Phase	Verantwortlicher	E-Mail
Pflichtenheft	Florian Lorenz	Lorenz@fim.uni-passau.de
Entwurf	Andreas Wilhelm	wilhelm@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

Entwurf

3. November 2010

Inhaltsverzeichnis

1 Einleitung

Dieses Dokument stellt den konzeptionellen Entwurf des e-puck Conquest Systems dar. Dabei handelt es sich um ein verteiltes System mit bis zu sechs e-puck Roboter und einem Android-Smartphone.
Die Roboter sollen ein Spielfeld möglichst zeiteffizient in Kooperation mit den anderen Teilnehmern erkunden. Auf dem Smartphone werden die gesammelten Kartendaten dargestellt. Außerdem kann ein e-puck zur manuellen Steuerung ausgewählt werden.

Die Kommunikation der Roboter wird über e-puck-Proxys auf dem Smartphone abgewickelt.

Dieses Dokument stellt den konzeptionellen Entwurf des e-puck Conquest Systems dar. Hierbei handelt es sich um ein verteiltes System mit bis zu sechs e-puck Roboter und einem Android-Smartphone.
Die Roboter haben die Aufgabe ein Spielfeld möglichst zeiteffizient in Kooperation mit den anderen Teilnehmern zu erkunden. Auf dem Smartphone werden die gesammelten Kartendaten dargestellt, außerdem kann ein e-puck zur manuellen Steuerung ausgewählt werden.

Der Entwurf des Systems wird zur besseren Übersicht in die Bereiche *e-puck Roboter*, *Smartphone*, *abstrakte Logik* und *Kommunikation* aufgeteilt.
Das Ziel ist ein möglichst hohes Maß an Qualität, Wartbarkeit und Erweiterbarkeit. Dazu ist ein sinnvolles Systemdesign unter Verwendung mehrerer verschiedener Entwurfsmuster und Architekturen in allen Bereichen erforderlich.

Weiterhin werden in den folgenden Abschnitten die verwendeten Datentypen und Schnittstellen der Komponenten erläutert. Die beschränkten Ressourcen der Roboter erzwingen hierbei einen möglichst effizienten Aufbau. Insbesondere stellt der interne Arbeitsspeicher sowie die Rechenleistung der e-pucks eine Einschränkung für den Entwurf dar.

2 e-Puck Roboter

2.1 Architektur

Die Software des e-puck wird als Schichtenarchitektur mit wachsendem Abstraktionsgrad entworfen.
Jede Schicht besteht aus einer oder mehreren Komponenten. Diese sind teilweise wiederum aus mehreren Schichten aufgebaut.

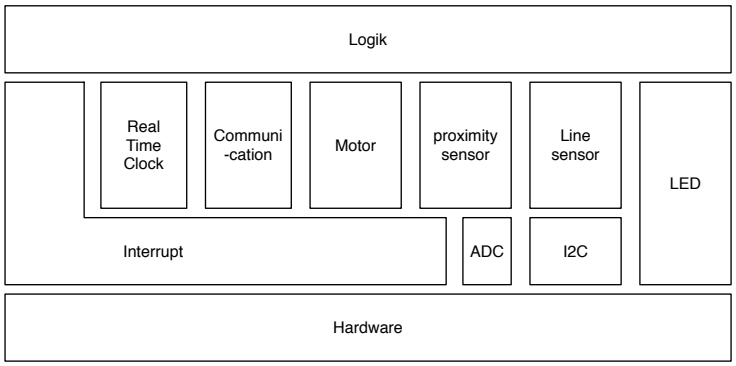


Abbildung 1: Architektur der Software des e-puck

Die folgenden Abschnitte beschreiben die einzelnen Komponenten der Architektur näher.

Komponente 'Interrupt' Die Komponente 'Interrupt' bietet einfache Funktionen zum globalen Ein- und Ausschalten von Interrupts und zur Festlegung von Interrupt-Prioritäten. Außerdem können gesetzte Interrupt-Flags gelöscht werden.

Diese Funktionen werden in den Komponenten 'Timer', 'Motor', 'Communication' und 'IR proximity sensor' sowie in der darüber liegenden Logikschicht verwendet.

Externe Datenstrukturen: keine

Strukturelle Gliederung der Komponente:

`hal_int.h, hal_int.c, hal_int_types.h`

Komponente 'Real-Time-Clock' Die 'Real-Time-Clock'-Komponente stellt eine Echtzeituhr bereit. Diese speichert die aktuelle Laufzeit des e-puck Roboter und löst in regelmäßigen Abständen Interrupts aus. Außerdem können Callbacks für zeitgesteuerte Funktionen anderer Komponenten registriert werden.

Strukturelle Gliederung:

`hal_rtc.h, hal_rtc.c, hal_rtc_types.h`

Externe Datenstrukturen: keine

Komponente 'Communication' Die Komponente 'Communication' stellt einfache Funktionen zum Aufbau und Verwaltung der Ringstruktur des Bluetooth-Netzwerks bereit. Sie enthält außerdem Funktionen zum Senden und Empfangen von Bluetooth-Nachrichten.

Die Modul besteht aus zwei aufeinander aufbauenden Komponenten mit wachsendem Abstraktionsgrad:

hal_uart1 Diese Komponente bildet die hardwarenahste Schicht des Kommunikationsmoduls.
Sie enthält Funktionen zur Initialisierung des UART. Insbesondere werden das Datenformat und die Datenrate festgelegt. Darüber hinaus können die Sende- und Empfangs-Interrupts des UART konfiguriert werden. Die zu den Interrupts gehörigen Interrupt Service Routinen sind ebenfalls Teil der Komponente.
Auch die Kontrolle des Transmitter (Tx)- und Receiver (Rx)-Ringpuffers wird in diesem Modul realisiert.
Das Senden und Empfangen von Nachrichten erfolgt asynchron.

Strukturelle Gliederung:

`hal_uart.h, hal_uart.c, hal_uart1_types.h`

Externe Datenstrukturen:
ringbuf_SContext_t: Zustand des Ringpuffers

```
typedef struct {
    uint8_t* lpui8Storage;
    uint16_t ui16Size;
    volatile uint16_t ui16ReadIndex;
    volatile uint16_t ui16WriteOffset;
} ringbuf_SContext_t;
```

com 'com' enthält Funktionen zur Generierung von Raw-Messages (Nachrichten, die vom Bluetooth-Modul versendet werden können) und zur Aufbereitung von eingehenden Raw-Nachrichten aus der darunter liegenden 'hal_uart'-Schicht.

Strukturelle Gliederung:

`btm.h, btm.c, btm_types.h`

Externe Datenstrukturen:
com_SMessage_t: Format der Raw-Nachrichten

```
typedef struct {
    uint16_t ui16type;
    uint8_t aui8Data[30];
} com_SMessage_t;
```

Komponente 'Motor' Das Modul 'Motor' stellt Funktionen zur Verfügung, die die Voraussetzung für die High-Level-Steuerung der Bewegung des e-puck bilden. Die Komponente initialisiert und kontrolliert die Timer für die Steuerung der beiden Schrittmotoren und gewährleistet deren korrekte Ansteuerung. Außerdem werden Funktionen bereit gestellt, mit denen die Geschwindigkeit der beiden Motoren eingestellt werden kann und deren Schrittzähler gelesen und gesetzt werden können.

Strukturelle Gliederung:

`hal_motor.h, hal_motor.c, hal_motor_types.h`

Komponente 'ADC' Die Komponente 'ADC' kapselt Funktionen zur Initialisierung der Analog-Digital-Wandler und zum einfachen Auslesen der Register mit den konvertierten Werten.
Das Funktionen der Komponente bilden eine wichtige Grundlage für die Verwendung der IR-Abstandssensoren und werden vom Modul 'IR proximity sensor' verwendet.
Strukturelle Gliederung:

`hal_adc.h, hal_adc.c, hal_adc_types.h`

Komponente 'I2C' Die Komponente 'I2C' initialisiert die Datenrate und die Masterfunktionalität des I2C-Moduls. Außerdem werden grundlegenden Übertragungsfunktionen wie Adressierung, Lesen und Schreiben zur Verfügung gestellt.

Strukturelle Gliederung:

`hal_i2c.h, hal_i2c.c`

Komponente 'IR proximity sensor' Es erfüllt im wesentlichen zwei Hauptaufgaben. Zunächst findet eine Initialisierung statt. Hierbei wird das Abtastintervall mit dem die Sensoren ausgelesen werden definiert und der Timer eingerichtet, mit dem das Abtastintervall betrieben wird. Weiterhin stellt dieses Modul die Funktionen zum Auslesen der IR-Sensoren zur Verfügung. Dabei werden Funktionen der 'ADC'-Komponente verwendet.

Strukturelle Gliederung:

`sen_prox.h, sen_prox.c, sen_prox_types.h`

Komponente 'Line sensor' Die Hauptaufgabe dieser logischen Komponente liegt darin, die Daten der IR-Abstandssensoren aus dem I2C-Bus auszulesen um sie später einem Regler zur Verfügung zu stellen, der gewährleistet, dass der jeweilige Roboter seine Linie nicht verlässt.

Strukturelle Gliederung:

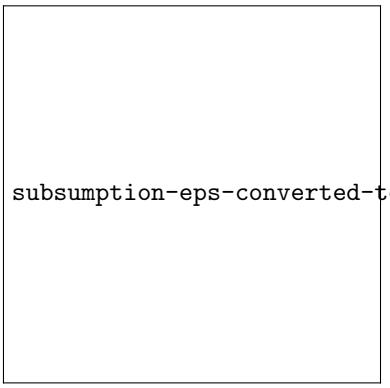
`sen_line.h, sen_line.c, sen_line_types.h`

Komponente 'LED' Zunächst werden sämtliche für die LEDs benötigten Hardware-Register initialisiert. Sobald dies geschehen ist, kann durch diese Komponente die Ansteuerung der LEDs am e-puck übernommen werden.

Strukturelle Gliederung:

`hal_led.h, hal_led.c`

Komponente 'Logik' Diese Komponente des Systems basiert auf allen zuvor genannten Komponenten und wird in einer Subsumption-Architektur umgesetzt. Hierbei handelt es sich um ein nach Prioritäten geordnetes Schichtenmodell, welches in jedem Roboter enthalten in den Dateien subs.c und subs.h enthalten ist. Jede Schicht definiert einen Verhaltensaspekt des jeweiligen e-pucks und besitzt Zugriff auf globale Variablen, welche Sensordaten speichern.
In der folgenden Grafik wird die Anordnung der Schichten nach Priorität in einem Stack veranschaulicht:



Niedrige Schichten beschreiben also die grundlegendsten und wichtigsten Verhaltensweisen eines e-pucks, während höhere Schichten zunehmend abstraktere Verhalten beschreiben und nur dann ausgeführt werden, wenn nicht bereits ein wichtigeres Verhalten ausgeführt wurde. Weitere, komplexere Verhaltensschichten können dadurch problemlos ergänzt werden.
In diesem System wird jeder e-puck einen Verhaltenskodex gemäß obiger Abbildung befolgen.
Je nach Sensordaten kommt ein bestimmtes Verhalten zur Ausführung, wobei höher liegende Verhaltensschichten ignoriert werden bis das angestrebte Ziel erreicht wurde bzw. kritische Daten vom Sensor gemeldet wurden. Anschließend wird eine Liste der registrierten Verhalten erneut von vorne durchlaufen.
Die einzelnen Verhaltensschichten befinden sich gekapselt in einzelnen C-Dateien, so dass sie einfach und schnell ausgetauscht, erweitert und gewartet werden können. Diese Schichten werden im Folgenden näher erläutert.

1. Schicht 1: Abgrund
   Diese Schicht wird aktiv sobald die Bodensensoren einen Abgrund erkennen. Diese kritische Situation muss mit oberster Priorität behandelt werden indem die Motoren sofort gestoppt werden und eine Meldung an das Smartphone erfolgt. Anschließend erwartet der e-puck eine Antwort mit der nächsten Bewegungsanweisung.

`subs_abyss.c, subs_abyss.h`

2. Schicht 2: Kollision
   Diese Schicht der Subsumption-Architektur wird ausgeführt sobald einer der acht IR-Sensoren, die rundherum außen am Roboter angebracht sind einen kritischen Wert liefert. Dann nämlich wurde erkannt, dass eine Kollision unmittelbar bevorsteht. Daraufhin wird eine Kollisionsmeldung an das Smartphone gesendet.

   `subs_collision.c, subs_collision.h`

3. Schicht 3: Knotenanalyse
   Hier werden die Daten der Bodensensoren verarbeitet und in eine History aufgenommen. Anhand dieser History kann entschieden werden ob der Roboter soeben einen Knoten passiert und welche Beschaffenheit dieser besitzt. Falls ein Knoten ermittelt werden konnte wird seine Position und Art an das Smartphone gemeldet und auf die nächste Anweisung gewartet.

   `subs_analyse.c, subs_analyse.h`

4. Schicht 4: Linienverfolgung
   Diese Schicht beinhaltet Algorithmen, welche die Daten der Bodensensoren bezüglich der Linienverfolgung beurteilen. Hierfür werden die Sensordaten in die Berechnungen des PID-Reglers aufgenommen, welcher sich darum kümmert, dass der e-puck die Linie auf der er sich befindet nicht verlässt.

   `subs_pid.c, subs_pid.h`

3 abstrakte Logik

3.1 Problemstellung

Damit die e-pucks ihre Ziele, wie zum Beispiel das Auffinden eines noch nicht erkundeten Knotens oder die Rückkehr zur Startposition, möglichst effizient erreichen können, wird ein Pfadplanungsalgorithmus benötigt. Dieser soll auf Grundlage einer bereits (teilweise) erkundeten Karte, einer Liste von möglichen Zielpunkten und einer Startposition eine Liste von Pfaden zu allen Zielen erstellen. Diese Liste soll nach aufsteigenden Wegkosten sortiert sein.
Außerdem soll die Funktion zur Bewertung der tatsächlichen Knotenkosten austauschbar sein. Dadurch lässt sich der Algorithmus für unterschiedliche Aufgaben verwenden.

3.2 Überblick

A* ist ein informierter Offline-Suchalgorithmus, der einen optimalen Pfad zwischen zwei Knoten in kürzester Laufzeit findet, falls dieser existiert. Alle Knoten werden mit einem der drei folgenden Typen identifiziert, welche jedoch nur in diesem Kontext gültig sind:

- unbekannt: Zu diesem Knoten ist noch kein Pfad bekannt
- bekannt (open): Zu diesem Knoten ist ein (suboptimaler) Pfad bekannt
- erkundet (closed): Zu diesem Knoten ist ein optimaler Pfad bekannt

Jeder erkundete oder bekannte Knoten besitzt eine Referenz auf seinen Vorgänger. Diese Verkettung stellt den Pfad zum Startknoten dar. Für erkundete Knoten ist dies auch der kürzeste Pfad. Außerdem besitzen alle bekannten und erkundeten Knoten einen Wegkostenzähler.
Der Container der bekannten Knoten verwendet einen Binary-Heap. Dieser ermöglicht das effiziente Entfernen des jeweils günstigsten als nächstes zu erkundenden Knotens. Der Schlüssel setzt sich dabei aus den aktuellen Wegkosten zum Start und einer Schätzung der Wegkosten zum aktuellen Ziel zusammen.
Der Container der erkundeten Knoten ist ein Binärbaum, um Suchoperationen zu beschleunigen.

3.3 Wegkosten und Heuristik

Der Algorithmus verwendet eine Bewertungsfunktion, um eine Aussage über die möglichen Gesamtwegkosten eines Knoten treffen zu können. Diese werden durch die Summe aus den tatsächlichen Wegkosten des Vorgängerknotens, den Übergangskosten zum aktuellen Knoten und einer Schätzung der verbleibenden Kosten bis zum Ziel errechnet.
Die Funktion zur Berechnung der Übergangskosten *calculateCosts()* ist austauschbar, wodurch sich das Verhalten des Algorithmus trimmen lässt. Es ist jedoch zu beachten, dass die tatsächlichen Kosten nie den Schätzwert unterschreiten dürfen. Damit z.B. eine Kollision aktiv vermieden wird, müssen die Kosten eines blockierten Knotens höher sein als die Kosten des kürzesten lokalen Umwegs.

Als Schätzfunktion (Heuristik) wird hier die Absolutsummennorm verwendet. Mit Hilfe der Heuristik wird eine zielgerichtete Knotenexpansion erreicht, da nur ein bekannter Knoten mit den geringsten Gesamtkosten als nächstes analysiert wird.

3.4 Datenstrukturen

Benötigt wird eine Knotenklasse *AStarNode*, die eine Referenz auf einen Vorgängerknoten vom selben Typ sowie eine Referenz auf einen Graphknoten *GraphNode* der Karte enthält. Außerdem enthält sie einen Integer, der die aktuellen Kosten bis zum Startknoten angibt.

3.5 Interface

Es wird das Interface *IPathFinder* von der Klasse *AStarPathFinder* implementiert:

- *AStarNode[] find(GraphNode start, GraphNode[] destinations)* führt eine Suche durch und gibt die kürzesten Pfade zurück
- *int calculateCosts(GraphNode from, GraphNode to)* berechnet die Übergangskosten von einem Knoten zu einem angrenzenden Knoten.

3.6 Ablauf

Der Algorithmus basiert auf der Version von Wikipedia [1], jedoch mit einer Erweiterung für die Suche mehrerer Zielknoten. Dazu wird nach jeder erfolgreichen Suche eines Zielknotens der Container der bekannten Knoten umsortiert, damit jeweils der günstigste Knoten für den nächsten Zielknoten extrahiert werden kann. Durch diese Modifikation muss die Berechnung nicht für jeden Zielknoten neu durchgeführt werden, da auf bereits berechnete Pfadabschnitte zurückgegriffen werden kann.

[1] Wikipedia A* Algorithmus

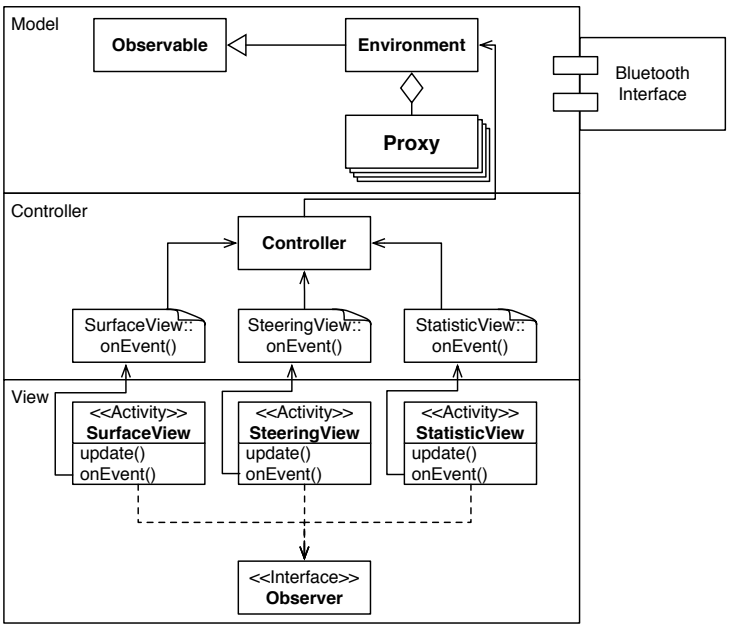


Abbildung 2: Model-View Controller Architektur

4 Smartphone

4.1 Model-View-Controller (MVC) Architektur

Die Daten- bzw. Benutzer-Interaktion der Android-Anwendung wird mit einer MVC-Architektur aufgebaut. Dies ermöglicht ein flexibles Programmdesign, das eine Wiederverwendbarkeit von Komponenten sowie eine reduzierte Komplexität gewährleistet. Die Daten, wie z.B. Karteninformationen oder Roboterpositionen, können dadurch von den zugehörigen Darstellungen getrennt werden (*Separation of Concerns*). Es werden hierbei 3 Views (Activities) verwendet, je ein Dialog für Kartenansicht, Steuerung und Statistik. Bei diesem Entwurf hat die View-Schicht keinen direkten Zugriff auf das Model, dieser wird vollständg vom Controller übernommen. Gemäß den Entwicklerrichtlinien für Android-Anwendungen [2] gehören die Activity-Klassen der View logisch zum Controller. Die Model-Schicht erhält über die Bluetooth-Schnittstelle Aktualisierungen der e-puck Roboter, welche den Zustand ändern.

Model Die Model-Schicht der Architektur speichert als zentrale Komponente sämtliche Daten bzgl. Karten- und e-puck-Informationen. Es beinhaltet außerdem die Applikationslogik der Android-Anwendung. Die Kommunikation nach außen findet über das angeschlossene Bluetooth-Interface statt. Die Klasse *Environment* erbt von der Klasse *Observable* und verarbeitet die Aktualisierungen der Roboter. Die Klasse wird als Singleton realisiert, da hier zentral der globale Zustand der Kartendaten gespeichert wird. Informationen der einzelnen

[2] `http://developer.android.com/guide/topics/ui/index.html`

Roboter werden in den Instanzen der Klasse *Proxy* verwaltet (siehe Kapitel ...). Diese Insanzen werden in einer Attributsliste der Klasse *Environment* gehalten. Bei Zustandsänderung der *Proxy*-Instanzen bzw. bei Änderung der Kartendaten werden die registrierten Views benachrichtigt.

View Die Präsentationskomponenten der MVC-Architektur sind für die Ausgabe der Model-Daten zuständig und bilden eine Abstraktionsschicht zwischen der Präsentation der Anwendung, dem Model und dem Benutzer. Die Schicht besteht aus den drei Klassen *SurfaceView*, *SteeringView* und *StatisticView*. *SurfaceView* ist für die Kartendarstellung verantwortlich, *SteeringView* stellt die Steuerungsbedienelemente dar und *StatisticView* beinhaltet statistische Informationen. Damit die View-Klassen als *Activities* für die Android-Applikation verwendet werden können, müssen sie das Interface *Activity* implementieren. Für die Verwendung als View der MVC-Architektur muss zusätzlich das Interface *Observer* implementiert werden. Jede einzelne View registriert sich bei der Klasse *Environment* als Observer, um bei Zustandsänderungen vom Model benachrichtigt zu werden. Benutzereingaben auf den Dialogen werden über die Ereignisbehandler-Methoden des *Activity*-Interface an das Model weitergegeben. Somit gehören diese Handler-Methoden im Sinne der MVC-Architektur der Controller-Schicht an.

Controller Der Controller nimmt Eingaben aus den verschiedenen View Klassen entgegen und leitet diese bereinigt und normalisiert and die Model-Schicht weiter. Hier wird also eine Abstraktionsschicht eingeführt, welche die Verbindung zwischen Benutzer-Interaktionen und dem Model beschreibt. Zur Controller-Schicht gehört neben den Ereignisbehandler-Methoden der View Klassen die Klasse *Controller*, welche für die zentrale Steuerung der Model Schicht verantwortlich ist.

4.2 Nachrichtenbehandlung

Wir verwenden das Chain-of-Responsibility-Pattern in unserem Projekt, um Nachrichten, die von den e-puck Robotern an das Handy geschickt werden zu analysieren und weiter zu verarbeiten. Die Grundidee hinter diesem Pattern ist, dass die Nachricht durch eine Liste von konkreten Handlerklassen, die von einer abstrakten Handlerklasse erben, gereicht wird und der richtige Handler die Nachricht verarbeitet. Sobald eine Nachricht von einem Handler erkannt und bearbeitet wurde gibt er true zurück. Falls sich der Handler nicht verantwortlich für die Nachricht fühlt gibt er sie an den nächsten Handler weiter. Falls auch der letzte Handler mit der Nachricht nicht umzugehen weiß, gibt er den Rückgabewert false zurück.Im Gegensatz zu herkömmlichen Nachrichtenbearbeitungen wird hier ein hohes Maß an Flexibilität erreicht und es das Hinzufügen eines neuen Nachrichtentyps wird erleichtert, da nur eine neue Klasse in die Liste der Handler hinzugefügt werden muss.

Vorteile:

- Der Absender braucht sich nicht zu kümmern, wer genau die Nachricht verarbeitet (implizieter Empfänger)

- Unabhängigkeit zwischen Sender und Empfänger (Entkopplung)
- Sehr gut erweiterbar, falls ein neuer Nachrichtentyp hinzugefügt werden soll
- Mehrere Klassen kümmern sich um die Verarbeitung, d.h. die Fehlersuche und -behandlung wird vereinfacht
- Größere Flexibilität bei der Bearbeitung von Nachrichten
- Mithilfe des Rückgabewerts wird erkannt, ob eine Nachricht behandelt wurde

Teilnehmer:

- Handler
  Definiert ein Interface für die Requests
  Besitzt Zeiger auf nächstes Listenelement
- Konkrete Handler
  Größere Flexibilität bei der Bearbeitung von Nachrichten
  Mithilfe des Rückgabewerts wird erkannt, ob eine Nachricht behandelt wurde
- Client
  Ruft Funktion handlerequest(Nachricht) auf ersten Listenelement der Handler auf

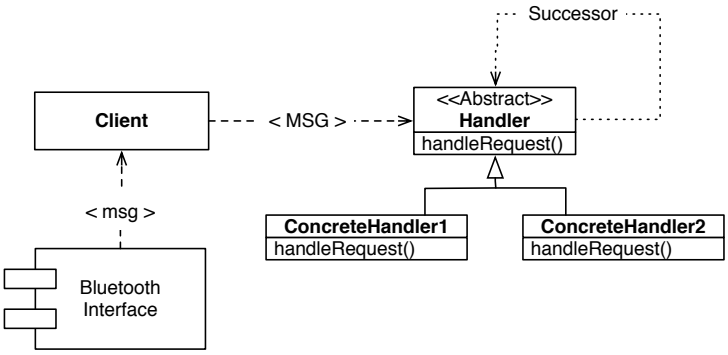


Abbildung 3: Chain of Responsibility - Entwurfsmuster

Erklärung der Abbildung:

1. Die Bluetoothschnittstelle leitet eine einkommende Nachricht weiter an eine Client-Klasse, die eine Liste aus konkreten Handlern, die von der abstrakten Klasse Handler erben, enthält.
2. Diese Klasse gibt die Nachricht an den ersten Handler seiner Liste weiter und ruft dort die Funktion handlerequest(Nachricht), die den Rückgabetyp Boolean hat, auf.

3. Der Handler leitet die Nachricht an seinen direkten Nachfolger weiter und ruft dort wieder handlerequest(Nachricht) auf, sofern er nicht für die Bearbeitung der Nachricht zuständig ist. In diesem Fall behandelt er die Nachricht entsprechend und gibt true zurück

4. Der Handler, der keinen Nachfolger mehr hat gibt den Wert false zurück und somit weiß der Client dass für die Nachricht kein entsprechender Handler zur Verfügung steht.

4.3 Repräsentation der e-puck Roboter

Der Zugriff auf einzelne e-pucks aus der Android Applikation heraus erfolgt durch Einsatz des Proxy Patterns. Dieses dient als Schnittstelle zwischen der Programmlogik und dem Bluetooth Interface. Die Proxy Klasse wird als abstrakte Oberklasse implementiert,wobei jede Instanz dieser Klasse, genau einen e-puck repräsentiert. Diese Stellvertreterobjekte speichern die Zustände der zugehörigen Roboter und beinhalten alle Funktionen um Daten des Roboters abzurufen oder zu setzen. Durch das Einsparen von Logik im Bluetooth Interface

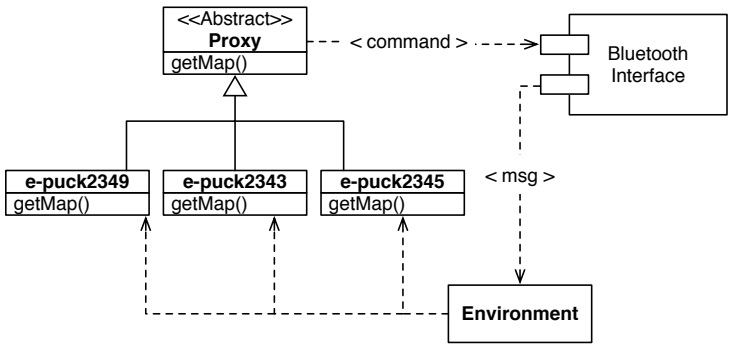


Abbildung 4: Proxy - Entwurfsmuster

und Auslagern in die Proxy Klasse ist die Austauschbarkeit der Robotermodelle gegeben und man bleibt flexibel für die Einführung neuer Befehle.

Beim Entdecken eines neuen, noch unbekannten e-pucks durch das Bluetooth Interface wird für diesen eine neue Instanz der Proxy Klasse angelegt. Die Kommunikation mit diesem Roboter ist ausschließlich über dieses Objekt möglich.

Das Erkunden von neuen Knoten wird vom e-puck zunächst an das Bluetooth Interface weitergegeben und die Informationen anschließend in der Environment Klasse verarbeitet. Die Zustände der einzelnen Roboter, wie zum Beispiel die aktuelle Position, wird aber an die entsprechenden Proxy Instanzen vermittelt und gespeichert.

Der andere Weg führt von der Benutzereingabe über das Environment, welches je nach Befehl die richtige Funktion am Objekt des zugehörigen e-pucks aufruft. Von dort werden Befehle über das Bluetooth Interface an den Roboter gesendet.

4.4 Erzeugung der Views

Ein Ziel der Anwendungs-Architektur ist die Möglichkeit, den Austausch von View-Elementen in einfacher Weise zu ermöglichen. Für diesen Zweck wird das Abstract-Factory Entwurfsmuster zur Erzeugung einer Gruppe von Steuerelementen verwendet. Die Art sowie das Aussehen der Benutzeroberfläche ist damit von der Logik getrennt und neue Darstellungen können einfach hinzugefügt werden. Die Klasse *Factory* dient als Muster zur Erzeugung ganzer "Famili-

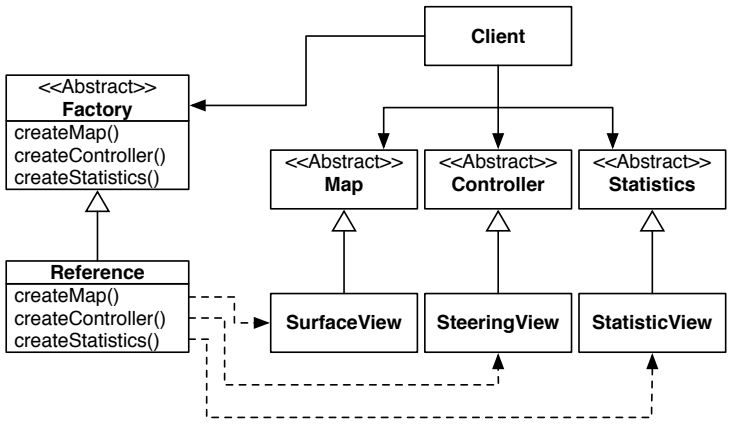


Abbildung 5: Abstract Factory - Entwurfsmuster

en"von Views. Um in dem Fall der Android-Anwendung die konkreten Dialoge für Karte, Steuerung und Statistik zu erstellen, existiert die Klasse *Reference*, welche von *Factory* erbt. Diese sorgt wiederum dafür, dass die konkreten Acitvity-Klassen *SurfaceView*, *SteeringView* und *StatisticsView* der Android - Applikation korrekt erzeugt werden. Um die Repräsentation der Dialoge für den Benutzer einfach austauschbar zu gestalten, erben diese Klassen von entsprechenden abstrakten Oberklassen. Der Client hat somit lediglich die Aufgabe, eine Instanz einer konkreten Factory-Klasse zu erzeugen, der restliche Ablauf kann für jede beliebige Familie von Dialogen gleich erfolgen.

Die Verwendung des Abstract Factory - Entwurfsmusters hat hier den Vorteil, dass konkrete Dialog-Klassen isoliert behandelt werden können. Außerdem macht es den Austausch ganzer Familien einfach und fördert Konsistenz über gesamte Dialog-Familien. Der entscheidende Punkt für den sinnvollen Einsatz, liegt bei dem Entwurf der abstrakten Activity-Klassen *Map*, *Controller* und *Statistics*. Zum Einen wird die Erweiterung der entsprechenden Schnittstellen aufwändig, sobald mehrere konkrete Ableitungen existieren. Zusätzlich müssen im Vorhinein sämtliche benötigten Methoden in diese Klassen aufgenommen werden, um die vollständige Funktionalität gewährleisten zu können.

5 Kommunikation