CARL VON OSSIETZKY universität OLDENBURG

DEPARTMENT FÜR INFORMATIK
SYSTEMSOFTWARE UND VERTEILTE SYSTEME

Konzeption und Implementierung einer universellen Management-Schnittstelle für HW/SW-Sensoren und Aktoren

Bachelorarbeit

16. Dezember 2013

Connor Fibich
Innsbruckerstraße 25
26131 Oldenburg

Erstprüfer Prof. Dr.-Ing. Oliver Theel
Zweitprüfer Prof. Dr. Andreas Winter

Erklärung zur Urheberschaft

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe.

Oldenburg, den 16. Dezember 2013

Connor Fibich

Im Rahmen dieser Bachelorarbeit wird eine Erweiterung für ein bereits bestehendes Rahmenwerk entwickelt, mit dem informatische Systeme innerhalb der von ihnen umgesetzten Funktionalität Sensoren und Aktoren optimieren und diese flexibel während der Laufzeit anbinden, entfernen und austauschen können. Nachteil dieses Rahmenwerks ist jedoch, dass das informatische System die Sensor- und Aktordaten in kurzen regelmäßigen Abständen selbst abrufen muss. Durch ein solches Vorgehen wird auch für das Abrufen derjenigen Werte, die für das System uninteressant sind, Energie verbraucht. Deshalb soll das Rahmenwerk um eine Event- und Kommunikationsstruktur erweitert werden, die die Rechenleistung der ohnehin benötigten Sensor- und Aktorknoten nutzbar macht, um die Anzahl der Abrufvorgänge zu reduzieren.
Diese Arbeit stellt die Entwicklung und die Implementierung der Erweiterung des Rahmenwerks vor und zeigt anhand eines Versuchsaufbaus, wie sich die Verwendung der Eventstruktur auf die Kommunikationen zwischen dem System und den Sensoren auswirkt.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

In heutigen informatischen Systemen werden häufig viele unterschiedliche Arten von Sensoren und Aktoren eingesetzt. Sie erlauben es dem System, seine Umgebung zu überwachen und zu beeinflussen. Zum Verwenden von Sensoren und Aktoren benötigen informatische Systeme oft Sensor-/Aktorknoten, die eine Kommunikationsschicht zwischen den Sensoren/Aktoren und dem System umsetzen. Das Betreiben solcher Sensor- und Aktorknoten erhöht jedoch den Energieverbrauch des Systems. Da tendenziell die Anzahl solcher Knoten in informatischen Systemen auch in Zukunft weiter ansteigen wird, werden Methoden gesucht, um dem damit steigenden Energieverbrauch des Gesamtsystems entgegenzuwirken.

Zu diesen Methoden gehört unter anderem der Ansatz, den Energieverbrauch eines Sensor- bzw. Aktorknotens selbst zu optimieren. Diese Optimierung wird unter anderem durch die Verwendung energietechnisch günstigerer Bauteile erreicht. Der Ansatz der Forschungsgruppe ExploIT Dynamics ist es, das Überwachen und Neukonfigurieren des Systems zur Laufzeit zu ermöglichen, um die Nutzung teurer Ressourcen einzudämmen. Im Rahmen dieser Neukonfigurationen kann es insbesondere erforderlich sein, "teure" Sensoren durch günstigere auszutauschen. Solche Umstrukturierungen zur Laufzeit erfordern eine gute Anpassungsfähigkeit des Systems, ähnlich den Wartbarkeits- und Übertragbarkeitskriterien für Software [ISO01].

Hierzu wurde im Rahmen der Bachelorarbeit "DSA: Dynamische Verwendung von Sensor- und Aktuatorknoten (HW/SW) zur Verbrauchsoptimierung von WSAN" von Jens Krayenborg [Kra13] ein Rahmenwerk zum Erstellen informatischer Systeme entwickelt. Dieses Rahmenwerk bietet Strukturen, um die dynamische Verwendung von beliebigen Sensor- und Aktorknoten zu ermöglichen, sowie eine einheitliche Schnittstelle, um Sensoren bzw. Aktoren anzusprechen. Bisher werden keine speziellen Strukturen, insbesondere hinsichtlich der Datenübertragung, für die Sensor- und Aktorknoten von diesem

Rahmenwerk gefordert. Deshalb steht dem Rahmenwerk das Potential der Knoten bezüglich der Interoperabilität und Konfigurierbarkeit nicht zur Verfügung.
So ist es bisher im DSA-Rahmenwerk vorgesehen, dass die Messdaten der Sensoren periodisch durch das System abgefragt (Polling) und auf Relevanz überprüft werden. Hier könnte das System durch eine bessere Ausnutzung der Interoperabilität zwischen System und Sensor- und Aktorknoten entlastet werden.
Auch könnten die Sensor- bzw. Aktorknoten nach vordefinierten Kriterien selbst prüfen, ob Messdaten relevant sind. Erweitert man dann die "Datenübermittlungsstrategie" vom reinen Polling um eine Interrupt- bzw. Eventstruktur, so können unnötige Kommunikationen vermieden werden. Eine solche Einsparung wird dadurch erreicht, dass der jeweilige Sensor nur noch dann Daten sendet, wenn diese nach definierten Kriterien als relevant einzustufen sind. Deshalb soll das Rahmenwerk nun auf die Implementation der Sensor- und Aktorknoten ausgeweitet und bezüglich der unterstützten "Datenübermittlungsstrategie" erweitert werden.

Als Teil dieser Implementation soll eine Selbstauskunftsfähigkeit über die angeschlossenen Sensoren/Aktoren ermöglicht werden. Die Selbstauskunftsfähigkeit erleichtert das Wiederverwenden der Sensoren/Aktoren.
Zusätzlich soll eine einheitliche Schnittstelle für Sensoren und Aktoren und deren Management entwickelt werden. Diese soll den Implementationsaufwand neuer Sensoren und Aktoren verringern.
Die Funktionalität dieser Erweiterungen soll mit einem Versuchsaufbau nachgewiesen und anschließend eine Quantifizierung der Einsparungen an einem Beispiel vorgenommen werden.

2. Zielsetzung

Im Rahmen dieser Bachelorarbeit soll eine Erweiterung des bereits bestehenden Rahmenwerks um die benötigten Strukturen und Schnittstellen vorgenommen werden, die für die Realisierungen einer Interrupt- bzw. Eventstruktur als "Datenübermittlungsstrategie" erforderlich sind. Außerdem sollen Templates für die Sensor- bzw. Aktorknoten-seitige Implementierung erstellt werden. Daraus ergeben sich folgende Ziele für die Erweiterung des Rahmenwerks:

1. Das Rahmenwerk soll mit einer Kommunikationsstruktur ein möglichst breites Spektrum an Anbindungsmöglichkeiten zur Verfügung stellen. Diese Struktur soll ermöglichen, Messdaten empfangen und verarbeiten zu können, die vom Sensor-/Aktorknoten aktiv gesendet wurden.

2. Das Rahmenwerk muss eine möglichst universelle Schnittstellendefinition für Sensor- und Aktorknoten bereitstellen. Diese soll die Kompatiblität zu den vom Rahmenwerk verwendeten Strukturen sicherstellen, ansonsten aber möglichst frei konfigurierbar sein.

3. Das Rahmenwerk soll eine einheitliche Schnittstelle zum schnellen Anbinden, Austausch und Management von Sensoren und Aktoren an die Sensor- bzw. Aktorknoten bereitstellen. Über diese Schnittstelle soll auch eine Selbstsauskunftfähigkeit und Konfigurierbarkeit der Sensoren und Aktoren ermöglicht werden.

4. Es muss die Möglichkeit bestehen, das Rahmenwerk auch als externen Dienst auf entfernten Systemen laufen lassen zu können. Die von diesem Dienst bereitgestellten Sensoren und Aktoren sollen dann über das lokale Rahmenwerk eingebunden werden können.

Dabei soll das Rahmenwerk in seinen ursprünglich definierten Fähigkeiten [Kra13, S. 3] möglichst nicht beeinträchtigt werden. Diese lauten wie folgt:

A Flexibilität bezüglich der verwendeten Kommunikationstechniken und Protokolle der Sensor- und Aktorknoten,

B Flexibilität bezüglich des Formats der Daten, die von den Sensor- und Aktorknoten empfangen und gesendet werden,

C dynamisches Ein- und Ausschalten der Sensor- und Aktorknoten,

D dynamisches Austauschen der Sensor- und Aktorknoten,

E dynamisches Entfernen und Hinzufügen von Sensor- und Aktorknoten,

F sowie Schaffen einer Transparenzschicht, die erreicht, dass das umliegende informatische System keine Kenntnisse über die Art der verwendeten Sensor- und Aktorknoten benötigt.

Die oben genannten Änderungen sollen anhand eines prototypischen Versuchsaufbaus, zum Beispiel einer Murmelbahn, hinsichtlich Effizienz bewertet und mit dem ursprünglichen Rahmenwerk verglichen werden.

3. Grundlagen

In diesem Kapitel soll eine kurze Beschreibung der Eigenschaften der wesentlichen von Sensor/Aktor-Anwendungen verwendeten Komponenten gegeben werden.

3.1. Eigenschaften der Sensoren

Sensoren messen bestimmte Eigenschaften der Umgebung. Diese Informationen werden an einer von dem spezifischen Sensor abhängigen Schnittstelle zur Auswertung und Weiterverarbeitung bereitgestellt. Das innerhalb dieser Arbeit zu entwickelnde Rahmenwerk unterstützt sowohl Hardware- als auch Softwaresensoren, denen unterschiedliche Konzepte zugrunde liegen, die im Folgenden beschrieben werden.

3.1.1. Hardwaresensoren

Hardwaresensoren sind Messgeräte, die in der Lage sind, Messgrößen der Umgebung in auswertbare Signale umzuwandeln.
Dabei kann es sich um physikalische, chemische, biologische oder sonstige Messgrößen handeln. Für jede Art dieser unterschiedlichen Messgrößen gibt es wiederum verschiedene Technologien, mit denen die entsprechenden Messgrößen ermittelt werden, so dass ein breites Spektrum an unterschiedlichen Hardwaresensoren unterstützt werden muss.
Dabei ist jedoch lediglich die Schnittstelle der Hardwaresensoren, über die die Messgröße ausgelesen wird, für das informatische System relevant. Auch bei den Schnittstellen gibt es eine große Varianz. Beispielsweise verfügt die Infrarotlichtschranke K153P [tem] über eine analoge Schnittstelle, über die der Wert ausgelesen werden muss, während eine Kamera, deren Bilder ausgewertet werden sollen, die aufgezeichneten Bilder in einem digitalen Format bereitstellt. Da das informatische System grundsätzlich nur digitale Werte weiterverarbeiten kann, müssen gegebenenfalls analoge Messsignale in einem Zwischenschritt mit einem Analog-Digital-Wandler aufbereitet werden.

Dieser Wandler stellt die digitale Schnittstelle bereit, auf die die Software zugreift. Obwohl sämtliche Schnittstellen innerhalb des Systems digital ausgelegt sind, weisen sie erhebliche Unterschiede in der Syntax und dem Format der zur Verfügung gestellten Daten auf.

Aktive / passive Sensoren

Hardwaresensoren werden darüber hinaus als aktiv oder passiv klassifiziert. Aktive Sensoren zeichnen sich dadurch aus, dass sie neben externen Energien auch zusätzliche Hilfsenergie benötigen, um die Messgröße erfassen zu können. Ein Beispiel für aktive Sensoren sind Lichtschranken. Diese bestehen aus einem Emitter, der als Referenz-Lichtquelle Hilfsenergien verwendet, und einem Photosensor der Abweichungen zu diesem Referenzwert misst.
Passive Sensoren hingegen verwenden ausschließlich externe Energien, um die Messgrößen zu erfassen. Zum Beispiel verwenden einfache Photosensoren das Umgebungslicht als externe Energie für ihre Messungen.

3.1.2. Softwaresensoren

Softwaresensoren sind Sensoren, die physikalisch nicht existieren, sondern die Sensorfunktion virtuell umsetzen. Zum einen messen sie auf Softwareebene die logischen Eigenschaften der Softwareumgebung. Zu diesen Eigenschaften gehören unter anderem Attribute, die vom System während der Laufzeit erzeugt werden, oder das Ein- bzw. Austreten aus Funktionsaufrufen. Andererseits simulieren Softwaresensoren andere physische Sensoren, indem sie Messwerte realer Sensoren verwenden und diese über empirische oder physikalische Modelle ableiten und hochrechnen, so dass diese Hardwaresensoren seltener ausgelesen werden müssen. Externe Dienste, die von anderen Systemen bereitgestellt werden, können ebenfalls als Softwaresensoren klassifiziert werden. Diese Dienste stellen in der Regel extern aufbereitete Messdaten in Form von logischen Größen zur Verfügung. Das informatische System unterscheidet nicht zwischen den unterschiedlichen Quellen der verwendeten Daten, sondern verarbeitet Informationen aus internen Softwaresensoren sowie externen Diensten gleichermaßen. Dabei muss das informatische System in der Lage sein, sämtliche verwendete Schnittstellen anzusprechen und aus zu lesen, die sich insbesondere aufgrund der unterschiedlichen Arten der externen Dienste erheblich unterscheiden können. Die Klassifizierung der internen

Softwaresensoren und externen Dienste erfolgt analog zu der dieser Arbeit zugrundeliegenden Ausarbeitung [Kra13].

3.2. Eigenschaften der Aktoren

Anders als Sensoren beeinflussen Aktoren ihre Umgebung. Die Funktionen, die mit den spezifischen Aktoren ausgeführt werden sollen, können dabei über deren Schnittstelle angesteuert werden. Ähnlich wie für die Sensoren ist auch für die Aktoren eine Unterscheidung zwischen Hardwareaktoren und Softwareaktoren sinnvoll.

3.2.1. Hardwareaktoren

Hardwareaktoren sind funktionale Elemente, über die das physikalische Umfeld des Aktors beeinflusst werden kann. Diese Elemente besitzen eine Schnittstelle, an die Signale angelegt werden können. Die Eingangssignale werden vom entprechenden Aktor in mechanische Bewegung oder andere Aktionen umgesetzt. Wie die Hardwaresensoren weisen Hardwareaktoren eine große Varianz in den verwendeten Schnittstellen auf. Ein gutes Beispiel zur Veranschaulichung dieser Varianz sind Motoren: So besitzt die Schnittstelle eines Gleichstrommotors zwei Signaleingänge. Die Differenz der Spannungen, die an diesen Eingängen angelegt werden, gibt vor, wie schnell und in welche Richtung sich der Motor dreht. Ganz anders ist die Schnittstelle eines Schrittmotors aufgebaut: Sie besteht i.d.R. aus 4, 6 oder 8 Eingängen. Wird an einem dieser Eingänge eine Spannung angelegt, wird der Motor in eine bestimmte Position gebracht. Um den Motor zu drehen, müssen die Eingänge in einer bestimmten Reihenfolge geschaltet werden. Die Reihenfolge und Geschwindigkeit, in der die Eingangssignale angelegt werden, geben dabei vor, in welche Richtung und wie schnell sich der Motor dreht.

3.2.2. Softwareaktoren

Wie die Softwaresensoren sind auch die Softwareaktoren physikalisch nicht existent. Sie beeinflussen i.d.R. das logische Umfeld des Systems auf der Softwareebene. So könnte beispielsweise ein Softwareaktor dazu genutzt werden, andere Prozesse zu steuern (starten, stoppen, Attribute ändern, etc.). Der Einflussbereich von Softwareaktoren kann jedoch auch über das logische Umfeld

hinausgehen. Die Beeinflussung der physikalischen Umgebung kann, in einem solchen Fall, auch durch Softwareaktoren mit Hilfe von externen Diensten erfolgen. Diese Dienste bieten den informatischen Systemen die Nutzung der von ihnen gesteuerten Aktoren über eine Softwareschnittstelle an, die von den Softwareaktoren genutzt werden kann.

4. Anforderungen und Rahmenbedingungen

In diesem Kapitel werden zunächst die Anforderungen, die an die Erweitung des Rahmenwerks gestellt werden, genauer definiert. Anschließend wird an Szenarien geprüft, ob diese durch die Anforderungen abgedeckt sind.

4.1. Anforderungserhebung

In der Problemstellung wurden bereits Ziele festgelegt, aus denen die wichtigsten Anforderungen abgeleitet werden.
Als Grundlage ergeben sich aus den Zielen zum verwendeten Basis-Rahmenwerk die folgenden Anforderungen:

Anforderung A: Das Rahmenwerk muss Schnittstellen zur Verfügung stellen oder vom Benutzer anfordern, die das volle Spektrum an Kommunikationstechniken und Protokollen der Sensoren und Aktoren unterstützt.

Anforderung B: Sämtliche gängige Datenformate müssen vom Rahmenwerk unterstützt werden. Außerdem muss gefordert werden, dass Sensoren und Aktoren ihre Daten in einem vom Rechner auswertbaren Format bereitstellen.

Anforderung C: Das Rahmenwerk muss über die Strukturen für Sensoren und Aktoren erkennen können, ob ein verwalteter Sensor oder Aktor mit der Sensor/Aktor-Anwendung verbunden ist. Neu in die Anwendung eingebundene Sensoren und Aktoren müssen automatisch aktiviert werden, nicht mehr verwendete Sensoren und Aktoren müssen entsprechend deaktiviert werden.

Anforderung D: Die vom Rahmenwerk verwendeten Strukturen für Sensoren und Aktoren müssen es der Sensor/Aktor-Anwendung ermöglichen, zur Laufzeit dynamisch zwischen den vom Rahmenwerk verwalteten Sensoren und Aktoren zu wechseln. Dafür muss die sofortige Verwendbarkeit der ausgetauschten Sensoren oder Aktoren gewährleistet sein, um Unterbrechungen im Programmablauf zu vermeiden. Das Rahmenwerk muss also selbst alle erforderlichen Anpassungen an den ausgetauschten Sensoren und Aktoren intern vornehmen. Es dürfen darüber hinaus keine Änderungen in der Sensor/Aktor-Anwendung erforderlich werden.

Anforderung E: Das Rahmenwerk muss der Sensor/Aktor-Anwendung Funktionen anbieten, mit denen Sensoren und Aktoren neu in die Verwaltungsstruktur des Rahmenwerks eingebunden oder aber aus dieser entfernt werden können.

Anforderung F: Um eine Transparenz-Schicht zu bilden, die es dem informatischen System ermöglicht, ohne Kenntnisse über die Art der verwendeten Sensoren und Aktoren deren vollen Funktionsumfang nutzen zu können, muss das Rahmenwerk eine Schnittstelle bereitstellen, die einen transparenten Zugriff erlaubt.

Aus den durch die Erweiterung hinzugekommenen Zielen ergeben sich darüber hinaus weitere Anforderungen:

Anforderung 1: Das Rahmenwerk muss so aufgebaut sein, dass der Informationsaustausch zwischen dem System und den Sensoren und Aktoren von beiden Seiten initiiert werden kann.

Anforderung 2: Für die Anbindung der Sensoren und Aktoren an das Rahmenwerk muss von diesem eine möglichst universell passende Schnittstelle gefordert werden. Diese ist durch den Benutzer auf Grundlage der tatsächlich eingesetzten Sensoren und Aktoren zu entwickeln und bereitzusstellen. Dem Benutzer muss eine Definition für diese Schnittstelle vorgegeben werden, die es erlaubt, ein breites Spektrum unterschiedlicher Schnittstellen solcher spezifischen Sensoren und Aktoren abzubilden.

Ziel drei bezieht sich auf das Zusammenspiel zwischen der Sensor/Aktor-Anwendung und dem Rahmenwerk bezüglich der Verwaltung und Konfiguration der Sensoren und Aktoren. Daraus lassen sich Anforderung 3 und 4 ableiten:

Anforderung 3: Das Rahmenwerk muss eine Managementschnittstelle bereitstellen. Über diese Schnittstelle muss die Funktionalität aus den Anforderungen C, D und E der Sensor/Aktor-Anwendung zur Verfügung gestellt werden. Darüber hinaus muss diese Schnittstelle Zugriff auf die Selbstauskunftsfunktionen der Sensoren und Aktoren bereitstellen sowie eine Konfiguration dieser Komponenten ermöglichen.

Anforderung 4: Innerhalb des Rahmenwerkes wird eine geeignete Verwaltungsstruktur benötigt, in der die verbundenen Sensoren und Aktoren verwaltet werden. Auf diese Struktur kann über die in Anforderung 3 geforderte Schnittstelle zugegriffen werden.

Anforderung 5: Um die Möglichkeit zu erhalten, das Rahmenwerk auch als Dienst verwenden zu können, wird eine Kommunikationsstruktur benötigt. Diese Struktur muss Verbindungen zu anderen Rahmenwerken verwalten und die über die Verbindungen übertragenen Funktionsaufrufe ausführen können.

4.2. Zusammengesetzte Sensoren

Häufig verwenden Sensor/Aktor-Anwendungen nicht nur einen einzelnen Sensor zur Bestimmung eines Messwertes, sondern eine Kombination verschiedener Sensoren, deren Werte in eine Auswertung eingehen, die den eigentlichen Messwert liefert. Betrachtet man als Beispiel die Temperaturüberwachung eines Raumes, so kann der Messwert eines einzelnen Sensors dabei nur Informationen zu einem Teilbereich des Raumes liefern. Abhängig von der Positionierung des Sensors innerhalb des Raumes, können große Differenzen der gemessenen Werten auftreten (z.B. in Abhängigkeit zu der Entfernung zur Heizung). Um den durchschnittlichen Temperaturwert entsprechend genau wiedergeben zu können, müssen deshalb mehrere Sensoren im Raum positioniert werden. Das Anwendungsprogramm verarbeitet die Werte dieser Temperatursensoren, indem es eine Auswertung vornimmt und zum Beispiel den Durchschnittswert ermittelt.

Eine solches Vorgehen würde ein Polling und damit eine vielzahl von Datenübertragungen erfordern, die jedoch gerade vermieden werden sollen. Diese Aufgabe könnte deshalb idealerweise ebenfalls vom Rahmenwerk übernommen werden, damit auch in diesem Fall wieder die durch Ziel 1 gewünschte optimierte Kommunikationsstruktur verwendet werden kann.

Aus diesem Anwendungsfall folgt somit folgende optionale Anforderung:

Das Rahmenwerk soll Strukturen bereitstellen, über die Sensoren und Aktoren dynamisch zusammengefasst werden können. Innerhalb dieser Struktur müssen die Messwerte der einzelnen Sensoren nach vorgegebenen Regeln vereinheitlicht werden. Da diese Regeln stark voneinander abweichen kön-

nen, muss die Struktur die Flexibilität liefern, diese Regeln erst nachträglich zu erstellen und eventuell auch zu änderen.

5. Architektur des Rahmenwerks

In diesem Kapitel wird die Architektur des Rahmenwerks vorgestellt und gezeigt wie diese durch ihre Gestaltung die in Kapitel 4 festgelegten Anforderungen umsetzt.

5.1. Ebenendefinition

Das DSA-Rahmenwerk kapselt die Konfiguration und Verwaltung von Sensoren und Aktoren von dem informatischen System ab. Durch diese Kapselung entsteht eine Ebenenstruktur für Sensor/Aktor-Anwendungen, die das DSA-Rahmenwerk verwenden.
Abbildung 5.1 zeigt die vier Ebenen, die durch die Verwendung das Rahmenwerk entstehen und im Folgenden näher betrachtet werden.

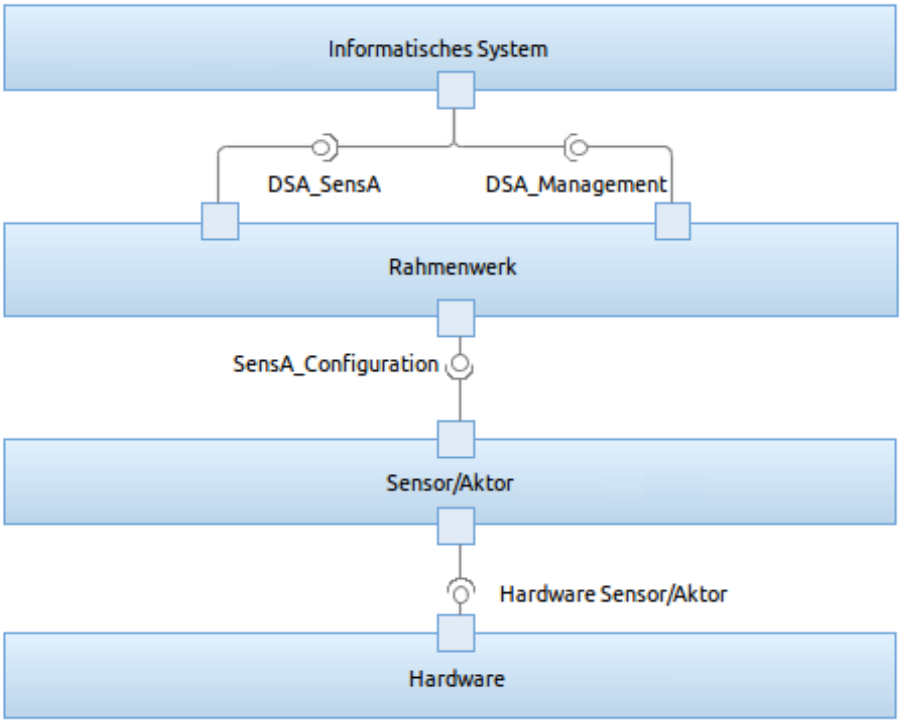


Abbildung 5.1.: Ebenen des Rahmenwerks

5.1.1. Ebene des informatischen Systems

Auf der Ebene des informatischen Systems laufen die Sensor/Aktor-Anwendungen. Das informatische System nutzt hierzu die vom Rahmenwerk gegebene transparente Schnittstelle für Sensoren und Aktoren (Anforderung F (siehe Kapitel 4)). Über diese Schnittstelle kann das System die Sensoren und Aktoren des Rahmenwerks verwenden, ohne deren genaue Beschaffenheit zu kennen.

Das System lagert die direkte Kontrolle über die Sensoren und Aktoren in das Rahmenwerk aus, auf das es über die in Anforderung 3 geforderte Managementschnittstelle Einfluß nimmt. Über diese Schnittstelle greift es außerdem auf Selbstauskunftsfunktionen der Sensoren und Aktoren zu. Mit den über die Selbstauskunftsfähigkeit gewonnenen Informationen können die Anwendungen eine Auswahlentscheidung für Sensoren und Aktoren treffen und diese dann dynamisch einbinden oder austauschen. Über diese Schnittstelle wird ebenfalls eine Konfiguration der Sensoren und Aktoren durch das System ermöglicht.

5.1.2. Sensor/Aktor-Ebene

Auf der Sensor/Aktor-Ebene werden die eigentlichen Implementationen der Schnittstellen der Sensoren und Aktoren nach den Vorgaben des DSA-Rahmenwerks gemäß Anforderung 2 erstellt. Dabei werden die jeweiligen Schnittstellen der spezifischen Sensoren und Aktoren verwendet und auf die vorgegebene Schnittstelle abgebildet. Eine solche Abbildung einer zu dem System inkompatiblen Schnittstelle auf eine einheitliche Schnittstelle, die vom System verwendet werden kann, wird auch als *Adapter-Pattern* [GJHV11, S. 139 ff.] bezeichnet.

Durch diese Abbildung wird auch auf dieser Ebene eine Kapselung bewirkt, die gegenüber dem DSA-Rahmenwerk eine Transparenzschicht bildet, so dass für das DSA-Rahmenwerk die genauen Schnittstellen-Spezifikationen der Sensoren und Aktoren nicht mehr relevant sind.

5.1.3. Hardware-Ebene

Zur Hardware-Ebene gehören die spezifischen Sensoren und Aktoren. Diese geben ihre eigenen Schnittstellen-Spezifikationen vor, über die eine Ansteuerung erfolgen muss. Wie bereits in der Definition für Sensoren und Aktoren festgestellt, bestehen große Unterschiede zwischen diesen Schnittstellen-Spezifikationen, je nach Art der verschiedenen Sensoren und Aktoren. Diese Schnittstellen werden in der Sensor/Aktor-Ebene vereinheitlicht.

5.1.4. Rahmenwerk-Ebene

Die Rahmenwerk-Ebene stellt dem informatischen System eine einheitliche transparente Schnittstelle zur Verwendung der jeweiligen Sensoren und Aktoren zur Verfügung. Darüber hinaus übernimmt das Rahmenwerk die zentralen Managementfunktionen zur Verwaltung der einzelnen Sensoren und Aktoren. Hierfür besteht eine weitere Schnittstelle, die den Zugriff auf diese Managementfunktionen erlaubt.
Auf dieser Ebene werden die auf der Sensor/Aktor-Ebene implementierten Sensoren und Aktoren verwaltet und mit der transparenten Schnittstelle für das informatische System verknüpft.

5.2. Komponenten des Rahmenwerks

Das Rahmenwerk setzt sich aus drei Komponenten zusammen, auf die die gewünschte Funktionalität aufgeteilt wird: der Sensor/Aktor-, der Management- und der Kommunikations-Komponente. Um die Anforderungen des Basis-Rahmenwerks weiterhin zu erfüllen, knüpfen die in diesem Abschnitt beschriebenen Komponenten an Strukturen des Basis-Rahmenwerks an. Im Folgenden werden zunächst die im Basis-Rahmenwerk verwendeten Strukturen vorgestellt, die den entsprechenden Komponenten zugeordnet werden können. Anschließend werden die Neuerungen beschrieben, die durch die Anforderungen an die Erweiterung erforderlich sind.

5.2.1. Sensor/Aktor-Komponente

In der Sensor/Aktor-Komponente werden die Strukturen zusammengefasst, die dem informatischen System den Zugriff auf die Sensoren und Aktoren erlauben. Abbildung 5.2. zeigt die bisherige Struktur, mit der das informatische

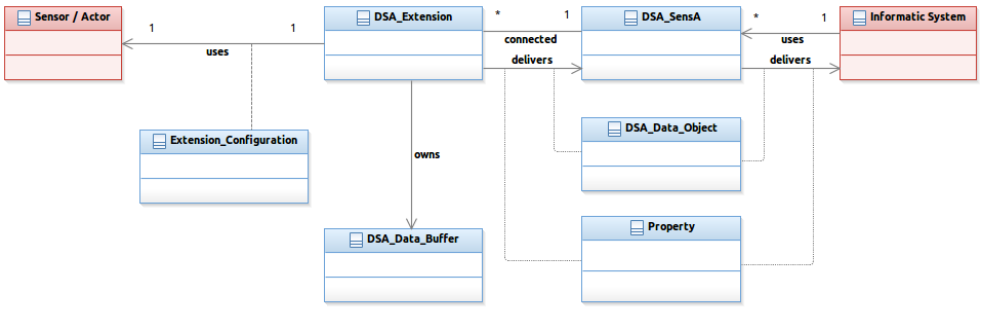


Abbildung 5.2.: Struktur für Sensoren und Aktoren im Basis-Rahmenwerk

System mit den Sensoren und Aktoren interagiert. Diese realisieren bereits die Forderungen von Anforderung 2, A, B, C, D und F.

Das informatische System nutzt die *DSA_SensA*-Klasse für den Zugriff auf die Sensoren und Aktoren. *Instanzen* der *DSA_SensA* bieten dem System einheitliche Funktionen, die eine transparente Verwendung von Sensoren und Aktoren erlauben. *DSA_SensA* können dazu mit den Sensoren und Aktoren verbunden werden. Mit der Verbindung des Sensors oder Aktors an den *DSA_SensA* erhält die *DSA_SensA*-Instanz Assoziationen, mit denen auf die Funktionalität des Sensors bzw. Aktors zugegriffen werden kann.

Die mit den *DSA_SensA* verbundenen Sensoren und Aktoren sind Instanzen der *DSA_Extension*-Klasse. Instanzen dieser Klasse verwalten die Daten der Sensoren bzw. Aktoren sowie die Verbindungen zu den *DSA_SensA*. *DSA_Extensions* sind in der Lage, über die mit ihnen verbundene Konfigurationsklasse auf die Funktionalität der Sensoren und Aktoren zuzugreifen.

Die *Extension_Configuration*-Klasse stellt die Schnittstelle für Sensoren und Aktoren dar. Sie wurde nach dem Template-Pattern [GJHV11, S. 325 ff.] entworfen. Klassen, die dieses Entwurfsmuster verwenden, sind *abstrakte* Klassen, deren Funktionen zum Teil nur als Schablonenmethoden angelegt wurden. Solche Funktionen müssen durch abgeleitete Klassen, die die abstrakte Klasse erweitern, implementiert werden. Über die Schablonenmethoden der *Extension_Configuration*-Klasse erfolgt der Zugriff auf die spezifischen Schnittstellen der Sensoren und Aktoren. Das auf der

Sensor/Aktor-Ebene (siehe Abschnitt 5.1.2.) verwendete Adapter-Pattern wird über die Ableitung der *Extension_Configuration*-Klasse realisiert. Instanzen dieser Klassen werden als Konfiguration an die *DSA_Extensions* gebunden.

Diese Struktur unterstützt nur einen aktiv vom informatischen System initiierten Austausch zwischen dem System und den Sensoren und Aktoren. Deshalb muss die Struktur zur Erfüllung von Anforderung 1 so angepasst werden, dass die Sensoren und Aktoren Daten aktiv senden können. Zur Umsetzung eines solchen Informationsaustauschs bieten sich insbesondere das Observer- oder das Mediator-Pattern [GJHV11, S. 293 ff. und S. 273 ff.] an.

Das Observer-Pattern erlaubt es, einer beliebigen Anzahl von Beobachtern (Observer) ein Subjekt direkt zu beobachten. Ein Subjekt stellt Funktionen bereit, die es Beobachtern ermöglichen, sich an diesem an- bzw. abzumelden, sowie eine Funktion zum Benachrichtigen der angemeldeten Beobachter. Dieser Aufbau unterstützt die angestrebte Dynamik, die durch das Rahmenwerk realisiert werden soll, da Beobachter sehr variabel an Subjekte gebunden werden können. Die Subjekte leiten somit nur Daten an Objekte weiter, die für diese relevant sind.

Im Gegensatz zum Observer-Pattern ermöglicht das Mediator-Pattern einen indirekten Informationsaustausch zwischen den Objekten. Durch das Mediator-Pattern wird der Struktur ein Vermittler hinzugefügt. Daten können von den Objekten an den Vermittler gesendet werden, der diese wiederum an alle anderen Objekte weiterleitet. Die Realisierung dieses Patterns ist im Rahmenwerk in zwei Varianten möglich. In der ersten Variante wird ein zentraler Vermittler eingesetzt. Bei der Verteilung der Daten würde ein zentraler Vermittler für jedes Objekt prüfen, ob die Daten für das Objekt relevant sind. Ein solches Vorgehen erzeugt einen großen Overhead an Auswertungen, der für jeden Datensatz entsteht, der verteilt werden soll. In der zweiten Variante wird ein Vermittler für jedes Objekt erstellt, das Daten senden möchte. Diese Variante vermeidet den Overhead, der durch die Verwendung eines zentralen Vermittlers entsteht, da der Vermittler die Daten nur noch an diejenigen Objekte verteilt, für die die Informationen relevant sind. Im

Vergleich zum Observer-Pattern unterscheidet sich diese Variante lediglich dadurch, dass das Subjekt in ein Objekt und einen Vermittler aufgespalten wurde. Die Aufteilung führt zu einem erhöhten Verwaltungsaufwand, für die verwendeten Objekte.

Aufgrund des unnötigen Overheads in der ersten Variante und dem erhöhten Verwaltungsaufwand in der zweiten Variante ergeben sich keine Vorteile des Mediator-Patterns gegenüber dem Observer-Pattern. Deswegen wurde im Modell des erweiterten Rahmenwerks das Observer-Pattern als geeigneter bewertet und deswegen verwendet. Der Informationsfluss, der mit diesem Muster realisiert werden soll, wird im Rahmenwerk durch die Sensoren und Aktoren initiiert. Die Daten werden dann von der *Extension_Configuration* aus über die *DSA_Extension* und die *DSA_SensA* an das System geleitet. Da die *DSA_Extension* und *DSA_SensA* diese Daten vor der Weiterleitung gegebenenfalls weiter verarbeiten, müssen alle zwischengeschalteten Objekte sowohl Beobachter sein, als auch beobachtet werden können.

In Abschnitt 4.2. wurde die optionale Anforderung gestellt, zusammengesetzte Sensoren in den Strukturen des Rahmenwerks zu realisieren. Um zu ermöglichen, dass ein Sensor oder Aktor sich aus einer unbestimmten Menge von Sensoren oder Aktoren zusammensetzen kann, werden spezielle Verwaltungsstrukturen benötigt. Diese müssen diejenigen Sensoren bzw. Aktoren verwalten, die zusammen den neuen Sensor bzw. Aktor bilden sollen. Außerdem muss die Möglichkeit bestehen, die Funktion festzulegen die das Zusammenwirken der Sensoren bzw. Aktoren erreicht werden soll. Dabei darf durch die Struktur nicht ausgeschlossen werden, dass die einzelnen Bestandteile eines solchen zusammengesetzten Sensors bzw. Aktors, selbst auch bereits zusammengesetzte Sensoren oder Aktoren sind.

Diese optionale Anforderung führt zu einer baumartigen Struktur für Sensoren und Aktoren, die idealerweise mit dem *Composite*-Pattern [GJHV11, S. 163 ff.] realisiert werden kann. Abbildung 5.3. zeigt den beispielhaften Aufbau dieses Entwurfmusters, das über die abstakte Klasse *Component* die einheitliche Verwendung der Klassen *Leaf* und *Composite* erlaubt. *Leaf* stellt

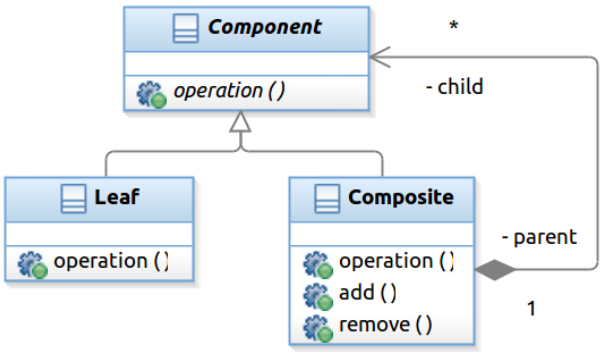


Abbildung 5.3.: Struktur des Composite-Patterns

dabei eine Implementation der *Component*-Klasse dar. Das Besondere dieses Entwurfsmusters liegt dabei im Entwurf der *Composite*-Klasse. Während diese selbst die *Component*-Klasse implementiert und somit *Component* als übergeordneten Typ besitzt, können ihr weitere Objekte vom Typ Component zugeordnet werden. Die Implementation der durch Component definierten Funktionen müssen dabei die Funktionen der Component-Instanzen berücksichtigen, die dem Composite zugeordnet wurden.

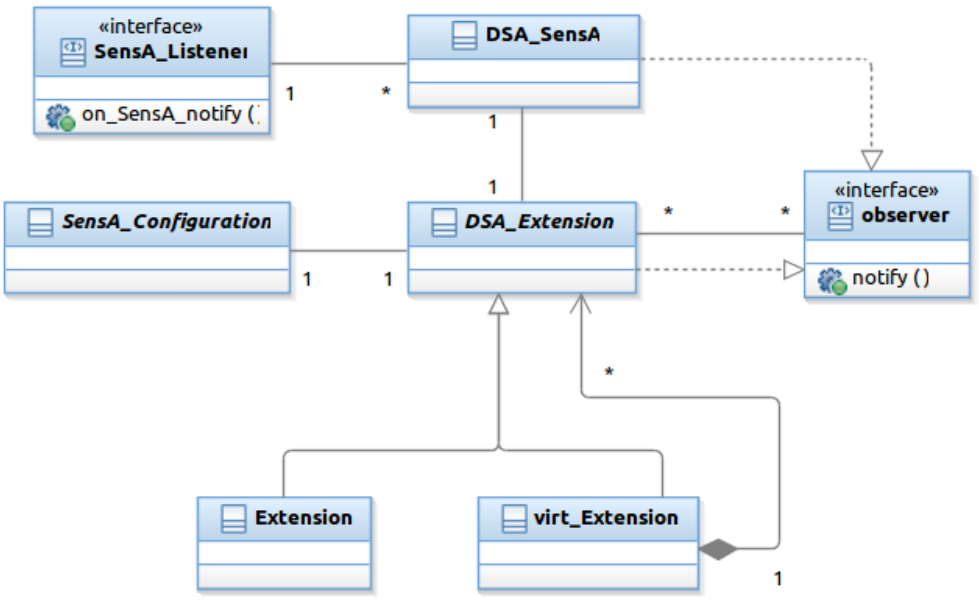


Abbildung 5.4.: Modell der Sensor/Aktor Komponente

In Abbildung 5.4. werden die Neuerungen des Modells für die Sensoren und Aktoren gezeigt, die durch die Einarbeitung der Observer- und Composite-

Struktur entstehen. Dabei bleibt die Transparenz der Schnittstelle für den Zugriff durch das informatische System auf die Sensoren und Aktoren des Rahmenwerks erhalten und wird weiterhin durch die *DSA_SensA*-Klasse gegeben. Diese implementiert die durch das Observer-Pattern hinzugekommene *Observer*-Schnittstelle.

Die Funktionalität der *DSA_Extension* des Basis-Rahmenwerks wird im neuen Modell durch *DSA_Extension*, *virt_Extension* und *Extension* umgesetzt. Diese drei Klassen bilden die gewünschte Composite Struktur. *DSA_Extension* ist dabei die abstrakte Klasse des Composite-Patterns. Die *Extension*-Klasse stellt die Impementierung der lokalen Sensoren und Aktoren. *virt_Extension* implementiert den letzten Teil des Patterns. Den *virt_Extensions* können beliebig *DSA_Extensions* hinzugefügt werden, deren Auswertung dann in den Berechnungen der *virt_Extension* berücksichtigt werden. Durch die Implementierung der Observer-Schnittstelle in der *DSA_Extension* können die von den spezifischen Sensoren und Aktoren ermittelten Daten nicht nur an die *DSA_SensA*, sondern auch an die jeweiligen *virt_Extensions* weitergegeben werden, die die Daten berücksichtigen sollen.

5.2.2. Management-Komponente

Durch die Strukturen der Management-Komponente werden die Sensoren und Aktoren verwaltet. Diese Komponente muss auch eine Managementschnittstelle bereitstellen, mit der eine Konfiguration der Sensoren und Aktoren ermöglicht wird.

Im Basis-Rahmenwerk wurde diese Aufgabe durch die Klassen *DSA_Factory* und *Extension_Container* erfüllt. Die *DSA_Factory* war dafür zuständig, neue *DSA_Extensions* zu erstellen und bereits bestehende sauber aus den Strukturen zu entfernen. Erstellte *DSA_Extension* wurden dabei in dem *Extension_Container* abgelegt. Der *Extension_Container* besaß Strukturen, in denen *DSA_Extension* abgelegt und jederzeit über einen einzigartigen Identifikator abgerufen werden können. Diese Klassen wurden nach dem *Singleton*-Pattern [GJHV11, S. 127 ff.] entworfen. Dieses Entwurfsmuster stellt sicher, dass innerhalb einer Programmausführung stets nur eine Instanz der *DSA_Factory* und des *Extension_Container* existiert. Dies ist erforder-

lich, damit jeweils ein klarer Punkt exisitert, über den die Sensoren und Aktoren erstellt und diese verwaltet werden.

Neben Anforderung 3 und 4, die durch einige Anpassungen des beschriebenen Aufbaus erfüllt werden können, muss in den Strukturen der Management-Komponente auch Anforderung 5 berücksichtigt werden. Durch diese Anforderung ist es notwendig, nicht nur lokale Sensoren und Aktoren verwalten zu können, sondern auch fremde Sensoren und Aktoren, die auf beliebigen entfernten Rahmenwerken ausgeführt werden. Solche entfernten Sensoren und Aktoren sollten idealerweise in einem von den lokalen Sensoren und Aktoren getrennten *Extension_Container* abgelegt werden. Dies ist unter anderem deshalb sinnvoll, weil sonst nicht gewährleistet werden kann, dass einer der auf dem lokalen System einzigartig vergebenen Identifikatoren, nicht ein zweites Mal von einem entfernten Sensor bzw. Aktor durch das dortige Management zugewiesen wird.

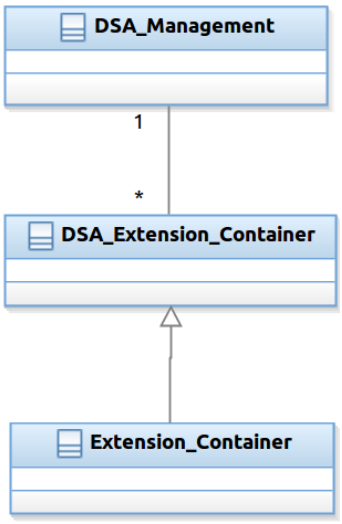


Abbildung 5.5.: Modell der Management Komponente

Abbildung 5.5. zeigt das neue Modell der Management-Komponente. Da davon auszugehen ist, dass eine spezielle Implementierung für Kontainer benötigt wird, die *DSA_Extensions* der entfernten Rahmenwerke verwaltet, wurde in diesem Modell zunächst eine Abstraktion der *Extension_Container* vorgenommen. *DSA_Extension_Container* ist dabei der abstrakte Kontainer

und *Extension_Container* eine spezielle Implementierung für die Verwaltung lokaler *DSA_Extensions*. Da anders als im Basis-Rahmenwerk mehr als ein *DSA_Extension_Container* existieren können muss, implementieren diese nicht mehr das Singleton-Pattern. Gleichzeitig wird aber auch nicht mehr der direkte Zugriff auf die *DSA_Extension_Container* von dem informatischen System erlaubt, sondern der Zugriff ist nur noch über die *DSA_Management* möglich.

Die durch Anforderung 3 verlangte Managementschnittstelle des Rahmenwerks wird durch die *DSA_Management*-Klasse gegeben. Diese ersetzt im neuen Modell die *DSA_Factory*-Klasse. So wie die *DSA_Factory* wurde das *DSA_Management* nach dem Singleton-Pattern entworfen. Diese Limitierung auf maximal eine Instanz des Managements ist notwendig, damit weiterhin ein klarer Punkt exisitert, über den die Sensoren erstellt und verwaltet werden. Diese Klasse ermöglich mit Hilfe von Assoziationen zu allen *DSA_Extension_Containern* den Zugriff auf die Verwaltungsstruktur für die Sensoren und Aktoren.

5.2.3. Kommunikations-Komponente

In der Kommunikations-Komponente werden die Strukturen zusammengefasst, die den Zugriff auf entfernte Sensoren und Aktoren erlauben, die ebenfalls mit den Strukturen des Rahmenwerks realisiert wurden. Da das Ziel, das Rahmenwerk auch auf anderen Systemen auszuführen und mit dem lokalen Rahmenwerk zu verbinden, bei der Entwicklung des Basis-Rahmenwerks noch nicht verfolgt wurde, existierten für die Kommunikations-Komponente noch keine Strukturen.

Für die Verwendung eines entfernten Sensors oder Aktors auf dem lokalen System muss die *DSA_Extension* des Sensors oder Aktors auch lokal vorhanden sein. Da entfernte Sensoren und Aktoren an das entsprechende System gebunden sind, ist es unmöglich, über die softwareseitige Replikation der *DSA_Extension* auf dem lokalen System den Sensor oder Aktor bereitzustellen. Die Notwendigkeit eines Replikats des entsprechenden Objekts kann mit Hilfe des *Proxy*-Patterns [GJHV11, S. 207 ff.] umgangen werden. In Strukturen, die nach diesem Muster entworfen sind, wird statt einer Kopie

des Objekts ein *Proxy* angelegt. Dieser Proxy besitzt dieselbe Struktur wie das *reale* Objekt, jedoch wird innerhalb dieser Struktur eine Verbindung zu dem eigentlichen Objekt hergestellt. Wie die Verbindung zwischen Proxy und realen Objekt zustande kommt, ist dabei jeweils von der Anwendung abhängig.

Ohne die Berücksichtigung entfernter Sensoren und Aktoren war eine solche Verbindung bisher direkt über Assoziationen zu den Objekten möglich. Da entfernte Systeme in Software vollständig voneinander getrennt sind, muss auch diese Verbindung über ein Kommunikationsmedium erfolgen können. Ein solches Kommunikationsmedium kann unter anderem eine TCP-Verbindung oder auch der i²c-Bus [NXP] sein. So wie die zukünftige Entwicklung bei den Sensoren und Aktoren ist heute ebensowenig absehbar, welche Protokolle und Kommunikationsmedien in Zukunft entwickelt und verwendet werden. Deshalb ist es sinnvoll, die durch Anforderung A geforderte Flexibilität der Kommunikationstechniken und Protokolle für Sensoren und Aktoren auch auf die Kommunikationsmedien und Protokolle für den Austausch zwischen verschieden Instanzen des Rahmenwerks auszuweiten. Für die Entwicklung der Kommunikations-Komponente muss das Kommunikationsmedium bzw. die Verbindungen, die die Datenübertragung mit diesem Medium realisieren, als *Black-Box* angesehen werden, an die durch die zu entwickelnde Struktur möglichst keine einschränkenden Anforderungen gestellt werden sollten.

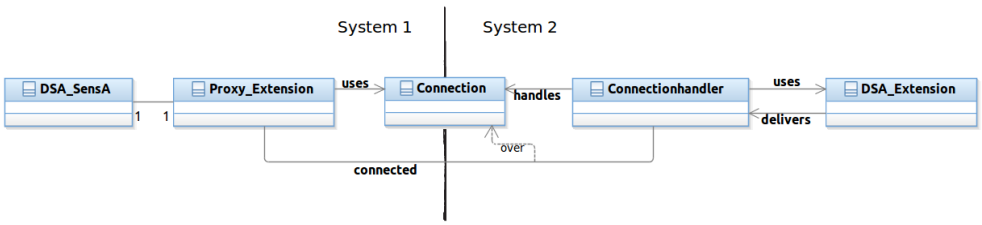


Abbildung 5.6.: Erster Entwurf des Modells einer angebundenen entfernten *DSA_Extension*

Da eine solche Verbindung nur ermöglicht, Daten zu übertragen, ist es erforderlich, die Funktionsaufrufe in übertragbare Daten umzuwandeln.

Diese Daten müssen am anderen Ende der Verbindung ausgewertet und wieder in die entsprechenden Aufrufe überführt werden. Für eventuelle Rückgabewerte muss auf die gleiche Weise, jedoch in umgekehrter Reihenfolge vorgegangen werden. Abbildung 5.6. zeigt das bisherige Modell einer entfernten *DSA_Extension*, die vom lokalen Rahmenwerk verwendet wird. Die *Proxy_Extension* implementiert die abstrakte Klasse *DSA_Extension*. Die Funktionsaufrufe dieser Klasse werden dabei als auswertbare Daten über Instanzen der *Connection*-Klasse an das entfernte System gesendet. Aufgrund der Black-Box-Darstellung für das Kommunikationsmedium und deren Verbindungen bleibt *Connection* hier erstmal undefiniert. Auf dem entfernten System werden die Daten von dem *Connectionhandler* ausgewertet und der entsprechende Aufruf ausgeführt.

In Abschnitt 5.2.1. wurde durch die Verwendung des Observer-Patterns ermöglicht, dass Daten von den *DSA_Extension* selbstständig an das informatische System übergegeben werden können. Das in Abbildung 5.6. dargestellte Modell ermöglicht bisher jedoch nur eine Übertragung, die vom lokalen System initiiert wurde. Deshalb muss das Proxy-Pattern in umgekehrter Richtung nochmal auf den Observer der *Proxy_Extension* angewandt werden. Abbildung 5.7. zeigt das Modell einer solchen Verbindung. Mit Hilfe der *Remote_Observer*-Klasse wird in diesem Modell die Observer-Schnittstelle der *Proxy_Extension* auf dem entfernten System erstellt. So wie bereits die *Proxy_Extensions* selbst, übernehmen die *Remote_Observer* die Aufgabe, Funktionsaufrufe des Observers auf dem entfernten System in Daten umzuwandeln, die anschließend über die *Connection* an das lokale System übertragen werden können. Auf dem lokalen System wird nun auch ein *Connectionhandler* benötigt, der eingehende Daten interpretiert und die entsprechenden Funktionen ausführt. Ob es sich bei der *Connection* um dieselbe handelt, die auch für die Verbindung der *Proxy_Extension* mit den entfernten *DSA_Extension* verwendet wird, muss dabei vom Kommunikationsmedium abhängig sein.

Der Aufbau einer Verbindung zu einer entfernten *DSA_Extension* ist jedoch nur ein Teil der Funktionalität, die durch die Kommunkations-Komponente gegeben werden muss. Der andere Teil ist der Zugriff auf die Managementschnittstelle des entfernten Rahmenwerks. Da davon ausgegangen werden

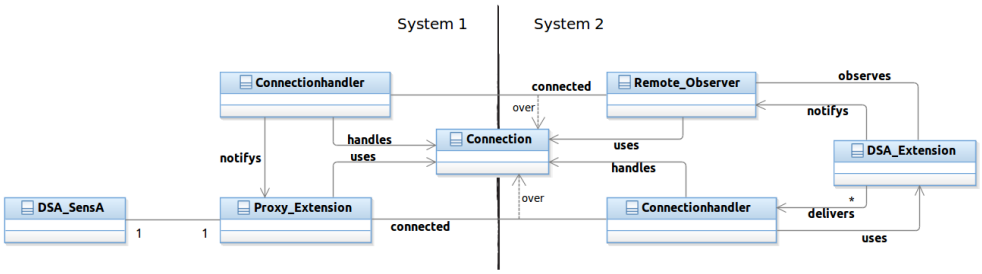


Abbildung 5.7.: Zweiter Entwurf des Modells einer angebundenen entfernten *DSA_Extension*

muss, dass sich der Zugriff auf das entfernte System auf das dort ausgeführte Rahmenwerk beschränkt, ist es unerlässlich die Managementschnittstelle des entfernten Rahmenwerks auch über die Kommunikations-Komponente ansprechen zu können. Der Zugriff kann jedoch auf die Auskünfte über die angeschlossenen *DSA_Extensions* und deren Umkonfigurierung eingeschränkt werden. Diese Einschränkung ist erforderlich, da für die Erstellung neuer *DSA_Extensions* auf dem entfernten System *Extension_Configurations* für die entsprechenden Sensoren und Aktoren vorhanden sein müssen. Solche Konfigurationen können aufgrund der eventuell unterschiedlichen Beschaffenheiten der Systeme, auf denen das Rahmenwerk ausgeführt wird, nicht einheitlich über ein Kommunikationsmedium so übertragen werden, dass diese ohne eine spezielle Aufbereitung auf dem System verwendet werden können.

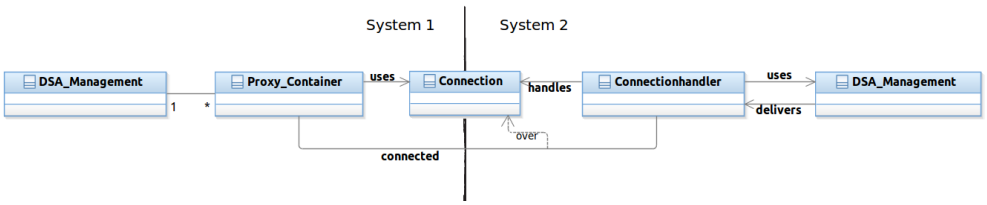


Abbildung 5.8.: Modell für den Zugriff auf das entfernte *DSA_Management*

Wie in Abschnitt 5.2.2. beschrieben, sollen die *DSA_Extensions* der externen Rahmenwerke in einem eigenen *Extension_Container* verwaltet werden. Durch die dort vorgenommene Abstraktion des *DSA_Extension_Containers* kann eine von *Extension_Container* verschiedene Implementierung erstellt werden, über die neben der Verwaltung der externen *Extensions* auch die

Zugriffe auf das Management erfolgen. Diese Implementierung muss so wie auch die *Proxy_Extension* die Funktionsaufrufe an das externe Management in übertragbare Daten umwandeln.

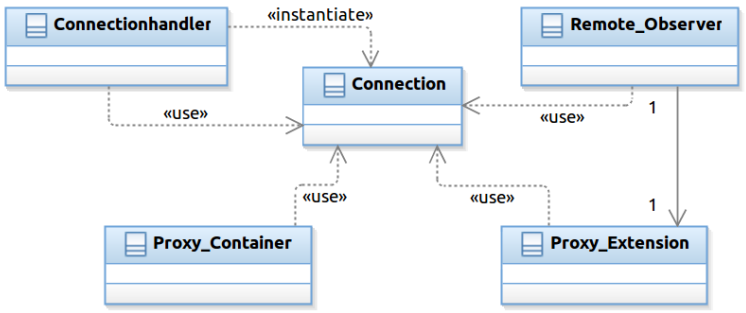


Abbildung 5.9.: Erster Entwurf der Kommunikations-Komponente

Das Modell für die Einbindungen externer *DSA_Extension* (Abbildung 5.7.) und *DSA_Management* (Abbildung 5.8.) muss nun vereinheitlicht werden. Abbildung 5.9. zeigt die Stuktur, die für den Zugriff auf das *DSA_Management* und die *DSA_Extensions* des entfernten Rahmenwerks benötigt wird. In dieser Struktur werden die Funktionsaufrufe der *Proxy_Container* und *Proxy_Extensions* über die *Connection* an das externe System geleitet. Diese Struktur realisiert die für das Proxy-Pattern benötigte Verbindung zwischen dem Proxy und dem realen Objekt über das Mediator-Pattern (siehe Abschnitt 5.2.1.). Der durch das Mediator-Pattern eingeführte Vermittler wird in diesem Modell durch die *Connectionhandler*-Klasse implementiert. Der *Connectionhandler* interpretiert die Daten, die über die bestehenden *Connection*-Instanzen gesendet wurden, und führt anschließend die entsprechenden Funktionsaufrufe des Managements oder der *Extension* aus.

In Abbildung 5.10. wird der Ablauf einer Verbindung zwischen einem Proxy und dem realen Objekt am Beispiel einer *Proxy_Extension* in Form eines Sequenzdiagramms gezeigt. Innerhalb dieses Ablaufes wurde davon ausgegangen, dass die *Connection* durch den *Connectionhandler* auf unspezifizierte Weise erstellt wird. Bisher wurde das Kommunikationsmedium bzw.

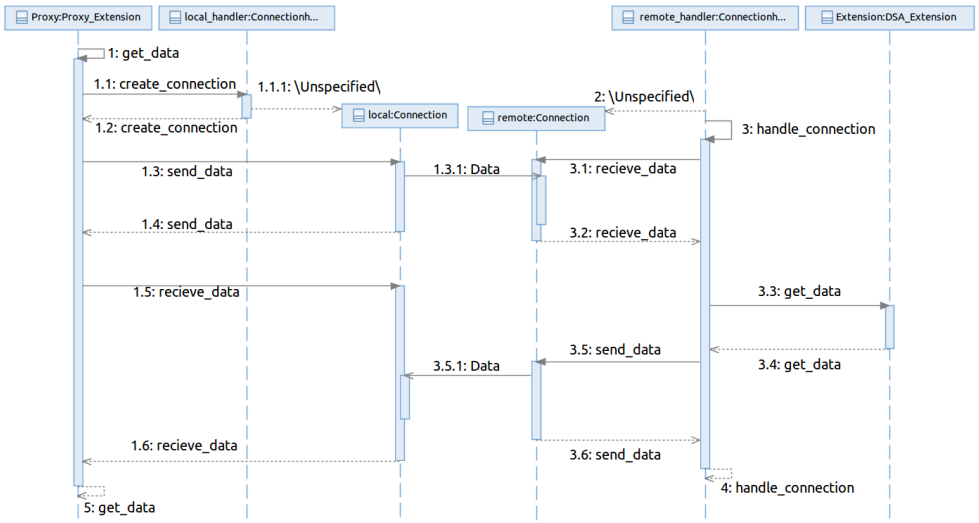


Abbildung 5.10.: Sequenzdiagramm der Verbindung einer *Proxy_Extension* zu einer entfernten *DSA_Extension*

die Verbindungen als Black-Box angesehen. Um weiterhin zu ermöglichen, dass das Rahmenwerk keine Informationen über spezifischen Kommunikationsmedien und Protokolle benötigt, muss durch das Rahmenwerk eine Schnittstelle für diese Black-Box vorgegeben werden.

Es muss somit, ähnlich wie in Abschnitt 5.1.2. über das Adapter-Pattern eine Abbildung der Schnittstelle dieser Black-Box auf die Schnittstelle des Rahmenwerks erfolgen. Hierzu ist es erforderlich, beim Verbindungsaufbau zwischen dem initiierenden und dem empfangenden Ende der Verbindung zu unterschieden. Auf der initiierenden Seite muss über die Schnittstelle eine Verbindung angefordert werden können, die anschließend durch einen *Proxy_Container* oder eine *Proxy_Extension* verwendbar ist. Die Black-Box muss auf der empfangenden Seite aktive Verbindungen erkennen und diese dem *Connectionhandler* zur Bearbeitung bereitstellen. Da unbekannt ist, wie am Ende der Übertragung mit den Verbindungen umgegangen wird, müssen diese wieder an die Black-Box zurückgegeben werden, die dann über das weitere Verfahren entscheidet.

Hier wird deutlich, dass die Black-Box im bisherigen Modell tatsächlich aus zwei voneinander abhängigen Teilen besteht. Den ersten Teil bildet die Verbindung selbst, die bisher über *Connection* dargestellt wurde. Der

zweite Teil ist der Manager, der abhängig von Kommunikationsmedium Verbindungen erstellen, annehmen und löschen bzw. verwalten kann. Deshalb muss das Modell der Kommunikations-Komponente um diesen zweiten, noch fehlenden Teil erweitert werden.

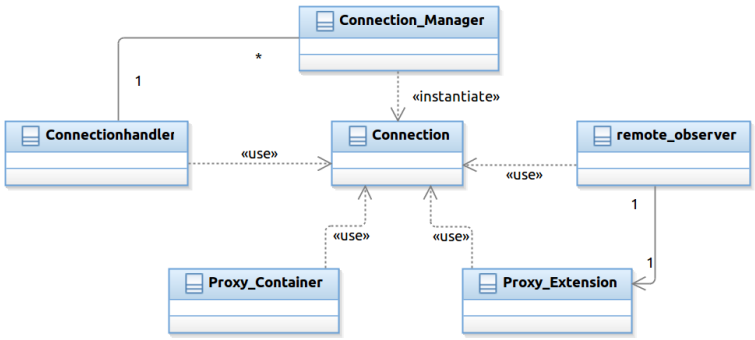


Abbildung 5.11.: Zweiter Entwurf der Kommunikations-Komponente

In Abbildung 5.11. wird der bisherige Entwurf der Struktur für die Kommunikations-Komponente gezeigt. In dieser Struktur greift der *Connectionhandler* auf die abstrakte Klasse *Connection_Manager* zu, um *Connections* zu erstellen. Durch die Abstraktion von *Connection_Manager* bildet diese Klasse den Adapter zu dem Management der Black-Box. Die durch diese Struktur erreichte Transparenz bezüglich des verwendeten Kommunikationsmediums und dessen Verbindungen führt jedoch zu dem Problem, dass bei der Verwendung mehrerer Verbindungsarten durch das Rahmenwerk nicht mehr nachvollzogen werden kann, welche *Connections* zu den verschiedenen *Connection_Managern* gehören. Um diese Zuordnung trotz der gewünschten Transparenz zu ermöglichen, bedarf es eines Identifikators, über den das Rahmenwerk von dem *Connection_Managern* eine *Connection* anfordern kann. Der *Connection_Manager* kann über diesen Identifikator klar erkennen, ob eine *Connection* das von ihm realisierte Kommunikationsmedium verwendet und somit kompatibel zu seinen Strukturen ist.

Da an diesem Punkt die bisher unspezifizierte Black-Box-Darstellung von *Connection* für eine Verwendung durch das Rahmenwerk nicht mehr ausreicht, muss diese Klasse hier genauer definitiert werden. Aufgrund der

am Anfang dieses Abschnittes beschriebenen Ausweitung von Anforderung A dürfen die Kommunikationsmedien und Protokolle, die für die Verbindung zwischen Instanzen des Rahmenwerks verwendet werden können, möglichst nicht eingeschränkt werden.

Damit die Definition der *Connection* nicht zu einer solchen Einschränkung führt, muss diese über die Beschaffenheit des umliegenen Systems erfolgen. Da über die *Connection* Daten gesendet werden sollen, ist es sinnvoll, diese zunächst genauer zu betrachten. In den Programmiersprachen C und C++, mit denen häufig Betriebssysteme realisiert werden [Tan09, S. 109], ist der kleinste Datentyp, der referenziert werden kann, ein *Byte*. Alle anderen Daten, die auf dem System verwendet werden, bestehen wiederum aus einer beliebigen Anzahl Bytes, auf die die Daten aufgeteilt werden. Daraus folgt für die Definition der *Connection*, dass es ausreichend ist, beliebig viele Bytes senden und empfangen zu können, um alle erdenklichen Daten zu übertragen. Da bereits das umliegende System mindestens erwartet, dass ein Kommunikationsmedium Bytes übertragen kann, hat diese Definition, wie durch Anforderung A gefordert, keine einschränkende Wirkung.

Weiterhin muss berücksichtigt werden, dass komplexere Kommunikationsmedien, wie zum Beispiel TCP, Daten in Paketen versenden. Jedes Paket benötigt neben den eigentlichen Daten Headerinformationen, die mit übertragen werden [Doy04, S. 78 ff.]. Die Headerinformationen beinhalten unter anderem Informationen über den Absender und Empfänger des Pakets. Da diese Header und auch eventuelle Kontrollmechanismen bei der Übertagung selbst Zeit beanspruchen, hat die Übertragung der Bytes der Daten Einfluss auf die zeitliche Effizenz der Übertragung. Deshalb ist es sinnvoll, Funktionen bereitzustellen, mit denen auch komplexeren Datentypen des Rahmenwerks direkt übertragen werden können. Da diese Funktionen nur durch einen Teil der Kommunikationsmedien sinnvoll verwendet werden können, muss eine Standardimplementation vorhanden sein, die die Übertragung intern über das Senden der Bytes löst. Somit ist trotz dieser Funktionen weiterhin Anforderung A erfüllt, da es optional ist, die bestehenden Implementationen durch für die Medien spezifische Implementationen auszutauschen.

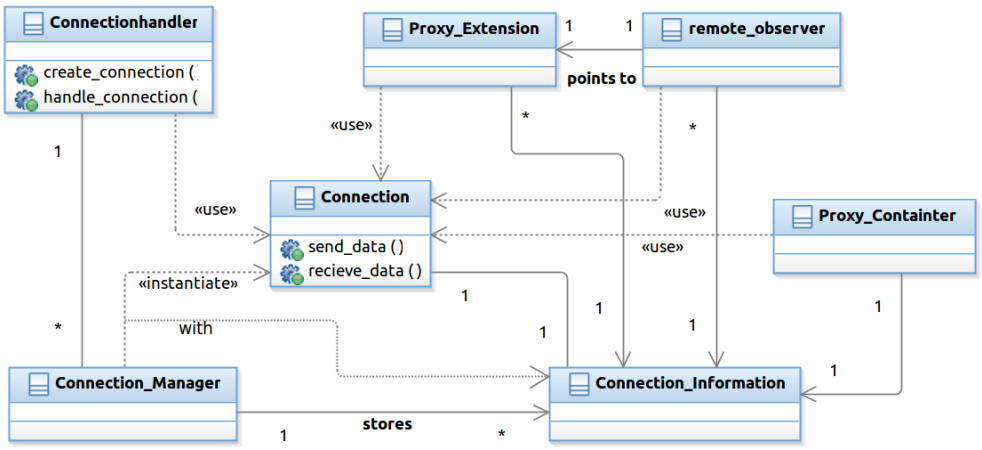


Abbildung 5.12.: Endgültiges Modell der Kommunikations-Komponente

In Abbildung 5.12. wird die endgültige Struktur der Kommunikations-Komponente gezeigt. Die Klassen *Connection*, *Connection_Manager* und *Connection_Information* realisieren die Schnittstelle zum Kommunikationsmedium. Bei diesen Klassen handelt es sich um abstrakte Klassen, die nach dem Template-Pattern entworfen sind. Durch die Implementation dieser Klassen können beliebige Kommunikationsmedien und Protokolle vom Rahmenwerk verwendet werden. Der *Connection_Manager* kann erkennen, ob eine *Connection_Information* mit dem implementierten Medium kompatibel ist. Durch Angabe einer solchen *Connection_Information* können über den *Connection_Manager Connections* erstellt werden. *Connection_Information* helfen dabei nicht nur bei der Zuordnung von *Connections* zu den *Connection_Managern*, sondern sie enthalten auch Addressinformationen, die für die Erstellung der *Connections* benötigt werden.

5.3. Überblick über die entworfene Architektur

Die Architektur des erweiterten Rahmenwerks setzt sich aus den drei Komponenten zusammen, deren Entwicklung im letzten Abschnitt beschrieben wurde. Abbildung 5.13. gibt einen Überblick über die gesamte Struktur und zeigt auch, wie die einzelnen Komponenten verbunden sind. Die Komponenten wurden dabei so ausgewählt, dass die Ziele dieser Arbeit und somit auch die Anforderungen auf diese aufgeteilt werden können. Die Sensor/Aktor- und die Management-Komponente bauen dabei auf den Strukturen des Basis-Rahmenwerks auf. Durch dieses Vorgehen wurde sichergestellt, dass

das erweiterte Rahmenwerk in seiner Funktionalität vollständig kompatibel zu dem Basis-Rahmenwerk ist und somit auch die Ziele und Anforderungen des Basis-Rahmenwerks weiterhin erfüllt bleiben.

Über diese Teilstrukturen wie dargestellt alle Anforderungen erfüllt, die in Kapitel 4 erhoben werden.

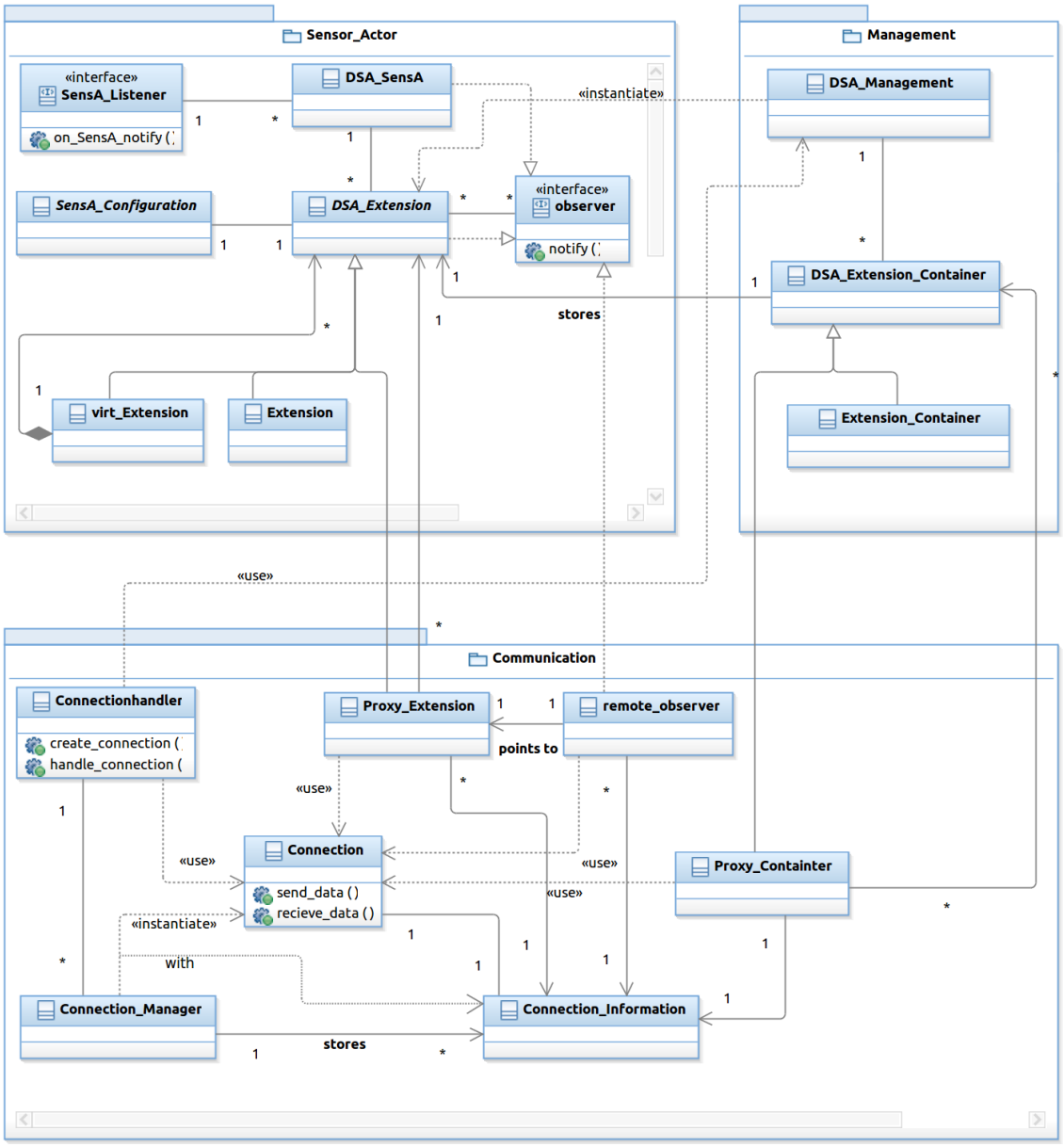


Abbildung 5.13.: Struktur des Rahmenwerks

6. Versuchsaufbau zur Validierung

Ziel dieses Versuchs ist die Validierung der durch die Erweiterung hinzugekommenen Funktionalitäten des Rahmenwerks. Insbesondere soll die beabsichtigte Einsparung bei der Anzahl der Kommunikationsvorgänge, die durch die neue Eventstruktur ermöglicht wird, nachgewiesen werden. Weiterhin dient der Versuch auch als Funktionstest für das entwickelte Rahmenwerk.

Damit durch den Versuchsaufbau gezeigt werden kann, dass die Zielsetzungen dieser Arbeit durch die Erweiterung des DSA-Rahmenwerks verwirklicht werden, muss er folgende Bedingungen erfüllen:

A Zur Erfüllung von Ziel 1 müssen die Sensoren selbstständig relevante Messdaten an das System senden.

B Zur Erfüllung von Ziel 2 müssen Sensoren und Aktoren mit dem Rahmenwerk verbunden werden, die unterschiedliche Schnittstellen bereitstellen.

C Zur Erfüllung von Ziel 3 müssen die Sensoren und Aktoren über die Managementschnittstelle des Rahmenwerks verwaltet werden.

D Zur Erfüllung von Ziel 4 muss das Rahmenwerk, das direkt durch die Versuchsanwendung genutzt wird, die Sensoren und Aktoren verwenden, die in einem externen Rahmenwerk eingebunden sind.

E Der Versuchsaufbau muss so gestaltet sein, dass er einen Vergleich zum Basis-Rahmenwerk ermöglicht.

Da die Erweiterung des Rahmenwerks so implementiert wurde, dass die Kompatibilität zum Basis-Rahmenwerk besteht ist (siehe Abschnitt 5.3.), können die Forderungen des Basis-Rahmenwerks bei der Validierung der Erweiterung vernachlässigt werden.

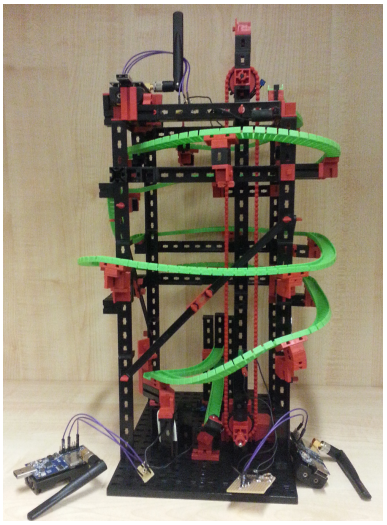


Abbildung 6.1.: Komplettansicht der Murmelbahn

6.1. Regelung einer Murmelbahn

Um Bedingung F zu erfüllen, wurde als Versuchsaufbau in dieser Arbeit ebenfalls eine Murmelbahn geregelt. Diese wurde mit zwei Sensoren und einem Aktor ausgestattet, die eine Regelung durch das informatische System erlauben. Abbildung 6.2 und 6.3 geben detailierte Teilansichten der in Abbildung 6.1 gezeigten Murmelbahn. Diese Murmelbahn besteht aus einer Laufbahn, an deren Enden Infrarot-Lichtschranken angebracht wurden, und einem Förderband, über das die Murmel wieder zum oberen Ende der Laufbahn befördert werden kann. Das Förderband wird mittels eines Motors angetrieben.

Durch die Verwendung von Lichtschranken am Anfang und Ende der Laufbahn entstehen zwei Kontrollpunkte, die durch ein informatisches System ausgewertet werden können. Der Kontrollpunkt am unteren Ende der Laufbahn wird durch Sensor S1 überwacht. Der zweite Kontrollpunkt befindet sich am oben am Startpunkt der Laufbahn und wird durch Sensor S2 ausgewertet.
Die Sensoren und Aktoren der Murmelbahn werden durch Sensorknoten vom Typ *MTM-CM5000MSP* [Max] der Firma *MAXFOR Technology Inc.* angesteuert. Diese sind dem Entwurf des *TelosB/TMoteSky* [Tel] der Universität Berkley nachempfunden. Als Microcontroller verwenden die Sensorknoten den TI MSP-430. Diese besitzen 10 kByte Arbeitsspeicher, 48

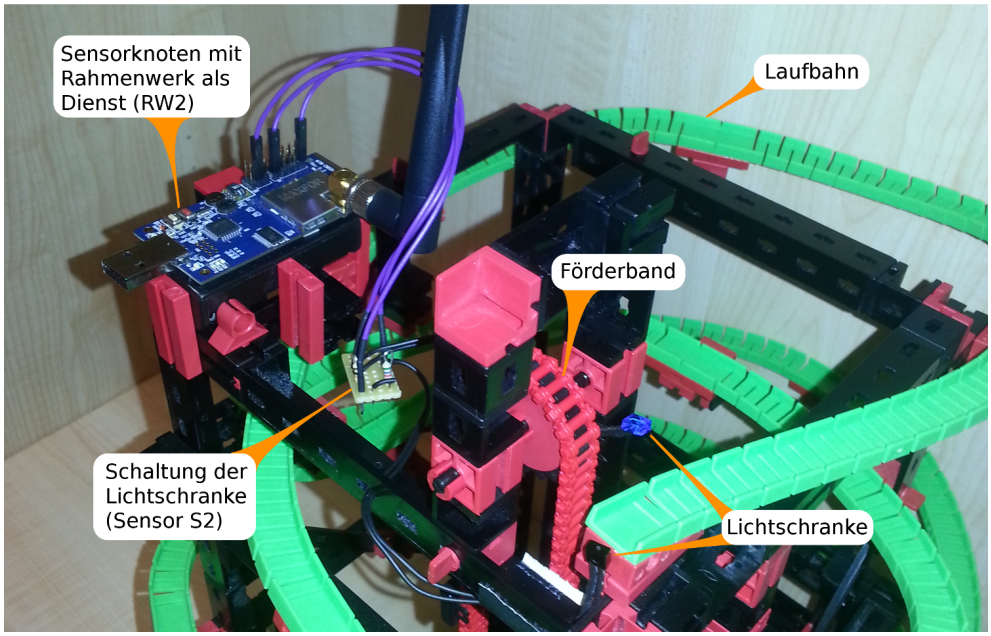


Abbildung 6.2.: Obere Teilansicht der Murmelbahn

kByte Flashspeicher und 1 MByte externen Speicher. Ein IEEE 802.15.4-konformes Funkmodul [I3E] erlaubt die drahtlose Kommunikation zwischen solchen Sensorknoten. Über 16 Pins können die Sensorknoten Sensor- und Aktorhardware ansteuern. Gängige Betriebssysteme für diese Sensorknoten sind *TinyOS* und *Contiki-OS* [con]. Contiki-OS bietet über sogenannte *Protothreads* ein Event-gesteuertes System, über das Sensoren und Aktoren leicht umgesetzt werden können. Im Rahmen dieses Versuchs wird deshalb Contiki-OS zur Implementation der Sensorknoten verwendet.

Die verwendeten Inforarot-Lichtschranken *K153P* [tem] der Firma *TEMIC TELEFUNKEN* sowie der an der Murmelbahn angebrachte Motor, können nicht direkt durch die Sensorknoten verwendet werden. Die Kompatibilität zwischen Sensorknoten und den Lichtschranken bzw. dem Motor wird durch Zwischenschaltung einer entsprechenden Schaltung erreicht.

Der Versuchsaufbau wird über ein Hauptsystem und drei sekundäre Systeme realisiert. Dabei werden die Sensoren und der Aktor an jeweils einem der sekundären Systeme (im Folgenden RW1, RW2 und RW3 genannt) angeschlossen. Diese drei Systeme erlauben durch die Ausführung des Rahmenwerks als Dienst eine externe Verwendung der angeschlossenen

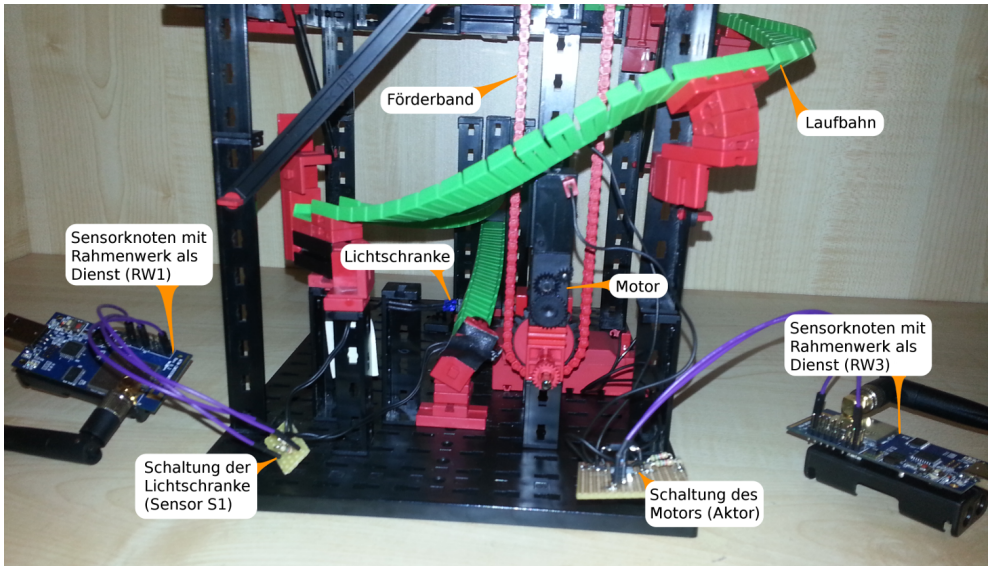


Abbildung 6.3.: Untere Teilansicht der Murmelbahn

Sensoren und Aktoren. Das Hauptsystem führt die Sensor/Aktor-Anwendung aus. Hier wird das Rahmenwerk genutzt, um die Sensoren und Aktoren der externen Rahmenwerke RW1, RW2 und RW3 anzusteuern, wodurch Bedingung D erfüllt wird.

Anders als im Versuchsaufbau des Basis-Rahmenwerks [Kra13, S. 25 ff.], in dem die Sensorknoten die Sensorschnittstelle implementieren, werden in diesem Versuchsaufbau die Sensorknoten als entfernte Rahmenwerke verwendet. Dazu wurde eine angepasste Version des Rahmenwerks auf diesen Knoten implementiert, die die gleiche Schnittstelle wie das normale Rahmenwerk als externer Dienst anbietet.

6.2. Beschreibung der Regelungssoftware

Der Versuchsaufbau wird auf dem Hauptsystem durch eine Regelungssoftware betrieben. Abbildung 6.4 zeigt den Zustandsautomaten, der durch die Regelungssoftware implementiert wurde.

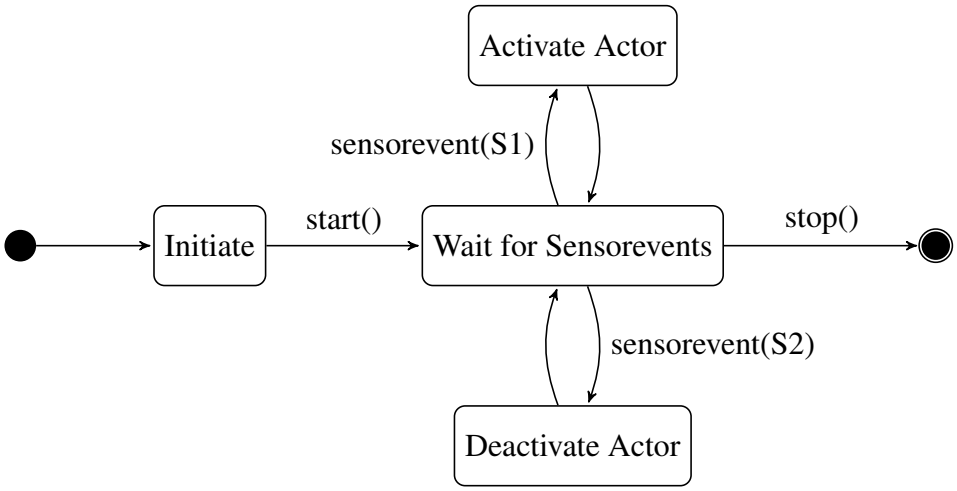


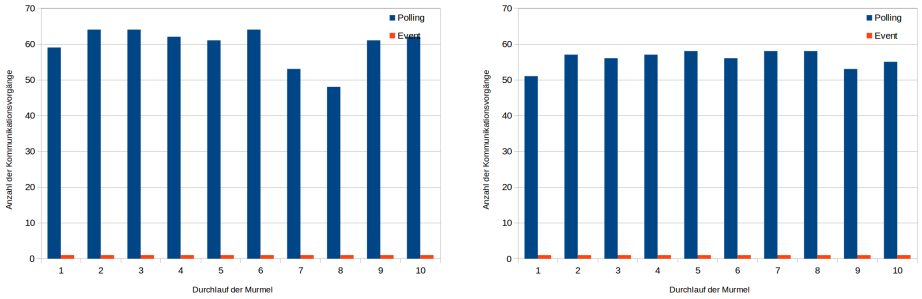
Abbildung 6.4.: Modell des Regelungsautomaten

Die hinzugekommene Eventstruktur erlaubt dabei auf das Polling der Sensordaten zu verzichten, da die Sensoren selbstständig relevante Messereignisse melden, wodurch Bedingung A erfüllt ist. Bei der Initialisierung des Versuchs werden die entfernten Rahmenwerke RW1, RW2 und RW3 eingebunden (Bedingung D). Auf die über diese Rahmenwerke verwalteten Sensoren und Aktoren wird anschließend über die Managementschnittstelle des Rahmenwerks zugegriffen (Bedingung C).

Damit ein direkter Vergleich zwischen der umgesetzten Eventstruktur und der im Basis-Rahmenwerk zum Abruf der Sensordaten verwendeten Pollingstrategie möglich ist, muss ein zweiter Zustandsautomat implementiert werden, der diese Pollingstrategie verwendet.

6.3. Versuchsdurchführung

Der Versuch wurde mit beiden Automaten über einen Zeitraum von jeweils 10 Durchläufen der Murmel ausgeführt. Durch Beobachtung der beiden Ausführungen zeigt sich die Funktionsfähigkeit beider Automaten. Es lassen sich visuell keine physischen Unterschiede im Ablauf der Ausführungsprozesse der Automaten feststellen. Dieser verläuft in beiden Fällen reibungslos und ohne Störungen. Die Schaltung des Motors erfolgt unmittelbar nachdem die Murmel Sensor S1 bzw. S2 passiert.



(a) Sensor S1 (b) Sensor S2

Abbildung 6.5.: Kommunikationsvorgänge des Regelungsautomaten

Zu jedem Durchlauf wurde durch die Regelungssoftware gezählt, wie viele Kommunikationsvorgänge zu den Sensoren nötig waren, bevor der Wert des jeweiligen Sensors zu einer Zusandsänderung in den entsprechenden Automaten führte. In Abbildung Abbildung 6.5. werden die für jeden Durchlauf benötigten Kommunikationen der Sensoren gegenüber gestellt. Die Auswertung dieser Messergebnisse zeigt, dass der Vergleichs-Automat, der die Sensoren über die Pollingstrategie ausliest, durchschnittlich $59,9$ Kommunikationsvorgänge für Sensor S1 und $55,9$ Kommunikationsvorgänge für Sensor S2 benötigt. Die Auswertung des Automaten, der die Eventstruktur des erweiterten Rahmenwerks nutzt, zeigt das für Sensor S1 und Sensor S2 nur jeweils ein Kommunikationsvorgang durchgeführt wird. Damit wird die Dimension der durch die Erweiterung des Rahmenwerkes erreichten Einsparung bei der Anzahl der innerhalb des Prozesses benötigten Kommunikationsvorgänge deutlich.

6.4. Ergebnis des Versuchs

Aus den Versuchsdurchführungen lassen sich im Wesentlichen drei Ergebnisse ableiten.
Zunächst zeigt die erfolgreiche Umsetzung der informatischen Systeme, die für die Realisierung der beiden Regelungsautomaten entwickelt wurden, die Verwendbarkeit des Rahmenwerks für solche Anwendungen.

Die Durchführung der Versuchsabläufe bestätigt die Funktionalität der

internen Strukturen des erweiterten Rahmenwerks sowohl für Automaten, die die Sensoren über die Pollingstrategie auslesen, als auch für Automaten, die Sensorereignisse über die Eventstruktur erwarten. Durch die Verwendung von entfernten Rahmenwerken konnte gleichzeitig die entwickelte Kommunikationsstruktur validiert werden, die für die Kommunikation zwischen den Rahmenwerken genutzt wird.

Darüber hinaus zeigt der Vergleich der Messergebnisse der beiden Automaten, die sich lediglich durch die verwendete Kommunikationsart unterscheiden, dass die Verwendung der neuen Eventstruktur die Anzahl der benötigten Kommunikationen pro Sensor nicht nur signifikant, sondern sogar konstant auf das mögliche Minimum reduziert.

Damit ist gezeigt, dass die Erweitung des Rahmenwerks die Zielsetzungen dieser Arbeit erfüllt.

7. Zusammenfassung und Ausblick

Das DSA-Rahmenwerk ist ein Werkzeug, mit dem die Realisierung informatischer Systeme erleichtert wird, die zur Überwachung und Beeinflussung der Umgebung Sensoren und Aktoren verwenden. Dieses Rahmenwerk wurde mit dem Ziel entwickelt, zur Laufzeit des informatischen Systems dynamische Anpassungen der verwendeten Sensoren und Aktoren zu ermöglichen. Durch diese Anpassungsfähigkeit können die Sensoren und Aktoren leicht ausgetauscht oder optimiert werden, ohne dass eine Anpassung der Implementierung des informatischen Systems notwendig wird.

In dieser Arbeit wurde das bestehende DSA-Rahmenwerk weiterentwickelt. Die Erweiterungen, die im Rahmen der Weiterentwicklung hinzugekommen sind, umfassen eine Eventstruktur und eine Ausweitung des Rahmenwerks über die Systemgrenzen hinaus.

Die vom ursprünglichen Rahmenwerk vorgesehene Pollingstrategie für die Auswertung der Sensoren, führt zu einer hohen Anzahl an Datenübertragungen zwischen den Sensoren und dem System, die bei energieintensiven Übertragungsmedien hohe Kosten verursachen. Das erweiterte Rahmenwerk führt deswegen eine Eventstruktur ein, die das selbstständige Übermitteln von Daten durch die Sensoren an das System ermöglicht. Diese Eventstruktur, erlaubt eine Auslagerung von Auswertungsprozessen zu den Sensoren. Dadurch wird die Anzahl der Datenübertragungen auf die Häufigkeit der für das System relevanten Werte reduziert.

Da Sensoren und Aktoren häufig über Sensorknoten angesteuert werden, die auch als informatische Systeme betrachtet werden müssen, wurde eine Ausweitung des Rahmenwerks über die Systemgrenzen hinaus vorgenommen. Diese Ausweitung erfolgt über die Nutzung des Rahmenwerks auf den entfernten informatischen Systemen. Da auf diesen Systemen mit Hilfe des Rahmenwerks dieselbe Grundlage zur Verwendung von Sensoren

und Aktoren geschaffen wurde, kann diese Grundlage genutzt werden, um den Zugriff auf die Sensoren und Aktoren der anderen Systeme zu ermöglichen. Zur Umsetzung der Kommunikation zwischen den verschiedenen Rahmenwerks-Instanzen wurde eine Kommunikationsstruktur entwickelt. Diese Struktur hält offen, welche konkrete Verbindungstechniken realisiert werden können. Diese Flexibilität ist notwendig, damit die Kompatibilität des Rahmenwerks auch zu Verbindungstechniken, die heute noch nicht vorhersehbar sind, gewährleistet ist.

Im Versuchsaufbau einer geregelten Murmelbahn wurde das Rahmenwerk dazu genutzt, Sensoren und Aktoren von entfernten Systemen einzubinden, die dort ebenfalls mit Hilfe des Rahmenwerks realisiert wurden. Durch die Umstellung vom Polling der Sensordaten auf eine Eventstruktur ist es gelungen, die Anzahl der Datenübertragungsvorgänge, die über das Funkmodul der Sensorknoten erfolgen, auf diejenigen Übertragungen einzuschränken, die für die Anwendung relevante Daten enthalten. Mit diesem Versuch, konnte auch gezeigt werden, dass die Zielsetzungen dieser Arbeit mit der entwickelten Erweiterung des Rahmenwerks erreicht wurden.

Ausblick Das in dieser Arbeit entwickelte, erweiterte DSA-Rahmenwerk ist die Version 2.0.0. Es setzt die Zielsetzungen für die Erweiterung um, ohne die Realsisierung der ursprünglichen Forderungen des Basis-Rahmenwerks zu beeinträchtigen. Neben der umgesetzten Eventstruktur bestehen noch weitere Ansätze, über die Energie gespart werden kann.

Einer dieser Ansätze ist eine Komprimierung der zu sendenden Messdaten. In dem Paper *Power Management in Wireless Sensor Networks: Challenges and Solutions* [AT13] wird vorgeschlagen, mit Hilfe von *Fuzzy-Transformationen* Messreihen auf einige Messpunkte und eine Funktion zu reduzieren. Im DSA-Rahmenwerk wird bei einem Abruf der gespeicherten Messreihe jeder Messwert einzeln übertragen. Hier könnten mit solchen Transformationen Übertragungsvorgänge und somit auch Energie eingespart werden.

Das Paper *What Does Model-Driven Data Acquisition Really Achieve*

in Wireless Sensor Networks? beschäftigt sich mit dem Prinzip der *Derivate-Based Predictions* [RCM+12]. Beim Einsatz dieses Prinzips, wird eine Sensorfunktion aus den gemessenen Werten des Sensors abgeleitet. Diese Funktion wird dann statt der Messereignisse als Event an die Systeme übertragen. Da die Systeme anschließend die Messwerte selbst über die Funktion berechnen können, kann auf das Senden einzelner Messereignisse verzichtet werden. Der Sensor muss allerdings weiterhin die Differenz zwischen den berechneten und den tatsächlichen Messwerten ermitteln. Über diese Differenz kann der Sensor ermitteln, ob die abgeleitete Funktion zu ungenau wird und entsprechend eine neue Funktion ableiten. Durch dieses Prinzip entstehen hyride Sensoren, die aus einem Hardwaresensor, der die Sensorfunktion ableitet, und einem Softwaresensor bestehen, der mit dieser Funktion die Werte berechnet. Die Möglichkeit solche hybriden Sensoren verwenden zu können, wäre auch für das DSA-Rahmenwerk wünschenswert.

Literaturverzeichnis

[AT13] ABDELAAL, Mohamed ; THEEL, Oliver: *Power Management in Wireless Sensor Networks: Challenges and Solutions*. 2013

[con] *Contiki-OS*. `http://www.contiki-os.org/`. – Zuletzt besucht: 28.08.2013

[Doy04] DOYLE, Jeff: *Routing TCP/IP*. Cisco Press, 2004. – Volume I, CCIE Professional Development. – ISBN 1–57870–041–8

[GJHV11] GAMMA, Erich ; JOHNSON, Ralph ; HELM, Richard ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2011. – ISBN 0–201–63361–2

[I3E] *IEEE 802.15.4 Standard*. `http://standards.ieee.org/getieee802/download/802.15.4-2011.pdf`. – Zuletzt besucht: 02.12.2013

[ISO01] *ISO/IEC 9126-1:2001*. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749`. Version: 2001

[Kra13] KRAYENBORG, Jens: *DSA : Dynamische Verwendung von Sensor- und Aktuatorknoten (HW/SW) zur Verbrauchsoptimierung von WSAN*, Universität Oldenburg, Bachelorarbeit, 2013

[Max] *Product Info - MTM-CM5000-MSP*. `http://maxfor.co.kr/eng/en_sub5_1_1.html`. – Zuletzt besucht: 03.12.2013

[NXP] NXP SEMICONDUCTORS: *I2C-bus specification and user manual*. `http://www.nxp.com/documents/other/UM10204_v5.pdf`. – Zuletzt besucht: 07.12.2013

[RCM+12] RAZA, Usman ; CAMERRA, Alessandro ; MURPHY, Amy L. ; PALPANAS, Themis ; PICCO, Gian P.: What does model-driven data acquisition really achieve in wireless sensor networks? In: GIORDANO, Silvia (Hrsg.) ; LANGHEINRICH, Marc (Hrsg.) ; SCHMIDT, Albrecht (Hrsg.): *PerCom*, IEEE. – ISBN 978–1–4673–0256–2, 85-94

[Tan09] TANNENBAUM, Andrew S.: *Moderne Betriebssysteme*. Pearson Education, 2009. – ISBN 978–3–8273–7342–7

[Tel] *Open WSN - TelosB*. `http://openwsn.berkeley.edu/display/OW/TelosB`. – Zuletzt besucht: 03.12.2013

[tem] *Temic K153P*. `http://www.pollin.de/shop/downloads/D120592D.PDF`. – Zuletzt besucht: 04.12.2013

A. Benutzeranleitung zum DSA-Rahmenwerk

In diesem Anhang wird das bestehende Benutzerhandbuch zur Verwendung des Basis-Rahmenwerks auf das erweiterte DSA-Rahmenwerk angepasst. Die Änderungen wurden dabei durch blaufarbige Schrift erkennbar gemacht.

A.1. Installation des Rahmenwerks

A.1.1. Voraussetzungen

Die Installation erfordert einen Rechner mit Linux System und das Programm `cmake`, mit dem die zur Übersetzung notwendige Makefile erzeugt wird. Dieses Programm kann für die meissten Linux Distributionen über den Paketmanager bezogen werden.

A.1.2. Installation

Um die Installation vorzubereiten, muss ein beliebig benannter Ordner erstellt werden, in den die Dateien des Ortes `<CDROOT>/QuellcodeDSA/` kopiert werden. In diesem neuen Ordner wird durch die beiden Anweisungen

```
cmake .
make
```

die dynamisch linkbare Bibliothek `libDSA.so` erstellt. Diese muss nur noch in den Ortner des Systems kopiert werden, in dem sich die dynamisch linkbaren Bibliotheken befinden. In den meissten Linux Distributionen handelt es sich um den Ordner `/usr/lib/`. Anschliessend müssen alle Dateien mit der Dateiendung `.h` in den Includeordner des Systems, für gewöhnlich `/usr/include/`, kopiert werden. Um die Übersicht im eigenen System zu wahren, empfielt es sich aber in diesem Ordner einen Unterordner anzulegen und die Headerdateien dort hinein zu kopieren. Sind die Kopiervorgänge abgeschlossen, dann muss die Bibliothek vom System geladen werden. Dies geschieht mit der Anweisung

```
ldconfig
```

Für die Kopiervorgänge und das laden der Bibliothek werden in der Regel Administratorrechte benötigt.

A.2. Verwendung des Rahmenwerks

Die folgenden Abschnitte geben eine einfache Einführung in die Verwendung des DSA-Rahmenwerks. Sie erläutern nacheinander die Erstellung eines *DSA_SensA* und einer *DSA_Extension*. Anschließend wird erläutert, wie Verbindungen zwischen diesen erzeugt und beendet werden. Darauf folgen Abschnitte mit Erklärungen zu Abfrage von Sensordaten, und wie Eigenschaftswerte verändert werden. Der letzte Abschnitt befasst sich mit Erklärungen zur Erstellung eigener *Connection_Manager*, mit der zugehörigen *Connection* und *Connection_Information*, und *Extension_Configuration*-Klassen.

A.2.1. Erstellung eines logischen *DSA_SensA*

Instanzen der *DSA_SensA* Klasse dienen als logische Schnittstelle zu spezifischen Sensoren und Aktuatoren. Sämtliche Zugriffe auf diese Sensoren und Aktuatoren verlaufen unter Benutzung des DSA-Rahmenwerks über die Nutzung von *DSA_SensA* Instanzen. Eine Instanz dieser Klasse wird durch den Aufruf einer Fabrikmethode der *DSA_Management* erzeugt. Wie dies geschieht, zeigt folgendes Listing.

```
1 #include <dsa_sensa.h>
2 #include <dsa_management.h>
3
4 class X
5 {
6  public:
7  DSA_SensA* s;
8
9  int main(void)
10  {
11  s = DSA_Management::get_instance()->get_DSA_SensA();
12  }
13  }
```

Da es sich bei der *DSA_Management* um eine als Singleton implementierte Klasse handelt, muss auf ihre Instanz durch die Anweisung

`DSA\_Management::get\_instance()`

zugegriffen werden. Mit der Anweisung `get_DSA_SensA()` wird eine Instanz der Klasse *DSA_SensA* erzeugt und die Referenz zurückgegeben. Diese muss in einer Pointervariablen gespeichert werden, damit sie später verwendet werden kann. Eine Erzeugung durch das Management ist nötig, damit jede erzeugte *DSA_SensA* Instanz eine eindeutige Identifikationsnummer erhält.

A.2.2. Erstellung einer DSA_Extension

DSA_SensA Instanzen alleine sind nicht fähig mit spezifischen Sensoren und Aktuatoren zu interagieren, da es sich dabei nur um eine Schnittstelle handelt. Zur Interaktion wird eine *DSA_Extension* Instanz benötigt, die über eine *Extension_Configuration* Instanz mit den Sensoren und Aktuatoren interagiert. Für dieses Beispiel wird angenommen, dass bereits eine solche *Extension_Configuration* implementiert wurde, und dass diese vom Typ Ex_Con ist.

```
1 #include <dsa.h>
2
3 class Y
4 {
5 public:
6    SensA_ID ext;
7    Ex_Con econ;
8
9    int main(void)
10   {
11      ext = DSA_Management::get_instance()->
           create_Extension(&econ);
12   }
13 }
```

Auch die *Extension_Configuration* Instanz muss mit einer Fabrikmethode der *DSA_Management* erzeugt werden. Diese Erzeugung sorgt dafür, dass die Instanz innerhalb des DSA-Rahmenwerks sicher verwaltet werden kann. Zur Erzeugung ist die Übergabe eines Zeigers auf eine *Extension_Configuration* Instanz notwendig, in diesem Fall wird der Zeiger auf die Instanz econ übergeben. Der Rückgabewert dieser Fabrikmethode ist nicht die Referenz auf die

erzeugte Instanz, sondern dessen Identifikationsnummer. Mit dieser wird später durch die *DSA_SensA* Instanz auf die *DSA_Extension* Instanz zugegriffen.

A.2.3. Konnektierung und Disconnektierung

Bevor mit einer *DSA_SensA* Instanz auf einen Sensor oder Aktuator zugegriffen werden kann, welches mit der entsprechenden Instanz der *DSA_Extension* Klasse geschieht, müssen diese beiden Instanzen miteinander Verbunden werden. Dies Funktioniert ganz einfach durch die entsprechende Funktion der *DSA_SensA* Instanz.

```
1 #include <dsa.h>
2
3 class Y
4 {
5 public:
6    DSA_SensA* s;
7    SensA_ID ext;
8    Ex_Con econ;
9    DSA_Management __man;
10
11    int main(void)
12    {
13       __man = DSA_Management::get_instance()
14       s = __man->get_DSA_SensA();
15       ext = __man->create_Extension(&econ);
16
17        __mans->connect_SensA(s, ext);
18    }
19 }
```

Der Aufruf __man->connect_SensA() benötigt die Identifikationsnummer der zu konnektierenden *DSA_Extension* Instanz und den Pointer zu der *DSA_SensA* Instanz. Das Management erstellt dann alle notwendigen Verbindungen zwischen dem *DSA_SensA* und der *DSA_Extension*. Durch den Aufruf von __man->disconnect_SensA(s) kann die Verbindung wieder entfernt werden.

A.2.4. Lesen von Sensordaten

Nach der Konnektion kann die *DSA_SensA* Instanz mit dem konkreten Sensor oder Aktuator interagieren. Um Sensordaten zu lesen, existieren in der Schnittstelle zwei Funktionen, die im folgenden Listing vorgestellt werden.

```
1  #include <dsa.h>
2
3  class Y
4  {
5  public:
6     DSA_SensA* s;
7     SensA_ID ext;
8     Ex_Con econ;
9     DSA_Management __man;
10    DSA_DO_Container ddoc*;
11
12    int main(void)
13    {
14       __man = DSA_Management::get_instance()
15       s = __man->get_DSA_SensA();
16       ext = __man->create_Extension(&econ);
17
18        __mans->connect_SensA(s, ext);
19       ddoc = s->get_all_Data(ext);
20    }
21 }
```

Durch die Funktion `get\_all\_Data()` werden alle Daten bezogen, die sich zu dem Zeitpunkt im Datenpuffer der *DSA_Extension* Instanz befinden. Die Funktion `get\_new\_data()` gibt einen Datencontainer zurück, der nur die Daten enthält, die durch diese *DSA_SensA* Instanz noch nicht bezogen wurden. Der DSA_DO_Container ist eine Containerklasse, die implementiert ist, um Kopien der Datenobjekte zu transportieren. Bei diesen Kopien handelt es sich nicht um die Originaldaten in dem Datenpuffer der *DSA_Extension* Instanz, damit diese vor Manipulation geschützt sind. Um auf die Daten zugreifen zu können, stellt der DSA_DO_Container drei Funktionen zur Verfügung. Um auf das letzte Element im Container zuzugreifen, wird die Funktion `last()` verwendet. Für den Zugriff auf das erste Element dient `first()`.

Beide Funktionen benötigen keine Parameter und geben eine Referenz auf das enthaltene Datenobjekt zurück. Mit der Funktion `at()` kann auf ein Datenobjekt an einer Bestimmten Position im Container zugreifen. Zusammen mit der Funktion `size()` kann damit der Container durchlaufen werden. Ein Beispiel dazu bietet das nächste Listing.

```
1 #include <dsa.h>
2 DSA_DO_Container ddoc;
3 DSA_Data_Object* do;
4
5 int main(void)
6 {
7    for (int i = 0; i < ddoc.size(); i++){
8       do = ddoc.at(i);
9    }
10 }
```

Um die Eventstruktur des Rahmenwerks nutzen zu können, müssen die Anwendung die Schnittstelle `SensA_Listener` implementieren. Diese kann anschließend an den *DSA_SensA* angemeldet werden. Ein Beispiel einer solchen Implementierung zeigt folgendes Listing:

```
1 #include <dsa.h>
2
3 class Y: SensA_Listener
4 {
5 public:
6    DSA_SensA* s;
7    SensA_ID ext;
8    Ex_Con econ;
9    DSA_Management __man;
10
11   int main(void)
12   {
13      __man = DSA_Management::get_instance();
14      s = __man->get_DSA_SensA();
15      s->set_listener(this);
16      ext =__man->create_Extension(&econ);
17
18      __man->connect_SensA(s, ext);
```

```
19
20      }
21
22      void on_SensA_notify(DSA_SensA_ID sensa,
           DSA_Data_Object* ddo){
23      /* hier werden die Reaktionen auf die
           Messergebnisse ausgefuehrt */
24      }
25  }
```

Durch den Aufruf von `s->set_listender(this)` wird der `SensA_Listener` zu den Beobachtern der *DSA_SensA* Instanz hinzugefügt. Der *DSA_SensA* leitet eingehende Messdaten an die `on_SensA_notify()` Funktion weiter.

A.2.5. Eigenschaftswerte ändern

Um Eigenschaften abzufragen ist es erforderlich den Typ dieser Eigenschaft zu kennen. Um eine Liste von Eigenschaftstypen zu erhalten, die für eine bestimmte *DSA_Extension* definiert sind, dient die Schnittstellenfunktion `property_Types()`. Diese Funktion benötigt ebenfallt die Identifikationsnummer der *DSA_Extension* Instanz.

Mit der Kenntnis des Eigenschaftstyps und des Wissens um den Wertetyp der betreffenden Eigenschaft, kann eine Property_Set_Object Instanz erzeugt werden, mit der eine Eigenschaft verändert wird.

```
1  #include <dsa.h>
2
3  Property_Set_Object* pso;
4  DSA_SensA* s;
5  SensA_ID ext;
6   ...
7
8  int eigenschaftszugriff(void)
9  {
10    pso = new Property_Set_Object(s,
11       "Eigenschaftstyp",
12       "Wertetyp",
```

```
13        "Wert")
14    s->set_Property(pso, ext);
15 }
```

Der erste Parameter des Konstruktors des Property_Set_Objects kann auch `NULL` sein. Die Instanz der *DSA_SensA* Klasse sorgt selbst dafür, dass dieser Wert durch den richtigen gesetzt wird. Dieser Wert ist immer die Selbstreferenz der *DSA_SensA* Instanz.

A.2.6. Eine eigene Konfigurationsklasse schreiben

Die Fähigkeit zur Interaktion mit Sensoren und Aktuatoren des DSA-Rahmenwerks steht und fällt in der sinnvollen Implementierung von Konfigurationsklassen, die von der abstrakten Klasse *Extension_Configuration* abgeleitet sind. In diesem Abschnitt wird ein Überblick über die zu implementierenden Funktionen gegeben. Für das Wie der Implementierung dieser Funktionen kann keine genaue Anleitung gegeben werden, da die Schnittstellen der Sensoren und Aktuatoren dieses Wie maßgeblich vorgeben.

Die *Extension_Configuration* Klasse beinhaltet insgesamt sechs virtuelle Funktionen, von denen vier abstrakt sind. Diese vier abstrakten Funktionen müssen implementiert werden, um eine Verwendung der Kindklasse der *Extension_Configuration* zu gestatten. Die Erstellung einer Kindklasse wird C++-typisch hergestellt.

```
1 #include <extension_configuration.h>
2
3 class Ext_Con : public Extension_Configuration
4 {
5    public:
6    DSA_Data_Object* get_Expenses();
7    FaultValue set_Property(Property_Set_Object* pso);
8    FaultValue update_data_buffer();
9    Permission_List* get_standard_Permissions();
10 }
```

Dadurch erhält die Klasse Ext_Con alle in der *Extension_Configuration* implementierten Fähigkeiten. Dazu gehören Attribute und Methoden, die von Instanzen der *DSA_Extension* Klasse verwendet werden. Die Methoden,

die alle mit dem Präfix `__DSA_` (<zwei Unterstriche> + <DSA> + <ein Unterstrich>) beginnen, dürfen nicht überladen werden.

Nachfolgend werden die einzelnen Methoden betrachtet und erläutert welchem Zweck diese dienen sollen. Eine Implementierung zur Erfüllung dieses Zwecks ist die Aufgabe des jeweiligen Entwicklers. Den jeweiligen Zweck erfüllen muss die Implementierung, da Instanzen der Klasse *DSA_Extension* diese Methoden für den jeweiligen Zweck aufrufen.

get_Expenses()
Diese Funktion soll eine Berechnung der Betriebskosten des Sensors oder Aktuators liefern. Wie diese Berechnung durchgeführt wird und ob dazu der Sensor oder der Aktuator befragt wird wird nicht vorgegeben. Ist eine Berechnung dieser Art nicht möglich, so kann auch ein leeres DSA_Data_Object zurückgegeben werden.

set_Property(Property_Set_Object* pso)
In dieser Funktion findet die Implementierung statt, die es ermöglicht Eingenschaftswerte zu verändern. Dies kann Eigenschaften des Sensors oder Aktuators betreffen. Aber auch Eigenschaften dieser Kindklasse der *Extension_Configuration* können hierdurch geändert werden, wenn hierfür Eigenschaften definiert wurden. Die Eigenschaften müssen alle vom Typ `Property` sein und in dem Container mit dem Bezeichner properties gespeichert werden. Dieser Datencontainer ist vom Typ `Property_List`, was in der aktuellen Implementierung des Rahmenwerks ein `std::vector<Property>` ist.

update_data_buffer()
Hier gehört die Implementierung zum Zugriff auf die Daten des Sensors rein. Wie diese Daten geholt werden ist Sensortypisch und wird an dieser Stelle nicht vorgegeben. Vorgegeben jedoch wird, dass diese Daten in ein Objekt vom Typ `DSA_Data_Object` abgelegt werden müssen. Dieses Objekt muss anschließend in den Puffer mit dem Bezeichner `data_buffer` gespeichert werden. Dieser Puffer ist vom Typ `DSA_Data_Buffer`.

make_decision(Claims* claims)

Die vierte virtuelle Methode ist die Methode `make_decision()`. Diese kann implementiert werden, um die Möglichkeit zu nutzen Eigenschafsänderungen zu beantragen. Da das Rahmenwerk keine Strategie vorgeben kann zur Entscheidung, welchem Antrag stattgegeben wird, ist es Aufgabe des Entwicklers diese Strategie zu implementieren. Die *DSA_Extension* wird diese Methode aufrufen und ihr als Parameter einen Container mit Anträgen übergeben. Diese Antrage sind alle vom Typ `Property_Set_Object`. Diese Methode soll nur entscheiden, welchem Antrag stattgegeben wird. Sie soll diese Eigenschaftsänderung nicht ausführen. Der gewünschte Rückgabewert muss die Position des stattgegebenen Antrags in der Liste kennzeichnen und ist vom Typ `Claim_Position`, was dem Typ unsigned int entspricht. Wenn keinem der Anträge entsprochen werden soll, dann kann der Wert `NOT_IMPLEMENTED` verwendet werden. Dieser führt dazu, dass die *DSA_Extension* keine Eigenschaftsänderung durchführt.

notify(DSA_Data_Object* ddo)

Diese virtuelle Methode ermöglicht es Abhängigkeiten von Sensoren zu realisieren. So könnte beispielsweise ein Softwaresensor immer eine bestimmte Zeit nach einem Messergebniss eines anderen Sensors einen Wert liefern. Mit der `notify()` Methode kann dieser Sensor über das Messergebniss informiert werden und seinen Timer starten.

get_standard_Permissions()

Über diese Methode kann das System die Zugriffrechte ermitteln, die bei der Verbindung eines *DSA_SensA* und einer *DSA_Extension* benötigt werden. Diese werden als Liste vom Typ `Permission_List` übergeben. Diese enthält für jede Eigenschaft, für die ein Recht bestehen soll, ein Eigenschaftstyp-Recht-Paar vom Typ `PP_Tupel`. Bei dem Eigenschaftstyp handelt es sich um eine Stringkonstante, die zu der betreffenden Eigenschaft in der *Extension_Configuration* passen muss. Die Rechtewerte sind vom Typ `Permission` und können die Werte `READ`, `WRITE` oder `READ_WRITE` einnehmen. Eigenschaften, an denen die *DSA_SensA* Instanz keine Rechte besitzen soll, werden der Rechteliste einfach nicht hinzugefügt. Wird statt der Referenz auf eine Rechteliste der Wert NULL zurückgeben, dann erhält die

DSA_SensA Instanz keine Rechte auf Eigenschaften. Sie kann aber dennoch verwendet werden, um Daten von Sensoren zu beziehen.

A.2.7. Ein neues Kommunikationsmedium dem Rahmenwerk bereitstellen

Die Fähigkeit auf Rahmenwerks-Instanzen auf entfernten Systemen zugreifen zu können, benötigt eine Implementierung der Klassen *Connection*, *Connection_Information* und *Connection_Manager*.

Erstellen der Connection_Information

Die Connection_Information dient als Identifikator einer Verbindung (*Connection*). Sinnvollerweise sollten in dieser auch die Addressinformationen der Verbindung gespeichert werden.

```
1 #include<connection_information.h>
2
3 class test_inf : Connection_Information
4 {
5
6 public:
7    bool operator ==(const Connection_Information& )
       const;
8 }
```

Die abstrakte Funktion `==()` muss dabei einen Vergleich der Instanzen dieser Connection_Information ermöglichen.

Erstellen der Connection

Für die Verwendung der spezifischen Verbindungen durch das Rahmenwerk stellt dieses die Template-Klasse *Connection* bereit. Diese beinhaltet 15 virtuelle Funktionen, von denen zwei abstrakt sind. Über diese beiden abstrakten Funktionen kann die gesamte Kommunikation realisiert werden.

```
1 #include<connection.h>
2 #include"test_inf.h"
3 class test_conn : Connection
4 {
5
```

```
6 public:
7     test_conn(test_inf* info) : Connection(info)
8     {
9     /*Konstruktor*/
10    }
11    int send_data(void* data, size_t len);
12    int receive_data(void* data, size_t len);
13 }
```

send_data()

Diese Funktion erlaubt dem Rahmenwerk, beliebig viele Bytes über die Verbindung zu senden. Als Parameter wird ein Pointer zu den Daten und die Anzahl der zu sendenden Bytes angegeben.

receive_data()

Diese Funktion erlaubt dem Rahmenwerk, beliebig viele Bytes über die Verbindung zu empfangen. Als Parameter wird ein Pointer zu den Daten und die Anzahl der zu empfangenden Bytes angegeben.

Für die anderen virtuellen Funktionen bestehen bereits Implementierungen, die über die `send_data()` und `receive_data()` realisiert werden. Sollte das Kommunikationsmedium die durch diese Funktionen übertragenen Objekte optimaler senden und empfangen können, können auch diese neu implementiert werden.

Erstellen des Connection_Managers

Diese Klasse dient zur Erstellung der *Connections*. Diese Klasse beinhaltet fünf abstrakte Funktionen, die für die spezifischen Kommunikationsmedien implementiert werden müssen:

```
1 #include<connection_manager.h>
2 #include"test_inf.h"
3 #include"test_conn.h"
4
5 class test_manager : public Connection_Manager{
6 public:
7     void start_listening();
8     void stop_listening();
```

```
9     bool check_compatible_conn_info(
          Connection_Information* info);
10    Connection* connect(Connection_Information* info);
11    void return_Connection(Connection* conn);
12
13 }
```

Eine Instanz des erstellten *Connection_Managers* wird über den Aufruf der folgender Funktion in das Rahmenwerk eingebunden:
`DSA_Management::get_instance()->add_Connection_Manager()`

Nachfolgend wird betrachtet und erläutert, welche Funktionalität durch diese Funktionen gegeben werden soll:

start_listening()
Über diese Funktion muss ein Prozess gestartet werden, der eingehende Verbindungen oder neue Daten auf bestehenden Verbindungen erkennt. Diese müssen dann durch den Aufruf der Funktion `Connectionhandler::get_instance()->handleconnection()` von diesem Prozess an das Rahmenwerk zur Bearbeitung gegeben werden.

stop_listening()
Diese Funktion stoppt den Prozess, mit dem der *Connection_Manager* die Verbindungen überwacht.

check_compatible_conn_info()
Mit dieser Funktion kann das Rahmenwerk prüfen, ob die *Connection_Information* des Rahmenwerks, die vom Typ der speziellen Implementation der *Connection_Information* entspricht. Für diese Funktion stellt die *Connection_Information* die generische Funktion `check_compatible()` bereit.

connect()
Diese Funktion erstellt eine Verbindung nach den Vorgaben des implemen-

tierten Kommunikationsmediums. Die übergebene *Connection_Information* enthält dabei die für den Verbindungsaufbau wichtigen Informationen.

return_Connection()
Da das Rahmenwerk keine Informationen über das implementierte Kommunikationsmedium besitzt, wird die Entscheidung, wie mit einer nicht mehr verwendeten *Connection* weiter verfahren wird, dem *Connection_Manager* überlassen. Über diese Funktion werden die *Connections* an diesen *Connection_Manager* zurückgegeben.

B. Quelltext des erweiterten DSA-Rahmenwerks in C++ implementiert

DSA_Framework/dsa.h

```
/*
 * File:   dsa.h
 * Author: krayn
 *
 * Created on October 2, 2013, 1:06 PM
 */

#ifndef DSA_H
#define  DSA_H

#include <dsa_management.h>

#endif   /* DSA_H */
```

DSA_Framework/dsa_types.h

```
#ifndef DSA_H
#define DSA_H

/**
 * @file dsa_types.h
 * In this header file all types, used by the DSA-Framework,
 * which are not standard by C++/11 are defined.
 *
 * @author Jens Krayenborg
 * @date 13.01.2013
 *
 * @version 1.0.0
 */

#include <string>
#include <list>
#include <utility>
#include <limits.h>
```

```
19 #include <ctime>
20 #include <stdint-gcc.h>
21
22 class Connection;
23 class Connection_Manager;
24 class Connection_Information;
25 class Property;
26 class DSA_Data_Object;
27 class Extension_Information;
28 class Remote_Observer;
29
30 #define STDBUFFERSIZE 50
31 #define HISTDATDIFF 0
32
33 /* Type definitions */
34 /** Timetype used by the DSA-Framework */
35 typedef time_t DSA_Time;
36
37 /** Sizetype used by the DSA-Framework */
38 typedef uint8_t dsa_data_size;
39
40 /** Sizetype used by the DSA-Framework */
41 typedef uint32_t dsa_size_t;
42
43 /** The Permissions which can be given a DSA_SensA instance
      to a Property. */
44 enum Permission {
45     READ, /** Permission only to read a Property */
46     WRITE, /** Permission only to write a Property */
47     READ_WRITE /** Permission to read and write a Property */
48 };
49
50 /** The type of a property as a string.
51  * This form of representation is necessary
52  * as an enumerated type after compiling a rule can not be
       changed.*/
53 typedef std::string Property_T;
54
55 /** A simple vector to hold property types. */
56 typedef std::list<Property_T> Property_T_List;
57
58 /** A simple vector to hold Property instances. */
59 typedef std::list<Property> Property_List;
```

```
60
61 /** A (P)roperty type (P)ermission tuple. Used for
62  * for giving a DSA_SensA instance a permission to one
       property.
63  */
64 typedef std::pair<Property_T, Permission> PP_Tupel;
65
66 /** A simple vector to hold PP_Tuples. */
67 typedef std::list< PP_Tupel > Permission_List;
68
69 /** The fault values defined for use in the DSA-Framework. */
70 enum FaultValue {
71     S_OK, /** Operation successfully completed */
72     V_OK, /** Operation virtually completed */
73     E_VALUEERROR, /** One given parameter has wrong value */
74     E_TYPEMISSMATCH, /** One given parameter has not matching
           type */
75     E_ILLEGAL, /** This function cannot legaly permfomed */
76     E_PROPNDEF, /** The given property is not defined */
77     E_UNKSENSA, /** The given DSA_SensA instance is unknown
           */
78     E_UNKEXT, /** The given DSA_Extension ID is unknown */
79     E_NOPERMISSION, /** The calling DSA_SensA instance has no
80                              permission to perform this
                                    request */
81     E_NOCLAIM, /** This property cannot be set by using
82                              the claiming implementation (
                                    obsolet)*/
83     E_RELATIONSHIP, /** Relation between DSA_SensA and
           DSA_Extension
84                              instance is in trouble */
85     E_NOT_IMPLEMENTED, /** The called function is not
           implemented yet */
86     E_COMMERROR, /** The communication could not be performed
           */
87     E_NULLPTR, /** One needed parameter is a NULL reference
           */
88     E_UNKCONN
89 };
90
91
92 /** The type for internal used ID's for DSA_SensA instances.
       */
```

```
93 typedef unsigned int DSA_SensA_ID;
94
95 /** The type for internal used ID's for DSA_Extension
       instances. */
96 typedef unsigned int Spec_ID;
97
98 /** A typeless pointer to reference memory space which holds
99  *  the data of a DSA_Data_Object instance. */
100 typedef void* DSA_Data;
101
102 /** A type hint to the data holdes in the memory space the
103  *  DSA_Data points to. */
104 typedef std::string DSA_Data_Type;
105
106 /** A type to store a location of a sensor/actor
107  */
108 typedef std::string SensA_Location;
109
110 /** A type to store a location of a sensor/actor
111  */
112 typedef std::string SensA_Type;
113
114 /** The type for internal used ID's for DSA_Data_Object
       instances. */
115 typedef uint32_t Data_ID;
116
117 /** The type of an DSA_Data_Object instance. */
118 enum DSA_Object_Type {
119     OT_DATA, /** The given DSA_Data_object contains sensing
           data */
120     OT_EXPENSE, /** The given DSA_Data_object contains
           expense data */
121     OT_ACCURACY /** The given DSA_Data_object contains
           accuracy data */
122 };
123
124 /** A simple vector to hold pointers to DSA_Data_Object
       instances. */
125 typedef std::list<DSA_Data_Object*> DSA_DO_List;
126
127 /** A simple vector to hold Data_ID's. */
128 typedef std::list<Data_ID> Data_History;
129
```

```
130 /** NOT CONTinuous*/
131 const DSA_Time NOT_CONT = -1;
132
133 /** The answer is unknown. */
134 const DSA_Time UNKNOWN = -2;
135
136 /** An error occured while performing the request. */
137 const unsigned int FAIL = UINT_MAX;
138
139 /** The called function is not implemented yet. */
140 const unsigned int NOT_IMPLEMENTED = UINT_MAX - 1;
141
142 /** The internal used type to point to a position in an
      application vector. */
143 typedef unsigned int Claim_Position;
144
145 /** A simple vector to hold DSA_SensAID's. */
146 typedef std::list<DSA_SensA_ID> ID_List;
147
148
149 /** The type used to identify a DSA_virt_SensA within a
      SensA_Container*/
150 typedef uint32_t Cont_SensA_ID;
151 /** The type used to identify SensA_Containers*/
152 typedef uint32_t Cont_ID;
153 typedef uint32_t Observer_ID;
154 /** A tuple used to identify a DSA_virt_SensA instance within
       the local process*/
155 typedef std::pair<Cont_ID, Cont_SensA_ID> SensA_ID;
156
157 /** A tuple that specifies a relationship between a
      DSA_Extension and
158  *  a DSA_SensA instance. */
159 typedef std::pair<SensA_ID, DSA_SensA_ID> Connection_Pair;
160
161 /** A simple vector to hold relationship tuples. */
162 typedef std::list<Connection_Pair> Connection_List;
163
164 typedef std::list<Extension_Information*> SensA_Info_list;
165 typedef std::list<SensA_ID> SensA_ID_List;
166
167 enum ConnectionResponse {
168     _GET_DATA,
```

```
169     _TEST_SENSA,
170     _TEST_CONTAINER,
171     _REG_OBS,
172     _S_OK,
173     _E_UNKSENSA,
174     _NOTIFY,
175     _E_CONNECTION,
176     _SENSA_INFO,
177     _RM_OBS,
178     _SET_PROP,
179     _GET_PROP,
180     _GET_PERMS,
181     _GET_SENSA_IDS,
182     _GET_EXP
183 };
184 #endif // DSA_H
```

DSA_Framework/dsa_sensa.h

```
1  /**
2   * @file   dsa_sensa.h
3   * @author Connor Fibich
4   *
5   * Created on October 14, 2013, 12:31 PM
6   */
7
8  #ifndef DSA_SENSA_H
9  #define  DSA_SENSA_H
10
11 //#include "dsa_management.h"
12 #include "sensa_observer.h"
13 #include "sensa_listener.h"
14 #include "dsa_extension.h"
15
16 class DSA_Extension;
17 class DSA_SensA : public SensA_Observer{
18     friend class DSA_Management;
19 public:
20     DSA_SensA(SensA_ID id);
21     DSA_SensA(const DSA_SensA& orig);
22     virtual ~DSA_SensA();
23
24     Property_T_List* property_Types();
25     Permission_List* permissions();
```

```
26      Property* get_Property(Property_T property_Type);
27      FaultValue set_Property(Property_Set_Object* prop);
28      DSA_DO_Container* get_all_Data();
29      DSA_DO_Container* get_new_Data();
30      size_t get_data_Buffer_size();
31
32      void notify(DSA_Data_Object* ddo);
33      void set_listener(SensA_Listener* sen);
34      SensA_ID get_connected_SensA();
35      SensA_ID id;
36  protected:
37
38      FaultValue connect_SensA(DSA_Extension* sens);
39      FaultValue disconnect_SensA();
40  private:
41      SensA_Listener* obs;
42      DSA_Extension* sensA;
43  };
44
45  #endif   /* DSA_SENSA_H */
```

DSA_Framework/dsa_sensa.cpp

```
1   /**
2    * @file   dsa_sensa.cpp
3    * @author Connor Fibich
4    *
5    * Created on October 14, 2013, 12:31 PM
6    */
7
8   #include "dsa_sensa.h"
9   #include "dsa_do_container.h"
10
11  DSA_SensA::DSA_SensA(SensA_ID id) : SensA_Observer() {
12      this->id = id;
13  }
14
15  DSA_SensA::DSA_SensA(const DSA_SensA& orig) {
16  }
17
18  DSA_SensA::~DSA_SensA() {
19  }
20
21  FaultValue DSA_SensA::connect_SensA(DSA_Extension* sens) {
```

```
22
23      if (sens == NULL) return E_UNKEXT;
24      sensA = sens;
25      sensA->register_observer(this);
26      return S_OK; //connect(this, id, perms);
27  }
28
29  FaultValue DSA_SensA::disconnect_SensA() {
30
31      if (sensA == NULL) return E_NULLPTR;
32
33      sensA->remove_observer(get_observer_ID());
34      sensA = NULL;
35      return S_OK; //sensA->connect(this, id, perms);
36  }
37
38  /**
39   * @brief Set the class listening to DSA_SensA notifications
40   * @param sen the class implementing the SensA_Listener
41   */
42  void DSA_SensA::set_listener(SensA_Listener* sen) {
43      obs = sen;
44  }
45
46  void DSA_SensA::notify(DSA_Data_Object* ddo) {
47      obs->on_sensA_notify(id.second, ddo);
48  }
49
50  SensA_ID DSA_SensA::get_connected_SensA() {
51      if (sensA == NULL) return SensA_ID(-1,-1);
52      return sensA->get_ID();
53  }
54
55  DSA_DO_Container* DSA_SensA::get_new_Data() {
56      if(sensA == NULL)
57          return 0;
58      return sensA->get_new_Data(id);
59  }
60
61  FaultValue DSA_SensA::set_Property(Property_Set_Object* prop)
        {
62      if(sensA == NULL)
63          return E_NULLPTR;
```

```
64      return sensA->set_Property(prop);
65  }
```

DSA_Framework/dsa_extension.h

```
1   /**
2    * @file   dsa_virt_sensa.h
3    * @Author Connor Fibich
4    *
5    * Created on September 30, 2013, 11:41 AM
6    */
7   #pragma once
8   //#ifndef DSA_VIRT_SENSA_H
9   //#define   DSA_VIRT_SENSA_H
10
11  #include "sensa_observer.h"
12  #include "dsa_types.h"
13  #include "dsa_do_container.h"
14  #include "extension_information.h"
15  #include "extension_configuration.h"
16  #include "extensions_sensas.h"
17  #include "dsa_sensa.h"
18  #include <vector>
19
20  class DSA_Extension : public SensA_Observer {
21  public:
22      //DSA_virt_SensA(SensA_ID sensaID);
23      DSA_Extension(SensA_ID sensaID, Extension_Configuration*
            conf);
24      virtual ~DSA_Extension();
25
26
27
28
29      virtual DSA_DO_Container* get_all_Data(SensA_ID sensA);
30      virtual DSA_DO_Container* get_new_Data(SensA_ID sensA);
31      Permission_List* get_permissions(SensA_ID sensA);
32      Property* get_Property(Property_T prop, SensA_ID sensA);
33      Property_T_List* get_Property_Types();
34      FaultValue set_Property(Property_Set_Object* value);
35
36      size_t get_data_buffer_size();
37      ID_List* connected_sensas();
38
```

```
39      /*
40       changed type from Spec_ID to SensA_ID;
41       */
42      SensA_ID get_ID();
43      virtual void register_observer(SensA_Observer* obs);
44      virtual void remove_observer(Observer_ID obsID);
45      void notify_observers(DSA_Data_Object* ddo);
46      /**
47       *
48       * @return information about the SensA (type, location,
            etc)
49       */
50      virtual Extension_Information* get_SensA_information();
51      virtual void notify(DSA_Data_Object* ddo);
52
53  protected:
54      friend class DSA_Management;
55      friend class Connectionhandler;
56      // only the DSA_Management has access to theese functions
57      virtual DSA_Data_Object* get_Expenses();
58      virtual FaultValue connect(SensA_ID id, Permission_List*
           perms);
59      virtual FaultValue disconnect(SensA_ID sensA);
60      virtual Permission_List* get_standard_Permissions();
61      std::list<SensA_Observer*> observers;
62      Extension_Configuration* configuration;
63  private:
64      //SensA_Configuration* configuration;
65      std::list<Extensions_SensAs> sensAs;
66
67      /* Operations */
68
69      FaultValue set_data_buffer_size(Property_Set_Object* pso)
           ;
70
71      Extensions_SensAs* search_SAEP(SensA_ID ref);
72      unsigned int count_sensas_with_rights(Property_T ptype,
           Permission perm);
73
74  };
75
76
77  //#endif /* DSA_VIRT_SENSA_H */
```

DSA_Framework/dsa_extension.cpp

```
1  /*
2   * Commentexplaination:
3   * @adopted from xy: Code passage flagged as "@adopted" is
       Code adopted from the previous DSA-Framework by Jens
       Krayenborg
4   * (xy is the file where the code is located in the previous
       Framework)
5   * @end_adopted: marks the end of an adopted code passage
6   * @change: notes what changes were made to the code.
7   */
8
9  #include <list>
10
11 #include "dsa_extension.h"
12
13 /*
14  * ============
15  * @adopted from dsa_extension.cpp
16  * @change parametertypes changed from DSA_SensA* to SensA_ID
17  * ============
18  */
19
20 FaultValue DSA_Extension::connect(SensA_ID sensA,
      Permission_List* perms) {
21     std::list<PropPermPet> ppp;
22
23     // Check if the DSA_SensA is allready connected to this
          extension.
24     if (search_SAEP(sensA) != NULL)
25         return E_ILLEGAL;
26
27     // OK. It's not. So check if the permission list is NULL.
28     if (perms != NULL){
29         // It's not NULL so fill the property and permission
              list.
30         for (Permission_List::iterator it = perms->begin();
31              it != perms->end();
32              ++it)
33         {
34             PP_Tupel* ppt = &(*it);
```

```
35              PropPermPet adding(configuration->
                    __DSA_get_Property(ppt->first), ppt->second,
                    false);
36              ppp.push_back(adding);
37          }
38      }
39
40      // And add the new one with its property and permission
            list to the sensa list.
41      sensAs.push_back(Extensions_SensAs(sensA, ppp,
            configuration->__DSA_get_data_buffer_size(),
            configuration->__DSA_get_data_buffer()));
42
43      return S_OK;
44  }
45  /**
46   * @brief DSA_Extension::disconnect
47   * Disconnecting a DSA_SensA instance. Also removing all
        management
48   * informations related to this connection from memory.
49   *
50   * @param sensA The DSA_SensA instance to be disconnected.
51   * @return      E_ILLEGAL if the DSA_SensA instance is not
52   *              connected.
53   *              S_OK otherwise.
54   */
55  FaultValue DSA_Extension::disconnect(SensA_ID sensA)
56  {
57      for (std::list<Extensions_SensAs>::iterator it = sensAs.
            begin();
58           it != sensAs.end();
59           ++it)
60      {
61          Extensions_SensAs* test = &(*it);
62          if (test->sensaID == sensA)
63          {
64              // Found the sensa so erase all management infos
                    related
65              // to this connection.
66              sensAs.erase(it);
67              return S_OK;
68          }
69      }
```

```
70      // Not found the DSA_SensA instance.
71      return E_ILLEGAL;
72  }
73
74
75  /**
76   * @brief DSA_Extension::get_all_Data
77   * Getting copies of all datas in the databuffer.
78   *
79   * @param sensA The DSA_SensA instance which calls this
        function.
80   * @return      A container with copies of all data from the
        buffer.
81   *              NULL if the DSA_SensA is not connected
82   */
83  DSA_DO_Container* DSA_Extension::get_all_Data(SensA_ID sensA)
        {
84      /* Only connected DSA_SensA instances have the permission
            to use
85       * use this function.
86       */
87      if (search_SAEP(sensA) == NULL) return NULL;
88
89      /* Let the Extension_Configuration update the buffer if
           necessary. */
90      configuration->update_data_buffer();
91
92      DSA_DO_List* ddolist = configuration->
           __DSA_get_data_buffer();
93      Extensions_SensAs* seap = search_SAEP(sensA);
94      DSA_DO_Container* c = new DSA_DO_Container;
95
96      /* Copy the contents of the buffer into the container */
97      c->add(ddolist);
98
99      // Update the history of the calling sensa.
100     for (DSA_DO_List::iterator i = ddolist->begin();
101             i != ddolist->end();
102             ++i) {
103         DSA_Data_Object* ddo = *i;
104         if (!seap->history.contains_DataID(ddo->dataID))
105             seap->history.add_to_buffer(ddo->dataID);
106     }
```

```
107
108     /* And return the container. */
109     return c;
110 }
111
112 /*
113  * @brief DSA_Extension::get_new_Data
114  * Getting copies of data objects in the databuffer.
115  * Getting only them were not received before from this
       DSA_SensA
116  * instance.
117  *
118  * @param sensA The DSA_SensA instance which calls this
       function.
119  * @return      A container with copies of new data from the
       buffer.
120  *              NULL if the DSA_SensA is not connected
121  */
122 DSA_DO_Container* DSA_Extension::get_new_Data(SensA_ID sensA)
       {
123     /* Only connected DSA_SensA instances have the permission
           to use
124      * use this function.
125      */
126     if (search_SAEP(sensA) == NULL) return NULL;
127
128     /* Let the Extension_Configuration update the buffer if
          necessary. */
129     configuration->update_data_buffer();
130
131     DSA_DO_List* ddolist = configuration->
          __DSA_get_data_buffer();
132     Extensions_SensAs* seap = search_SAEP(sensA);
133     DSA_DO_Container* nD = new DSA_DO_Container;
134
135     /* Iterate throgh the buffer an copy data objects into
          the
136      * container which were not received from the calling
           sensa
137      * before.
138      */
139     for (DSA_DO_List::iterator it = ddolist->begin();
140             it != ddolist->end();
```

```
141             ++it) {
142         DSA_Data_Object* ddo = *it;
143         if (!seap->history.contains_DataID(ddo->dataID)) {
144             nD->add(ddo);
145             seap->history.add_to_buffer(ddo->dataID);
146         }
147     }
148
149     // Andreturn the container.
150     return nD;
151 }
152
153 /*
154  * =======
155  *  @end_adopted dsa_extension.cpp
156  * =======
157  */
158
159 DSA_Data_Object* DSA_Extension::get_Expenses() {
160     return configuration->get_Expenses();
161 }
162
163 Extension_Information* DSA_Extension::get_SensA_information()
        {
164
165 }
166
167 void DSA_Extension::notify(DSA_Data_Object* ddo) {
168     configuration->notify(ddo);
169 }
170
171 void DSA_Extension::notify_observers(DSA_Data_Object* ddo) {
172     SensA_Observer* dsaObs;
173     for (std::list<SensA_Observer*>::iterator it = observers.
           begin();
174             it != observers.end();
175             ++it) {
176         dsaObs = *it;
177         dsaObs->notify(ddo);
178     }
179 }
180
181 void DSA_Extension::register_observer(SensA_Observer* obs) {
```

```
182     observers.push_back(obs);
183 }
184
185 void DSA_Extension::remove_observer(Observer_ID obsID) {
186
187     SensA_Observer* dsaExt;
188     for (std::list<SensA_Observer*>::iterator it = observers.
           begin();
189             it != observers.end();
190             ++it) {
191          dsaExt = *it;
192          if (dsaExt->get_observer_ID() == obsID) {
193             observers.erase(it);
194             return;
195          }
196     }
197 }
198
199 SensA_ID DSA_Extension::get_ID() {
200     return configuration->ID;
201 }
202
203 DSA_Extension::DSA_Extension(SensA_ID sensaID,
       Extension_Configuration* conf) : SensA_Observer() {
204     configuration = conf;
205     configuration->ID = sensaID;
206 }
207
208 DSA_Extension::~DSA_Extension() {
209
210 }
211
212 Permission_List* DSA_Extension::get_standard_Permissions() {
213     configuration->get_standard_Permissions();
214 }
215 /*
216  * @adopted from dsa_extension.cpp
217  * @change parameter changed type from DSA_SensA* to SensA_ID
218  */
219
220 /**
221  * @brief DSA_virt_SensA::search_SAEP
222  * Search the management informations of a given DSA_SensA
```

```
223  * instance.
224  *
225  * @param ref   The DSA_SensA instance from which the
       management
226  *              infos to be found.
227  * @return      The management infos of the given sensa.
228  *              NULL if them cannot be found.
229  */
230 Extensions_SensAs* DSA_Extension::search_SAEP(SensA_ID ref) {
231     for (std::list<Extensions_SensAs>::iterator it = sensAs.
           begin();
232              it != sensAs.end();
233              ++it) {
234          Extensions_SensAs* seap = &(*it);
235          if (seap->sensaID == ref)
236              return seap;
237     }
238     return NULL;
239 }
240 /*
241  * @end_adopted
242  */
243
244 #include <iostream>
245 #include <stdio.h>
246 FaultValue DSA_Extension::set_Property(Property_Set_Object*
       value)
247 {
248     Extensions_SensAs* seap;
249     PropPermPet* ppp;
250
251     if (value->property_Type.compare("__DSA_bc") == 0)
252         return set_data_buffer_size(value);
253
254     // Checking if the property is generaly setable for this
           extension.
255     if ((configuration->__DSA_get_Property(value->
           property_Type)) == NULL)
256         return E_PROPNDEF;
257
258     /* Only connected DSA_SensA instances have the permission
           to use
259      * use this function.
```

```
260      */
261     if ((seap = search_SAEP(value->sensA)) == NULL)
262         return E_UNKSENSA;
263
264     // DSA_SensA was found. So search the property which
           should be set.
265     if ((ppp = seap->search_PPP(value->property_Type)) ==
           NULL)
266         return E_NOPERMISSION;
267
268     // The property was found, so check the permission.
269     if (ppp->permission == READ)
270         return E_NOPERMISSION;
271
272     // The property was found and the DSA_SensA has the
273     // permission to set it.
274     // So, if this property set has to be claimed,
275     // we have to set the petition to true.
276     // Otherwise simply set the property and return.
277     FaultValue returnValue = E_ILLEGAL;
278     Property* property = configuration->__DSA_get_Property(
           value->property_Type);
279
280     if (configuration->__DSA_isClaimProp(value->property_Type
           ))
281     {
282         // The setting of this property has to be claimed.
283         Claim_Position decision;
284
285         // Add the application to the other claims for this
               property.
286         property->claims.setClaim(value);
287         ppp->petition = true;
288
289         // Checking how much DSA_SensA instances with write
               permission
290         // to this property are connected to this
               dsa_extension.
291         unsigned int nop = count_sensas_with_rights(value->
               property_Type, WRITE);
292
293         // Checking if all sensas with permission have
               claimed a property set.
```

```
294         if (nop == property->claims.number_of_claims()){
295
296             // If so then let the extension configuration
                   decide which
297             // apllication to use for setting the property.
298             decision = configuration->make_decision(&(
                   property->claims));
299
300             // If the decision is NOT_IMPLEMENTED than the
                   make_decision()
301             // function ist not implemented or the algorithm
                   was not
302             // able to decide wich aplication to use. Is this
                    will be so
303             // then we will do nothing and return.
304             if (decision != NOT_IMPLEMENTED) {
305
306                 // OK... we are able to set the property. So
                       get the
307                 // application from the claims object and
                       give it to
308                 // the configuration to set the real property
                       .
309                 Property_Set_Object* pso = property->claims.
                       claim_at(decision);
310                 returnValue = configuration->set_Property(pso
                       );
311                 if (returnValue == S_OK  returnValue == V_OK)
312                     // If the setting was ok than we have to
313                     // change the value in the property
                           object
314                     // describing the real property.
315                     property->value.assign(pso->value);
316             } else return E_NOT_IMPLEMENTED;
317         } else
318             // Not all connected sensas with rights have
                   claimed yet,
319             // so we do nothing and return V_OK.
320             return V_OK;
321     } else {
322         // OK ... It's not a property for which the setting
                has to be claimed.
323         returnValue = configuration->set_Property(value);
```

```
324         if (returnValue == S_OK  returnValue == V_OK)
325             // If the setting was ok than we have to
326             // change the value in the property object
327             // describing the real property.
328             property->value.assign(value->value);
329     }
330     return returnValue;
331 }
332
333 /**
334  * @brief DSA_Extension::set_data_buffer_size
335  * Setting the data buffer to a new size.
336  *
337  * @param pso   The object holding the  new value.
338  * @return      S_OK on success,
339  *              E_UNKSENSA if the calling sensa is not
       connected,
340  *              E_NOPERMISSION if the sensa has not the
       permission
341  *              to set the buffer size.
342  */
343
344 #include <sstream>
345 FaultValue DSA_Extension::set_data_buffer_size(
      Property_Set_Object* pso)
346 {
347     PropPermPet* ppp;
348     Extensions_SensAs* seap;
349
350     /* Only connected DSA_SensA instances have the permission
           to use
351      * use this function.
352      */
353     if ((seap = search_SAEP(pso->sensA)) == NULL)
354         return E_UNKSENSA;
355
356     // DSA_SensA was found. So search the property which
         should be set.
357     if ((ppp = seap->search_PPP(pso->property_Type)) == NULL)
358         return E_NOPERMISSION;
359
360     // The property was found, so check the permission.
361     if (ppp->permission == READ)
```

```
362         return E_NOPERMISSION;
363
364     // Convert the value to the correct format.
365     std::stringstream ss;
366     size_t s;
367     ss << pso->value;
368     ss >> s;
369
370     /* Iterate through all connected sensas and set their
           history buffer to
371      * the same size as the new data buffer size.
372      */
373     for (std::list<Extensions_SensAs>::iterator it = sensAs.
           begin();
374          it != sensAs.end();
375          ++it)
376     {
377         Extensions_SensAs* seap = &(*it);
378         seap->history.set_Size(s);
379     }
380     // Set the data buffer size to the new one.
381     configuration->__DSA_set_data_buffer_size(s);
382     return S_OK;
383 }
384
385 unsigned int DSA_Extension::count_sensas_with_rights(
       Property_T ptype, Permission perm)
386 {
387     Extensions_SensAs* es;
388     int c = 0;
389
390     for (std::list<Extensions_SensAs>::iterator it = sensAs.
           begin();
391          it != sensAs.end();
392          ++it)
393     {
394         es = &(*it);
395         if (es->has_right_to(ptype, perm))
396             c += 1;
397     }
398
399     return c;
400 }
```

DSA_Framework/virt_extension.h

```
/*
 * File:   virtual_sensa.h
 * Author: krayn
 *
 * Created on October 1, 2013, 3:33 PM
 */

#ifndef VIRTUAL_SENSA_H
#define  VIRTUAL_SENSA_H

#include "dsa_extension.h"
#include "dsa_data_object.h"

class Virtual_Extension : public DSA_Extension {
public:
    Virtual_Extension(SensA_ID sensaID);
    Virtual_Extension(const Virtual_Extension& orig);
    virtual ~Virtual_Extension();
    FaultValue add_SensA(DSA_Extension* sensa);
    FaultValue remove_SensA(SensA_ID sensaID);
    FaultValue set_virtual_Rule();
    DSA_DO_Container* get_new_data();

    void notify(DSA_Data_Object* ddo);

private:
    std::list<DSA_Extension*> sensas;
};

#endif   /* VIRTUAL_SENSA_H */
```

DSA_Framework/virt_extension.cpp

```
/*
 * File:   virtual_sensa.cpp
 * Author: krayn
 *
 * Created on October 1, 2013, 3:33 PM
 */

#include "virt_extension.h"
#include "proxy_extension.h"

```

```
11 Virtual_Extension::Virtual_Extension(SensA_ID sensaID) :
       DSA_Extension(sensaID, new Proxy_Configuration()) {
12 }
13
14 //Virtual_SensA::Virtual_SensA(const Virtual_SensA& orig) {
15 //}
16
17 Virtual_Extension::~Virtual_Extension() {
18 }
19
20 FaultValue Virtual_Extension::add_SensA(DSA_Extension* sensa)
       {
21     sensa->register_observer(this);
22     sensas.push_back(sensa);
23     return S_OK;
24 }
25
26 FaultValue Virtual_Extension::remove_SensA(SensA_ID sensaID)
       {
27     DSA_Extension* dsaSen;
28     for (std::list<DSA_Extension*>::iterator it = sensas.
          begin();
29              it != sensas.end();
30              ++it) {
31          dsaSen = *it;
32          if (dsaSen->get_ID().first == sensaID.first && dsaSen
             ->get_ID().second == sensaID.second) {
33              sensas.erase(it);
34              return S_OK;
35          }
36     }
37 }
38
39 FaultValue Virtual_Extension::set_virtual_Rule() {
40     return S_OK;
41 }
42
43 DSA_DO_Container* Virtual_Extension::get_new_data() {
44     return NULL;
45 }
46 void Virtual_Extension::notify(DSA_Data_Object* ddo) {
47         notify_observers(ddo);
48 }
```

DSA_Framework/sensa_observer.h

```
/*
 * File:   sensa_observer.h
 * Author: krayn
 *
 * Created on September 30, 2013, 11:37 AM
 */

#ifndef SENSA_OBSERVER_H
#define  SENSA_OBSERVER_H
#include "dsa_types.h"
#include "dsa_data_object.h"
#include "notify_rule.h"

class SensA_Observer {
public:
    SensA_Observer();
    virtual void notify(DSA_Data_Object* ddo) = 0;
    Observer_ID get_observer_ID();
    FaultValue set_notify_rule();
    FaultValue remove_notify_rule();
    bool check_relevant(DSA_Data_Object* ddo);

private:
    Observer_ID id;
    Notify_Rule* rules;
};

#endif   /* SENSA_OBSERVER_H */
```

DSA_Framework/sensa_observer.cpp

```
#include "sensa_observer.h"
#include "dsa_management.h"

SensA_Observer::SensA_Observer() {
    this->id = DSA_Management::get_instance()->
       get_new_observer_ID();
}

Observer_ID SensA_Observer::get_observer_ID() {
    return this->id;
}

```

```
12
13      bool SensA_Observer::check_relevant(DSA_Data_Object* ddo)
            {
14
15      }
```

DSA_Framework/sensa_listener.h

```
1  /**
2   * @file   sensa_listener.h
3   * Classes that want to receive events from Sensors and
        Actors have to implement this Class.
4   * @author: Connor Fibich
5   *
6   * Created on October 14, 2013, 2:00 PM
7   */
8
9  #ifndef SENSA_LISTENER_H
10 #define  SENSA_LISTENER_H
11
12 #include "dsa_data_object.h"
13
14 /**
15  *
16  */
17 class SensA_Listener{
18     public:
19         /**
20          * @brief This function receives the Data_Objects
                from DSA_SensA's. The Class using the DSA
                Framework can use this to take the nessesary
                actions to the incoming Data Objects
21          * @param sensa The Id of the DSA_SensA sending the
                Data_Object
22          * @param ddo The Date Object send by the SensA
23          */
24         virtual void on_sensA_notify(DSA_SensA_ID sensa,
              DSA_Data_Object* ddo) = 0;
25 };
26
27 #endif   /* SENSA_LISTENER_H */
```

DSA_Framework/extensions_sensas.h

```
1  #ifndef EXTENSIONS_SENSAS_H
```

```
2  #define EXTENSIONS_SENSAS_H
3
4  /**
5   * @file extensions_sensas.h
6   * This file describes the PropPermPet and the
       Extensions_SensAs classes.
7   * The first is a pure data holding class to manage, which
       permission
8   * is given to which property. And if a DSA_SensA instance
       has given
9   * an application (petition) for setting this property.
       Objects of
10  * this class are used by the second class in this file.
11  *
12  * Tis second class is designed to manage which DSA_SensA
       instance
13  * own which permission to which properties. And if it has
       given a
14  * petition to a property. To perform this, it uses instances
         of the
15  * first class defined in this file.
16  * Additionaly instances of this class holds the data history
         for
17  * the DSA_SensA instance which is related to this instance.
18  *
19  * These two classes never should be used directly by the DSA
       -Framework
20  * using informatik system. It is designed only for internal
       use.
21  *
22  * @author Jens Krayenborg
23  * @date 13.01.2013
24  *
25  * @version 1.0.0
26  */
27
28 #include "property.h"
29 #include "dsa_history_buffer.h"
30 #include "dsa_data_object.h"
31
32 //class DSA_SensA;
33
34 class PropPermPet
```

```
35 {
36 public:
37     /* Constructor */
38     PropPermPet(Property* prop, Permission perm = READ, bool
          pet = false);
39     /* Attributes */
40     Property* property;
41     Permission permission;
42     bool petition;
43 };
44
45 class Extensions_SensAs
46 {
47 public:
48     /* Constructor */
49     Extensions_SensAs( SensA_ID id, std::list<PropPermPet>
          props, size_t history_size, DSA_DO_List* db);
50
51     /* Attributes */
52     SensA_ID sensaID;
53     std::list<PropPermPet> properties;
54     DSA_History_Buffer history;
55     bool has_right_to(Property_T ptype, Permission perm);
56
57     /* Operations */
58     PropPermPet* search_PPP(Property_T pt);
59 };
60
61 #endif // EXTENSIONS_SENSAS_H
```

DSA_Framework/extensions_sensas.cpp

```
1 /**
2  * @file extensions_sensas.cpp
3  * This file implements the constructor and methods
4  * for the class discribed in extensions_sensas.h file.
5  *
6  * @author Jens Krayenborg
7  * @date 13.01.2013
8  *
9  * @version 1.0.0
10 */
11
12 #include "extensions_sensas.h"
```

```
13
14 /**
15  * @brief PropPermPet::PropPermPet
16  * The constructor for the PropPermPet class.
17  */
18 PropPermPet::PropPermPet(Property* prop,
19                          Permission perm,
20                          bool pet)
21 {
22     property = prop;
23     permission = perm;
24     petition = pet;
25 }
26
27 /**
28  * @brief Extensions_SensAs::Extensions_SensAs
29  * The constructor.
30  *
31  *
32  * @param id            The identification number of the
       DSA_SensA
33  *                      instance sensAc points to.
34  * @param props         The properties, permissions and
35  *                      petitions objects for all properties
36  *                      the instance sensAc has rights to.
37  * @param history_size  The size, the history should have.
38  *                      Have to be equal to the buffer size
39  *                      of the extension.
40  * @param db            The current content of the
41  *                      data buffer of the calling extension.
42  */
43 /*removed DSA_SensA* sensAc as parameter
44  * @param sensAc        The reference to the DSA_SensA
       instance
45  *                      for which these management infos are
       used.
46  */
47 Extensions_SensAs::Extensions_SensAs(SensA_ID id, std::list<
      PropPermPet> props, size_t history_size, DSA_DO_List* db)
48 {
49     sensaID = id;
50     properties = props;
51     history.set_Size(history_size);
```

```
52
53      // Have to do this to prevent sensA from getting all old
            data while calling get_new_data.
54      for (DSA_DO_List::iterator i = db->begin();
55           i != db->end();
56           i++){
57          DSA_Data_Object* ddo = *i;
58          history.add_to_buffer(ddo->dataID);
59      }
60  }
61
62  /**
63   * @brief Extensions_SensAs::search_PPP
64   * Getting the property permission petition object
65   * with the property identified by the given
66   * property type.
67   *
68   * @param pt     The given property type.
69   * @return       The property permission petition object
70   *               if it could be found.
71   *               NULL if not.
72   */
73  PropPermPet* Extensions_SensAs::search_PPP(Property_T pt)
74  {
75      for (std::list<PropPermPet>::iterator it = properties.
            begin();
76           it != properties.end();
77           ++it){
78          PropPermPet* ppp = &(*it);
79          if (ppp->property->property_Type.compare(pt) == 0)
                return ppp;
80      }
81      return NULL;
82  }
83
84  /**
85   * @brief Extensions_SensAs::has_right_to
86   * Prooving if the DSA_SensA instance identified by the
        reference
87   * containing in this instance has the given permission to
        the
88   * property identified by the given property type.
89   *
```

```
90  * @param ptype     The property given property type.
91  * @param perm      The permission to check.
92  * @return          The result.
93  *                  False if the property could not
94  *                  be found or the sensa has not the
95  *                  permission asked for. True
96  *                  otherwise.
97  */
98 bool Extensions_SensAs::has_right_to(Property_T ptype,
      Permission perm)
99 {
100     PropPermPet* p = search_PPP(ptype);
101     if (p == NULL) return false;
102     /* If asking for READ then READ_WRITE is also true, so
          checking
103      * against WRITE. */
104     switch (perm){
105     case READ:  if (p->permission == WRITE) return false;
106     case WRITE: if (p->permission == READ) return false;
107     }
108
109     return true;
110 }
```

DSA_Framework/extension_configuration.h

```
1  #ifndef SENSA_CONFIGURATION_H
2  #define SENSA_CONFIGURATION_H
3
4  /**
5   * @file extension_configuration.h
6   * This class realizes the Extension_Configuration.
7   * This is an abstract class and gives the model
8   * for implementing the spezializations for the
9   * use of the template pattern in interrelation with
10  * the DSA_Extension class.
11  *
12  * The methods with the prefix __DSA_ shpuld never
13  * be reimplemented/overloaded. Otherwise the
14  * consistent and correct funktionality of the
15  * DSA-Framework cannot be ensured.
16  *
17  * The method make_decision() may, but not have
18  * to be reimplemented. If one would use the
```

```
19  * application funktionality of this framework this
20  * method must be reimplemented.
21  *
22  * @author Jens Krayenborg
23  * @date 13.01.2013
24  *
25  * @version 1.0.0
26  *
27  * Modified by
28  * @author Connor Fibich
29  * @date 15.10.13
30  *
31  * @version 1.0.0
32  */
33
34 #include "property.h"
35 #include "dsa_data_object.h"
36 #include "dsa_data_buffer.h"
37
38
39 //Renamed to SensA_Configuration
40 class Extension_Configuration
41 {
42 public:
43     /* Operations */
44     /**
45      * @brief get_Expenses
46      * This function ist to implement. It should calculate
47      * the actual costs, the sensor or actuator will
48      * consume.
49      *
50      * @return The data object which contains the costs
51      * of the sensor/actuator.
52      */
53     virtual DSA_Data_Object* get_Expenses() = 0;
54
55     /**
56      * @brief set_Property
57      * This function ist to implement. It should perform
58      * the setting of the property given in pso to the
59      * value given in pso.
60      *
61      * @param pso   The object, which holds the values for
```

```
62       *              setting the property.
63       * @return      A FaultValue if somethings went wrong.
64       *              One can use any value, because it still
65       *              will be sent back to the informatic
66       *              system.
67       */
68      virtual FaultValue set_Property(Property_Set_Object* pso)
            = 0;
69
70      /**
71       * @brief update_data_buffer
72       * This function ist to implement. It should perform
73       * the fetching of data from the sensor. The
            DSA_Extension
74       * will call this function before fetching the
            data_buffer.
75       * The data fetched from the sensor have to be put in
76       * a DSA_Data_Object and stored in the data_buffer.
77       * @return      A FaultValue if somethings went wrong.
78       *              One can use any value, because it still
79       *              will be sent back to the informatic
80       *              system.
81       */
82      virtual FaultValue update_data_buffer() = 0;
83
84      virtual Permission_List* get_standard_Permissions() = 0;
85      /**
86       * @brief make_decision
87       * You have to implement this function, if you like to
88       * use the application system. This function have to
89       * implement the strategie to decide which application
90       * should be used to set a property.
91       * @param claims A container with the applications
92       * @return      The position in claims, where the
            application
93       *              resides, which should be used to set the
94       *              property. If none of them should be used
95       *              use NOT_IMPLEMENTED. This will cause in
96       *              no property setting.
97       */
98      virtual Claim_Position make_decision(Claims* claims);
99
100     virtual void notify(DSA_Data_Object* ddo);
```

```
101
102     bool __DSA_isClaimProp(Property_T propType);
103     Property* __DSA_get_Property(Property_T propType);
104     Property* __DSA_get_Property(uint8_t index);
105     uint8_t __DSA_get_Property_index(Property_T propType);
106     DSA_DO_List* __DSA_get_data_buffer();
107     void __DSA_set_data_buffer_size(size_t size);
108     size_t __DSA_get_data_buffer_size();
109
110     /* Attributes */
111     SensA_ID ID;
112     Property_List properties;
113     DSA_Data_Buffer data_buffer;
114 };
115
116 #endif // EXTENSION_CONFIGURATION_H
```

DSA_Framework/extension_configuration.cpp

```
1  /**
2   * @file sensa_configuration.cpp
3   * previously known as extension_configuration.cpp.
4   *
5   * This file implements the constructor and methods
6   * for the class discribed in sensa_configuration.h file.
7   *
8   * @author Jens Krayenborg
9   * @date 13.01.2013
10  *
11  * @version 1.0.0
12  *
13  * Modified by
14  * @author Connor Fibich
15  * @date 15.10.13
16  *
17  * @version 1.0.0
18  */
19
20
21 #include "extension_configuration.h"
22
23 /**
24  * @brief SensA_Configuration::__DSA_isClaimProp
25  * Prooving if the property of the given type is
```

```
26  * a property for which the sensas has to claim when
27  * setting the property.
28  *
29  * @param propType  The type of the property.
30  * @return          True if the property is a claim property.
31  */
32 bool Extension_Configuration::__DSA_isClaimProp(Property_T
      propType)
33 {
34     if (__DSA_get_Property(propType)->claiming == true)
35         return true;
36
37     return false;
38 }
39
40 /**
41  * @brief SensA_Configuration::__DSA_get_Property
42  * Getting the property of the given type.
43  *
44  * @param propType  The type of the property.
45  * @return          The property.
46  *                  NULL if no property with the given
47  *                  type was found.
48  */
49 Property* Extension_Configuration::__DSA_get_Property(
      Property_T propType)
50 {
51     Property* p;
52     for (Property_List::iterator i = properties.begin();
53          i != properties.end();
54          ++i)
55     {
56         p = &(*i);
57         if (p->property_Type.compare(propType) == 0)
58             return p;
59     }
60     return NULL;
61 }
62
63 /**
64  * @brief SensA_Configuration::make_decision
65  * The dummy function of making a decision. Just returning
66  * the value NOT_IMPLEMENTED.
```

```
67  *
68  * @param claims    Not used.
69  * @return          Ever NOT_IMPLEMENTED
70  */
71 Claim_Position Extension_Configuration::make_decision(Claims
       *claims)
72 {
73     return NOT_IMPLEMENTED;
74 }
75
76 /**
77  * @brief SensA_Configuration::__DSA_get_data_buffer
78  * Getting the data buffer of this sensa_configuration
79  * instance.
80  *
81  * @return  the reference to the data buffer.
82  */
83 DSA_DO_List* Extension_Configuration::__DSA_get_data_buffer()
84 {
85     return data_buffer.get_buffer();
86 }
87
88      void Extension_Configuration::notify(DSA_Data_Object*
           ddo){}
89 /**
90  * @brief SensA_Configuration::__DSA_set_data_buffer_size
91  * Setting the size of the data buffer of this
92  * sensa_configuration instance to the given size.
93  *
94  * @param size  The new size for the buffer.
95  */
96 void Extension_Configuration::__DSA_set_data_buffer_size(
       size_t size)
97 {
98     data_buffer.set_Size(size);
99 }
100
101 /**
102  * @brief SensA_Configuration::__DSA_get_data_buffer_size
103  * Getting the date buffer size. It reflects not the real
       number
104  * of containing elements. Just the maximum number of
       containing
```

```
105  * elements.
106  *
107  * @return  The current maximum size of the buffer.
108  */
109 size_t Extension_Configuration::__DSA_get_data_buffer_size()
110 {
111     return data_buffer.get_size();
112 }
113
114
115 //-- new functions by Connor Fibich
116
117 /**
118  * @brief SensA_Configuration::__DSA_get_Property
119  * Getting the property with the given index.
120  *
121  * @param index  The index of the property.
122  * @return          The property.
123  *                  NULL if no property with the given
124  *                  type was found.
125  */
126 Property* Extension_Configuration::__DSA_get_Property(uint8_t
      index)
127 {
128     Property* p;
129     int j = 0;
130     for ( Property_List::iterator i = properties.begin();
131           i != properties.end(),j!=index;
132           ++i,++j)
133     {
134         p = &(*i);
135         if(j==index)
136        return p;
137     }
138     return NULL;
139 }
140
141 /**
142  * @brief SensA_Configuration::__DSA_get_Property_index
143  * Getting the index of the property of the given type.
144  *
145  * @param propType  The type of the property.
146  * @return          The index of the property.
```

```
147  *                   NULL if no property with the given
148  *                   type was found.
149  */
150 uint8_t Extension_Configuration::__DSA_get_Property_index(
      Property_T propType)
151 {
152     Property* p;
153     int j = 0;
154     for ( Property_List::iterator i = properties.begin();
155           i != properties.end();
156           ++i,++j)
157     {
158         p = &(*i);
159         if(p->property_Type.compare(propType) == 0)
160        return j;
161     }
162     return NULL;
163 }
```

DSA_Framework/extension_information.h

```
1  /*
2   * File:   extension_information.h
3   * Author: Connor Fibich
4   *
5   * Created on September 30, 2013, 4:50 PM
6   */
7
8  #ifndef EXTENSION_INFORMATION_H
9  #define  EXTENSION_INFORMATION_H
10 #include "dsa_types.h"
11 class Extension_Information {
12 public:
13     Extension_Information();
14     Extension_Information(const Extension_Information& orig);
15     virtual ~Extension_Information();
16
17     SensA_ID id;
18     uint8_t version;
19     SensA_Type type;
20     DSA_Data_Type dataType;
21     SensA_Location location;
22
23 private:
```

```
24 };
25
26 #endif   /* EXTENSION_INFORMATION_H */
```

DSA_Framework/extension_information.cpp

```
1 /*
2  * File:   sensa_information.cpp
3  * Author: krayn
4  *
5  * Created on September 30, 2013, 4:50 PM
6  */
7
8 #include "extension_information.h"
9
10 Extension_Information::Extension_Information() {
11 }
12
13 Extension_Information::Extension_Information(const
     Extension_Information& orig) {
14 }
15
16 Extension_Information::~Extension_Information() {
17 }
```

DSA_Framework/dsa_management.h

```
1 /*
2  * File:   dsa_management.h
3  * Author: krayn
4  *
5  * Created on September 30, 2013, 12:03 PM
6  */
7
8 #ifndef DSA_MANAGEMENT_H
9 #define  DSA_MANAGEMENT_H
10
11 #include "dsa_extension_container.h"
12 #include "dsa_types.h"
13 #include "connection_information.h"
14 #include "proxy_container.h"
15 #include "extension_container.h"
16 #include "dsa_sensa.h"
17
18
```

```
19
20 class DSA_Management {
21 public:
22     ~DSA_Management();
23     static DSA_Management* get_instance();
24     void connect_remote_container(Connection_Information*
          conn);
25     FaultValue connect_SensA(DSA_SensA* sensA, SensA_ID sID);
26     FaultValue disconnect_SensA(DSA_SensA* sensA);
27     DSA_Extension* get_SensA(SensA_ID sensaID);
28     SensA_ID create_Extension(Extension_Configuration* sensa)
          ;
29     SensA_Info_list get_SensA_list();
30     FaultValue set_Property(SensA_ID sensaID, Property prop);
31     Property_List get_Propertys(SensA_ID sensaID, Property
          prop);
32     Observer_ID get_new_observer_ID();
33     DSA_SensA* get_DSA_SensA();
34     DSA_Data_Object* get_Expenses(DSA_SensA* sensa);
35     DSA_Data_Object* get_Expenses(SensA_ID sensa);
36     SensA_ID_List get_SensA_ID_list();
37     void add_Connection_Manager(Connection_Manager* man);
38 protected:
39     // Constructor
40     DSA_Management();
41
42 private:
43     // Forcing the copy-contructor to be private,
44     // to eliminate the chance to use it to declare
45     // additional instances of this class.
46     DSA_Management(DSA_Management&);
47     void add_container(DSA_Extension_Container* cont);
48     // Attributes
49     static DSA_Management* instance;
50     std::list<DSA_Extension_Container*> containers;
51     Cont_ID last_cont_ID;
52     Observer_ID last_obs_ID;
53     SensA_ID last_sensa_ID;
54 };
55
56 #endif   /* DSA_MANAGEMENT_H */
```

DSA_Framework/dsa_management.cpp

```
1  /**
2   * @file dsa_management.cpp
3   *
4   */
5
6  //#include <list>
7
8  #include "dsa_management.h"
9
10 /**
11  * Constructor of the DSA_Management. Creates the local
        SensA_Container.
12  */
13 DSA_Management::DSA_Management() {
14      last_cont_ID = 1;
15      last_obs_ID = 0;
16      last_sensa_ID.first = 0;
17      last_sensa_ID.second = 0;
18      add_container(new Extension_Container(last_cont_ID));
19 }
20
21 /**
22  * @brief DSA_Management::~Extension_Container
23  * A simple destructor.
24  */
25 DSA_Management::~DSA_Management() {
26      Connectionhandler::get_Instance()->~Connectionhandler();
27      if (instance != 0) delete(instance);
28 }
29
30 DSA_Management* DSA_Management::instance = 0;
31
32 /**
33  * @brief DSA_Management::get_instance
34  * Getting the one and only instance of this container.
35  *
36  * @return  The one and only instance.
37  */
38 DSA_Management* DSA_Management::get_instance() {
39      if (instance == 0)
40          instance = new DSA_Management();
41      return instance;
```

```
42 }
43
44 void DSA_Management::add_container(DSA_Extension_Container*
       cont) {
45     containers.push_back(cont);
46 }
47
48 void DSA_Management::connect_remote_container(
       Connection_Information* conn) {
49     Connectionhandler::get_Instance()->
           add_Connection_Information(conn);
50     last_cont_ID++;
51     Proxy_Container* pCon;
52     pCon = new Proxy_Container(last_cont_ID, conn);
53     if (pCon->test_connect() == S_OK) {
54         add_container(pCon);
55     }
56 }
57
58 /**
59  *
60  * @return a list containing information about all SensA
       stored within the connected Containers
61  */
62 SensA_Info_list DSA_Management::get_SensA_list() {
63     SensA_Info_list list;
64     DSA_Extension_Container* dsaCont;
65     for (std::list<DSA_Extension_Container*>::iterator it =
           containers.begin();
66              it != containers.end();
67              ++it) {
68         dsaCont = *it;
69           SensA_Info_list list2 = dsaCont->get_SensA_list();
70         list.merge(list2);
71
72     }
73     return list;
74 }
75
76 DSA_Extension* DSA_Management::get_SensA(SensA_ID sensaID) {
77     DSA_Extension_Container* dsaCont = NULL;
78     for (std::list<DSA_Extension_Container*>::iterator it =
           containers.begin();
```

```
79              it != containers.end();
80              ++it) {
81
82          if ((*it)->get_ID() == sensaID.first) {
83              dsaCont = *it;
84          }
85      }
86      if (dsaCont == NULL) {
87          return NULL;
88      }
89      return dsaCont->get_SensA(sensaID.second);
90  }
91
92  Observer_ID DSA_Management::get_new_observer_ID() {
93      return ++last_obs_ID;
94  }
95
96
97  /**
98   * @brief Create a DSA_virt_SensA with the given
        SensA_Configuration
99   * @param conf the Configuration of the SensA
100  * @return The SensA_ID of the created SensA
101  */
102 SensA_ID DSA_Management::create_Extension(
      Extension_Configuration* conf) {
103     Extension_Container* cont = (Extension_Container*) (
          containers.front());
104     return cont->add_SensA(new DSA_Extension(cont->
          last_sensa_id,conf));
105 }
106
107 /**
108  * @brief Create a DSA_SensA that can be used by the
        informatic system.
109  * @return a pointer of a new DSA_SensA
110  */
111 DSA_SensA* DSA_Management::get_DSA_SensA() {
112     last_sensa_ID.second++;
113     return new DSA_SensA(last_sensa_ID);
114 }
115
```

```
116 FaultValue DSA_Management::connect_SensA(DSA_SensA* sensA,
      SensA_ID sID) {
117     DSA_Extension* sen = get_SensA(sID);
118     Permission_List* perms = sen->get_standard_Permissions();
119      FaultValue ret = sen->connect(sensA->id, perms);
120      ret  = sensA->connect_SensA(sen);
121 }
122
123 FaultValue DSA_Management::disconnect_SensA(DSA_SensA* sensA)
       {
124     SensA_ID sID = sensA->get_connected_SensA();
125
126     FaultValue ret  = sensA->disconnect_SensA();
127     if(ret == S_OK){
128     DSA_Extension* sen = get_SensA(sID);
129     ret = sen->disconnect(sensA->id);
130     }
131     return ret;
132 }
133
134
135 DSA_Data_Object* DSA_Management::get_Expenses(SensA_ID sID)
136 {
137
138     DSA_Extension* sen = get_SensA(sID);
139     if(sen == NULL)
140         return NULL;
141    return sen->get_Expenses();
142 }
143
144 DSA_Data_Object* DSA_Management::get_Expenses(DSA_SensA*
      sensA)
145 {
146    return get_Expenses(sensA->get_connected_SensA());
147 }
148
149
150     SensA_ID_List DSA_Management::get_SensA_ID_list(){
151         SensA_ID_List list;
152         DSA_Extension_Container* dsaCont;
153         for (std::list<DSA_Extension_Container*>::iterator it
              = containers.begin();
154             it != containers.end();
```

```
155             ++it) {
156             dsaCont = *it;
157             SensA_ID_List list2 = dsaCont->get_SensA_ID_list
                    ();
158             list.merge(list2);
159         }
160         return list;
161     }
162
163
164
165     void DSA_Management::add_Connection_Manager(
            Connection_Manager* man){
166         man->cont_id = ++last_cont_ID;
167         Connectionhandler::get_Instance()->
                add_Connection_Manager(man);
168     }
```

DSA_Framework/dsa_extension_container.h

```
1  /*
2   * File:   dsa_extension_container.h
3   * Author: Connor Fibich
4   *
5   * Created on September 30, 2013, 12:02 PM
6   */
7
8  #ifndef DSA_EXTENSION_CONTAINER_H
9  #define  DSA_EXTENSION_CONTAINER_H
10
11 #include "dsa_extension.h"
12 #include "extension_information.h"
13 #include "dsa_types.h"
14
15 class DSA_Extension_Container {
16 public:
17     DSA_Extension_Container(Cont_ID id);
18     Cont_ID get_ID();
19     virtual SensA_Info_list get_SensA_list();
20     virtual Extension_Information* get_SensA_information(
           Cont_SensA_ID sensID) = 0;
21     virtual DSA_Extension* get_SensA(Cont_SensA_ID sensID) =
           0;
22     virtual SensA_ID_List get_SensA_ID_list();
```

```
23 protected:
24     std::list<DSA_Extension*> sensas;
25     Cont_ID id;
26 private:
27 };
28
29 #endif   /* DSA_EXTENSION_CONTAINER_H */
```

DSA_Framework/dsa_extension_container.cpp

```
1  /*
2   * File:   dsa_sensa_container.cpp
3   * Author: krayn
4   *
5   * Created on September 30, 2013, 5:48 PM
6   */
7
8  #include "dsa_extension_container.h"
9
10 DSA_Extension_Container::DSA_Extension_Container(Cont_ID id)
       {
11     this->id = id;
12 }
13
14 Cont_ID DSA_Extension_Container::get_ID() {
15     return id;
16 }
17
18 SensA_Info_list DSA_Extension_Container::get_SensA_list() {
19     DSA_Extension* dsaSen;
20     SensA_Info_list list;
21     for (std::list<DSA_Extension*>::iterator it = sensas.
          begin();
22              it != sensas.end();
23              ++it) {
24         dsaSen = *it;
25         list.push_back(dsaSen->get_SensA_information());
26     }
27     return list;
28 }
29
30 SensA_ID_List DSA_Extension_Container::get_SensA_ID_list() {
31     DSA_Extension* dsaSen;
32     SensA_ID_List list;
```

```
33      for (std::list<DSA_Extension*>::iterator it = sensas.
           begin();
34              it != sensas.end();
35              ++it) {
36
37           dsaSen = *it;
38           list.push_back(dsaSen->get_ID());
39      }
40      return list;
41  }
```

DSA_Framework/extension_container.h

```
1  /*
2   * File:   extension_container.h
3   * Author: Connor Fibich
4   *
5   * Created on September 30, 2013, 12:27 PM
6   */
7
8  #ifndef EXTENSION_CONTAINER_H
9  #define  EXTENSION_CONTAINER_H
10
11 #include "dsa_extension_container.h"
12
13 class Extension_Container : public DSA_Extension_Container {
14 public:
15      Extension_Container(Cont_ID id);
16      ~Extension_Container();
17      DSA_Extension* get_SensA(Cont_SensA_ID sensID);
18      SensA_Info_list get_SensA_list();
19      Extension_Information* get_SensA_information(
           Cont_SensA_ID sensID);
20      SensA_ID add_SensA(DSA_Extension* sensA);
21      void remove_SensA(SensA_ID sensID);
22      SensA_ID last_sensa_id;
23
24 protected:
25
26 private:
27      Extension_Container(Extension_Container&);
28
29      // Attributes
30 };
```

```
31
32 #endif   /* EXTENSION_CONTAINER_H */
```

DSA_Framework/extension_container.cpp

```
1  /*
2   * File:   sensa_container.cpp
3   * Author: krayn
4   *
5   * Created on September 30, 2013, 12:27 PM
6   */
7
8  #include <list>
9
10 #include "extension_container.h"
11
12 Extension_Container::Extension_Container(Cont_ID id) :
       DSA_Extension_Container(id) {
13     last_sensa_id.first = id;
14     last_sensa_id.second = 1;
15 }
16
17 //SensA_Container::SensA_Container(const SensA_Container&
       orig) {
18 //}
19
20 Extension_Container::~Extension_Container() {
21 }
22
23 SensA_ID Extension_Container::add_SensA(DSA_Extension* ext) {
24     sensas.push_back(ext);
25     last_sensa_id.second++;
26     return ext->get_ID();
27 }
28
29 DSA_Extension* Extension_Container::get_SensA(Cont_SensA_ID
       sID) {
30     DSA_Extension* dsaSen;
31     std::list<DSA_Extension*> copy = std::list<DSA_Extension
           *> (sensas);
32     for (std::list<DSA_Extension*>::iterator it = copy.begin
           ();
33              it != copy.end() && !sensas.empty();
34              ++it) {
```

```
35          dsaSen = *it;
36          if (dsaSen->get_ID().second == sID) {
37              return dsaSen;
38          }
39      }
40      return NULL;
41  }
42
43  SensA_Info_list Extension_Container::get_SensA_list() {
44      DSA_Extension* dsaSen;
45      SensA_Info_list list;
46      for (std::list<DSA_Extension*>::iterator it = sensas.
           begin();
47              it != sensas.end() && !sensas.empty();
48              ++it) {
49          dsaSen = *it;
50          list.push_back(dsaSen->get_SensA_information());
51      }
52      return list;
53  }
54
55  Extension_Information* Extension_Container::
       get_SensA_information(Cont_SensA_ID sensID) {
56      DSA_Extension* dsaSen;
57      for (std::list<DSA_Extension*>::iterator it = sensas.
           begin();
58              it != sensas.end() && !sensas.empty();
59              ++it) {
60          dsaSen = *it;
61          if(dsaSen->get_ID().second == sensID)
62          return dsaSen->get_SensA_information();
63      }
64      return NULL;
65  }
```

DSA_Framework/proxy_extension.h

```
1  /*
2   * File:   Proxy_SensA.h
3   * Author: krayn
4   *
5   * Created on September 30, 2013, 5:48 PM
6   */
7
```

```
8  #ifndef PROXY_SENSA_H
9  #define  PROXY_SENSA_H
10
11 #include "dsa_types.h"
12 #include "dsa_extension.h"
13 #include "connection_information.h"
14 #include "connectionhandler.h"
15 #include "connection.h"
16
17 class Proxy_Configuration : public Extension_Configuration {
18     /* Operations */
19
20     public:
21     virtual DSA_Data_Object* get_Expenses();
22
23     virtual FaultValue set_Property(Property_Set_Object* pso)
          ;
24
25     virtual FaultValue update_data_buffer();
26
27     virtual Permission_List* get_standard_Permissions();
28
29     virtual Property_List* __DSA_get_remote_Propertys();
30
31     Connection_Information* conn_info;
32     Permission_List standard_perms;
33     SensA_ID sensaID;
34     Connection_Manager* man;
35 };
36
37 class Proxy_Extension : public DSA_Extension {
38 public:
39     Proxy_Extension(SensA_ID sensaID,SensA_ID rid,
          Connection_Information* conn);
40     virtual ~Proxy_Extension();
41     FaultValue test_connect();
42     Extension_Information* get_SensA_information();
43    // DSA_DO_Container* get_new_data();
44     void register_observer(SensA_Observer* obs);
45     void remove_observer(Observer_ID obsID);
46     void notify(DSA_Data_Object* ddo);
47
48 protected:
```

```
49      friend class Proxy_Container;
50      // Proxy_SensA(const Proxy_SensA& orig);
51      FaultValue update_Config();
52
53  private:
54      FaultValue update_SensA_information(uint8_t version);
55      Proxy_Configuration* get_conf();
56      Extension_Information* sensA_info;
57      Connection_Information* conn_info;
58  };
59
60  #endif   /* PROXY_SENSA_H */
```

DSA_Framework/proxy_extension.cpp

```
1  /*
2   * File:   Proxy_SensA.cpp
3   * Author: krayn
4   *
5   * Created on September 30, 2013, 5:48 PM
6   */
7
8  #include "proxy_extension.h"
9  #include <stdlib.h>
10 #include <iostream>
11
12 Proxy_Extension::Proxy_Extension(SensA_ID sensaID, SensA_ID
       rid, Connection_Information* conn) : DSA_Extension(sensaID
       , new Proxy_Configuration()) {
13     conn_info = conn;
14     get_conf()->conn_info = conn;
15     get_conf()->sensaID = rid;
16     configuration->properties.push_back(*(new Property("
          active", "bool", "false", READ_WRITE)));
17     configuration->properties.push_back(*(new Property("
          interval", "int", "1", READ_WRITE)));
18     get_conf()->man = Connectionhandler::get_Instance()->
          get_Connection_Manager(conn);
19 }
20
21 //Proxy_SensA::Proxy_SensA(const Proxy_SensA& orig) {
22 //}
23
24 Proxy_Extension::~Proxy_Extension() {
```

```
25 }
26
27 /**
28  * @brief Function to easyly cast the configuration to a
       Proxy configuration
29  * @return
30  */
31 Proxy_Configuration* Proxy_Extension::get_conf() {
32     return static_cast<Proxy_Configuration*> (configuration);
33 }
34
35 FaultValue Proxy_Extension::test_connect() {
36     Connection* conn = Connectionhandler::get_Instance()->
          create_connection(get_conf()->conn_info);
37     ConnectionResponse resp = conn->receive_response();
38     conn->send_response(_TEST_SENSA);
39     uint32_t i = get_conf()->sensaID.first;
40     conn->send_data(&i, sizeof (uint32_t));
41     i = get_conf()->sensaID.second;
42     conn->send_data(&i, sizeof (uint32_t));
43     resp = conn->receive_response();
44     switch (resp) {
45         case _S_OK:
46             Connectionhandler::get_Instance()->
                  get_Connection_Manager(conn->info)->
                  return_Connection(conn);
47             return S_OK;
48             break;
49
50         default: case _E_UNKSENSA:
51             Connectionhandler::get_Instance()->
                  get_Connection_Manager(conn->info)->
                  return_Connection(conn);
52             return E_UNKSENSA;
53             break;
54     }
55
56 }
57
58 FaultValue Proxy_Extension::update_SensA_information(uint8_t
       version) {
59     if (sensA_info == NULL  sensA_info->version != version) {
```

```
60          Extension_Information* inf = new
               Extension_Information();
61          inf->id = configuration->ID;
62          SensA_ID rid;
63          rid = get_conf()->sensaID;
64          Connection* conn = Connectionhandler::get_Instance()
               ->create_connection(conn_info);
65          conn->receive_response();
66          conn->send_response(_SENSA_INFO);
67          conn->send_data(&(rid.first), sizeof (Cont_ID));
68          conn->send_data(&(rid.second), sizeof (Cont_SensA_ID)
               );
69          conn->receive_SensA_info(inf);
70          conn->send_response(_S_OK);
71          Connectionhandler::get_Instance()->
               get_Connection_Manager(conn_info)->
               return_Connection(conn);
72
73          sensA_info = inf;
74      }
75      return S_OK;
76  }
77
78  Extension_Information* Proxy_Extension::get_SensA_information
        () {
79      if (sensA_info == NULL) {
80          update_SensA_information(0);
81      }
82
83      return sensA_info;
84  }
85
86  void Proxy_Extension::notify(DSA_Data_Object* ddo) {
87       if (sensA_info == NULL)
88      update_SensA_information(0);
89      //ddo->dataType = sensA_info->dataType;
90      if (ddo->typ == OT_DATA){
91          ddo->specID = configuration->ID;
92          ddo->dataType = sensA_info->dataType;
93          notify_observers(ddo);
94      }
95  }
96  /*
```

```
97 DSA_DO_Container* Proxy_SensA::get_new_data() {
98     Connection* conn = Connectionhandler::get_Instance()->
           create_connection(conn_info);
99     ConnectionResponse resp = conn->receive_response();
100
101    conn->send_response(_GET_DATA);
102    uint32_t i = get_conf()->sensaID.first;
103    conn->send_data(i);
104    i = get_conf()->sensaID.second;
105    conn->send_data(i);
106    resp = conn->receive_response();
107    switch (resp) {
108        case _S_OK:
109            Connectionhandler::get_Instance()->
                   get_Connection_Manager(conn->info)->
                   return_Connection(conn);
110            break;
111
112        default: case _E_UNKSENSA:
113
114
115            Connectionhandler::get_Instance()->
                   get_Connection_Manager(conn->info)->
                   return_Connection(conn);
116            break;
117    }
118
119 }
120  */
121
122 void Proxy_Extension::register_observer(SensA_Observer* obs)
       {
123     if (observers.empty()) {
124         Connection* conn = Connectionhandler::get_Instance()
               ->create_connection(conn_info);
125         ConnectionResponse resp = conn->receive_response();
126
127         conn->send_response(_REG_OBS);
128         uint32_t i = get_ID().second;
129         conn->send_data(&i, sizeof (uint32_t));
130         i = get_conf()->sensaID.first;
131         conn->send_data(&i, sizeof (uint32_t));
132         i = get_conf()->sensaID.second;
```

```
133         conn->send_data(&i, sizeof (uint32_t));
134         resp = conn->receive_response();
135         switch (resp) {
136             case _S_OK:
137
138                 observers.push_back(obs);
139                 Connectionhandler::get_Instance()->
                       get_Connection_Manager(conn->info)->
                       return_Connection(conn);
140                 break;
141
142             default: case _E_UNKSENSA:
143                 Connectionhandler::get_Instance()->
                       get_Connection_Manager(conn->info)->
                       return_Connection(conn);
144                 break;
145         }
146
147     } else
148         observers.push_back(obs);
149 }
150
151 void Proxy_Extension::remove_observer(Observer_ID obsID) {
152     DSA_Extension::remove_observer(obsID);
153     if (observers.empty()) {
154         Connection* conn = Connectionhandler::get_Instance()
              ->create_connection(conn_info);
155         ConnectionResponse resp = conn->receive_response();
156
157         conn->send_response(_RM_OBS);
158         uint32_t i = get_ID().second;
159         conn->send_data(&i, sizeof (uint32_t));
160         i = get_conf()->sensaID.first;
161         conn->send_data(&i, sizeof (uint32_t));
162         i = get_conf()->sensaID.second;
163         conn->send_data(&i, sizeof (uint32_t));
164         resp = conn->receive_response();
165         switch (resp) {
166             case _S_OK:
167                 Connectionhandler::get_Instance()->
                       get_Connection_Manager(conn->info)->
                       return_Connection(conn);
168                 break;
```

```
169
170             default: case _E_UNKSENSA:
171                 Connectionhandler::get_Instance()->
                       get_Connection_Manager(conn->info)->
                       return_Connection(conn);
172                 break;
173         }
174
175     }
176 }
177
178 DSA_Data_Object* Proxy_Configuration::get_Expenses() {
179     Connection* conn = man->connect(conn_info);
180     ConnectionResponse resp = conn->receive_response();
181     DSA_Data_Object* ddo;
182     ddo = new DSA_Data_Object(1);
183
184     ddo->typ = OT_EXPENSE;
185     ddo->specID = ID;
186     ddo->dataType.assign("Runtime_V");
187     conn->send_response(_GET_EXP);
188     uint32_t i = sensaID.first;
189     conn->send_data(&i, sizeof (uint32_t));
190     i = sensaID.second;
191     conn->send_data(&i, sizeof (uint32_t));
192     conn->receive_Data_Object(ddo);
193     Connectionhandler::get_Instance()->get_Connection_Manager
          (conn->info)->return_Connection(conn);
194
195
196     return ddo;
197 }
198
199
200 FaultValue Proxy_Configuration::set_Property(
        Property_Set_Object* pso) {
201     //Connection* conn = Connectionhandler::get_Instance()->
          create_connection(conn_info);
202     Connection* conn = man->connect(conn_info);
203     ConnectionResponse resp = conn->receive_response();
204     uint8_t id = __DSA_get_Property_index(pso->property_Type)
          ;
205     conn->send_response(_SET_PROP);
```

```
206     uint32_t i = sensaID.first;
207     conn->send_data(&i, sizeof (uint32_t));
208     i = sensaID.second;
209     conn->send_data(&i, sizeof (uint32_t));
210     conn->send_Property_Set_Object(&id, pso);
211     resp = conn->receive_response();
212     switch (resp) {
213         case _S_OK:
214             man->return_Connection(conn);
215             return S_OK;
216             break;
217 
218         default: case _E_UNKSENSA:
219             man->return_Connection(conn);
220             break;
221     }
222 
223 }
224 
225 FaultValue Proxy_Configuration::update_data_buffer() {
226 
227     Connection* conn = man->connect(conn_info);
228     ConnectionResponse resp = conn->receive_response();
229     conn->send_response(_GET_DATA);
230     uint32_t i = sensaID.first;
231     conn->send_data(&i, sizeof (uint32_t));
232     i = sensaID.second;
233     conn->send_data(&i, sizeof (uint32_t));
234     i = ID.second;
235     conn->send_data(&i, sizeof (uint32_t));
236     dsa_size_t k;
237     conn->receive_data(&k,sizeof (dsa_size_t));
238 
239     while(k--){
240         DSA_Data_Object* ddo = new DSA_Data_Object(1);
241         ddo->typ = OT_DATA;
242         ddo->specID = ID;
243         conn->receive_Data_Object(ddo);
244         data_buffer.add_to_buffer(ddo);
245     }
246 
247 
248     return S_OK;
```

```
249 }
250
251 Property_List* Proxy_Configuration::
        __DSA_get_remote_Propertys() {
252     Property_List* _PA = new Property_List;
253     if (properties.empty()) {
254
255         Connection* conn = Connectionhandler::get_Instance()
                ->create_connection(conn_info);
256         ConnectionResponse resp = conn->receive_response();
257         conn->send_response(_GET_PROP);
258         uint32_t i = sensaID.first;
259         conn->send_data(&i, sizeof (uint32_t));
260         i = sensaID.second;
261         conn->send_data(&i, sizeof (uint32_t));
262         conn->receive_Property_list(&properties);
263         Connectionhandler::get_Instance()->
                get_Connection_Manager(conn->info)->
                return_Connection(conn);
264
265     }
266     _PA = &properties;
267
268     //_PA->push_back(PP_Tupel("interval", READ_WRITE));
269     //_PA->push_back(PP_Tupel("active", READ_WRITE));
270     return _PA;
271
272 }
273
274 Permission_List* Proxy_Configuration::
        get_standard_Permissions() {
275     Permission_List* _PA = new Permission_List;
276     if (standard_perms.empty()) {
277
278         Connection* conn = Connectionhandler::get_Instance()
                ->create_connection(conn_info);
279         ConnectionResponse resp = conn->receive_response();
280         conn->send_response(_GET_PERMS);
281         uint32_t i = sensaID.first;
282         conn->send_data(&i, sizeof (uint32_t));
283         i = sensaID.second;
284         conn->send_data(&i, sizeof (uint32_t));
285         conn->receive_Perm_list(_PA);
```

```
286         standard_perms = *_PA;
287         Connectionhandler::get_Instance()->
               get_Connection_Manager(conn->info)->
               return_Connection(conn);
288     } else {
289         _PA = &standard_perms;
290     }
291     //_PA->push_back(PP_Tupel("interval", READ_WRITE));
292     //_PA->push_back(PP_Tupel("active", READ_WRITE));
293     return _PA;
294
295 }
296
297
298 #include <unistd.h>
299
300 FaultValue Proxy_Extension::update_Config() {
301     get_conf()->properties.clear();
302     get_conf()->__DSA_get_remote_Propertys();
303     get_conf()->standard_perms.clear();
304     get_conf()->get_standard_Permissions();
305     update_SensA_information(0);
306 }
```

DSA_Framework/remote_observer.h

```
1 /*
2  * File:   remote_observer.h
3  * Author: krayn
4  *
5  * Created on October 2, 2013, 11:30 AM
6  */
7
8 #ifndef REMOTE_OBSERVER_H
9 #define  REMOTE_OBSERVER_H
10
11 #include "sensa_observer.h"
12 #include "connection_information.h"
13 #include "connection.h"
14 #include "connectionhandler.h"
15 #include "dsa_do_container.h"
16
17 class Remote_Observer : public SensA_Observer {
18 public:
```

```
19      Remote_Observer(Cont_SensA_ID obsID,
           Connection_Information* conninfo);
20      Remote_Observer(const Remote_Observer& orig);
21      virtual ~Remote_Observer();
22      void notify(DSA_Data_Object* ddo);
23      DSA_DO_Container* get_new_Data(SensA_ID id);
24
25      Cont_SensA_ID sensaID;
26      Connection_Information* conn_info;
27  private:
28  };
29
30  #endif   /* REMOTE_OBSERVER_H */
```

DSA_Framework/remote_observer.cpp

```
1  /*
2   * File:   remote_observer.cpp
3   * Class to identify a local SensA on a remote System
4   * Author: krayn
5   *
6   * Created on October 2, 2013, 11:30 AM
7   */
8
9  #include "remote_observer.h"
10
11 Remote_Observer::Remote_Observer(Cont_SensA_ID sensaID,
       Connection_Information* conninfo) : SensA_Observer() {
12      this->sensaID = sensaID;
13      this->conn_info = conninfo;
14 }
15
16 Remote_Observer::Remote_Observer(const Remote_Observer& orig)
       {
17 }
18
19 Remote_Observer::~Remote_Observer() {
20 }
21
22 void Remote_Observer::notify(DSA_Data_Object* ddo) {
23      Connection* conn = Connectionhandler::get_Instance()->
           create_connection(conn_info);
24      conn->receive_response();
25      conn->send_response(_NOTIFY);
```

```
26      conn->send_data(&sensaID, sizeof (Cont_SensA_ID));
27
28      conn->send_Data_Object(ddo);
29
30      Connectionhandler::get_Instance()->get_Connection_Manager
            (conn_info)->return_Connection(conn);
31
32  }
33
34  DSA_DO_Container* Remote_Observer::get_new_Data(SensA_ID id)
        {
35      DSA_Extension* sen = DSA_Management::get_instance()->
            get_SensA(id);
36      DSA_DO_Container* con;
37      con = sen->get_new_Data(SensA_ID(Connectionhandler::
            get_Instance()->get_Connection_Manager(conn_info)->
            cont_id, get_observer_ID()));
38
39
40
41      return con;
42  }
```

DSA_Framework/proxy_container.h

```
1  /*
2   * File:   proxy_sensa_container.h
3   * Author: krayn
4   *
5   * Created on September 30, 2013, 4:15 PM
6   */
7
8  #ifndef PROXY_SENSA_CONTAINER_H
9  #define  PROXY_SENSA_CONTAINER_H
10
11 #include "dsa_extension_container.h"
12 #include "dsa_types.h"
13 #include "connection_information.h"
14 #include "proxy_extension.h"
15 #include "connectionhandler.h"
16
17 class Proxy_Container : public DSA_Extension_Container {
18 public:
```

```
19      Proxy_Container(Cont_ID contID, Connection_Information*
           conninfo);
20      virtual ~Proxy_Container();
21      FaultValue test_connect();
22      SensA_Info_list get_SensA_list();
23      Extension_Information* get_SensA_information(
           Cont_SensA_ID sensID);
24      DSA_Extension* get_SensA(Cont_SensA_ID sensID);
25      SensA_ID_List get_SensA_ID_list();
26
27      SensA_ID convert_SensA_ID(SensA_ID id, bool dir);
28
29
30  protected:
31      // Proxy_SensA_Container(const Proxy_SensA_Container&
           orig);
32
33  private:
34      Connection_Information* connectioninfo;
35      std::list<std::pair<SensA_ID, SensA_ID> > rlconv;
36      Cont_SensA_ID last_SensA_ID;
37      char* address;
38      int port;
39
40  };
41
42  #endif   /* PROXY_SENSA_CONTAINER_H */
```

DSA_Framework/proxy_container.cpp

```
1   /*
2    * File:   proxy_sensa_container.cpp
3    * Author: krayn
4    *
5    * Created on September 30, 2013, 4:15 PM
6    */
7
8   #include <string.h>
9
10  #include <stdlib.h>
11  #include <iostream>
12  #include "proxy_container.h"
13
```

```
14 Proxy_Container::Proxy_Container(Cont_ID contID,
       Connection_Information* conninfo) :
       DSA_Extension_Container(contID) {
15      connectioninfo = conninfo;
16      connectioninfo->id = contID;
17      last_SensA_ID = 0;
18 }
19
20 //Proxy_SensA_Container::Proxy_SensA_Container(const
       Proxy_SensA_Container& orig) {
21 //}
22
23 Proxy_Container::~Proxy_Container() {
24 }
25
26 DSA_Extension* Proxy_Container::get_SensA(Cont_SensA_ID
       sensID) {
27      DSA_Extension* dsaSen;
28      for (std::list<DSA_Extension*>::iterator it = sensas.
           begin();
29              it != sensas.end();
30              ++it) {
31          dsaSen = *it;
32          if (dsaSen->get_ID().second == sensID) {
33              return dsaSen;
34          }
35      }
36      SensA_ID sensAid, rid;
37      sensAid.first = get_ID();
38      sensAid.second = sensID;
39      rid = convert_SensA_ID(sensAid, true);
40      //delete(dsaSen);
41      Proxy_Extension* proxySen = new Proxy_Extension(sensAid,
           rid, connectioninfo);
42      if (proxySen->test_connect() == S_OK) {
43          proxySen->update_Config();
44          std::cout << "ok" << std::endl;
45          sensas.push_back(proxySen);
46          return proxySen;
47      } else {
48
49          std::cout << "err" << std::endl;
50          return NULL;
```

```
51      }
52
53  }
54
55  SensA_ID Proxy_Container::convert_SensA_ID(SensA_ID id, bool
        dir) {
56      SensA_ID cid;
57      std::pair<SensA_ID, SensA_ID> rlc;
58      for (std::list<std::pair<SensA_ID, SensA_ID> >::iterator
            it = rlconv.begin(); it != rlconv.end(); ++it) {
59
60          rlc = *it;
61          if (dir) {
62              if (rlc.first == id) {
63                  return rlc.second;
64              }
65          } else {
66              if (rlc.second == id) {
67                  return rlc.first;
68              }
69          }
70      }
71      return SensA_ID(-1, -1);
72  }
73
74  FaultValue Proxy_Container::test_connect() {
75      Connection* conn = Connectionhandler::get_Instance()->
            create_connection(connectioninfo);
76      ConnectionResponse resp = conn->receive_response();
77      conn->send_response(_TEST_CONTAINER);
78      resp = conn->receive_response();
79      switch (resp) {
80          case _S_OK:
81          Connectionhandler::get_Instance()->
                get_Connection_Manager(connectioninfo)->
                return_Connection(conn);
82              return S_OK;
83              break;
84
85          default: case _E_UNKSENSA:
86          Connectionhandler::get_Instance()->
                get_Connection_Manager(connectioninfo)->
                return_Connection(conn);
```

```
87              return E_UNKSENSA;
88              break;
89      }
90
91  }
92
93  SensA_Info_list Proxy_Container::get_SensA_list() {
94      get_SensA_ID_list();
95
96      Connection* conn;
97      ConnectionResponse resp;
98
99      SensA_Info_list list;
100     for (std::list<std::pair<SensA_ID, SensA_ID> >::iterator
           it = rlconv.begin(); it != rlconv.end(); ++it) {
101         std::pair<SensA_ID, SensA_ID> rlc;
102         rlc = *it;
103         conn = Connectionhandler::get_Instance()->
              create_connection(connectioninfo);
104         resp = conn->receive_response();
105         conn->send_response(_SENSA_INFO);
106         conn->send_data(&(rlc.second.first), sizeof (Cont_ID)
              );
107         conn->send_data(&(rlc.second.second), sizeof (
              Cont_SensA_ID));
108         Extension_Information* inf = new
              Extension_Information();
109         inf->id = rlc.first;
110         conn->receive_SensA_info(inf);
111         conn->send_response(_S_OK);
112         list.push_back(inf);
113         Connectionhandler::get_Instance()->
              get_Connection_Manager(connectioninfo)->
              return_Connection(conn);
114     }
115     return list;
116 }
117
118 Extension_Information* Proxy_Container::get_SensA_information
      (Cont_SensA_ID sensID) {
119
120     Extension_Information* inf = new Extension_Information();
121         inf->id = SensA_ID(id,sensID);
```

```
122         SensA_ID rid;
123         rid = convert_SensA_ID(inf->id, true);
124         Connection* conn = Connectionhandler::get_Instance()
               ->create_connection(connectioninfo);
125         conn->receive_response();
126         conn->send_response(_SENSA_INFO);
127         conn->send_data(&(rid.first), sizeof (Cont_ID));
128         conn->send_data(&(rid.second), sizeof (Cont_SensA_ID)
               );
129         conn->receive_SensA_info(inf);
130         conn->send_response(_S_OK);
131         Connectionhandler::get_Instance()->
               get_Connection_Manager(connectioninfo)->
               return_Connection(conn);
132
133     return inf;
134 }
135
136
137 SensA_ID_List Proxy_Container::get_SensA_ID_list(){
138     SensA_ID_List list;
139     Connection* conn = Connectionhandler::get_Instance()->
           create_connection(connectioninfo);
140     ConnectionResponse resp = conn->receive_response();
141     conn->send_response(_GET_SENSA_IDS);
142     dsa_size_t i;
143     conn->receive_data(&i, sizeof (dsa_size_t));
144     while (i--) {
145         SensA_ID sid;
146         conn->receive_data(&(sid.first), sizeof (Cont_ID));
147         conn->receive_data(&(sid.second), sizeof (
               Cont_SensA_ID));
148         conn->send_response(_S_OK);
149         if (convert_SensA_ID(sid, false) == SensA_ID(-1, -1))
               {
150             std::pair<SensA_ID, SensA_ID> rlc;
151             rlc.first = SensA_ID(id, ++last_SensA_ID);
152             rlc.second = sid;
153             rlconv.push_back(rlc);
154         }
155         list.push_back(sid);
156     }
```

```
157     Connectionhandler::get_Instance()->get_Connection_Manager
           (connectioninfo)->return_Connection(conn);
158     return list;
159 }
```

DSA_Framework/connection.h

```
1
2
3  /**
4   * @file:   Connection.h
5   * Functions that send/receive Objects can be reimplemented
       if its needed by the Connection-Protocol
6   * @author: Connor Fibich
7   *
8   *
9   * Created on September 30, 2013, 4:05 PM
10  */
11 #pragma once
12 //#ifndef CONNECTION_H
13
14 //#define   CONNECTION_H
15
16 #include "connection_information.h"
17 #include "property.h"
18
19 class Connection {
20 public:
21
22     Connection(Connection_Information* conninfo);
23     virtual ~Connection();
24     /**
25       * @brief send data to the connection-partner.
26       * @param buffer, the char* to the data that needs to be
            send
27       * @param len, size of the data
28       * @return
29       */
30     virtual int send_data(void* buffer, size_t len) = 0;
31     /**
32       * @brief receive data from the connection-partner.
33       * @param buffer, the char* to the data-buffer where the
            received data will be stored
34       * @param len, expected size of the data-packet.
```

```
35       * @return
36       */
37      virtual int receive_data(void* buffer, size_t len) = 0;
38
39
40      /**
41       * @brief Interprets the next Data-Packet as a
            ConnectionResponse
42       * @return ConnectionResponse represented by the data-
            packet
43       */
44      virtual ConnectionResponse receive_response();
45      /**
46       * @brief Formats the ConnectionResponse for the used
            connection-protocol
47       */
48      virtual void send_response(ConnectionResponse resp);
49      virtual void send_ready();
50      /**
51       *
52       * @brief Receives a Property_Set_Object.
53       * @param prop, Pointer of Property_Set_Object in which
            the received Property_Set_Object will be saved
54       * @return
55       */
56      virtual ConnectionResponse receive_Property_Set_Object(
           Property_Set_Object* prop);
57      virtual ConnectionResponse receive_SensA_info(
           Extension_Information* info);
58      virtual ConnectionResponse receive_Perm_list(
           Permission_List* perms);
59      virtual ConnectionResponse receive_Property_list(
           Property_List* prop);
60      virtual ConnectionResponse receive_Data_Object(
           DSA_Data_Object* ddo);
61
62      /**
63       * @brief Sends a Property_Set_Object.
64       * @param prop, Pointer of Property_Set_Object that will
            be send
65       * @return
66       */
```

```
67      virtual ConnectionResponse send_Property_Set_Object(
           uint8_t* id, Property_Set_Object* prop);
68      virtual ConnectionResponse send_SensA_info(
           Extension_Information* info);
69      virtual ConnectionResponse send_Perm_list(Permission_List
           * perms);
70      virtual ConnectionResponse send_Property_list(
           Property_List* prop);
71      virtual ConnectionResponse send_Data_Object(
           DSA_Data_Object* ddo);
72      Connection_Information* info;
73  };
74
75
76  //#endif /* CONNECTION_H */
```

DSA_Framework/connection.cpp

```
1   #include <stdint.h>
2   #include <stdlib.h>
3
4   #include "connection.h"
5   #include "extension_information.h"
6
7   Connection::Connection(Connection_Information* conninfo) {
8       this->info = conninfo;
9   }
10
11  Connection::~Connection() {
12
13  }
14
15
16
17  #include <stdlib.h>
18  #include <iostream>
19  #include <string.h>
20
21  ConnectionResponse Connection::receive_Property_Set_Object(
       Property_Set_Object* prop) {
22      /*
23    SensA_ID sensA;
24    Property_T property_Type;
25    std::string value_Type;
```

```
26   std::string value;
27      */
28
29     uint8_t size;
30     //receive property_Type
31     //uint8_t id;
32     //receive_data(&id,1);
33     int resp = receive_data(&size, sizeof (uint8_t));
34     char* data = (char*) malloc(size);
35     resp = receive_data(data, size);
36     prop->value.assign(data);
37
38     free(data);
39     return _S_OK;
40 }
41
42 /**
43  *
44  * @param prop, Pointer of Property_Set_Object that will be
       send
45  * @return
46  */
47 ConnectionResponse Connection::send_Property_Set_Object(
      uint8_t* id, Property_Set_Object* prop) {
48
49
50
51     uint8_t temp;
52     temp = (prop->value.size() + 1) * sizeof (char); //size
          of chararray with \0
53
54     send_data(id, sizeof (uint8_t));
55     send_data(&temp, sizeof (uint8_t));
56     send_data(&(prop->value)[0], temp);
57
58     /*uint8_t* msg = (uint8_t*) malloc(size); //allocate a
          bytearray with the correct size to hold the size and
          values of a Property_Set_Object
59
60      memcpy(msg, &id, sizeof (uint8_t));
61       memcpy((msg+ sizeof (uint8_t)), &temp, sizeof (uint8_t)
            );
```

```
62      memcpy((msg + 2*sizeof (uint8_t)),&(prop->value)[0],
           temp);
63      memcpy((msg + 2*sizeof (uint8_t) + temp), &n, sizeof (
           char));
64
65      * Each value of a Property_Set-Object is writen into the
           bytearray.
66      * the 4 bytes leading each value hold the size_t of that
           value.
67      *
68     memcpy(msg, &temp, sizeof (uint8_t)); //write the size_t
        of property_type
69     memcpy((msg + sizeof (uint8_t)), &(prop->property_Type)
        [0], temp); //write property_Type in the next bytes
70     memcpy((msg + sizeof (uint8_t) + temp), &n, sizeof (char)
        );
71     std::cout << (size_t) * msg << (msg + sizeof (uint8_t))
        << (msg + sizeof (uint8_t) + temp);
72     send_data(msg, size);
73
74     size = sizeof (uint8_t) + sizeof (char); // calculate
        size of the Property_Set_Object
75     size += (prop->value_Type.size() + 1) * sizeof (char);
76     realloc(msg, size);
77     temp = (prop->value_Type.size() + 1) * sizeof (char);
78     memcpy(msg, &temp, sizeof (uint8_t));
79     memcpy(msg + sizeof (uint8_t), &(prop->value_Type)[0],
        temp);
80     memcpy(msg + sizeof (uint8_t) + temp, &n, sizeof (char));
81     std::cout << (size_t) * msg << (msg + sizeof (uint8_t))
        << (msg + sizeof (uint8_t) + temp);
82     send_data(msg, size);
83
84     size = sizeof (uint8_t) + sizeof (char); // calculate
        size of the Property_Set_Object
85     size += (prop->value.size() + 1) * sizeof (char);
86     realloc(msg, size);
87     temp = (prop->value.size() + 1) * sizeof (char);
88     memcpy(msg, &temp, sizeof (uint8_t));
89     memcpy(msg + sizeof (uint8_t), &(prop->value)[0], temp);
90     memcpy(msg + sizeof (uint8_t) + temp, &n, sizeof (char));
91      *
92
```

```
93      free(msg);
94       */
95      return _S_OK;
96  }
97
98  ConnectionResponse Connection::receive_SensA_info(
        Extension_Information* info) {
99      /*
100   SensA_ID sensA;
101   Property_T property_Type;
102   std::string value_Type;
103   std::string value;
104      */
105     uint8_t size = 1;
106
107     receive_data(&(info->version), sizeof (uint8_t));
108     //receive property_Type
109     int resp = receive_data(&size, sizeof (uint8_t));
110     char* data = (char*) malloc(size);
111     resp = receive_data(data, size);
112     info->type.assign(data);
113     //receive value_Type
114     resp = receive_data(&size, sizeof (uint8_t));
115     realloc(data, size);
116     resp = receive_data(data, size);
117     info->location.assign(data);
118
119     //receive value
120     resp = receive_data(&size, sizeof (uint8_t));
121     realloc(data, size);
122     resp = receive_data(data, size);
123     info->dataType.assign(data);
124
125     free(data);
126     return _S_OK;
127 }
128
129 ConnectionResponse Connection::send_SensA_info(
        Extension_Information* info) {
130
131     send_data(&(info->version), 1);
132
133     uint8_t temp;
```

```
134     temp = (info->type.size() + 1) * sizeof (char); //size of
            chararray with \0
135
136
137
138     /*
139      * Each value of a Property_Set-Object is writen into the
             bytearray.
140      * the 4 bytes leading each value hold the size_t of that
             value.
141      */
142     send_data(&temp, sizeof (uint8_t)); //write the size_t of
            property_type
143     send_data(&(info->type)[0], temp); //write property_Type
           in the next bytes
144
145
146     temp = (info->location.size() + 1) * sizeof (char); //
           size of chararray with \0
147     send_data(&temp, sizeof (uint8_t)); //write the size_t of
            property_type
148     send_data(&(info->location)[0], temp); //write
           property_Type in the next bytes
149
150
151     temp = (info->dataType.size() + 1) * sizeof (char); //
           size of chararray with \0
152     send_data(&temp, sizeof (uint8_t)); //write the size_t of
            property_type
153     send_data(&(info->dataType)[0], temp); //write
           property_Type in the next bytes
154
155     return _S_OK;
156 }
157
158 ConnectionResponse Connection::receive_Perm_list(
      Permission_List* perm) {
159     uint8_t k;
160     receive_data(&k, 1);
161     uint8_t size;
162
163     for (uint8_t i = 0; i < k; i++) {
164         PP_Tupel ppt;
```

```
165         uint8_t u;
166         receive_data(&size, 1);
167         char* d = (char*) malloc(size);
168         receive_data(d, size);
169         ppt.first.assign(d);
170         receive_data(&u, 1);
171         send_response(_S_OK);
172         switch (u) {
173             case 1:
174                 ppt.second = READ;
175                 break;
176             case 2:
177                 ppt.second = WRITE;
178                 break;
179             case 3:
180                 ppt.second = READ_WRITE;
181                 break;
182         }
183         perm->push_back(ppt);
184         free(d);
185     }
186 }
187
188 ConnectionResponse Connection::send_Perm_list(Permission_List
      * perm) {
189     uint8_t k = perm->size();
190     uint8_t temp;
191
192     send_data(&k, 1);
193     for (Permission_List::iterator it = perm->begin();
194             it != perm->end();
195             ++it) {
196         PP_Tupel* ppt = &(*it);
197         temp = (ppt->first.size() + 1) * sizeof (char); //
              size of chararray with \0
198         send_data(&temp, sizeof (uint8_t)); //write the
              size_t of property_type
199         send_data(&(ppt->first)[0], temp); //write
              property_Type in the next bytes
200         switch (ppt->second) {
201             case READ:
202                 k = 1;
203                 break;
```

```
204             case WRITE:
205                 k = 2;
206                 break;
207             case READ_WRITE:
208                 k = 3;
209                 break;
210         }
211         send_data(&k, 1);
212 
213         receive_response();
214     }
215 }
216 
217 ConnectionResponse Connection::receive_Property_list(
      Property_List* prop) {
218     dsa_size_t k;
219     uint8_t u;
220     receive_data(&k, sizeof (dsa_size_t));
221     dsa_data_size size;
222 
223     for (int i = 0; i < k; i++) {
224         Property ppt = *(new Property(false));
225         receive_data(&size, sizeof (dsa_data_size));
226         char* d = (char*) malloc(size);
227         receive_data(d, size);
228         ppt.property_Type.assign(d);
229         receive_data(&size, 1);
230         realloc(d, size);
231         receive_data(d, size);
232         ppt.value_Type.assign(d);
233 
234         receive_data(&size, 1);
235         realloc(d, size);
236         receive_data(d, size);
237         ppt.value.assign(d);
238         receive_data(&u, 1);
239         send_response(_S_OK);
240         switch (u) {
241             case 1:
242                 ppt.permission = READ;
243                 break;
244             case 2:
245                 ppt.permission = WRITE;
```

```
246                 break;
247             case 3:
248                 ppt.permission = READ_WRITE;
249                 break;
250         }
251         prop->push_back(ppt);
252         free(d);
253     }
254 }
255
256 ConnectionResponse Connection::send_Property_list(
    Property_List* prop) {
257     dsa_size_t k = prop->size();
258     send_data(&k, sizeof (dsa_size_t));
259
260     for (Property_List::iterator it = prop->begin(); it !=
        prop->end(); ++it) {
261         Property p = *it;
262         dsa_data_size s = (p.property_Type.size() + 1) *
            sizeof (char);
263         send_data(&s, sizeof (dsa_data_size));
264         send_data(&(p.property_Type)[0], s);
265         s = (p.value_Type.size() + 1) * sizeof (char);
266         send_data(&s, sizeof (dsa_data_size));
267         send_data(&(p.value_Type)[0], s);
268         s = (p.value.size() + 1) * sizeof (char);
269         send_data(&s, sizeof (dsa_data_size));
270         send_data(&(p.value)[0], s);
271         uint8_t perm = 0;
272         switch (p.permission) {
273             case READ:
274                 perm = 1;
275                 break;
276             case WRITE:
277                 perm = 2;
278                 break;
279             case READ_WRITE:
280                 perm = 3;
281                 break;
282         }
283         send_data(&perm, sizeof (uint8_t));
284         receive_response();
285     }
```

```
286 }
287
288 ConnectionResponse Connection::receive_response() {
289     ConnectionResponse resp;
290     receive_data(&resp, sizeof (uint32_t));
291     return resp;
292 }
293 void Connection::send_ready() {
294     send_response(_S_OK);
295 }
296
297 void Connection::send_response(ConnectionResponse resp) {
298     send_data(&resp, sizeof (uint32_t));
299 }
300
301 ConnectionResponse Connection::receive_Data_Object(
      DSA_Data_Object* ddo) {
302     uint8_t s;
303     int32_t t;
304 ddo->timeStamp = 0;
305     receive_data(&s, sizeof (dsa_data_size));
306     ddo->_size = s;
307     realloc(ddo->dataValue, s);
308     receive_data(ddo->dataValue, s);
309     receive_data(&(ddo->dataID), sizeof (Data_ID));
310     receive_data(&t, sizeof (int32_t));
311     ddo->timeStamp = t;
312     send_response(_S_OK);
313     return _S_OK;
314
315 }
316
317 ConnectionResponse Connection::send_Data_Object(
      DSA_Data_Object* ddo) {
318     send_data(&(ddo->_size), sizeof (dsa_data_size));
319     send_data(ddo->dataValue, ddo->_size);
320     send_data(&(ddo->dataID), sizeof (Data_ID));
321     send_data(&(ddo->timeStamp), sizeof (uint32_t));
322     return receive_response();
323 }
```

DSA_Framework/connectionhandler.h

```
1 /*
```

```
 * File:   connectionhandler.h
 * Author: krayn
 *
 * Created on September 30, 2013, 4:03 PM
 */

#ifndef CONNECTIONHANDLER_H
#define  CONNECTIONHANDLER_H

#include "connection_information.h"
#include "connection.h"
#include "connection_manager.h"
#include "dsa_types.h"
#include "dsa_management.h"
#include "dsa_data_object.h"
#include "remote_observer.h"

class Connectionhandler {
public:

    virtual ~Connectionhandler();
    void handleconnection(Connection* conn);
    Connection* create_connection(Connection_Information*
       coninf);
    static Connectionhandler* get_Instance();
    void add_Connection_Manager(Connection_Manager* man);
    FaultValue add_Connection_Information(
       Connection_Information* info);
    Connection_Manager* get_Connection_Manager(
       Connection_Information* info);
private:
    Connectionhandler();
    Connectionhandler(const Connectionhandler& orig);
    static void* notify_th(void* vptr);
    static Connectionhandler* instance;
    std::list<Connection_Manager*> connman;


    /**
     * @brief Function to make handleconnection more pretty.
     */
    void __get_data(Connection* conn);
    void __reg_obs(Connection* conn);
```

```
42      void __rem_obs(Connection* conn);
43      void __notify(Connection* conn);
44      void __set_prop(Connection* conn);
45      void __test_cont(Connection* conn);
46      void __test_sensa(Connection* conn);
47      void __sensa_info(Connection* conn);
48      void __sensa_id(Connection* conn);
49      void __get_perms(Connection* conn);
50      void __get_props(Connection* conn);
51      void __get_exps(Connection* conn);
52
53
54  };
55
56  #endif   /* CONNECTIONHANDLER_H */
```

DSA_Framework/connectionhandler.cpp

```
1  /*
2   * File:   connectionhandler.cpp
3   * Author: krayn
4   *
5   * Created on September 30, 2013, 4:03 PM
6   */
7
8  #include <stdlib.h>
9  #include <iostream>
10 #include "connectionhandler.h"
11 Connectionhandler* Connectionhandler::instance = 0;
12
13 Connectionhandler::Connectionhandler() {
14 }
15
16 Connectionhandler::Connectionhandler(const Connectionhandler&
       orig) {
17 }
18
19 Connectionhandler::~Connectionhandler() {
20     if (instance != 0) {
21         for (std::list<Connection_Manager*>::iterator it =
              instance->connman.begin(); it != instance->connman
              .end(); ++it) {
22             (*it)->~Connection_Manager();
23         }
```

```
24          instance = 0;
25      }
26  }
27
28  Connectionhandler* Connectionhandler::get_Instance() {
29      if (instance == 0)
30          instance = new Connectionhandler;
31      return instance;
32  }
33
34  void Connectionhandler::add_Connection_Manager(
        Connection_Manager* man) {
35      man->start_listening();
36      connman.push_back(man);
37  }
38
39  Connection_Manager* Connectionhandler::get_Connection_Manager
        (Connection_Information* info) {
40
41      Connection_Manager* man;
42      for (std::list<Connection_Manager*>::iterator it =
            connman.begin();
43               it != connman.end();
44               ++it) {
45          man = *it;
46          if (man->check_compatible_conn_info(info)) {
47              return man;
48          }
49      }
50  }
51
52  FaultValue Connectionhandler::add_Connection_Information(
        Connection_Information* info) {
53      return get_Connection_Manager(info)->
            add_Connection_Information(info);
54  }
55
56  Connection* Connectionhandler::create_connection(
        Connection_Information* conninf) {
57      Connection* conn;
58      Connection_Manager* man = get_Connection_Manager(conninf)
            ;
59      conn = man->connect(conninf);
```

```
60      return conn;
61  }
62
63  struct conddo {
64      SensA_ID sensaID;
65      DSA_Data_Object* ddo;
66  };
67
68  /**
69   * @brief Handles Connections. The Data send via the
        connection is translated into function calls and their
        results are then send back.
70   * @param conn The connection that is to be handled
71   */
72  void Connectionhandler::handleconnection(Connection* conn) {
73      std::cout << "s - handling" << std::endl;
74      {
75          //If the connection is already known, switch the
              connection_information with the previously created
76          //else add the connection-information to the known
              connections
77          Connection_Manager* man = Connectionhandler::
              get_Instance()->get_Connection_Manager(conn->info)
              ;
78          if (man->check_known_Connection(conn->info)) {
79              conn->info = man->get_Connection_Information(conn
                  ->info);
80          } else {
81              man->add_Connection_Information(conn->info);
82          }
83      }
84
85      conn->send_ready();
86      ConnectionResponse command = conn->receive_response();
87      std::cout << "s - " << command << std::endl;
88      switch (command) {
89          case _GET_DATA:
90              __get_data(conn);
91              break;
92          case _TEST_CONTAINER:
93              __test_cont(conn);
94              break;
95          case _TEST_SENSA:
```

```
96              __test_sensa(conn);
97              break;
98          case _REG_OBS:
99              __reg_obs(conn);
100             break;
101         case _RM_OBS:
102             __rem_obs(conn);
103             break;
104         case _NOTIFY:
105             __notify(conn);
106             break;
107         case _SET_PROP:
108             __set_prop(conn);
109             break;
110         case _GET_PROP:
111             __get_props(conn);
112             break;
113         case _SENSA_INFO:
114             __sensa_info(conn);
115             break;
116         case _GET_PERMS:
117             __get_perms(conn);
118             break;
119         case _GET_SENSA_IDS:
120             __sensa_id(conn);
121             break;
122         case _GET_EXP:
123             __get_exps(conn);
124             break;
125         case _E_CONNECTION:
126             break;
127     }
128
129 }
130
131 /*void* Connectionhandler::notify(void* vptr) {
132     conddo* ddo = (conddo*) vptr;
133     ddo->conn->send_response(_S_OK);
134     ddo->sensaID.first = ddo->conn->info->id;
135     ddo->sensaID.second = ddo->conn->receive_data();
136
137     int i;
138     i = ddo->conn->receive_data();
```

```
139     ddo->ddo->typ = OT_DATA;
140  *((int*) (ddo->ddo->dataValue)) = i;
141 }*/
142
143 void* Connectionhandler::notify_th(void* vptr) {
144     Connection* conn = (Connection*) vptr;
145
146     SensA_ID sensaID;
147     DSA_Data_Object* ddo;
148     ddo = new DSA_Data_Object(1);
149     ddo->typ = OT_DATA;
150         time(&(ddo->timeStamp));
151
152     sensaID.first = conn->info->id;
153     conn->receive_data(&(sensaID.second), sizeof (
            Cont_SensA_ID));
154
155     conn->receive_Data_Object(ddo);
156     if(ddo->timeStamp == NOT_CONT)
157         time(&(ddo->timeStamp));
158
159     ddo->specID = sensaID;
160     sensaID = sensaID;
161     Connectionhandler::get_Instance()->get_Connection_Manager
            (conn->info)->return_Connection(conn);
162
163
164     DSA_Management::get_instance()->get_SensA(sensaID)->
            notify((ddo));
165
166 }
167
168 void Connectionhandler::__get_data(Connection* conn) {
169     SensA_ID sensaID;
170     Cont_SensA_ID rids;
171     conn->receive_data(&rids, sizeof (Cont_SensA_ID));
172     conn->receive_data(&(sensaID.first), sizeof (Cont_ID));
            //conn->info->id;
173     conn->receive_data(&(sensaID.second), sizeof (
            Cont_SensA_ID));
174     Remote_Observer* obs2;
175     obs2 = get_Connection_Manager(conn->info)->
            get_remote_Observer(conn->info, rids);
```

```
176     DSA_DO_Container* con;
177     con = obs2->get_new_Data(sensaID);
178     dsa_size_t k = con->size();
179     conn->send_data(&k, sizeof (dsa_size_t));
180 
181     for (int i = 0; i < k; i++) {
182         conn->send_Data_Object(con->at(i));
183     }
184     get_Connection_Manager(conn->info)->return_Connection(
           conn);
185 }
186 
187 void Connectionhandler::__reg_obs(Connection* conn) {
188     SensA_ID sensaID;
189     std::cout << "reg_obs" << std::endl;
190     Cont_SensA_ID rid;
191     conn->receive_data(&rid, sizeof (Cont_SensA_ID));
192     conn->receive_data(&(sensaID.first), sizeof (Cont_ID));
           //conn->info->id;
193     conn->receive_data(&(sensaID.second), sizeof (
           Cont_SensA_ID));
194     Remote_Observer* obs;
195     obs = get_Connection_Manager(conn->info)->
           get_remote_Observer(conn->info, rid);
196     DSA_Extension* sens = DSA_Management::get_instance()->
           get_SensA(sensaID);
197     sens->register_observer(obs);
198     sens->connect(SensA_ID(get_Connection_Manager(conn->info)
           ->cont_id, obs->sensaID), sens->
           get_standard_Permissions());
199     conn->send_response(_S_OK);
200 }
201 
202 void Connectionhandler::__rem_obs(Connection* conn) {
203     std::cout << "rem_obs" << std::endl;
204 
205     SensA_ID sensaID;
206     Cont_SensA_ID rid;
207     conn->receive_data(&rid, sizeof (Cont_SensA_ID));
208     conn->receive_data(&(sensaID.first), sizeof (Cont_ID));
           //conn->info->id;
209     conn->receive_data(&(sensaID.second), sizeof (
           Cont_SensA_ID));
```

```
210     Remote_Observer* obs;
211     obs = get_Connection_Manager(conn->info)->
           get_remote_Observer(conn->info, rid);
212     DSA_Extension* sens = DSA_Management::get_instance()->
           get_SensA(sensaID);
213     sens->remove_observer(obs->get_observer_ID());
214     sens->disconnect(SensA_ID(get_Connection_Manager(conn->
           info)->cont_id, obs->sensaID));
215     conn->send_response(_S_OK);
216     get_Connection_Manager(conn->info)->return_Connection(
           conn);
217 }
218
219 void Connectionhandler::__notify(Connection* conn) {
220     pthread_t th;
221     pthread_create(&th, NULL, &notify_th, conn);
222
223
224
225 }
226
227 void Connectionhandler::__test_cont(Connection* conn) {
228     conn->send_response(_S_OK);
229     get_Connection_Manager(conn->info)->return_Connection(
           conn);
230 }
231
232 void Connectionhandler::__test_sensa(Connection* conn) {
233     SensA_ID sensaID;
234     std::cout << "test_sensa" << std::endl;
235     conn->receive_data(&(sensaID.first), sizeof (Cont_ID));
           //conn->info->id;
236     conn->receive_data(&(sensaID.second), sizeof (
           Cont_SensA_ID));
237     DSA_Extension* sens = DSA_Management::get_instance()->
           get_SensA(sensaID);
238     if (sens != NULL) {
239          conn->send_response(_S_OK);
240     } else {
241          conn->send_response(_E_UNKSENSA);
242     }
243     get_Connection_Manager(conn->info)->return_Connection(
           conn);
```

```
244 }
245
246 void Connectionhandler::__set_prop(Connection* conn) {
247     SensA_ID sensaID;
248     std::cout << "SET_PROP" << std::endl;
249     conn->receive_data(&(sensaID.first), sizeof (Cont_ID));
           //conn->info->id;
250     conn->receive_data(&(sensaID.second), sizeof (
           Cont_SensA_ID));
251     uint8_t id;
252     conn->receive_data(&id, 1);
253     Property_Set_Object* pso;
254     pso = new Property_Set_Object(sensaID, "", "", "");
255
256     Property* ps;
257     ps = DSA_Management::get_instance()->get_SensA(sensaID)->
           configuration->__DSA_get_Property(id);
258     pso->property_Type = ps->property_Type;
259     pso->value_Type = ps->value_Type;
260
261
262
263     conn->receive_Property_Set_Object(pso);
264
265     Remote_Observer* obs1;
266     obs1 = get_Connection_Manager(conn->info)->
           get_remote_Observer(conn->info, sensaID.second); //
           hand over the pso
267     std::cout << pso->value << std::endl;
268     conn->send_response(_S_OK);
269     get_Connection_Manager(conn->info)->return_Connection(
           conn);
270 }
271
272 void Connectionhandler::__sensa_info(Connection* conn) {
273     SensA_ID sensaID;
274     conn->receive_data(&(sensaID.first), sizeof (Cont_ID));
           //conn->info->id;
275     conn->receive_data(&(sensaID.second), sizeof (
           Cont_SensA_ID));
276
277     DSA_Extension* sens = DSA_Management::get_instance()->
           get_SensA(sensaID);
```

```
278     Extension_Information* inf;
279     inf = sens->get_SensA_information();
280     conn->send_SensA_info(inf);
281     get_Connection_Manager(conn->info)->return_Connection(
            conn);
282 }
283
284 void Connectionhandler::__sensa_id(Connection* conn) {
285     SensA_ID_List ids;
286     ids = DSA_Management::get_instance()->get_SensA_ID_list()
            ;
287     dsa_size_t s;
288     s = ids.size();
289     conn->send_data(&s, sizeof (dsa_size_t));
290     for (SensA_ID_List::iterator it = ids.begin(); it != ids.
            end(); ++it) {
291         SensA_ID sid;
292         sid = *it;
293         conn->send_data(&(sid.first), sizeof (Cont_ID));
294         conn->send_data(&(sid.second), sizeof (Cont_SensA_ID)
                );
295         conn->receive_response();
296     }
297     get_Connection_Manager(conn->info)->return_Connection(
            conn);
298 }
299
300 void Connectionhandler::__get_perms(Connection* conn) {
301     SensA_ID sensaID;
302     conn->receive_data(&(sensaID.first), sizeof (Cont_ID));
            //conn->info->id;
303     conn->receive_data(&(sensaID.second), sizeof (
            Cont_SensA_ID));
304     DSA_Extension* sens = DSA_Management::get_instance()->
            get_SensA(sensaID);
305     Permission_List* l;
306     l = sens->get_standard_Permissions();
307     conn->send_Perm_list(l);
308     get_Connection_Manager(conn->info)->return_Connection(
            conn);
309 }
310
311 void Connectionhandler::__get_props(Connection* conn) {
```

```
312     SensA_ID sensaID;
313     conn->receive_data(&(sensaID.first), sizeof (Cont_ID));
           //conn->info->id;
314     conn->receive_data(&(sensaID.second), sizeof (
           Cont_SensA_ID));
315     DSA_Extension* sens = DSA_Management::get_instance()->
           get_SensA(sensaID);
316     Property_List properties = sens->configuration->
           properties;
317     conn->send_Property_list(&properties);
318     get_Connection_Manager(conn->info)->return_Connection(
           conn);
319 }
320
321 void Connectionhandler::__get_exps(Connection* conn) {
322     SensA_ID sensaID;
323     conn->receive_data(&(sensaID.first), sizeof (Cont_ID));
           //conn->info->id;
324     conn->receive_data(&(sensaID.second), sizeof (
           Cont_SensA_ID));
325
326     DSA_Extension* sens = DSA_Management::get_instance()->
           get_SensA(sensaID);
327     DSA_Data_Object* exp;
328     exp = sens->get_Expenses();
329     conn->send_Data_Object(exp);
330     get_Connection_Manager(conn->info)->return_Connection(
           conn);
331
332 }
```

DSA_Framework/connection_information.h

```
1 /*
2  * File:   connection_information.h
3  * Author: krayn
4  *
5  * Created on September 30, 2013, 5:54 PM
6  */
7
8 #ifndef CONNECTION_INFORMATION_H
9 #define  CONNECTION_INFORMATION_H
10
11 #include "dsa_types.h"
```

```
12 #include "connection_manager.h"
13
14 class Connection_Information {
15 public:
16     Connection_Information();
17     Connection_Information(const Connection_Information& orig
       );
18     virtual ~Connection_Information();
19
20     /**
21      * @brief This function is used to determine if a
          Connection_Information is compatible to a derived
          Connection_Information.
22      * @return
23      * true, if this instance of Connection_Information is
          compatible to the derived Connection_Information
24      *
25      * false, if it is not compatible.
26      */
27     template <typename T> bool check_compatible() {
28         T* a = dynamic_cast<T*> (this);
29         if (a != 0) {
30             return true;
31         } else
32             return false;
33     };
34     virtual bool operator ==(const Connection_Information& b)
           const = 0;
35     void set_ID(int id);
36     int id;
37 private:
38 };
39
40 #endif   /* CONNECTION_INFORMATION_H */
```

DSA_Framework/connection_information.cpp

```
1 /*
2  * File:   connection_information.cpp
3  * Author: krayn
4  *
5  * Created on September 30, 2013, 5:54 PM
6  */
7
```

```
8  #include "connection_information.h"
9
10 Connection_Information::Connection_Information() {
11 }
12
13
14
15 Connection_Information::Connection_Information(const
       Connection_Information& orig) {
16 }
17
18
19 Connection_Information::~Connection_Information() {
20 }
21
22
23
24 void Connection_Information::set_ID(int id) {
25     this->id = id;
26 }
```

DSA_Framework/connection_manager.h

```
1  /*
2   * File:   Connection_Manager.h
3   * Author: krayn
4   *
5   * Created on October 2, 2013, 1:41 PM
6   */
7
8  #ifndef CONNECTION_MANAGER_H
9  #define  CONNECTION_MANAGER_H
10
11 #include "connection_information.h"
12 #include "dsa_types.h"
13 #include "connection.h"
14 #include "remote_observer.h"
15
16 class Connection_Manager {
17 public:
18     Connection_Manager();
19     Connection_Manager(const Connection_Manager& orig);
20     virtual ~Connection_Manager();
```

```
21      virtual Connection* connect(Connection_Information*
           conninf) = 0;
22      /**
23       * @brief start listening for Connections. Created
            connections are then send to the connectionhandler by
             calling
24       * Connectionhandler::get_Instance()->handleconnection();
25       */
26      virtual void start_listening() = 0;
27      virtual void stop_listening() = 0;
28      FaultValue add_Connection_Information(
           Connection_Information* conn);
29      bool check_known_Connection(Connection_Information* conn)
           ;
30      virtual bool check_compatible_conn_info(
           Connection_Information* info)=0;
31      Connection_Information* get_Connection_Information(
           Connection_Information* conn);
32      Remote_Observer* get_remote_Observer(
           Connection_Information* conn, Cont_SensA_ID sensID);
33      virtual void return_Connection(Connection* conn)=0;
34      Cont_ID cont_id;
35  private:
36      int type;
37      std::list<Connection_Information*> conns;
38      std::list<Remote_Observer*> observers;
39
40  };
41
42  #endif   /* CONNECTION_MANAGER_H */
```

DSA_Framework/connection_manager.cpp

```
1  /*
2   * File:   Connection_Manager.cpp
3   * Author: krayn
4   *
5   * Created on October 2, 2013, 1:41 PM
6   */
7
8  #include <ios>
9
10 #include "connection_manager.h"
11 #include "remote_observer.h"
```

```
12
13 Connection_Manager::Connection_Manager() {
14 }
15
16 Connection_Manager::Connection_Manager(const
       Connection_Manager& orig) {
17 }
18
19 Connection_Manager::~Connection_Manager() {
20 }
21
22 Connection_Information* Connection_Manager::
       get_Connection_Information(Connection_Information* conn) {
23     Connection_Information* coninf;
24     for (std::list<Connection_Information*>::iterator it =
          conns.begin();
25              it != conns.end() && !conns.empty();
26              ++it) {
27         coninf = *it;
28         if (*coninf == *conn) {
29             return coninf;
30         }
31     }
32 }
33
34 FaultValue Connection_Manager::add_Connection_Information(
       Connection_Information* conn) {
35     if (check_compatible_conn_info(conn)) {
36         conns.push_back(conn);
37         return S_OK;
38     }
39     return E_UNKCONN;
40 }
41
42 bool Connection_Manager::check_known_Connection(
       Connection_Information* conn) {
43     Connection_Information* coninf;
44     for (std::list<Connection_Information*>::iterator it =
          conns.begin();
45              it != conns.end() && !conns.empty();
46              ++it) {
47         coninf = *it;
48         if (*coninf == *conn) {
```

```
49              return true;
50          }
51      }
52      return false;
53  }
54
55  Remote_Observer* Connection_Manager::get_remote_Observer(
        Connection_Information* conn, Cont_SensA_ID sensID) {
56      Remote_Observer* obs;
57      for (std::list<Remote_Observer*>::iterator it = observers
            .begin();
58              it != observers.end() && !observers.empty();
59              ++it) {
60          obs = *it;
61          if (*(obs->conn_info) == *conn && obs->sensaID ==
                sensID) {
62              return obs;
63          }
64      }
65      obs = new Remote_Observer(sensID, conn);
66      observers.push_back(obs);
67      return obs;
68  }
```

DSA_Framework/dsa_data_buffer.h

```
1  #ifndef DSA_DATA_BUFFER_H
2  #define DSA_DATA_BUFFER_H
3
4  /**
5   * @file dsa_data_buffer.h
6   * This class realizes a databuffer to store pointers to
        DSA_Data_Object objects.
7   * Any methods of this class should never be user outside of
        the DSA-Framework.
8   * This class provides methods to set the size of the
        databuffer and to get the size.
9   * Also this class provides a function to get a reference to
        the nested DSA_DO_List data_buffer
10  * and one function to add a DSA_Data_Object reference into
        the buffer.
11  *
12  * @author Jens Krayenborg
13  * @date 13.01.2013
```

```
 *
 * @version 1.0.0
 */

#include "dsa_types.h"
#include "dsa_data_object.h"

class DSA_Data_Buffer
{
public:
    /* Constructor */
    DSA_Data_Buffer(size_t size = STDBUFFERSIZE);
    ~DSA_Data_Buffer();

    /* Oberations */
    void set_Size(size_t size);
    void add_to_buffer(DSA_Data_Object* ddo);
    DSA_DO_List* get_buffer();
    size_t get_size();

private:
    Data_ID nextID;
    unsigned int bufferSize;
    DSA_DO_List data_buffer;
};

#endif // DSA_DATA_BUFFER_H
```

DSA_Framework/dsa_data_buffer.cpp

```
/**
 * @file dsa_data_buffer.cpp
 * This file implements the constructor and methods
 * for the class discribed in dsa_data_buffer.h file.
 *
 * @author Jens Krayenborg
 * @date 13.01.2013
 *
 * @version 1.0.0
 */

#include "dsa_data_buffer.h"

/**
```

```
15  * @brief DSA_Data_Buffer::DSA_Data_Buffer
16  * The constructor for this class.
17  *
18  * @param size The sought size of this buffer.
19  */
20 DSA_Data_Buffer::DSA_Data_Buffer(size_t size)
21 {
22     bufferSize = size;
23     nextID = 1;
24 }
25
26 /**
27  * @brief DSA_Data_Buffer::~DSA_Data_Buffer
28  * A simple destructor.
29  */
30 DSA_Data_Buffer::~DSA_Data_Buffer()
31 {
32 }
33
34 /**
35  * @brief DSA_Data_Buffer::set_Size
36  * This procedure sets the sought size of the buffer. If the
       buffer
37  * is smaller than the sought size, simply the bufferSize
       value
38  * will be setted. Otherwise the supernumerary
       DSA_Data_Object
39  * instances will be deleted completely from the memory and
       their
40  * pointers will be erased from the holding vector.
41  *
42  * @param size sought size of this buffer.
43  */
44 void DSA_Data_Buffer::set_Size(size_t size)
45 {
46     if (data_buffer.size() > size)
47     {
48         int diff = data_buffer.size() - size;
49         int j = diff;
50         while (j > 0)
51         {
52             DSA_Data_Object* ddo = data_buffer.front();
53             delete(ddo);
```

```
54              data_buffer.pop_front();
55              j--;
56          }
57      }
58      bufferSize = size;
59  }
60
61  /**
62   * @brief DSA_Data_Buffer::add_to_buffer
63   * Adding a pointer to a DSA_Data_Object instance to this
       buffer.
64   * If, while performing this procedure, the size of the
       pointer holding
65   * vector becomes greater than the bufferSize value of this
       buffer,
66   * the oldest DSA_Data_Object instance will be deleted
       completely from
67   * the memory and his pointer will be erased from this buffer
       .
68   *
69   * @param ddo The pointer to be added.
70   */
71  void DSA_Data_Buffer::add_to_buffer(DSA_Data_Object* ddo)
72  {
73      ddo->dataID = nextID;
74      nextID++;
75
76      data_buffer.push_back(ddo);
77      if (data_buffer.size() > bufferSize){
78          DSA_Data_Object* ddo = *(data_buffer.begin());
79          data_buffer.pop_front();
80          delete(ddo);
81      }
82  }
83
84  /**
85   * @brief DSA_Data_Buffer::get_buffer
86   * The function to get the reference to the pointer holding
87   * vector.
88   *
89   * @return The reference to the pointer holding vector.
90   */
91  DSA_DO_List* DSA_Data_Buffer::get_buffer()
```

```
92 {
93     return &data_buffer;
94 }
95
96 /**
97  * @brief DSA_Data_Buffer::get_size
98  * Delivering the sought size of this buffer. It is not the
99  * actual size, this could be smaller than the return value
100 * of this function.
101 *
102 * @return The sought size of this buffer.
103 */
104 size_t DSA_Data_Buffer::get_size()
105 {
106     return bufferSize;
107 }
```

DSA_Framework/dsa_history_buffer.h

```
1  #ifndef DSA_HISTORY_BUFFER_H
2  #define DSA_HISTORY_BUFFER_H
3
4  /**
5   * @file dsa_history_buffer.h
6   * This class realizes a buffer to store Data_ID's.
7   * Any methods of this class should never be user outside of
       the DSA-Framework.
8   * This class provides methods to set the size of the buffer
       and to get the size.
9   * Also this class provides a function to get a reference to
       the nested content
10  * and one function to add a Data_ID into the buffer.
11  * A History_Buffer is user by the DSA_Extension to manage
       which DSA_Data_Objects
12  * were still be delivered to a DSA_SensA while performing
       his method get_new_data().
13  *
14  * @author Jens Krayenborg
15  * @date 13.01.2013
16  *
17  * @version 1.0.0
18  */
19
20 #include "dsa_types.h"
```

```
21
22 class DSA_History_Buffer
23 {
24 public:
25     DSA_History_Buffer(size_t size = STDBUFFERSIZE);
26     ~DSA_History_Buffer();
27
28     /* Oberations */
29     void set_Size(size_t size);
30     void add_to_buffer(Data_ID did);
31     Data_History* get_buffer();
32     bool contains_DataID(Data_ID did);
33
34 private:
35     size_t bufferSize;
36     Data_History content;
37 };
38
39 #endif // DSA_HISTORY_BUFFER_H
```

DSA_Framework/dsa_history_buffer.cpp

```
1  /**
2   * @file dsa_history_buffer.cpp
3   * This file implements the constructor and methods
4   * for the class discribed in dsa_history_buffer.h file.
5   *
6   * @author Jens Krayenborg
7   * @date 13.01.2013
8   *
9   * @version 1.0.0
10  */
11
12 #include "dsa_history_buffer.h"
13
14 /**
15  * @brief DSA_History_Buffer::DSA_History_Buffer
16  * The constructor.
17  *
18  * @param size  The initial size of the buffer.
19  */
20 DSA_History_Buffer::DSA_History_Buffer(size_t size)
21 {
22     bufferSize = size+HISTDATDIFF;
```

```
23 }
24
25 /**
26  * @brief DSA_History_Buffer::~DSA_History_Buffer
27  * A simple destructor.
28  */
29 DSA_History_Buffer::~DSA_History_Buffer()
30 {
31 }
32
33 /**
34  * @brief DSA_History_Buffer::set_Size
35  * This procedure sets the sought size of the buffer. If the
       buffer
36  * is smaller than the sought size, simply the bufferSize
       value
37  * will be setted. Otherwise the supernumerary
       DSA_Data_Object
38  * instances will be deleted.
39  */
40 void DSA_History_Buffer::set_Size(size_t size)
41 {
42     if (content.size() > size+HISTDATDIFF)
43     {
44         int diff = content.size() - size+HISTDATDIFF;
45         for (int i = 0; i < diff; i++){
46             content.pop_front();
47         }
48     }
49     bufferSize = size+HISTDATDIFF;
50 }
51
52 /**
53  * @brief DSA_History_Buffer::add_to_buffer
54  * Adding a data id to this buffer.
55  * If, while performing this procedure, the size of the data
       id holding
56  * vector becomes greater than the bufferSize value of this
       buffer,
57  * the oldest data id will be deleted from this buffer.
58  *
59  * @param ddo The pointer to be added.
60  */
```

```
61 void DSA_History_Buffer::add_to_buffer(Data_ID did)
62 {
63     content.push_back(did);
64     if (content.size() > bufferSize){
65         content.pop_front();
66     }
67 }
68
69 /**
70  * @brief DSA_History_Buffer::get_buffer
71  * Getting the inlaying data id holding vector.
72  *
73  * @return  The reference of the vector.
74  */
75 Data_History* DSA_History_Buffer::get_buffer()
76 {
77     return &content;
78 }
79
80 /**
81  * @brief DSA_History_Buffer::contains_DataID
82  * Performs if the given data id is somewhere in the buffer.
83  *
84  * @param did   The data id from which the caller will
85  *              know if it is in the buffer.
86  * @return      True if the given data id is in the buffer,
87  *              fasle otherwise.
88  */
89 bool DSA_History_Buffer::contains_DataID(Data_ID did)
90 {
91     for(Data_History::reverse_iterator i = content.rbegin();
92         i != content.rend();
93         ++i)
94     {
95         Data_ID d = *i;
96         if (d == did){
97             return true;
98         }
99     }
100    return false;
101 }
```

DSA_Framework/dsa_data_object.h

```
1  #ifndef DSA_DATA_OBJECT_H
2  #define DSA_DATA_OBJECT_H
3
4  /**
5   * @file dsa_data_object.h
6   * This class realizes the DSA_Data_Object's.
7   * This is a very simple class to construct objects to hold
       all informations
8   * which should be delivered by the DSA-Framework to a using
       informatic system.
9   * For memory save copying of content from one
       DSA_Data_Object to another
10  * the function assign, or the special designed copy
       constructor should
11  * be used. This is very importent because otherwise it could
        be that
12  * the management information, which are used by e.q. malloc
       () will become
13  * corrupted.
14  *
15  * @author Jens Krayenborg
16  * @date 13.01.2013
17  *
18  * @version 1.0.0
19  */
20
21 #include <stdlib.h>
22 #include "dsa_types.h"
23 #include <cstring>
24
25 class DSA_Data_Object {
26 public:
27     /* Constructor */
28     DSA_Data_Object(size_t size);
29     DSA_Data_Object(const DSA_Data_Object& o);
30     ~DSA_Data_Object();
31     void assign(DSA_Data_Object* o);
32
33     /* Attributes */
34     Data_ID dataID;
35     size_t _size;
36     DSA_Object_Type typ;
```

```
37      SensA_ID specID;
38      DSA_Time timeStamp;
39      DSA_Data_Type dataType;
40      DSA_Data dataValue;
41  };
42
43  #endif // DSA_DATA_OBJECT_H
```

DSA_Framework/dsa_data_object.cpp

```
1   /**
2    * @file dsa_data_object.cpp
3    * This file implements the constructor and methods
4    * for the class discribed in dsa_data_object.h file.
5    *
6    * @author Jens Krayenborg
7    * @date 13.01.2013
8    *
9    * @version 1.0.0
10   */
11
12  #include "dsa_data_object.h"
13
14  /**
15   * @brief DSA_Data_Object::DSA_Data_Object
16   * The constructor.
17   *
18   * @param size  The size of the memoryspace to allocate.
19   *              It is importent that this size matches
20   *              the datasize which will be stored in
21   *              this data object.
22   */
23  DSA_Data_Object::DSA_Data_Object(size_t size) {
24      dataType = "UNKNOWN";
25      dataValue = malloc(size);
26      _size = size;
27  }
28
29  /**
30   * @brief DSA_Data_Object::DSA_Data_Object
31   * This is a copy constructor for memory save duplicating
32   * a dsa_data_object.
33   *
34   * @param o     The data object to be copied.
```

```
35  */
36 DSA_Data_Object::DSA_Data_Object(const DSA_Data_Object &o) :
37 dataID(o.dataID), _size(o._size), typ(o.typ),
38 specID(o.specID), timeStamp(o.timeStamp) {
39     dataType.assign(o.dataType);
40     dataValue = malloc(o._size);
41     // Using memcpy to perform a memory save copy.
42     memcpy(dataValue, o.dataValue, o._size);
43 }
44
45 /**
46  * @brief DSA_Data_Object::~DSA_Data_Object
47  * A destructor.
48  */
49 DSA_Data_Object::~DSA_Data_Object() {
50     if (dataValue != NULL) free(dataValue);
51 }
52
53 /**
54  * @brief DSA_Data_Object::assign
55  * This function copies the values of a given
56  * data object in a memory save way.
57  *
58  * @param o     The data object from which the data
59  *              should be copied.
60  */
61 void DSA_Data_Object::assign(DSA_Data_Object *o) {
62     dataID = o->dataID;
63     _size = o->_size;
64     typ = o->typ;
65     specID = o->specID;
66     timeStamp = o->timeStamp;
67     dataType.assign(o->dataType);
68     // Using memcpy to perform a memory save copy.
69     memcpy(dataValue, o->dataValue, o->_size);
70 }
```

DSA_Framework/dsa_do_container.h

```
1 #ifndef DSA_DO_CONTAINER_H
2 #define DSA_DO_CONTAINER_H
3
4 /**
5  * @file dsa_do_container.h
```

```
6  * This class realizes a contaibner to deliver
       DSA_Data_Object's to
7  * the informatic system which uses the DSA-Framework. It is
       designed
8  * to manage memory save adding and removing such objects.
       Importent
9  * to know is, that the objects which an object of this class
        holds
10 * should never be deletet maualy by the using informatic
       system.
11 * This will lead into a memory protection fault. If the
       objects holded
12 * by an instance of this class no more are needed, simply
       delete the
13 * instance of this class. Its destructor will perform all
       needed steps
14 * to prevent memory faults like leaks and so on.
15 *
16 * @author Jens Krayenborg
17 * @date 13.01.2013
18 *
19 * @version 1.0.0
20 */
21
22 #include "dsa_types.h"
23 #include "dsa_data_object.h"
24
25 class DSA_DO_Container
26 {
27 public:
28     DSA_DO_Container();
29     ~DSA_DO_Container();
30
31     DSA_DO_List* content();
32     DSA_Data_Object* last();
33     DSA_Data_Object* first();
34     DSA_Data_Object* at(unsigned int index);
35
36     void add(DSA_Data_Object* ddo);
37     void add(DSA_DO_List* dol);
38     void erase(unsigned int index);
39     void clear();
40
```

```
41      size_t size();
42      bool is_empty();
43
44      DSA_DO_List _content;
45  };
46
47  #endif // DSA_DO_CONTAINER_H
```

DSA_Framework/dsa_do_container.cpp

```
1   /**
2    * @file dsa_do_container.cpp
3    * This file implements the constructor and methods
4    * for the class discribed in dsa_do_container.h file.
5    *
6    * @author Jens Krayenborg
7    * @date 13.01.2013
8    *
9    * @version 1.0.0
10   */
11
12  #include "dsa_do_container.h"
13
14  /**
15   * @brief DSA_DO_Container::DSA_DO_Container
16   * A standard constructor.
17   */
18  DSA_DO_Container::DSA_DO_Container()
19  {
20  }
21
22  /**
23   * @brief DSA_DO_Container::~DSA_DO_Container
24   * A standard destructor.
25   */
26  DSA_DO_Container::~DSA_DO_Container()
27  {
28      clear();
29  }
30
31  /**
32   * @brief DSA_DO_Container::content
33   * Get the content of this container.
34   *
```

```
35  * @return The content.
36  */
37 DSA_DO_List* DSA_DO_Container::content()
38 {
39     return &_content;
40 }
41
42 /**
43  * @brief DSA_DO_Container::last
44  * Getting the last element in this container.
45  *
46  * @return The last data object in this container.
47  */
48 DSA_Data_Object* DSA_DO_Container::last()
49 {
50     return _content.back();
51 }
52
53 /**
54  * @brief DSA_DO_Container::first
55  * Getting the first element in this container.
56  *
57  * @return The first data object in this container.
58  */
59 DSA_Data_Object* DSA_DO_Container::first()
60 {
61     return _content.front();
62 }
63
64 /**
65  * @brief DSA_DO_Container::at
66  * Getting a data object at a special position.
67  *
68  * @param index The position of the data object.
69  * @return      A pointer to the data object.
70  *              NULL if index is greater than this
71  *              contents size-1.
72  */
73 DSA_Data_Object* DSA_DO_Container::at(unsigned int index)
74 {
75     DSA_Data_Object* ddo = NULL;
76     int i = 0;
77     for(DSA_DO_List::iterator it = _content.begin();
```

```
78          it != _content.end();
79          ++it, i++){
80          if (i == index){
81              ddo = *it;
82              break;
83          }
84      }
85      return ddo;
86  }
87
88  /**
89   * @brief DSA_DO_Container::add
90   * Adding a copy of a data object to the container.
91   *
92   * @param ddo   The data object to be added.
93   */
94  void DSA_DO_Container::add(DSA_Data_Object* ddo)
95  {
96      /* Add a copy of the given data object.
97       */
98      DSA_Data_Object* n = new DSA_Data_Object(*ddo);
99      _content.push_back(n);
100 }
101
102 /**
103  * @brief DSA_DO_Container::add
104  * Adding copies of all data objects from a list.
105  *
106  * @param dol   The list containing the objects.
107  */
108 void DSA_DO_Container::add(DSA_DO_List* dol)
109 {
110     /* Going through the list and add a copy of each
111      * object to this container.
112      */
113     for (DSA_DO_List::iterator i = dol->begin();
114          i != dol->end();
115          ++i)
116     {
117         DSA_Data_Object* s = *i;
118         DSA_Data_Object* n = new DSA_Data_Object(*s);
119         _content.push_back(n);
120     }
```

```
121 }
122
123 /**
124  * @brief DSA_DO_Container::erase
125  * Deleting one data object from the container.
126  *
127  * @param index     The position where the data object
128  *                  resides.
129  */
130 void DSA_DO_Container::erase(unsigned int index)
131 {
132     DSA_Data_Object* ddo = NULL;
133     int i = 0;
134     for(DSA_DO_List::iterator it = _content.begin();
135         it != _content.end();
136         ++it, i++){
137         if (i == index){
138             ddo = *it;
139             delete ddo;
140             _content.erase(it);
141             break;
142         }
143     }
144 }
145
146 /**
147  * @brief DSA_DO_Container::clear
148  * Clearing the whole content of this container.
149  */
150 void DSA_DO_Container::clear()
151 {
152     /* Going through the list an deleting each object
153      * in this container. Also delete it comletely from
154      * the memory.
155      */
156     for (DSA_DO_List::iterator i = _content.begin();
157          i != _content.end();
158          ++i)
159     {
160         DSA_Data_Object* d = *i;
161         delete(d);
162     }
163     _content.clear();
```

```
164 }
165
166 /**
167  * @brief DSA_DO_Container::size
168  * Delivers the number of containing data objects.
169  *
170  * @return  The number of objects in this container.
171  */
172 size_t DSA_DO_Container::size()
173 {
174     return _content.size();
175 }
176
177 /**
178  * @brief DSA_DO_Container::is_empty
179  * Saying if this container is empty.
180  *
181  * @return  A boolean value.
182  */
183 bool DSA_DO_Container::is_empty()
184 {
185     return _content.empty();
186 }
```

DSA_Framework/property.h

```
1  #ifndef PROPERTY_H
2  #define PROPERTY_H
3
4  /**
5   * @file property.h
6   * This file describes the classes for property related
       purposes.
7   * It contains the definitions of three classes.
8   *
9   * The first is the Property_Set_Object. This is a pure
10  * data holding class. It is needed to set a property.
11  *
12  * The second is the class needed by the claiming system
13  * in the DSA-Framework and kold the applications for
14  * setting one property.
15  *
16  * The third is the Porperty class. It is only a data holding
17  * class, which describes the values for a property.
```

```
18  *
19  * @author Jens Krayenborg
20  * @date 13.01.2013
21  *
22  * @version 1.0.0
23  */
24
25 #include "dsa_types.h"
26 #include <vector>
27
28 //class DSA_SensA;
29
30 class Property_Set_Object
31 {
32 public:
33     /* Constructor */
34     Property_Set_Object(SensA_ID sID, Property_T propType,
        std::string valType, std::string val);
35
36     /* Attributes */
37     SensA_ID sensA;
38     Property_T property_Type;
39     std::string value_Type;
40     std::string value;
41 };
42
43 class Claims
44 {
45 public:
46     /* Constructor */
47     Claims();
48
49     /* Operations */
50     FaultValue removeClaim(SensA_ID sensAc);
51     void setClaim(Property_Set_Object* pso);
52     Property_Set_Object* claim_at(Claim_Position position);
53     size_t number_of_claims();
54
55 private:
56     /* Operations */
57     Claim_Position get_Claim_from(SensA_ID sensAc);
58
59     /* Attributes */
```

```
60      std::vector<Property_Set_Object> applications;
61  };
62
63  class Property
64  {
65  public:
66      /* Constructors */
67      Property(bool claim = false);
68      Property(std::string propType, std::string valType, std::
            string val, Permission perm = READ, bool claim = false
            );
69
70      /* Attributes */
71      Property_T property_Type;
72      std::string value_Type;
73      std::string value;
74      Permission permission;
75      bool claiming;
76      Claims claims;
77  };
78
79  #endif // PROPERTY_H
```

DSA_Framework/property.cpp

```
1   /**
2    * @file property.cpp
3    * This file implements the constructor and methods
4    * for the class discribed in property.h file.
5    *
6    * @author Jens Krayenborg
7    * @date 13.01.2013
8    *
9    * @version 1.0.0
10   */
11
12  #include "property.h"
13
14  /**
15   * @brief Property_Set_Object::Property_Set_Object
16   * The constructor for the data holding object.
17   *
18   * @param sensAc    The sensa which will set the property.
19   * @param propType  The type of the property to set.
```

```
20  * @param valType   The type of the propertyvalue.
21  * @param val       The new value for the propertyvalue.
22  */
23 Property_Set_Object::Property_Set_Object(SensA_ID sensAc,
24         Property_T propType,
25         std::string valType,
26         std::string val)
27 {
28     sensA = sensAc;
29     property_Type = propType;
30     value_Type = valType;
31     value = val;
32 }
33
34 /**
35  * @brief Claims::Claims
36  * Simple constructor.
37  */
38 Claims::Claims(){}
39
40 /**
41  * @brief Claims::get_Claim_from
42  * Getting the position of the application which
43  * was received from the DSA_SensA instance to
44  * which the given reference points to.
45  *
46  * @param sensAc    The sensa from whom the application to
       get.
47  * @return          The position of the application from the
48  *                  given sensa.
49  *                  FAIL if there is no application from the
50  *                  given sensa.
51  */
52 Claim_Position Claims::get_Claim_from(SensA_ID sensAc)
53 {
54     for (Claim_Position i = 0;
55          i < applications.size();
56          i++)
57     {
58         Property_Set_Object pso = applications.at(i);
59         if (pso.sensA == sensAc)
60             return i;
61     }
```

```
62      return FAIL;
63  }
64
65  /**
66   * @brief Claims::removeClaim
67   * Removing the application from the given sensa.
68   *
69   * @param sensAc     The sensa whose application
70   *                   should be removed.
71   * @return           E_NOCLAIM if there is no application
72   *                   from the given sensa.
73   *                   S_OK if everything is fine.
74   */
75  FaultValue Claims::removeClaim(SensA_ID sensAc)
76  {
77      Claim_Position cp;
78      if ((cp = get_Claim_from(sensAc)) == FAIL)
79          return E_NOCLAIM;
80
81      applications.erase(applications.begin()+cp);
82      return S_OK;
83  }
84
85  /**
86   * @brief Claims::setClaim
87   * Setting the application to the sensa identified
88   * in the pso. If there is just one application from this
89   * sensa, then remove this before setting the new one.
90   *
91   * @param pso   The application.
92   */
93  void Claims::setClaim(Property_Set_Object* pso)
94  {
95      Claim_Position cp;
96      if ((cp = get_Claim_from(pso->sensA)) != FAIL)
97          applications.erase(applications.begin()+cp);
98
99      applications.push_back(*pso);
100 }
101
102 /**
103  * @brief Claims::claim_at
104  * Getting the application from the given position.
```

```
105  *
106  * @param position  The position from the application to get.
107  * @return          The application to get.
108  *                  NULL if the position is out of range.
109  */
110 Property_Set_Object* Claims::claim_at(Claim_Position position
     )
111 {
112     return &(applications.at(position));
113 }
114
115 /**
116  * @brief Claims::number_of_claims
117  * Getting the number of stored applications.
118  *
119  * @return  The number of stored applications.
120  */
121 size_t Claims::number_of_claims()
122 {
123     return applications.size();
124 }
125
126 /**
127  * @brief Property::Property
128  * A default property describing object.
129  *
130  * @param claim     Defining if this is a property for
131  *                  which the value setting have to be
132  *                  claimed or not.
133  */
134 Property::Property(bool claim)
135 {
136     property_Type = "UNKNOWN";
137     value_Type = "UNKNOWN";
138     value = "UNKNOWN";
139     permission = READ;
140     claiming = claim;
141 }
142
143 /**
144  * @brief Property::Property
145  * A property describing object.
146  *
```

```
147  * @param propType  The type of the property.
148  * @param valType   The type of the value of the property.
149  * @param val       The value of the property.
150  * @param perm      The global permissions of the property.
151  * @param claim     Defining if this is a property for
152  *                  which the value setting have to be
153  *                  claimed or not.
154  */
155 Property::Property(Property_T propType,
156          std::string valType,
157          std::string val,
158          Permission perm,
159          bool claim)
160 {
161     property_Type = propType;
162     value_Type = valType;
163     value = val;
164     permission = perm;
165     claiming = claim;
166 }
```

DSA_Framework/notify_rule.h

```
1  /**
2   * @file   notify_rule.h
3   * @author krayn
4   *
5   * Created on November 19, 2013, 4:52 PM
6   */
7  /*
8   * boundry - low - high
9   * function - func -error
10  * flanks
11  * flu
12  * fld
13  * true
14  * false
15  *
16  */
17 #ifndef NOTIFY_RULE_H
18 #define  NOTIFY_RULE_H
19
20 class Notify_Rule {
21 public:
```

```
22      Notify_Rule();
23      Notify_Rule(const Notify_Rule& orig);
24      virtual ~Notify_Rule();
25  private:
26      bool lastval;
27  };
28
29  #endif   /* NOTIFY_RULE_H */
```

DSA_Framework/notify_rule.cpp

```
1  /**
2   * @file   notify_rule.cpp
3   * @author krayn
4   *
5   * Created on November 19, 2013, 4:52 PM
6   */
7
8  #include "notify_rule.h"
9
10 Notify_Rule::Notify_Rule() {
11 }
12
13 Notify_Rule::Notify_Rule(const Notify_Rule& orig) {
14 }
15
16 Notify_Rule::~Notify_Rule() {
17 }
```