold in der Höhle, ohne dabei vom Wumpus verspeist zu werden.

Karte - virtuelle Repräsentation der Wumpus-Welt auf Rechnerseite

LEGO NXT brick - Grundbaustein von Robotern der Firma LEGO.

leJOS - Java Virtual Machine für den LEGO NXT brick.

Raum - Der Raum ist ein Teil des Spielfeldes, in dem die Roboter ihre Aktionen ausführen. Wenn ein Roboter sich geradeaus bewegt, wechselt er den Raum.

Spieler - siehe **Agent**

Spielfeld - mit Holzplatten begrenztes Labyrinth, in dem sich die Roboter bewegen

Wumpus - Monster, das in der Höhle lebt.

XSD-Schema - Ein schematischer Aufbau einer XML-Datei eines bestimmten Typs, an den sich alle Instanzen halten müssen.