„ ()

- Patientendialogsystem (PDS) -“

Autor:

QS-Freigabe: tt.mm.jjjj

# Überblick

## Zweck

Das vorliegende Dokument dokumentiert die grundsätzlichen Designentscheidungen und beschreibt die Dekomposition des zu erstellenden SW-Produkts statisch in SW-Komponenten und ggf. dynamisch in Prozessen. Die Zusammenhänge zwischen SW-Komponenten und Prozessen werden dargestellt. Ferner werden die externen (zur Aussenwelt) und die internen Schnittstellen (zwischen den SW-Komponenten untereinander) identifiziert.

## Inhaltsverzeichnis

[0 Überblick 2](#_Toc532738472)

[0.1 Zweck 2](#_Toc532738473)

[0.2 Inhaltsverzeichnis 3](#_Toc532738474)

[1 Grundsatzentscheidungen 4](#_Toc532738475)

[1.1 Entwicklungsumgebung 4](#_Toc532738476)

[1.2 GUI 4](#_Toc532738477)

[1.3 Datenhaltung 4](#_Toc532738478)

[1.4 Netzwerkverteilung 4](#_Toc532738479)

[2 Grobstruktur 6](#_Toc532738480)

[3 Paketbeschreibungen 7](#_Toc532738481)

[3.1 Fachkonzept 7](#_Toc532738482)

[3.1.1 SCRS-Subsystem 7](#_Toc532738483)

[3.1.2 Roboter-Subsystem 7](#_Toc532738484)

[3.1.3 Dialog-Subsystem 8](#_Toc532738485)

[3.2 UI 9](#_Toc532738486)

[3.3 Datenhaltung 9](#_Toc532738487)

[4 Feinspezifikation (interne Dokumenation) 9](#_Toc532738488)

[4.1 Klassendiagramm 9](#_Toc532738489)

[4.2 Modulspezifikationen 9](#_Toc532738490)

# Grundsatzentscheidungen

## Entwicklungsumgebung

* Ubuntu 16.04 LTS
* Programmiersprache C++
* Qt 5.11.2
* ROS Kinetic
* IDE QtCreator unter Verwendung von CMake oder catkin
* Eclipse Paho MQTT C Client 1.3.0 (unterstützt MQTT 3.1.1)

## GUI

Die hier erstellte erste Version des PDS wird nicht über ein grafisches User Interface (GUI) sondern über ein textuelles User Interface (TUI) für die Schnittstelle zum Patienten verfügen. Die Entwicklungsumgebung eröffnet jedoch die Möglichkeit, ein GUI unter Verwednung von Qt zu entwickeln. Die SW-Architektur wird so ausgelegt, das ein Austausch des TUI durch ein GUI ohne bzw. nur mit marginalen Änderungen an den Fachkonzept-Komponenten möglich ist.

Die Nutzerschnittstelle zum Administrator wird durch die Standard-Web-Oberfläche des verwendeten Datenbankmanagemeentsystems (DBMS) zur Verfügung gestellt (siehe 1.3).

## Datenhaltung

Zur Verwaltung der Patienten- und Rauminformationen wird das DBMS MySQL verwendet.

## Netzwerkverteilung

Eine Verfeinerung des Systemkontextdiagramms aus dem Pflichtenheft führt zu einer Aufteilung des PDS in die Komponenten (siehe Abbildung 1)

* Smart Connected RFID Sensor (SCRS)
* Patientendialog (PD) und
* Krankenhaus- und Gebäudeadminstration (KGA)

Das SCRS wird als fertige Komponente und als eigenständiges Hardwaresystem in das Gesamtsystem über das MQTT-Protokoll integriert.

Für die Komponente KGA wird ebenfalls nichts eigenes entwickelt. Hierfür werden die entsprechenden Frondends des DBMS für die Administration der Datenbanken verwendet. Das DBMS, die Datenbank und sowie die Frontends werden auf den jetzigen Linux-PC für die Ausführung der Roboter-Software installiert.

Die neu zu entwickelnde Komponente PD wird auf dem gleichen Linux-Rechner wie das DBMS installiert.

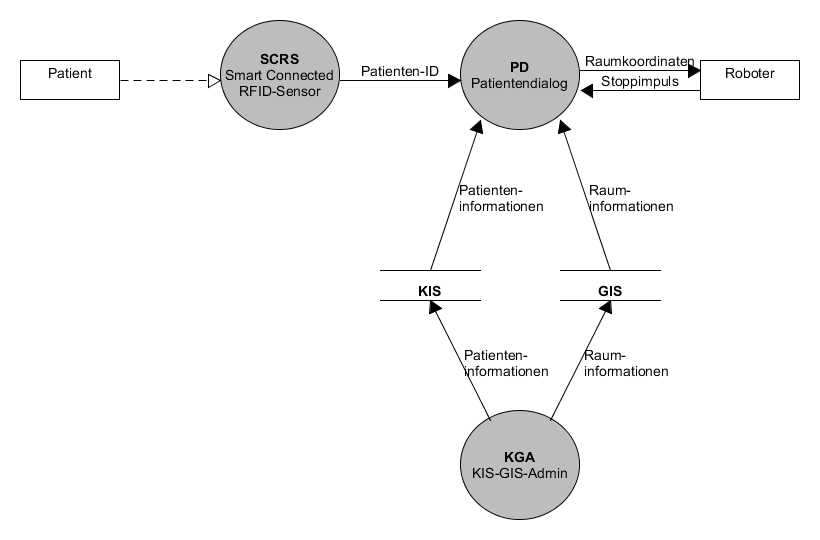


Abbildung 1: Datenflussdiagramm (DFD 1)

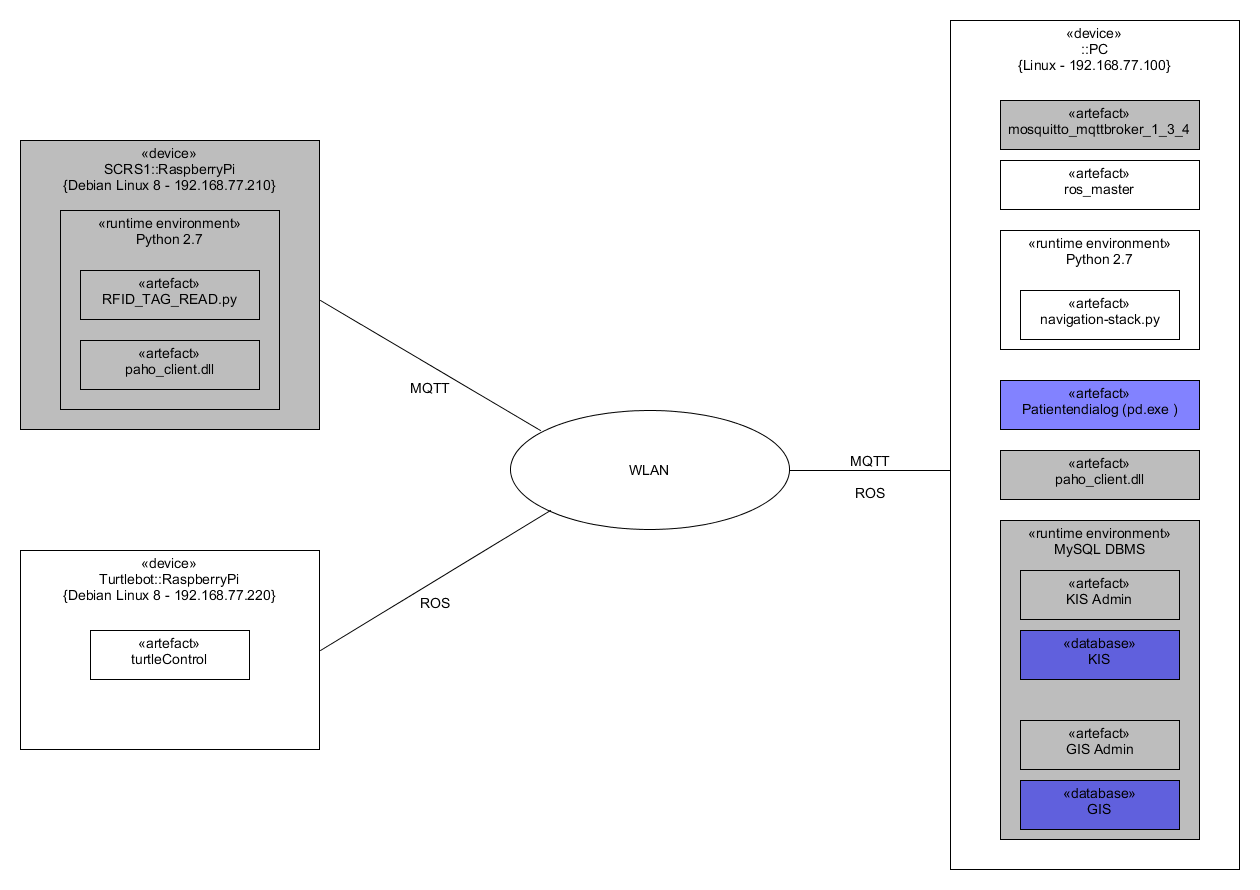


Abbildung 2: Verteilungsdiagramm

# Grobstruktur

Die Anwendung wird in strikter 3-Schichtenarchitektur realisiert, d.h. die Komponenten der jeweils tieferliegenden Schicht kennen nicht die Komponenten der darüberliegenden Schicht.

* GUI-Schicht
* Fachkonzept-Schicht
* Datenhaltung-Schicht

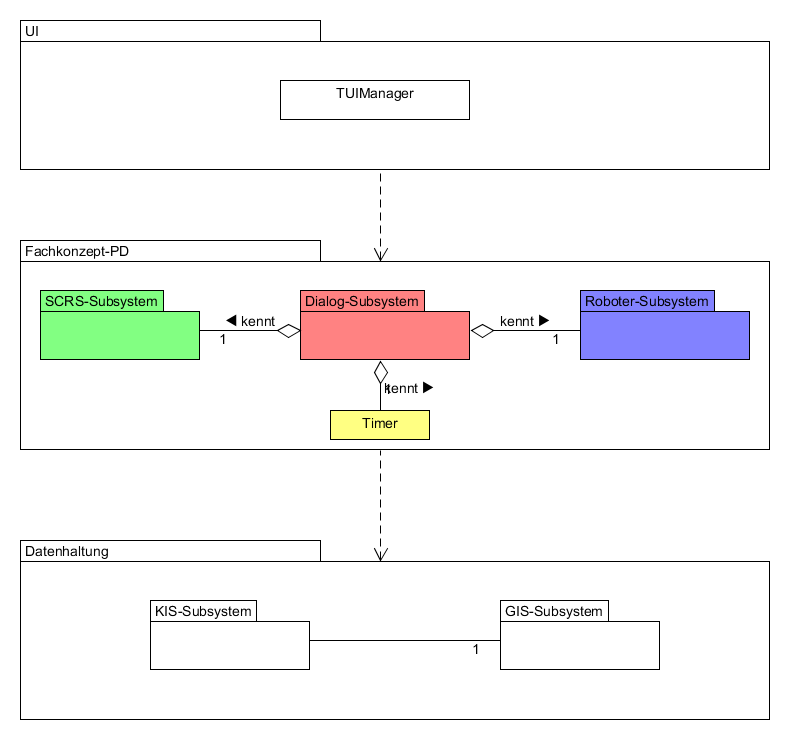


Abbildung 3: Paket-Diagramm

# Paketbeschreibungen

## Fachkonzept

Das Fachkonzept besteht aus drei Subsystemen (siehe Abbildung 3) und einen Timer. Das Dialog-Subsystem ist die Kernkomponente des Fachkonzepts, die den Dialog mit dem Patienten steuert. Die beiden anderen Subsysteme dienen als Proxy für die externen Systeme SCRS und Roboter. Die Subsysteme SCRS und Roboter werden so ausgelegt, dass sie unabhängig vom Dialogsubsystem sind. Deren Anbindung an das Dialogsubsystem erfolgt über ein Observer-Pattern, wobei die Subsysteme SCRS und Roboter die Subjekt-Rolle einnehmen. Gleiches gilt für den Timer.

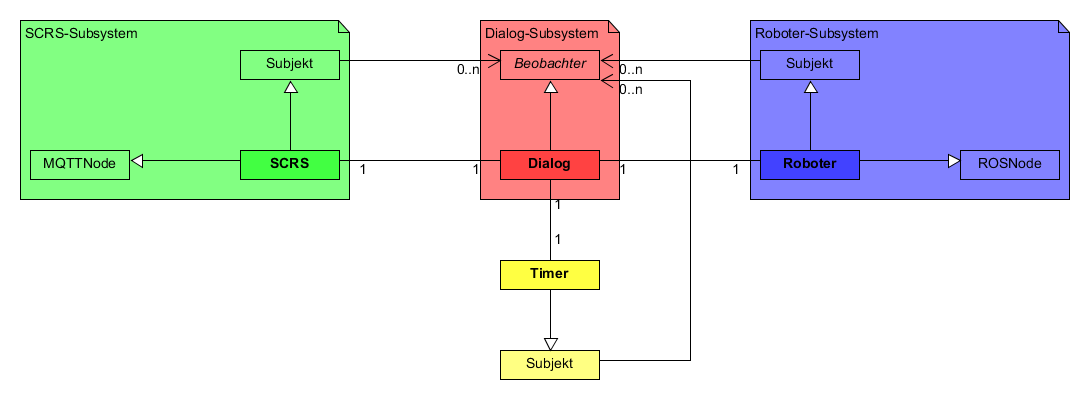


Abbildung 4: Klassendiagramm der Fachkonzept-Subsysteme

### SCRS-Subsystem

Das SCRS besteht aus den beiden Klassen "MQTTNode" und "SCRS". Die Klasse "MQTTNode", die aus einem Vorgängerprojekt wiederverwendet und hier modizifiziert wird, kapselt den Empfang und das Versenden von MQTT-Nachrichten. Sie basiert auf dem Paho Eclipse MQTT C-Client, welcher für den asynchronen Empfang von Nachrichten die Registrierung einer statischen Funktion erwartet. Diese statische Funktion wird in der Klasse "MQTTNode" über "msgarrvd()" zur Verfügung gestellt. Nutzerklassen wie SCRS sollten sich idealerweise von "MQTTNote" ableiten und einfach eine virtuelle Methode "onMessageReceived()" überschreiben, um eine Nachricht zu empfangen und zu verarbeiten. Um dies zu erreichen benötigt die statische Funktion "msgarrvd()" sicheren Zugriff auf ein Objekt der Klasse "MQTTNode" bzw. einer davon abgeleiteten Klasse. Dies kann nur über ein Klassenattribut "m\_instanz" geschehen, in welchem eine Referenz auf ein solches Objekt gespeichert wird. Dieses Design hat zur Folge, dass es nur eine Instanz eines MQTTNode´s bzw. einer davon abgeleiteten Klasse geben kann. Ggf. sollte der Einsatz des Singleton-Muster in Erwägung gezogen werden um die Verwendung dieser Klasse sicherer zu machen. Aktuell ist dies aber nicht umgesetzt.

### Roboter-Subsystem

Das Roboter-Subsystem besteht aus den beiden Klassen "ROSNode" und "Roboter". Wie im SRCS-Subsystem stellt die Klasse "ROSNode" die generische Funktionalität zum Senden und Empfangen von ROS-Nachrichen zur Verfügung. Die Klasse "Roboter" abstrahiert den realen Service-Roboter und stellt für diesen quasi eine Proxy dar. Sie nutzt die Funktionalität der Klasse "ROSNode".

Die Klasse "ROSNode" kapselt grundsätzlich die übliche ROS-Vorgehensweise für das Versenden und Empfangen und Nachrichten mittels Klassen "ros::Publisher" und "ros::Subscriber". Zum asynchronen Empfang von Nachrichten muss, wie bei MQTT, eine Call-Back-Funktion ("msgReceivedCallback()") bei ROS registriert werden. Anders als bei MQTT kann diese Funktion jedoch eine Objektmethode sein. Dies vereinfacht es, Nutzerklassen wie z.B. der Klasse "Roboter" den Empfang von Nachrichten mitzuteilen. Hierfür wird in der Klasse "ROSNode" die Call-Back-Methode als virtuell deklariert. Die Nutzerklasse leitet sich dann einfach von "ROSNode" ab und überschreibt die virtuelle Methode "msgReceivedCallback()". Da in ROS zum Empfangen von Nachrichten die Kontrolle periodisch oder dauerhaft mit "ros::spinOnce()" bzw. "ros::spin()" übergeben werden muss, würde dies zur Blockierung der Restanwendung führen. Deswegen wird das "spinning" in einem Thread ausgelagert. Das Threading wird über Klasse "std::thread" der Standard-C++-Library realisiert.

### Dialog-Subsystem

Der Dialog kann - wie bereits in der Systemspezifikation ausgeführt - über einen Zustandsautomaten beschrieben werden. Ereignisse, die zu Zustandsübergängen führen, können sowohl durch die beiden anderen Subsysteme "SCRS" und "Roboter" als auch durch das UI-Paket und den Timer ausgelöst werden. Die zentrale Klasse „Dialog“ des Dialog-Subsystem muss folglich als Zustandsmaschine ausgelegt werden. Hierfür wird das Zustands-Entwufsmuster verwendet, in der jeder Zustand durch eine separate Klasse repräsentiert wird.

Abbildung 5 skizziert den durch Klasse "Dialog" zu realisierenden Zustandsautomaten mit vier erlaubten Zuständen und den möglichen Ereignissen. Abbildung 6 zeigt dessen Umsetzung über das Zustands-Entwurfsmuster. Die Klasse "Dialog" delegiert die eintreffenden Ereignisse an entsprechende Zustandsklassen. In diesen Klassen wird dann das korrekte Verhalten inklusive der Zustandsübergänge bei Eintritt eines Ereignisses implementiert. Um eine Zustandsänderung durchführen zu können, benötigen die Zustandsklassen Zugriff auf die "Dialog"-Klasse, in der der aktuelle Zustand gespeichert ist. Deshalb gibt es auch eine Assoziation von der "Dialogzustand"-Klasse in Richtung "Dialog". Diese Assoziation wird an der abgeleiteten Zustandsklassen vererbt, die diese dann nutzen können.

Das onTimer60-Ereignis wird anders als die anderen onTimer-Ereignisse nicht durch ein Objekt der Klasse Timer ausgelöst. Dieses Timer-Ereignis soll eine Eingabe beenden. Bei der hier verwendeten textuellen Nutzerschnittstelle ist dies eine Konsoleneingabe. Der entsprechende blockierende Funktionsaufruf würde durch eine Timer-Unterbrechung nie zum Ende kommen. Deshalb wird dieses Timer-Ereignis durch eine Zähl-/Warteschleife mit nicht blockierender Keyboard-Abfrage innerhalb des TUI-Threads realisiert.

int stopZeit = 10; // in sec

int t=0,c;

while(!kbhit() && t<stopZeit )

{

usleep(1000000);

t++;

}

if(kbhit())

{

c = getchar();   
 …  
 }else …

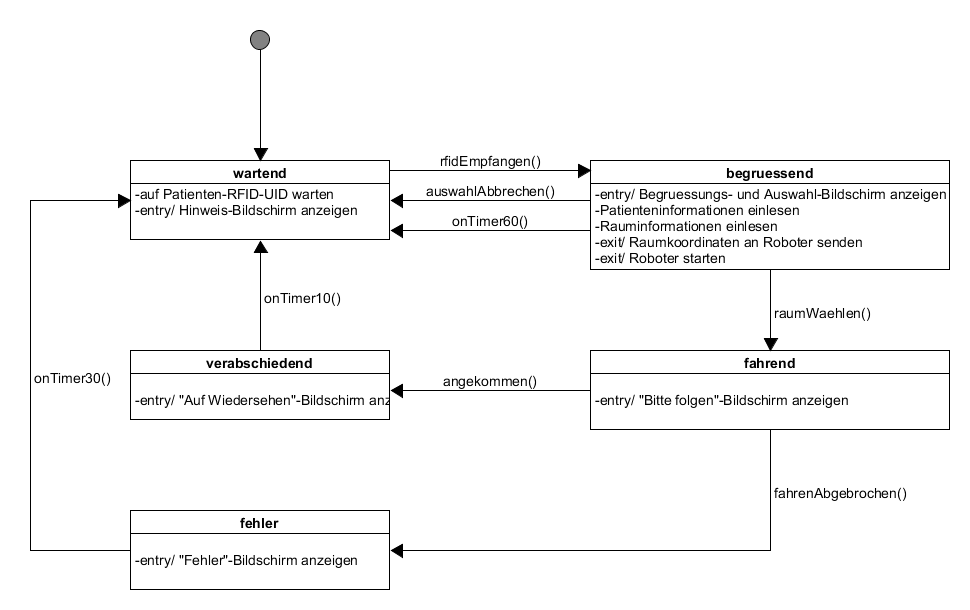


Abbildung 5: Zustandsautomat für den Dialog mit dem Patienten

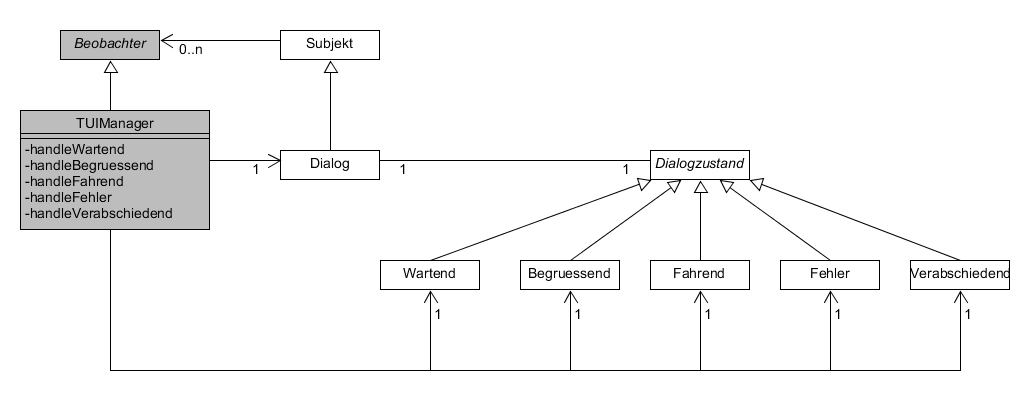


Abbildung 6: Umsetzung des Zustandsautomaten über das Zustands-Muster

## UI

Die Verbindung des UI mit dem Fachkonzept erfolgt über das Observer-Pattern. Hierbei ist die Klasse "Dialog" das beobachtete Objekt, welches seinen Beobachter, die TUI-Klasse "TUIManager" über Änderungen des Dialogzustands benachrichtigt. Die Klasse "TUIManager" enthält für jeden Zustand des Dialog-Subsystems eine Behandlungsmethode, die quasi als View und Controller für die entsprechende Zustandsklasse dient. Sie rendert die zu dem Zustand passenden Informationen und nimmt Nutzereingaben entgegen. Der Einfachheit halber wurde hier darauf verzichtet, für jede Zustandsklasse des Dialog-Subsystems eine eigene View/Controller-Klasse zu implementieren.

## Datenhaltung

# Feinspezifikation (interne Dokumenation)

## Klassendiagramm

## Modulspezifikationen