Entwicklung eines Webservice für Android-Apps zum „Self-Checkout“ an Kassensystemen

Bachelorarbeit im Studiengang Technische Informatik

des Fachbereichs Informatik und Medien der Beuth Hochschule für Technik Berlin

Vorgelegt von:	Christian Rau
	Lichtenrader Damm168
	12305 Berlin
	Martrikel-Nr.:s750062
Betreuende Lehrkraft:	Professorin Dr. Heike Ripphausen-Lipa
Gutachter:	Prof. Dr. Tramberend
Betreuer bei der Wincor Nixdorf AG:	Dr.-Ing. Sönke Kannapinn (Diplom-Informatiker)

Inhaltsverzeichnis

Geheimhaltungsvereinbarung

Aus der und im Zusammenhang mit der Bachelorarbeit der bei der Wincor Nixdorf AG (WN) zur Erstellung einer Bachelorarbeit tätigen Studenten der Beuth Hochschule für Technik Berlin, Herrn Christian Rau (Student), erhalten die bestellten Prüfer gegebenenfalls Kenntnis von Informationen, die Betriebsgeheimnisse der Wincor Nixdorf AG darstellen oder die die Wincor Nixdorf AG üblicherweise nicht an Dritte gibt, z.B. Informationen über technische Verfahren, technische Problemstellungen, Konstruktionsunterlagen, Fertigungseinrichtungen und Geschäftsvorgänge (insgesamt nachfolgend als „Informationen“ bezeichnet). Die bestellten Prüfer erklären sich hiermit bereit, diese Informationen geheim zu halten und auch nicht außerhalb der Betreuung dieser Bachelorarbeit und über das für die ordnungsgemäße Betreuung der Bachelorarbeit erforderliche Maß hinaus zu verwenden. Die Pflicht zur Geheimhaltung beginnt mit Aufnahme der Betreuung des Studenten und endet (02.10.12) nach Einreichen der Bachelorarbeit durch den Studenten zur Bewertung.

Die bestellten Prüfer versichern deshalb Folgendes:

1. Die an der Prüfung Beteiligten werden alle Informationen geheim halten, Dritten nicht zugänglich machen und diese auch nicht außerhalb der Betreuung der Bachelorarbeit verwenden, solange und soweit diese nicht bereits allgemein bekannt sind oder werden, ohne dass die Prüfer dies zu vertreten haben, den Prüfern bereits vor ihrer Mitteilung ohne Pflicht zur Geheimhaltung rechtmäßig bekannt waren, von den Prüfern unabhängig und ohne Rückgriff auf die von dem Studenten erhaltenen Informationen erarbeitet worden sind, oder die den Prüfern von einem Dritten rechtmäßig und ohne Geheimhaltungsverpflichtung mitgeteilt wurden oder die Wincor Nixdorf AG im Einzelfall einer Weitergabe oder Nutzung schriftlich zugestimmt hat.

2. Die Prüfer werden die Bachelorarbeit Dritten außerhalb des Prüfungsverfahrens nicht zugänglich machen und sie erst veröffentlichen oder die Ergebnisse der Bachelorarbeit öffentlich bekanntgeben (z.B. in einem wissenschaftlichen Vortrag oder einem Aufsatz), wenn die Bachelorarbeit und deren Ergebnisse von der Wincor Nixdorf AG hierzu schriftlich freigegeben wurden.

3. Diese Erklärung und die Pflicht zur Geheimhaltung steht den Rechten und Pflichten des Studenten und der Prüfer nach der Prüfungsordnung nicht entgegen und behindert insbesondere nicht die Durchführung des Verfahrens zur Prüfung und Bewertung der Bachelorarbeit als Prüfungsleistung des Studenten sowie die dazu erforderliche Vorlegung der Bachelorarbeit an Dritte.

Berlin, den 02.10.12	***Berlin***, den 02.10.12	***Berlin***, den 02.10.12
Wincor Nixdorf AG	Student	Betreuende Lehrkraft (Prüfer/in) der Beuth Hochschulefür Technik Berlin
Dr.-Ing. Sönke Kannapinn	*Christian Rau*	*Prof. Dr. Heike Ripphausen-Lipa*

Schutzklausel Wincor Nixdorf

1 Einleitung

Selbstbedienung findet heute in vielen Bereichen statt und wird immer selbstverständlicher. Man denke nur an Bankautomaten oder an den letzten Einkauf in einem Einrichtungskaufhaus wo man auch hierzulande sehen kann, wie Selbstbedienungskassen, so genannte Self-Checkout-Kassensysteme, den Menschenandrang im Kassenbereich bewältigen. Im Rahmen des Praxissemesters und während meiner Werkstudententätigkeit bei Wincor Nixdorf durfte ich mich in die Weiterentwicklung solcher Systeme einbringen. Auch Wincor Nixdorf stellt unter anderem Selbstbedienungskassen für den Self-Checkout (SCO) her. Dazu bietet Wincor Nixdorf passend die entsprechenden Dienstleistungen und Software an. Die Software TPiSCAN, die in Berlin und teilweise in den USA entwickelt wird, ist eine Java-basierte Handels-Softwarelösung, die im Filialbetrieb auf Self-Checkout-Kassensystemen von Wincor Nixdorf eingesetzt wird. TPiSCAN erweitert dabei die eigentliche Abverkaufssoftware – auch „Point of Sale“ (POS) genannt, welche zur Verwaltung der Transaktionen und der dahinter angeschlossenen Logistik sowie zur Beschaffung aktueller Artikel- und Rabattinformationen dient, um die im Self-Checkout (SCO) erforderlichen Funktionen und Geräte. Für TPiSCAN existiert von Wincor Nixdorf eine einheitliche Schnittstelle, um die separaten „Point Of Sale“ (POS)-Applikationen der unterschiedlichsten Hersteller anzubinden. Etwa 40 solcher unterschiedlicher POS-Integrationen sind auch dank der Schnittstelle bereits erfolgt, unter anderem auch mit den WN-eigenen POS-Produkten TP.net und TPLinux.

Auf den Markt und in den Alltag der Endverbraucher drängen immer mehr neuartige Internet-fähige Geräte, so genannte Smartphones und Tablet-PCs mit teilweise neuen Software-Architekturen. Der Markt um „Mobile Computing“ ist zurzeit immer noch stark im Wachstum und die Endgeräte werden immer leistungsfähiger. Sie verfügen mittlerweile sogar über eine Vielzahl von Sensoren und Kommunikationstechnologien, was deren Einsatzbreite und Flexibilität zusätzlich erhöht. Auch der Bereich des E-Commerce hat sich in den letzten Jahren rasant entwickelt. vgl. [5] Kunden erwarten heute auch vom stationären Handel, dass er sich nicht nur im Internet präsentieren, sondern auch seine Produkte darüber verkaufen kann. Er muss seine Ware also über mehrere Kanäle vertreiben. Dabei erwartet der Kunde natürlich, dass alle Artikel auf allen Vertriebskanälen zu denselben Preisen und Rabattmöglichkeiten verfügbar sind. Die oben genannten historisch gewachsenen Abverkaufssysteme (POS), die im stationären Handel breitflächig betrieben werden, sind meist nicht auf die Anforderungen im E-Commerce ausgelegt, weshalb dann parallel zusätzliche Systeme mit Webtechnologien für den E-Commerce installiert werden. Dabei war es in der Vergangenheit problematisch, Preise, Bestände, Rabattaktionen, Umsätze, Kundentransaktionen und Sortimente in Echtzeit mit allen Vertriebskanälen abzugleichen. Es ist zu beobachten, dass sich E-Commerce auch im „Mobile Computing-Bereich“ verstärkt durchsetzt. Durch diese Entwicklungen sind nun auch neue Einkaufsszenarien denkbar, die sogar mehrere Vertriebskanäle bündeln und verschmelzen lassen können. In der Branche wird hier auch von Multichannelstrategie gesprochen.

Wie sieht so eine Vision aus? Ein Kunde soll beispielsweise in Zukunft in der Lage sein, in einem Geschäft einzukaufen, indem er selbst Artikel mit seinem Smartphone erfasst. Dabei benutzt das Smartphone die integrierte Kamera um die Barcodes auf den Verpackungen einzulesen. Der Kunde bezahlt im Kassenbereich, indem seine Transaktion, die alle seine eingescannten Artikel enthält, an ein SCO-Terminal übergeben wird. Das Szenario hat für den Kunden den Vorteil, dass er nicht mehr so viel Zeit im Kassenbereich verbringen muss. Zum anderen kann er bei dieser Art des Einkaufens sein Informationsbedürfnis über die Artikel und Preise auch im Ladengeschäft stillen, wie er es bisher auch vom E-commerce von zu Hause aus gewohnt ist. Er könnte dabei auch gezielt über mögliche Rabattaktionen informiert werden. Des Weiteren besteht die Möglichkeit, einen virtuellen Einkaufszettel zu führen, den man bereits von zu Hause aus angelegt haben könnte und im Ladengeschäft mit Hilfe des Smartphones abarbeiten könnte. Für das Anlegen eines Einkaufszettels oder um von zu Hause aus zu bestellen, könnte die App auch eine E-Commerce-Komponente enthalten oder mit einer im Browser des Handys verknüpft sein. In beiden Fällen verbindet er sich zu einer POS-Applikation, die auf irgendeinem Server des Händlers läuft.

Um die erfolgreichen, bestehenden Integrationen zu den POS-Applikationen der Händler über die oben genannte Schnittstelle von Wincor Nixdorf wieder zu verwenden und zu schützen, soll das Einkaufen über eine Android Smartphone App als logische Konsequenz ebenfalls über eine Bibliothek für Wincor Nixdorfs Schnittstelle erfolgen. Während meines Praxissemesters wurde bereits eine App als Prototyp umgesetzt, die die TCP-Socket-basierte Implementierung zur Kommunikation mit dem POS einsetzt.

Wegen der Beliebtheit von service-orientierten Architekturen bei Geschäftsleuten und der heute besonders im Java-Enterprise-Umfeld guten Unterstützung von Web Services soll diese Bibliothek zukünftig intern statt des proprietären Protokolls Webservices verwenden. Dabei stellt sich die Frage, ob und wie sich das bestehende Interface als Webservice implementieren lässt. Dabei muss auch untersucht werden, ob die technischen Anforderungen, die schon für die Referenzimplementierung galten, sich auch für den dann entworfenen Webservice umsetzen lassen. Dabei sind zwei wesentliche Punkte von besonderem Interesse. Zum einen soll der Webservice durch den Codegenerator von der Interface-Spezifikation abgeleitet werden. Zum anderen soll eine so erzeugte Bibliothek für die Programmier Sprache Java auch für ein Smartphone als Webservice-Client verwendbar sein.

2 Aufgabenstellung

Die grundlegende technische Idee bei Wincor Nixdorf für neue Multichannel Produkte, die für mehrere Vertriebskanäle geeignet sind, besteht darin, die in der SCO- Software TPiSCAN vielfach bewährte POS-Adapter-Schnittstelle über Webservices zu nutzen. Ein Smartphone kann so mit einer POS-Instanz eine Verkaufstransaktion direkt durchführen. Dieses Konzept ermöglicht den direkten Zugang zu aktuellen Artikelinformationen mit all ihren möglichen Rabatten und lässt sich ideal in die vorherrschende IT-Infrastruktur des stationären Handels integrieren.

Die hier vorliegende Bachelor-Arbeit soll folglich herausarbeiten, wie Wincor Nixdorfs einheitliche POS-Adapter Schnittstelle für Self-Checkout Systeme mittels Web Services formuliert werden kann. Es soll ein beispielhafter Entwurf mit Implementierung in der Programmiersprache Java erfolgen. Das Ergebnis soll in Form einer Bibliothek vorliegen, die auch einem Client mit Android-Plattform ermöglicht, den umzusetzenden Webservice zu konsumieren. Denkbare Implementierungen für andere Programmiersprachen wie Objective C und Smartphone Plattformen wie zum Beispiel Apples Iphone mit iOS Betriebssystem brauchen nicht weiter untersucht werden. Zudem sollen in der Arbeit auch keine Technologieevaluationen von Webservices im Allgemeinen unternommen werden. Zudem sind Implementierungen in Richtung Sicherheit und Authentifizierbarkeit der Kommunikation über den Webservice für den Rahmen dieser Arbeit zu vernachlässigen.

Teil der Aufgabe soll es sein, eine Beispiel-Implementierung für Smartphones mit Android Plattform zu erstellen. Diese Client-Applikation soll die Funktionsfähigkeit für einen ersten Anwendungsfall als Beispiel demonstrieren. Dabei soll eine Demo-Applikation (App) die schon während meiner Praxisphase erstellt wurde, entsprechend angepasst werden, so dass sie fortan über Webservices kommuniziert. Diese App soll als Grundlage für erste Verkaufspräsentationen dienen, um zukünftig gebündeltes, mobiles Einkaufen demonstrieren zu können.

3 Fachliches Umfeld und Technologien

3.1 Point of Sale / checkout

In der englischen Wikipedia Enzyklopädie steht: „Point of sale (POS) (also sometimes referred to as point of purchase (POP) or checkout is the location where a transaction occurs. A "checkout" refers to a POS terminal or more generally to the hardware and software used for checkouts, the equivalent of an electronic cash register. A POS terminal manages the selling process by a salesperson accessible interface. The same system allows the creation and printing of the receipt.“ [2]

Ferner wird in diesem Zusammenhang auch von einem EPOS gesprochen, wenn man beim „Point of Sale“ nur den Kassenbereich an sich meint. Vgl. [3]

3.2 TPiSCAN und „Self checkout“

Die Wincor Nixdorf AG stellt neben Geldautomaten, Kiosksystemen, Kassensystemen mit deren Peripherie und Flaschenautomaten auch selbstbedienungsfähige Computerkassen her. Diese sogenannten Self-Checkout-Lösungen ermöglichen es dem Kunden, Waren ohne einen Kassierer zu erfassen und zu bezahlen. Dies kann für den Kunden die Wartezeit im Kassenbereich enorm reduzieren. Wie oben bereits erwähnt, wurde für TPiSCAN eine Schnittstelle definiert, um mit der eigentlichen Abverkaufssoftware dem POS zu kommunizieren.

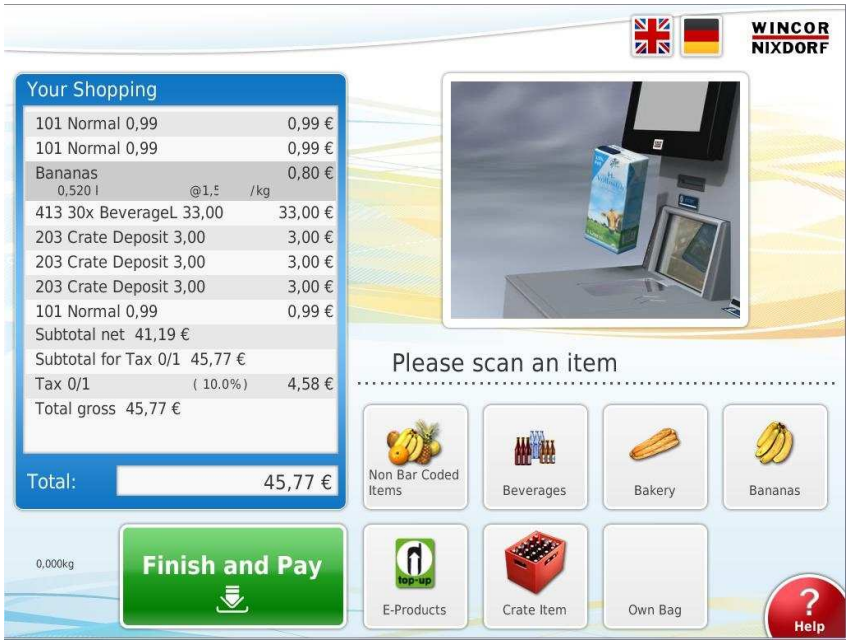

Abbildung 1 TPiSCAN Bedienoberfläche Quelle: WN intern

3.3 Wincor Nixdorfs POS-Schnittstelle

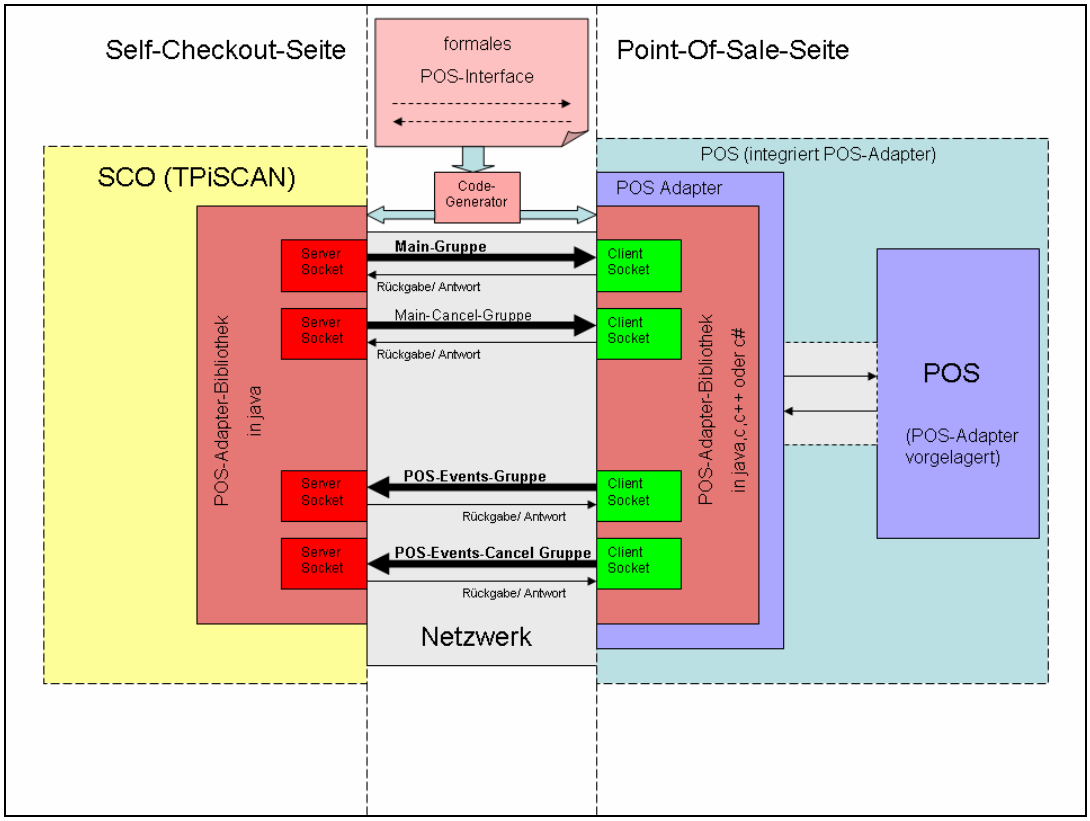

Abbildung 2 POS-Adapter (eigene Darstellung)

Wincor Nixdorfs Schnittstelle zur Kommunikation mit dem POS ist formal spezifiziert und wurde durch ein proprietäres, netzwerk-basiertes, binäres Protokoll implementiert. Viele verschiedene Integrationen mit POS-Fremd-Applikationen sind bereits über diese Schnittstelle erfolgt. Die POS-Applikationen müssen dabei nicht selbst Java-Software sein. Die Schnittstelle besteht im Wesentlichen aus Methoden-Aufrufen, die komplexe Datenstrukturen zwischen Aufrufer und Aufgerufenem austauschen. Genauer geschieht dies heute über TCP Socket-Verbindungen. Die technischen Details des Protokolls sind in Code-Bibliotheken gekapselt, die für verschiedene Programmiersprachen wie Java, C oder C# und verschiedene Betriebssystemplattformen bereitgestellt werden.

Die Schnittstellen-Spezifikation, die in der Version 2.0 zur Verfügung steht, definiert formal die Kommunikation zwischen POS und SCO. Sie besteht grundsätzlich aus Methoden mit Eingabe- und Ergebnisparametern, die entweder in der Richtung SCO nach POS ausgeführt werden können (d.h. SCO ruft auf und POS führt aus) oder umgekehrt in der Richtung POS nach SCO (d.h. POS ruft auf und SCO führt aus). Methodenaufrufe werden in serialisierter Form zusammen mit ihren Parametern übertragen. Die komplexen Datentypen, die für die Parameter benötigt werden, sind ebenfalls in der Spezifikation genau definiert.

Wie aus Abbildung 2 ersichtlich ist, sind vier Kommunikationskanäle für die Schnittstelle vorgesehen. Der Grund hierfür ist, dass die Methoden in Gruppen eingeteilt sind, wobei aus jeder Gruppe stets höchstens eine Methode aktiv sein kann. Man spricht hier auch von einer Synchronisationsgruppe. Die Schnittstelle definiert vier solcher Gruppen. Es gibt eine „Main-Gruppe“, die Methoden für die Richtung SCO zum POS enthält. In der umgekehrten Richtung gibt es entsprechend die „POS-Events-Gruppe“. Zu den beiden Gruppen gibt es eine dazugehörige „Cancel-Gruppe“ (entsprechend gleiche Richtung), die einen Main- bzw. POS-Events-Call abbrechen kann. Es könnte z.B. der Fall sein, dass auf die Antwort eines Methodenaufrufs zu lange gewartet werden muss. Als Konsequenz könnte die Aufrufende Seite den Aufruf stoppen. Da immer nur ein laufender Aufruf(„Call“) über eine Syncronisationsgruppe erlaubt ist, wird das durch einen Methoden-spezifischen Call über die entsprechende „Cancel-Gruppe“ veranlasst.

Jede Syncronisationsgruppe verfügt, wie oben angedeutet, über einen eigenen TCP-Socket, über den die Methodenaufrufe mit ihren Parametern als Bytestrom durch Serialisieren und Deserialisieren gesendet und empfangen werden kann. Beim Empfang einer Methode kommt es direkt zur Ausführung der durch die Empfängerseite implementierten Methode. Die Antwort aus Ergebnisparametern wird so ebenfalls über den gleichen TCP-Socket zurück übertragen.

In der Java Implementierung ist die Serialisierung und Deserialisierung in den Methoden und Datentypen fest codiert und benutzt aus Interoperabilitätsgründen keine Java Mechanismen. Für die TCP-Sockets existiert zudem eine Handler-Klasse, die den Verbindungsaufbau und Fehler in der Kommunikation behandelt.

Die Schnittstelle wurde in den Programmiersprachen C++, C# und Java in Form von inhaltlich identischen Bibliotheken umgesetzt. POS-Entwicklern ermöglicht das, einen spezifischen POS-Adapter für ihre POS-Software zu implementieren. Die einfache Integration von TPiSCAN durch Wincor Nixdorfs POS-Schnittstelle war in der Vergangenheit ein gutes Verkaufsargument und somit auch einer der Gründe für dessen Erfolg.

Für die Umsetzung der Schnittstelle in die einzelnen Programmiersprachen wurde statt auf ein UML-Werkzeug, wie z.B.: Enterprise Architect von Sparx Systems [4], auf einen von WN eigens entwickelten Codegenerator gesetzt.

3.4 Wincor Nixdorfs Codegenerator für die POS-Schnittstelle

Große Teile der Bibliotheken für die POS- Schnittstelle, die für verschiedene Programmiersprachen bereit stehen, werden mit Hilfe eines firmeneigenen Code-Generator-Werkzeugs erzeugt. Der Hintergrund für die Verwendung und Entwicklung dieses Tools ist der, dass WN auf die Abhängigkeit von teuren UML-basierten Entwicklungswerkzeugen verzichten möchte.
Allgemein sei hier kurz die Funktionsweise des Codegenerators erläutert. Es existiert eine Textdatei mit einer bestimmten Syntax. Sie beschreibt die POS- Schnittstelle und wurde mit Hilfe der offiziellen Spezifikation manuell erstellt. Der Codegenerator liest diese maschinen-lesbare Datei ein und erzeugt intern ein objektorientiertes Datenmodell der beschriebenen Schnittstellen-Methoden und Datenstrukturen. Danach führt er ein Programm aus, welches durch sogenannte Template-Dateien gesteuert wird. Als Ausgabe des Programms wird Programmcode erzeugt, der eine weitgehend vollständige Implementierung der Schnittstelle darstellt. Der Codegenerator unterstützt als Ausgabesprachen C++, C# und Java. Die Template-Dateien sind für die jeweiligen Programmiersprachen spezifisch. Der Aufbau einer Template- Datei folgt einer formalen Grammatik, die von einer formalen Sprache beschrieben wird. Mit dem Codegenerator kann so programmatisch, iterativer und objektorientierter Programmcode erzeugt werden.
Für den Codegenerator steht bei WN seit kurzer Zeit eine kleine Dokumentation zur Erstellung solcher Template-Dateien zur Verfügung. Die Kurzdokumentation ist der Arbeit als Anhang beigefügt.

3.5 Demo-POS

Die Abteilung benutzt für die Entwicklung von TPiSCAN ein eigens entwickeltes „Demo-POS“, welches das Verhalten eines POS simuliert und dabei das Gegenstück der POS-Adapter-Schnittstelle in Java implementiert. Die SCO-Seite, also TPiSCAN, benutzt ebenfalls die Java-Variante der Bibliothek, wie auch die in der Praxisphase entwickelnde Android-App (siehe oben).

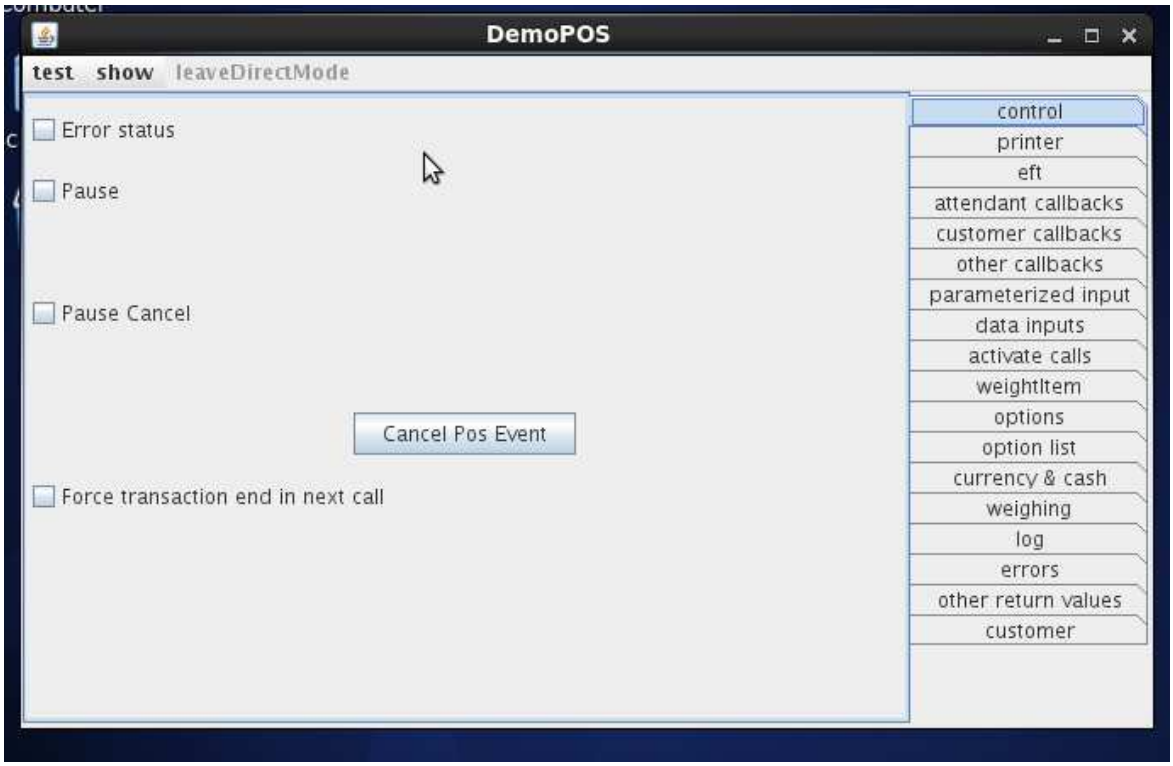

Abbildung 3 Bildschirmfoto vom Demo-POS mit grafischer Bedienoberfläche

3.6 Webservice-Protokolle

3.6.1 SOAP

SOAP steht als Abkürzung für „Simple Object Access Protocol”. SOAP ist ein Standard der vom W3C Konsortium unterstützt wird. Es dient als standardisiertes Verfahren für den Nachrichtenaustausch zwischen verteilten Systemen. Die Abkürzung SOAP wird heute als ein Eigenname behandelt, da die Abkürzung in der Entstehungsphase ungünstig gewählt wurde. Wer sich schon einmal intensiver mit SOAP auseinandergesetzt hat, weiß, dass sich mit SOAP auch komplexe Kommunikationsmodelle umsetzen lassen. Dabei muss die Kommunikation nicht unbedingt nur dafür verwendet werden, um z. B. auf Daten, in Form von Dokumenten, oder auf typisierte Objekte zuzugreifen. Es kann auch dafür genutzt werden, um Prozesse oder Berechnungen, z.B. auf einem Server, auszulösen. Von einem Protokoll im eigentlichen Sinne spricht man erst, wenn es zusammen mit einem Netzwerkprotokoll zur physischen Datenübertragung verwendet wird.

SOAP definiert dabei die logischen Strukturen („Envelope“) zum Verpacken der Kommunikationsdaten. SOAP baut in Folge immer auf einem physikalischen Netzwerkprotokoll auf. Häufig handelt es sich hierbei um HTTP, da in Unternehmen die Ports für HTTP in den Firewalls in der Regel nicht gesperrt werden. So können Dienste entwickelt werden die durch Webservices über Netzwerkgrenzen hinaus miteinander kommunizieren und dabei mit bestehenden Netzwerkarchitekturen kompatibel sind. Auch da wo der externe Zugang zu unternehmensinternen Systemen standardisiert nur über Webserver erreichbar sein soll, bietet sich SOAP über HTTP oder HTTPS an. Für mobile Apps und Web-Anwendungen gilt dies für gewöhnlich auch. SOAP-Nachrichten können jedoch als Alternative z. B. auch direkt über TCP/IP-Sockets mit beliebigen Ports versendet werden.

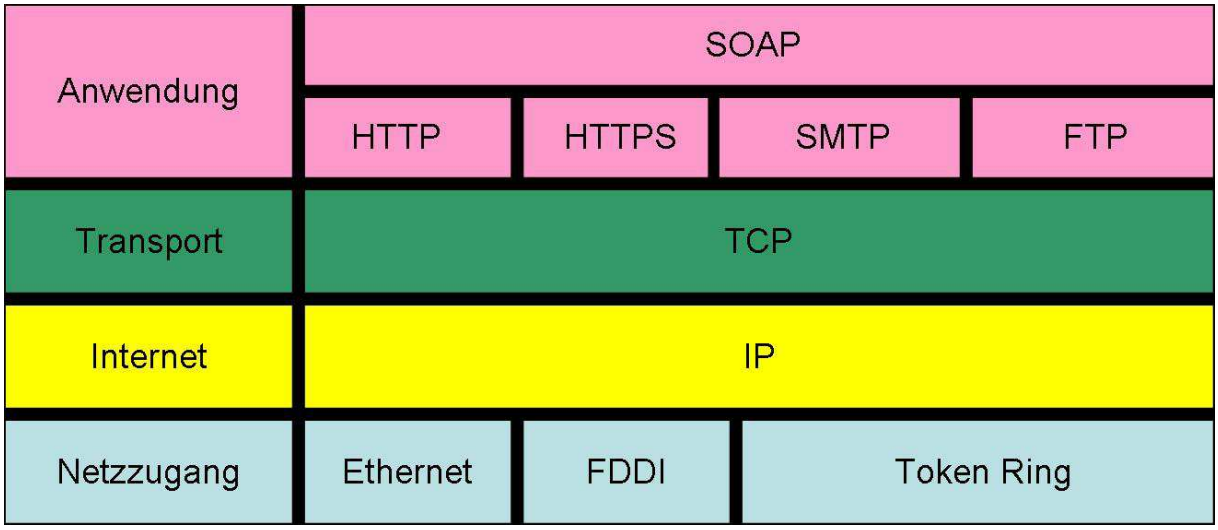

Abbildung 4: SOAP im TCP/IP-Protokoll-Stapel

In Abbildung 4 wird deutlich wie sich SOAP im TCP/IP-Protokoll-Stapel einfügt. Im Prinzip baut SOAP historisch gesehen auf XML-RPC (XML Remote Procedure Call) auf, wobei man sich auf XML als Übertragungsformat festgelegt hat. SOAP ist im Prinzip nichts anderes als ein XML-Derivat, durch das SOAP-Nachrichten formuliert werden, die an eine Protokollschicht übergeben und dann übertragen werden können. Wie bei XML-RPC können entfernte Prozeduren bzw. Methoden über ein Transportprotokoll wie z. B. HTTP aufgerufen werden. SOAP stellt ein Rahmenwerk (Framework) zur Verfügung, um textuelle wie auch binäre Daten in SOAP-Nachrichten einzubinden. Nachdem die SOAP-Nachrichten versendet wurden, ermöglicht das Framework sie dann wieder interpretieren zu können.

Durch die SOAP-Spezifikation vom W3C[19] werden Konventionen festgelegt wie entfernte Methoden bzw. Prozeduren aufgerufen werden. Für die dafür benötigten XML-Konstrukte wurden eigene Namensräume festgelegt. Die SOAP Spezifikation definiert wie, Nachrichten aufgebaut sind und wie sie mit einem beliebigen Transportprotokoll zu versenden sind. SOAP gibt es aktuell in der Version 1.2 die abwärtskompatibel zu Version 1.1 ist, welche noch oft eingesetzt wird.

Eine SOAP-Nachricht, auch Envelope(eng. Umschlag) genannt verpackt eine Nachricht in SOAP konformes XML. Das Grundgerüst ist wie folgt aufgebaut und besteht zunächst aus folgenden Grundelementen.

```
<?xml version="1.0" encoding=”UTF-8” ?>

<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">

      <soap:Header>

      </soap:Header>

      <soap:Body>

      </soap:Body>

</soap:Envelope>
```

Envelope ist das Wurzelelement und umschließt die Nachricht mit ihrem Inhalt. Für den Envelope existiert der XML-Namensraum Präfix „soap“. Mit ihm werden die Namensräume der SOAP-Version 1.1 (http://schemas.xmlsoap.org/soap/envelop) und der SOAP-Version 1.2 (http://www.w3.org/2003/05/soap-envelope) angegeben.

Der Header oder Kopfelement ist ein optionales Element, welches nur einmal zu Beginn der Nachricht angegeben wird. Er beinhaltet Informationen über das folgende Body-Element.

Das Bodyelement wird immer angegeben und kapselt die Nutzdaten einer SOAP-Nachricht in einem wohlgeformten XML-Dokument. Es existieren für SOAP auch Erweiterungen die binäre Daten im SOAP-Body einbinden können (z.B. MTOM).

3.6.1.1 SOAP-Nachrichtenverfahren

Für SOAP wurden vier Nachrichtenverfahren definiert, wovon allerdings nur noch zwei, auf Grund der Verfügbarkeit von XML-Schema, im Allgemeinen hauptsächlich benutzt werden. Von besonderem Interesse sind hier „document/literal“ und „RPC/literal“.

„document/literal“ beinhaltet wie oben beschrieben beliebige wohlgeformte XML-Dokumente. Eine Nachricht wird gesendet ohne eine direkte beantwortende Nachricht zu bekommen. Eine Antwort kann hier nur asynchron erfolgen.

„RPC/literal“ hier wird der Body der Nachricht anders formatiert. Es ermöglicht einen Methodenaufruf beim Diensteanbieter. Hierzu werden die Nachrichten als Request (Anfrage), Response (Antwort) oder als Fault-Message (Fehlernachricht) ausgelegt. Ein Request ruft eine Methode an einem Dienst auf und übergibt dabei die benötigten Parameter. Wenn die Anfrage erfolgreich beim Diensteanbieter ausgeführt wurde, wird eine Antwort mit den Ausgangsparametern der Methode zurück geschickt. Sollte es zu einem Fehler bei der Kommunikation kommen wird eine Fehlernachricht als Antwort zurück geschickt.

3.6.1.2 WSDL

WSDL steht für Web Service Description Language. Wie der Name schon sagt, können mit der WSDL Dienste beschreiben werden. Da die WSDL einen Dienst abstrakt definiert, ist es möglich nicht nur SOAP-Webservices zu beschreiben. WSDL baut auf XML auf und lässt sich um beliebige Netzwerkprotokolle mit eigenen Nachrichtenformaten erweitern. Weil WSDL ein weiteres XML-Derivat ist, ist es genauso Plattformunabhängig wie XML. In einer WSDL-Datei beschreiben die Elemente types, message, portType, binding, port und service einen Webservice. Wie eine WSDL-Datei aufgebaut ist lässt sich gut in der W3C Spezifikation nachlesen (siehe [20] und [21]).

3.6.2 REST

REST ist eine mögliche Alternative zur Umsetzung des zu entwickelnden Webservice. Jedoch hat man sich bei WN während der Bearbeitungszeit dieser Arbeit vorläufig gegen eine Umsetzung mit REST, zu Gunsten von SOAP, entschieden. Folgend soll REST hier jedoch kurz vorgestellt. Unterschiede zu SOAP die zu der Entscheidung beigetragen haben SOAP an Stelle von Rest in dieser Arbeit zu verwenden, werden unter 5.1 erläutert.

REST stellt einen Architekturstil dar, der auf HTTP beruht. Der Fakt, dass REST auf dem Protokoll HTTP gründet, macht REST allerdings nicht selber zu einem Protokoll. Vielmehr stellt REST Richtlinien auf, wie Dienste im Internet zu entwerfen sind. Dabei wird, im Gegensatz zu SOAP, auch darauf verzichtet, genau festzulegen, wie Nachrichten formatiert werden(siehe [18] S.367).

3.7 SOA als Architekturansatz

SOA steht für Serviceorientierte Architektur. SOA ist ein Management- und Architekturprinzip, wobei die Architektur aus verteilten Diensten besteht, die über Schnittstellen miteinander kommunizieren. Ein Dienst implementiert dabei einen Prozess bzw. eine Geschäfts-Funktion. Die Dienste sind fachlich wieder verwendbar und können miteinander zu größeren Geschäftsprozessen kombiniert („orchestriert“) werden. Ein Dienst wird unabhängig von einer verwendeten Technologie und konkreten Implementierung definiert. Eine SOA kann also durch die Geschäftsebene eines Unternehmens definiert werden. Ein solcher Ansatz ist folglich auch bei den Entscheidungsträgern in Unternehmen beliebt.

Üblicherweise werden Webservices zur Umsetzung einer solchen SOA in der Praxis verwendet. Der Begriff SOA wird mit dem Begriff Webservice oft verwendet. Eine Umsetzung von Wincor Nixdorfs POS-Schnittstelle als Webservice könnte Handelsunternehmen zukünftig in die Lage versetzen, ihr POS wie einen Dienst zu betrachten und als solchen in einer SOA zu planen. Die Umsetzung einer mobilen „Self-Checkout“-Applikation stellt einen ersten Anwendungsfall und einen ersten Schritt in diese Richtung dar.

3.8 Android

In dieser Arbeit soll unter anderem ein Android-Programm so erweitert werden, dass es mit einem zuvor entworfenen Websevice kommunizieren kann. Aus diesem Grund ist es für das Verständnis der Arbeit wichtig, sich mit Android als Betriebssystem und Softwareplattform zuvor auseinander gesetzt zu haben.

„... Android ist sowohl ein Betriebssystem als auch eine Software-Plattform für mobile Geräte wie Smartphones, Mobiltelefone, Netbooks und Tablets… .“[5] Google treibt gemeinsam mit 84 weiteren Unternehmen im Rahmen der „Open Handset Alliance“ die Entwicklung der Plattform voran (vgl.[6]). Als kompakten Einstieg in die Thematik sei an dieser Stelle auf den gut belegten Wikipedia Artikel zu Android verwiesen, um nicht unnötig allgemein Bekanntes zu wiederholen. Ein Ausdruck des Artikels liegt dem Anhang bei.

Für die einzelnen Plattformversionen von Android existieren korrespondierende „API-Levels“, also Versionen der Programmierschnittstelle. Die bisherige Entwicklung der „Selfcheckout“-App wurde für API-Level 7 programmiert, was Android in der Version 2.2.x entspricht (vgl. [8]). „Man sollte in der Praxis immer das kleinstmögliche „Build Target“ wählen, damit die Anwendung auf möglichst vielen Endgeräten lauffähig ist. “ ([7], S.3, 1.1). Mit Build Target ist hier wieder die Plattformversion gemeint. Auf einem neueren Gerät mit einem höheren „API-Level“ können Anwendungen ausgeführt werden, die für eine ältere Android-Plattformversion programmiert wurde (vgl. [8]). Aufgrund der Abhängigkeit zu einer Bibliothek vom Zxing-Projekt, die das Einscannen von Barcodes über die Kamera des Smartphones ermöglicht, musste das minimale API-Level auf 7 fest gelegt werden. Dies wird über das „minSdkVersion-Tag“ im Android-Manifest festgelegt (vgl. [9]). Das Android-Manifest ist eine XML Datei im Wurzelverzeichnis einer App. In ihr werden alle nötigen Informationen über die App festgelegt, die das System benötigt bevor, es sie ausführen kann (siehe [10]). Dafür wurden eine Menge von XML-Elementen und -Attributen festgelegt (vgl. [10], [9]). Man kann davon ausgehen, dass die App nach [11] mit dem API- Level 7 immer noch auf mehr als 95 Prozent aller Geräte ausgeführt werden kann, die zur Zeit weltweit aktiv betrieben werden.

4 Pflichtenheft

Die Anforderungen an die Arbeit lassen sich in zwei Teilbereiche einteilen. Wobei der Erste, also die POS-Schnittstelle durch Webservices umzusetzen, den Kern der Arbeit darstellt. Im zweiten Teilbereich lassen sich bestimmte Anforderungen an die Android App ableiten, wobei hier beispielhaft ein Anwendungsfall („Use Case“) umgesetzt werden soll.

4.1 Funktionale Anforderungen der Webservice Umsetzung

Die funktionalen Anforderungen an die Schnittstelle und somit auch an den Webservice lassen sich in funktionale Gruppen unterteilen, für die bereits alle Methoden und Datentypen definiert wurden. Eine Auflistung der Methoden und deren Einordnung in die jeweiligen funktionalen Gruppen erfolgt unter 4.1.1. Diese Methoden sollen durch Webservices als Webmethoden umgesetzt werden. Die Funktionalitäten der einzelnen Methoden werden nicht implementiert, da sie nur Aufrufe an das POS auslösen bzw. von POS-Herstellern implementiert werden müssen. Hierfür soll eine Bibliothek erstellt werden, die dafür benutzt werden kann, das POS-Interface durch Webservices als Server bereit zu stellen. Die Bibliothek soll auch einem Client Funktionalitäten bieten, um den Webservice anzusprechen. Genauer soll hier ein Smartphone mit dem Betriebssystem Android als Client dienen.

4.1.1 Methoden der POS-Schnittstelle und Einordnung in funktionale Gruppen

Basisfunktionalitäten:

getStatus, signOn, signoff, getGlobalProperties

Behandeln von Transaktionen:

startTransaction, recallTransaction, storeTransaction, preparePayment, unpreparePayment, transactionFinished, voidTransaction, addSubtotal

Artikel (“Items”) verkaufen:

getItemDetails, addItem, addItemList, addItemByDepartment, voidEntry, priceChange, getEtopupDetails, addEtopup, weighItem, computePrice, getCrateInfo

Preisnachlässe “Markdowns”:

addParticularDiscount, addBlanketDiscount

Bezahl Funktionalitäten “Payments”:

addTender, performPayment, payWithCard

Kundenkarten behandeln:

processCustomerCard, performCustomerCard, unregisterCustomer

Coupons, Gutscheine “Vouchers”, Pfandrückgabe “Refounds”

addSimpleCoupon, addCoupon, addSimpleVoucher, addVoucher, addSimpleRefund, addRefund

Geld-Management Funktionalitäten für Geldgeräte:

currentBalance, loan, pickup, tillReport

Sonstige (die Keiner Gruppe zugeordnet werden):

Special, journalData, directMode, print, checkError, handleAnomaly, disconnect

Abbrüche von Calls (müssen unabhängig von den oben aufgeführten Aufrufen möglich sein) :

Cancel, cancelPosEvent

Neben der cancelPosEvent-Methode existieren Methoden, die bei WN als Call-Backs bezeichnet werden. Das betrifft alle Methoden, die in Richtung POS zum SCO aufgerufen werden, also hauptsächlich zu der POS-Events-Synchronisationsgruppe der Schnittstelle gehören. Für sie lassen sich im Folgenden noch zwei weitere funktionale Untergruppen aufstellen.

Dialog Call-Backs:

attendantStatusText, customerStatusText, attendantInputRequest, customerInputRequest, attendantConfirmationRequest, customerConfirmationRequest, nbAttConfRequest

Sonstige Call-Backs:

signatureRequired, printerNearEndOfPaper, printerOutOfPaper, printerProblem, activate, deviceStatusChange, posStatusChange, getDeviceStatusList

4.2 Nichtfunktionale Anforderungen der Webservice Umsetzung

Die Web-Service-Version der Schnittstelle soll auch durch Einsatz des Wincor Nixdorfs Code-Generator-Werkzeugs erzeugt werden. Das macht nachträgliche Änderungen an der Schnittstellenbeschreibung schneller und flexibel umsetzbar.

Die Serverkomponente (Dispatcher/ Forwarder/ Multiplexer), die den Webservice zur Verfügung stellt, soll später in der Lage sein, mehrere Smartphones an mehrere Instanzen einer POS-Applikation zu vermitteln. Die Bibliothek sollte also entsprechend benutzbar sein.

Da für die Skalierung des Konzeptes nicht nur bei Wincor Nixdorf auf Java-Enterprise-Umgebungen gesetzt wird, sollte die Serverkomponente auf einem „Applikationsserver“ wie Apache Tomcat oder JBoss Applikationsserver umsetzbar sein.

4.3 Funktionale Anforderungen Mobile Shopping App

Neben der Umsetzung der Web-Service-Version der Schnittstelle existiert die Anforderung, eine Demonstrations-App für mobiles „selfcheckout“ zu entwickeln.

Die Verfügbarkeit einer Demonstrations-App ist besonders für Entscheidungsfindungen im Unternehmen wie auch für die Gewinnung von Investoren wichtig und kann sich positiv auf die Fortführung der durch diese Arbeit verfolgten Bestrebungen auswirken. Die Umsetzung der App ist folglich für den zukünftigen Erfolg der POS-Schnittstelle als Werbservice Version essenziell.

Mit der App soll ein einfacher Anwendungsfall (Usecase) umgesetzt werden. Ein Kunde soll in der Lage sein, mit der App einen Artikel am POS einzukaufen. Der entsprechende Usecase wird durch das Usecase-Diagramm in Abbildung 5 visualisiert.

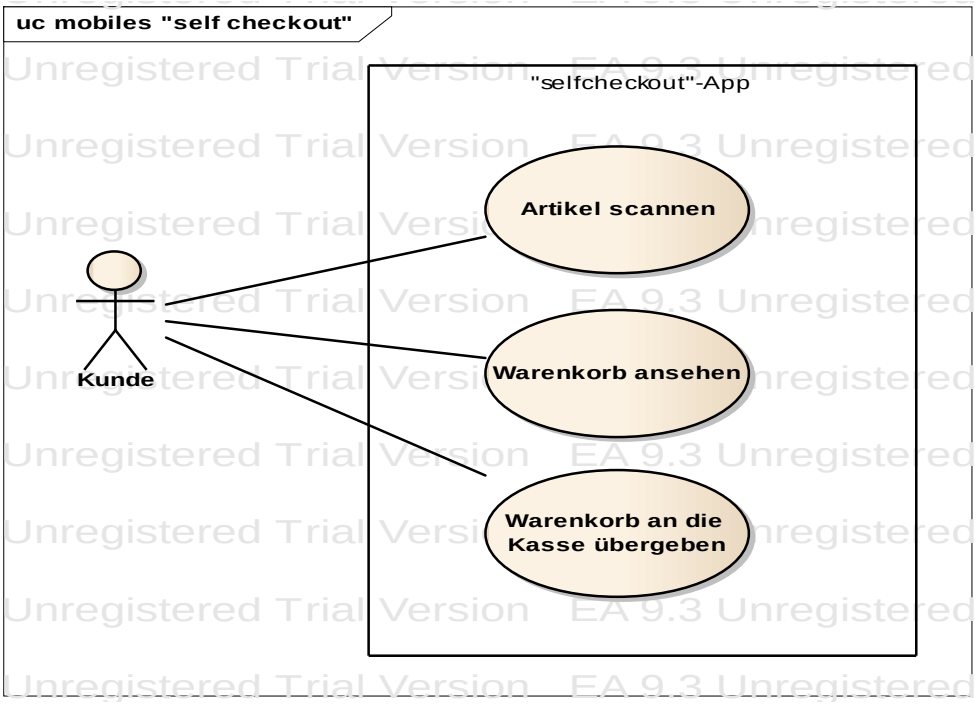

Abbildung 5 Use Case Diagramm der "selfcheckout"-App

4.4 Nichtfunktionale Anforderungen Mobile Shopping App

Außer der allgemeinen einfachen Bedienbarkeit der App durch den App- Bediener (Kunde) existieren keine nichtfunktionellen Anforderungen. Es kann eine in der Praxisphase erstellte App erweitert bzw. verändert werden.

5 Lösungsansätze

5.1 Lösungsansätze für den Webservice

Es existieren drei verschiedene Arten, wie man Webservices heute umsetzen kann. Die zwei dafür benötigten Technologien wurden unter 3.6 bereits vorgestellt.

Zum einen gibt es REST (Representational State Transfer), den populären Ansatz des ressourcen-orientierten Schnittstellenentwurfs. Hier stehen über URL identifizierbare Ressourcen im Mittelpunkt und ein Schnittstellenentwurf, der sich an die Architektur von HTTP anlehnt.

Zum Anderem gibt es noch den nachrichtenorientierten Ansatz, wobei vor allem Dokumente als Nachricht versendet werden. Hier kann SOAP mit dem dokumet/literal-basierten Nachrichtenverfahren für die Umsetzung verwendet werden.

Der dritte Ansatz ist schnittstellen- oder auch methoden- bzw. prozedurorientiert. Hier sind Aufrufe von Methoden/Prozeduren von zentraler Bedeutung. Sie sind vergleichbar mit denen einer Programmiersprache mit typisierten Parametern und Rückgabewerten. Es wird hier aber von einem RPC (Remote Procedure Call) gesprochen, da die Methoden aus der „Ferne“ z.B. über ein Netzwerk und HTTP aufgerufen werden können. Auch hier lässt sich SOAP mit dem RPC/literal-basierten Nachrichtenverfahren zur Umsetzung verwenden.

Die Wincor Nixdorf POS-Schnittstelle hat laut Schnittstellen-Spezifikation[17] 36 komplexe Daten-Typen die 66 Methoden als Parameter oder Ausgangsparameter dienen. Zu dem wurden in der Java Implementierung für die Ausgangsparameter jeder Methode Wrapper-Klassen erstellt um sie als einen Rückgabewert zu kapseln. Die Schnittstelle lässt sich als methodenorientiert beschreiben, weshalb eine Umsetzung mit SOAP nach dem dritten Ansatz sehr nahe liegt. Es werden durch Methodenaufrufe Prozesse angestoßen die laut Spezifikation der POS-Schnittstelle immer eine Antwort zurückgeliefert wird. Das entspricht dem RPC/literal-basierten Kommunikation per SOAP. Als Lösungsansatz scheidet SOAP mit dem dokumet/literal Nachrichtenverfahren aus. In [18] steht, „Alle zu versendenden Nachrichten werden in einer Ein-Weg-Kommunikation („One-Way Messaging) per plain-XML (also reinem XML) versandt. Ein-Weg-Kommunikation, weil im Gegensatz zu einer RPC/literal-basierten Kommunikation keine direkte Antwort an den Dienstenutzer erfolgen muss, bzw. wenn diese erfolgt, dann über eine eigene SOAP-Nachricht.“ Der RPC/literal basierte Ansatz eignet sich somit am besten zur Umsetzung der POS-Schnittstelle als Webservice.

Zu Beginn der Bearbeitungszeit wurde bei WN (nach gemeinsamer Beratung) entschieden, im Rahmen dieser Arbeit bei der Lösung der Aufgabe auf eine Umsetzung mit SOAP zu setzen. Es ist allgemein bekannt, dass REST-Umsetzungen in der Regel performanter sind, da sie JSON statt SOAP-XML als Übertragungsformat nutzen können. Eine Umsetzung mit REST war zwar eine Alternative, wurde jedoch erst einmal als mögliche Lösung für zukünftige Untersuchungen vorbehalten. Ein weiterer Grund für den Vorbehalt war der, dass vermutet wurde, dass das methoden-orientierte Design der Schnittstelle erst in eine ressourcen-orientierte Sicht umgewandelt werden müsste, um dem REST Architekturstil gerecht werden zu können. Es bestanden Zweifel, ob sich das in allen Fällen überhaupt vollständig umsetzen lassen würde. Die Erstellung von zwei Prototypen und einer damit verbundenen Untersuchung hätte sicherlich auch den Rahmen dieser Arbeit gesprengt.

Bei der Entwicklung eines Webservices basierend auf SOAP gibt es zwei übliche Vorgehensweisen. Zum einen die des „contract first“-Ansatzes, wobei der Vertrag (eng. Contract) in Form einer WSDL-Datei spezifiziert wird. Die dafür benötigten Datentypen müssen dazu noch in der XML-Schemasprache XSD spezifiziert werden. Danach kann aus diesen Beschreibungen Java Code generiert werden, der dann den Webservice implementiert. Dieser Code ist dann noch so anzupassen, dass bei einem Aufruf die entsprechende Geschäftslogik ausführt wird, die dann das Ergebnis als passenden Datentyp für die Antwort des Webservice liefern kann. Zum anderen existiert alternativ dazu die Vorgehensweise des „code first“-Ansatzes. Hier wird genau anders herum vorgegangen. Hier definiert man die Webmethoden und Datentypen des Webservice programmatisch mit Hilfe des Webservice-Frameworks. Dies ist besonders dann üblich, wenn Programmcode existiert der bereits eine bestehende Schnittstelle beschreibt. Bei der Schnittstelle von WN traf genau dieser Fall zu, deshalb fällt die Wahl auf den „code first“-Ansatz leicht. Um dann noch eine Beschreibung des Webservice (WSDL) über HTTP zu veröffentlichen, wird dazu eine WSDL-Datei aus dem Programmcode durch das Framework generiert. Als Framework soll hier JAX-WS („Java API for XML Web Services“) [12] aus dem Webservice-Framework Metro [13] dienen. Es wären noch andere Frameworks wie Apache Axis [16] denkbar gewesen, jedoch versteht sich Metro als die Referenzimplementierung (RI) für SOAP unter Java und ist bereits in der Java Standard Edition enthalten. In dieser Arbeit wird das JDK(Java Development Kit) von der Firma Oracle in der Version 6u33 verwendet. Es enthält das JAX-WS und JAXB -API in der Version 2.1 seit der JDK-Version 6u4 (siehe [14]).

Für die Clientseite kann man sich zur Entwicklungszeit alle Klassen, inklusive Datentypen, für die Benutzung eines Webservices durch das Werkzeug wsimport [15] aus der veröffentlichten WSDL-Datei erzeugen lassen. Dies gilt insbesondere auch für den „contract first“-Ansatz.

Der Lösungsansatz besteht darin, Webmethoden zu erstellen, die den alten Methodenklassen weitestgehend entsprechen. Die Klassen der bisherigen Schnittstelle enthalten sowohl Methoden, um sich selbst zu serialisieren und auf einen Byte-Strom zu schreiben, als auch welche, um sich selbst von einem Byte-Strom zu deserialisieren. Die Klassen der Datentypen verfügen ebenfalls über solche Funktionalitäten, um das Vorgehen der Methoden-Serialisierung auch für die Parameter der Methoden zu unterstützen. Auch auf die in der Java-Implementierung vorhandenen Wrapper-Klassen für die methodenspezifischen Rückgabewerte, die als Antwort auf einen Methodenaufruf zurück übertragen werden, trifft dies zu. Über Sockethandler-Klassen können die Byte-Ströme über speziell definierte TCP-Sockets versendet werden.

Die Webmethoden werden diese Implementierung ersetzen bzw. eine alternative Implementierung der Schnittstellen-Methoden darstellen. Die Methoden werden dabei auch in serialisierter Form versendet werden, wobei es sich dann jedoch um SOAP-konformes XML, welches über http versendet wird, handelt.

5.2 Lösungsansätze für den Android-Client

Wie in 5.1 beschrieben existiert die Möglichkeit, sich aus der Beschreibung des Webservice in Form einer WSDL-Datei sich den Webservice-Client generieren zu lassen. Leider existieren aber unter Android nicht die üblichen Webservice-Frameworks bzw. SOAP-Implementierungen. Weder JAX-WS mit JAXB noch Apache Axis werden nativ unterstützt. Das hat den Hintergrund, dass Android zwar in Java programmiert wird, aber nicht die Java-VM als Laufzeitumgebung benutzt. Android benutzt die von Google entwickelte Dalvik Virtual Maschine (DVM) mit einem eigenem Bytecode, der durch den Crosscompiler „dx“ aus Java-Bytecode generiert wird (siehe [7] S.21) . Die Verwendung der DVM hat nicht nur eine verbesserte Ausführung des Programmcodes auf Registerbasierten Microprozessoren zur Folge. Das Vorgehen stellt auch einen Lizenztrick dar, da echter Java-Bytecode und die Java-VM lizenzrechtlich geschützt sind. So kommt es auch, dass nicht alle Klassen der Java Standard Edition zur Verfügung stehen. Das Einfügen aller benötigten Pakete zur Nachrüstung eines entsprechenden Frameworks würde zudem eine resultierende App unverhältnismäßig vergrößern. Das ist auch für unseren Anwendungsfall schlecht, da davon ausgegangen werden muss, dass die App mitunter erst im Laden heruntergeladen und installiert wird. Um Performance und Ladezeiten gering zu halten, wird oft die leichtgewichtige SOAP-Bibliothek ksoap2 für Android verwendet. Ksoap wurde in der Vergangenheit bereits für Java ME (Micro Edition) basierte Geräte vielfach erfolgreich eingesetzt. Ksoap wird folglich auch in dieser Arbeit verwendet.

Der zu erfüllende Use Case der Android App wird bereits durch die in der Praxisphase erstellte App umgesetzt. Es reicht also aus sich auf die Nutzbarkeit des Webservices von Android aus zu konzentrieren. Eine zentrale Klasse ermöglicht den Zugriff auf das POS die Änderung dieser Klasse dahingehend das sie fort an über den Webservice kommuniziert ermöglicht es also den Use Case direkt umzusetzen.

5.3 Lösungsansätze für zusätzliche Erweiterungen (Mehrbenutzerfähigkeit)

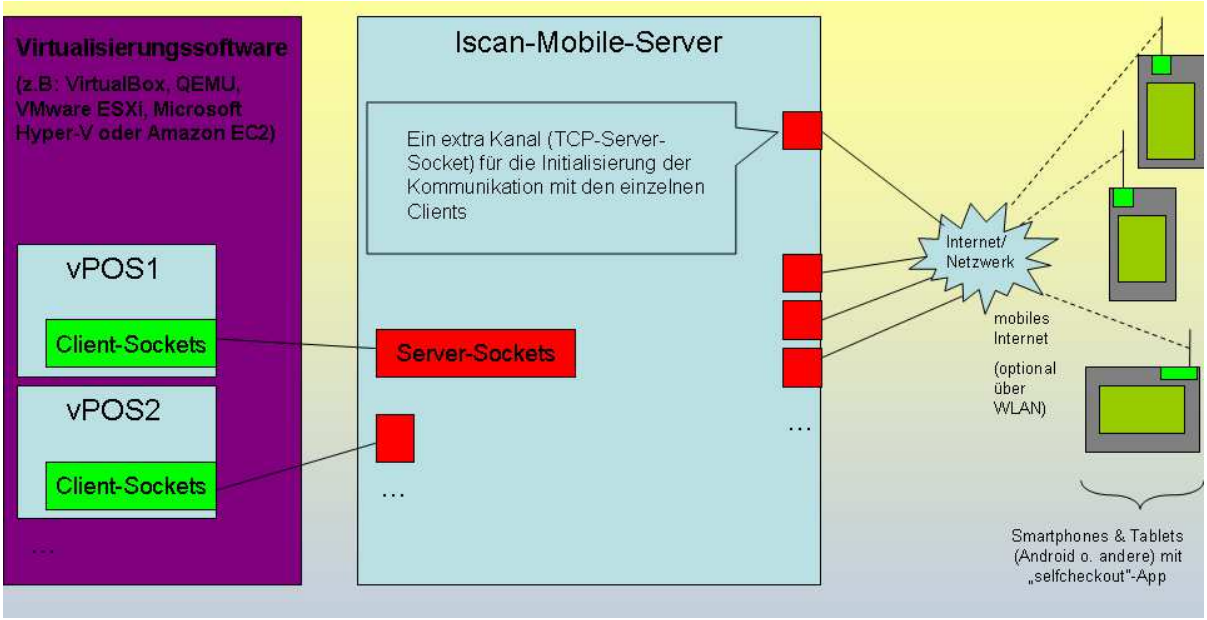

Abbildung 6 Serverarchitektur mit TCP-Sockets (eigene Darstellung)

Für den Anwendungsfall, wie er für mehrere „selfcheckout"-Apps erforderlich wäre, müssen die Aufrufe eines Clients an den Server genau einem dedizierten POS zugeordnet werden. Die allgemeine Vision sieht vor, dass die Kommunikation zwischen App und POS vom Server, der den Webservice zur Verfügung stellt, weitergeleitet wird. Der Server tritt hierbei auch als eine Art Vermittler und Einteiler („dispatcher") von freien POS-Instanzen auf. Für diese Erweiterung des „Use Case" aus 4.3 steht ein Demo-Server (siehe

Abbildung 6), der bereits während des Praxissemesters erstellt wurde, als Grundlage zur Verfügung. Bei der Erstellung dieses Servers, nennen wir ihn hier „Iscan-Mobile-Server", wurde die bestehende POS-Schnittstelle so modifiziert, dass sie das Weiterleiten („forwarding") der Methodenaufrufe unterstützt. Als alternative Architektur wäre es allerdings auch denkbar, dass das POS selber den Webservice zur Verfügung stellt. Demonstrativ würde dann das Demo-POS aus 3.5 den Webservice implementieren müssen. Jedoch müsste dann auch das POS die Mehrbenutzer-Fähigkeit im Sinne des mobilen „selfcheckouts" umsetzen. Das würde auch bedeuten, dass die Händler ihre POS-Software anpassen lassen müssten. Dies gilt es natürlich zu vermeiden, da dadurch unvorhersehbare Kosten verursacht würden. Zudem bestehen bereits viele Integrationen mit der POS-Schnittstelle von WN, die es auszunutzen gilt.

Eine Trennung der Client Aufrufe und eine damit verbundene Zuordnung eines Clients zu einer POS-Instanz lässt sich mit Hilfe der SOAP-Erweiterung WS-Adressing und der Implementierung eines Handlers erreichen. Bevor eine Nachricht gesendet oder dem Dienst übergeben wird durchläuft sie den Händler. Bei Empfang einer Nachricht sorgt er dafür, dass die Anfrage in der Webmethode vom zuvor zugeteilten POS bearbeitet wird.

6 Systementwurf

Wie in 5.1 angedeutet wird das Serialisieren und versenden über TCP/IP-Sockets der Methodenklassen (Abbildung 8 und Abbildung 7) durch Webservice-Methoden ersetzt. Genauer heißt das, dass durch die Definition der Webservice-Methoden eine Webservice Beschreibung, also aller Methoden und Datentypen, in Form einer WSDL-Datei erzeugt wird. Sie ist z.B. durch ihre Veröffentlichung dem Client bekannt. Dieser ist in der Lage, durch die Informationen aus der WSDL, SOAP-Envelops zu erzeugen. Sie können über HTTP an den Server verschickt werden. Die SOAP-Envelops beschreiben, durch SOAP konformes XML, die entsprechenden Methodenaufrufe („Requests“) aus der WSDL-Datei. Auf der Serverseite wird der „Request“ dann nach Ausführung der korrespondierenden Webservice-Methode durch eine Antwort(„Response“) beantwortet. Die Serialiesierung und Deserialisierung übernimmt fortan ein SOAP-Framework. In Abbildung 8 sieht man eine Übersicht über die Methodenklassen wie sie in der bisherigen Java-basierten Implementierung der POS-Schnittstelle zu finden sind. Sie erben alle von einer abstrakten Basis-Klasse „BasicMethod“. Die Auswahl an Methodenklassen stellt einen repräsentativen Ausschnitt aus der POS-Schnittstelle dar, um den aus Abbildung 5 dargestellten Use Case der "selfcheckout"-App vollständig abwickeln zu können. Die Methoden stammen alle aus der Main-Synchronisationsgruppe mit Ausnahme von CancelMethod, PosStatusChangeMethod und CancelPosEventMethod die jeweils Vertreter der anderen Synchronisationsgruppen sind. Bei näherer Betrachtung der GetGlobalProperties-Methoden-Klasse im Klassendiagramm in Abbildung 7 kann man die Methoden zur Serialisierung („append-/write-Methoden“) als auch die zur Deserialisierung („read-Methoden“) erkennen. Der De- oder Serialisierungsvorgang einer MethodenKlasse traversiert bei Anwendung baumartig durch ihre enthaltenen verschachtelten komplexen Datentypen. Bei der Realisierung müssen also auch diese Verschachtelungen der Datentypen in eine SOAP konforme Darstellung gebracht werden.

Eine alternativer Systementwurf könnte statt der Verwendung von JAX-WS und der Einführung von Methoden, die als Webmethoden annotiert sind, sämtliche Klassen die über Serialisierungs- und Deserialisierungs-Methoden verfügen um weitere Methoden zur Serialisierung und Deserialisierung von SOAP-Envelopes erweitern. Bei diesem alternativen Systementwurf würde man auf ein SOAP-Framework komplett verzichten und alles selber implementieren müssen. Über die Erzeugung und Veröffentlichung einer WSDL-Datei auf dem Server müsste dann auch extra gesorgt werden. Der Verzicht auf ein Framework erschwert es dann auch SOAP konform zu bleiben, weil bei der Implementierung dann genau darauf geachtet werden muss alle Einzelheiten der SOAP-Spezifikation einzuhalten. Der Vorteil wäre dann, dass man ein einheitliches Vorgehensweise für alle vom Codegenerator unterstützten Programmiersprachen hätte.

Um sprichwörtlich „Das Rad nicht neu erfinden zu müssen“ sollen SOAP-Frameworks genutzt werden die auf den einzelnen Plattformen und Programmiersprachen zur Verfügung stehen.

Im Falle von Android soll das wie in 5.2 die leichtgewichtige SOAP-Bibliothek ksoap2 für Android sein.

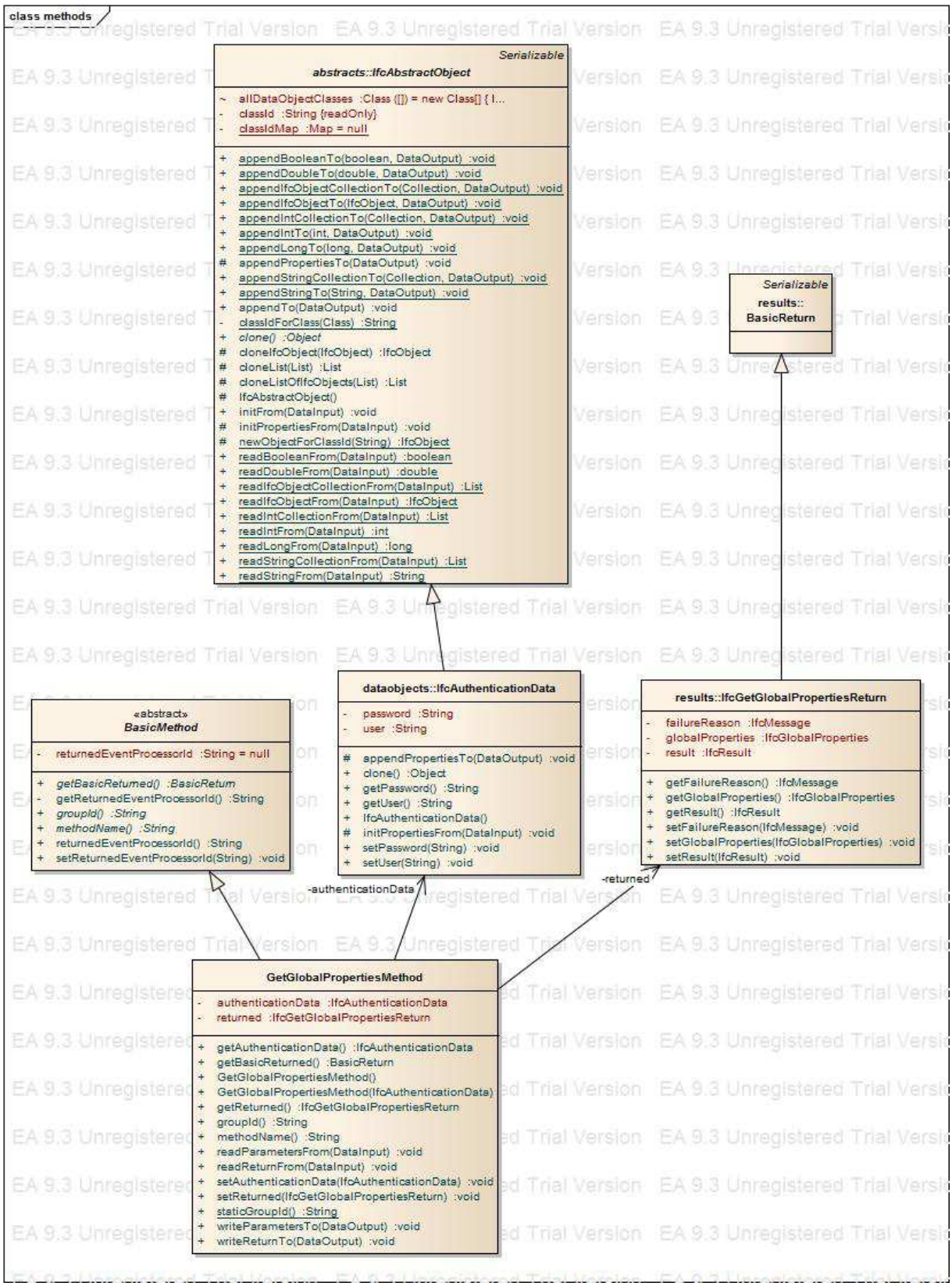

Abbildung 7 Klassendiagramm zu GetGlobalPropertiesMethod

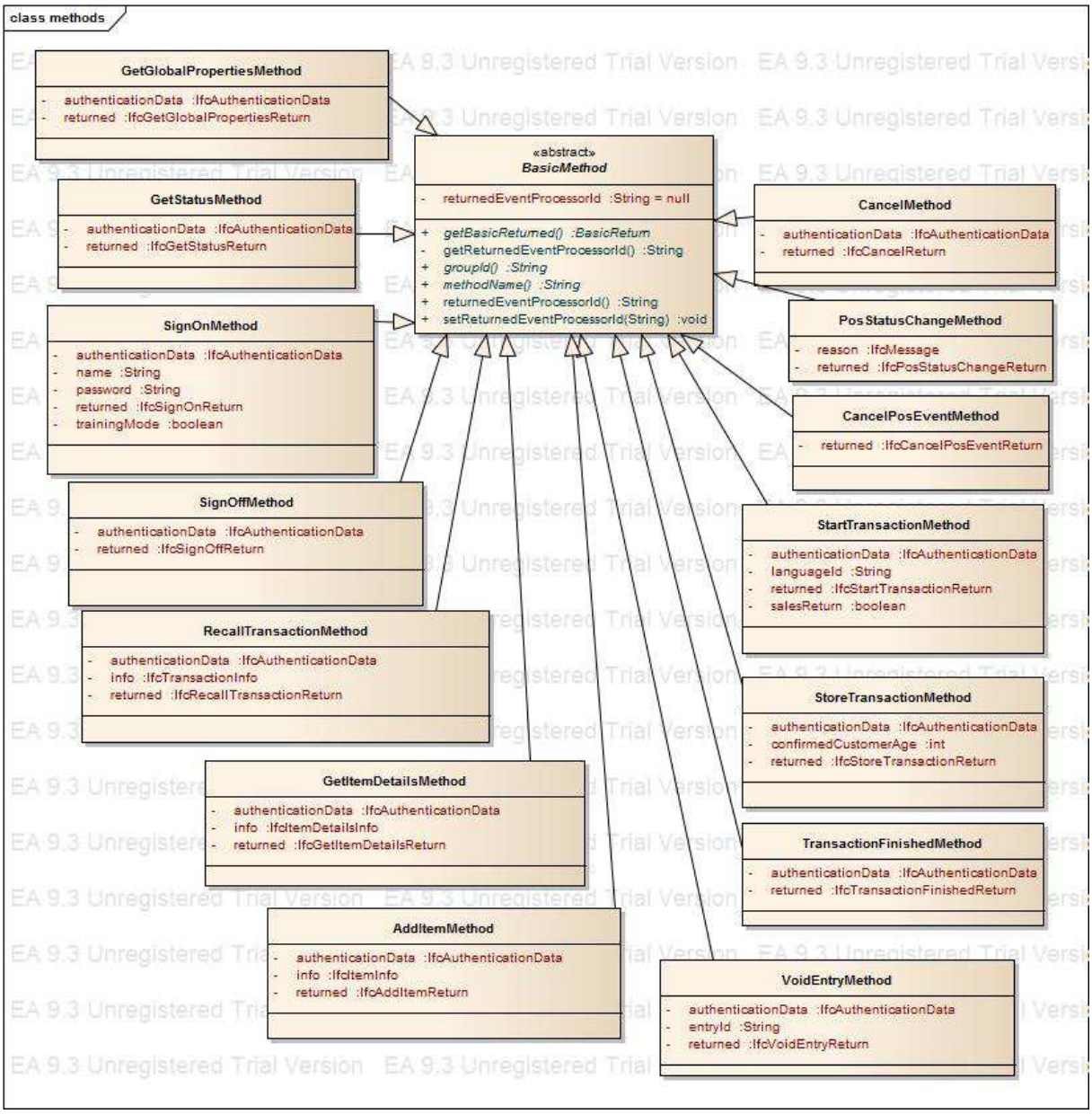

Abbildung 8 Übersicht Methodenklassen (eine Auswahl)

7 Realisierung

7.1 Realisierung der POS-Schnittstelle als Webservice

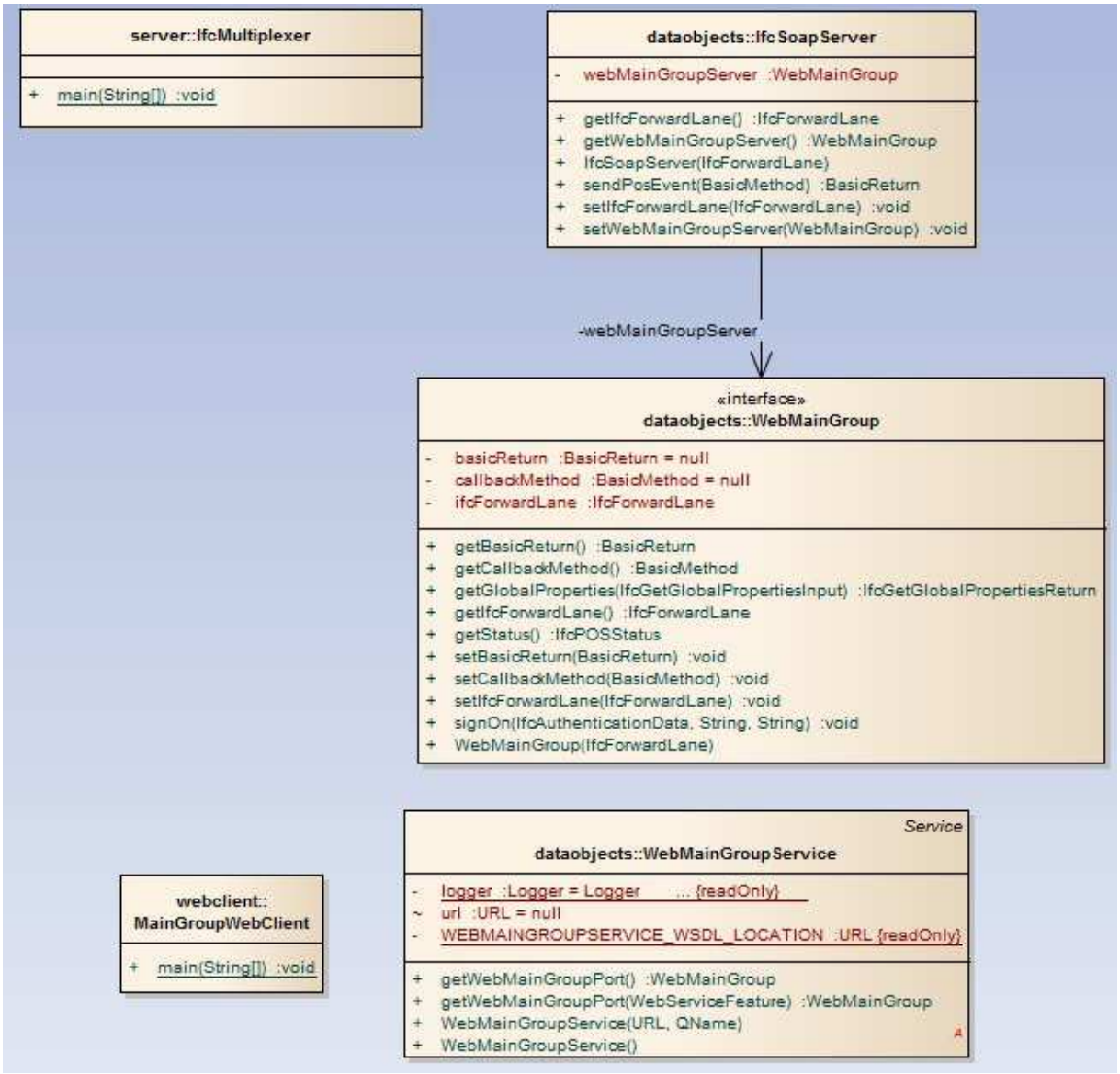

Abbildung 9 Klassendiagramm zum WebMainGroup-Webservice

Zur Erstellung eines Webservice-Endpunktes, also des Servers, der den Webservice für einen Client bereitstellt, braucht man in Java mit JAX-WS neben den Datentyp-Klassen für Argumente und Rückgabewerte der Webservice-Methoden in der Regel drei Klassen.

Die erste Klasse definiert den Webservice-Endpunkt durch ein Interface(„WebMainGroup“ vgl. Abbildung 9). Eine weitere Klasse implementiert genau dieses Interface(„IfcSoapServer“ vgl. Abbildung 9). Die Implementierenden Klassen könnten auch durch bereits vorhandene Klassen, wie POJOs („plain old java objects“) oder andere Klassen mit Methoden, die eine Art Schnittstelle darstellen, realisiert werden. Dies wird in JAX-WS über Annotationen realisiert.

Über sie können vorhandene Methoden als Webmethoden gekennzeichnet werden. In unseren Fall wären das die Methoden aus der POS-Schnittstellen-Implementierung. Diese stellen die Methoden in der Implementierung allerdings nur als Datentyp-Klasse dar. Es existiert also keine Klasse, die die POS-Schnittstellen-Methoden selber als Methoden im eigentlichen Sinne einer Methode hat. Aus diesem Grund kann keine vorhandene Klasse zur Implementierung des Webservice-Endpunkt-Interface benutzt werden. Sie wird extra geschrieben und bekommt den Namen „WebMainGroup“ (vgl. Abbildung 9). Sie enthält Methodenrümpfe, wie sie für die MainGroup-Synchronisationsgruppe in der POS-Schnittstellen-Beschreibung definiert sind. Am Ende wird für jede Synchronisationsgruppe ein Webservice-Endpunkt erstellt, um die Synchronität der Methodenaufrufe für die von einander unabhängigen Synchronisationsgruppen zu gewährleisten. Dies ist eine Anforderung, die für die POS-Schnittstelle im Allgemeinen gefordert wird und somit auch von der Webservice-Version umgesetzt werden soll. Jeder Webservice-Endpunkt enthält also der Synchronisationsgruppe entsprechende Methoden, die als Webmethoden gekennzeichnet sind und somit ein dazugehöriges Interface implementieren. Die dritte Klasse, die zur Erstellung eines Webservice-Endpunktes benötigt wird, ist die, die den Webservice-Endpunkt instanziert und veröffentlicht (IfcMultiplexer).

IfcSoapServer wurde wie folgt implementiert, wobei die Annotationen zur Kennzeichnung der Webmehoden von besonderer Bedeutung sind. Durch Sie hat das SOAP-Framework alle nötigen Informationen um den Webservice zu publizieren. Hier das Interface WebMainGroup:

```
@WebService(name = "WebMainGroup", targetNamespace =
"http://dataobjects.ifccommon_2_0.iscan.retail.wn.com/")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface WebMainGroup
{


    @WebMethod
    @WebResult(partName = "GetGlobalPropertiesReturn")
    public IfcGetGlobalPropertiesReturn getGlobalProperties(
        @WebParam(name = "IfcGetGlobalPropertiesInput") IfcGetGlobalPropertiesInput
ifcGetGlobalPropertiesInput);

 …
}
```

Callbacks wurden durch zwei Hilfs-Webmethoden umgesetzt hier für die PosEvent-Synchronisationgruppe

```
        // Hilfs Webmethoden ermöglichen Callbacks!!
        @WebMethod
        @WebResult(partName = "return")
        public BasicMethod callPosEventGroupBack();

        @WebMethod
        @WebResult(partName = "return")
        public IfcResult sendPosEventResponseBack(BasicReturn basicReturn);
```

„Der Solicit-Response Interaktionstyp wird von JAX-WS nicht unterstützt.“([] S.21)

Aus Diesem Grund mussten Callbacks asynchron gestaltet werden.

In der Implementierenden Klasse sieht der Code wie folgt aus:

```
@WebMethod
    @WebResult(partName = "GetGlobalPropertiesReturn")
    public IfcGetGlobalPropertiesReturn getGlobalProperties(
        @WebParam(name = "IfcGetGlobalPropertiesInput") IfcGetGlobalPropertiesInput
ifcGetGlobalPropertiesInput){

       IfcGetGlobalPropertiesReturn IfcGetGlobalPropertiesReturn =

ifcForwardLane.sendGetGlobalProperties(ifcGetGlobalPropertiesInput);
        return IfcGetGlobalPropertiesReturn;
    }

    /*
       @WebMethod
       public BasicMethod callPosEventGroupBack(){
              long startTime = new Date().getTime();
              BasicMethod basicMethod = callback(startTime);
              callbackMethod=null;
              return basicMethod;
       }


       //notification callable..ueber Setter
       @WebMethod(exclude=true)
       public  BasicMethod callback(long startTime){

              /*
              //for testing
              //try {
              //     Thread.sleep(51);
                     //Thread.sleep(30);
              //} catch (InterruptedException e) {

              //     e.printStackTrace();
              //}


              Date timestamp = new Date();
              long runningTime = new Date().getTime()-startTime;
              System.out.println(runningTime);
              CallPosEventLaterMethod callPosEventLaterMethod = new
CallPosEventLaterMethod();

              while(runningTime<60){
                     System.out.println(runningTime);
                     if (runningTime >= 50 && runningTime < 60){
                            System.out.println("@"+runningTime+"  return
callPosEventLaterMethod");
                            return callPosEventLaterMethod;
                     }
                     if (callbackMethod!=null && runningTime < 50){
                            System.out.println("@"+runningTime+"  return
callbackMethod: "+callbackMethod.methodName());
                            return callbackMethod;
                     }
                     runningTime = new Date().getTime()-startTime;
              }

              return callPosEventLaterMethod;//it is not allowed to return null
       }
       @WebMethod
```

```
       public IfcResult sendPosEventResponseBack(BasicReturn basicReturn){

              //TODO notify about basicReturn
              this.basicReturn = basicReturn;
              IfcResult result = new IfcResult();
              result.setId(result.IFC_OK);
              return result;
       }
    */
```

ifcForwardLane.sendGetGlobalProperties(ifcGetGlobalPropertiesInput); ermöglicht denMethodenaufruf an das POS durch ifcForwardLane weiter zu leiten und liefert den entsprechenden Returnwert.

Die Datentyp Classen wurden mit JAX-B Annotationen versehen damit sie für den Webservice entsprechend geparst werden können. Hier ein Beispiel (IfcAuthenticationData):

…

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "AuthenticationData", propOrder = { "user", "password", })
//@XmlRootElement
//this class represent a top-level XML node(extends IfcAbstractObject)
//but Inheritance of IfcAbstractObject is not needed in wsdl representation
public class IfcAuthenticationData extends IfcAbstractObject
{

    private String user;
    private String password;

    public IfcAuthenticationData()
    {
    }
…..

}
```

Für sein Elter werden die Annotationen so gesetzt:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "AbstractObject")
@XmlSeeAlso({ IfcCustomer.class, IfcDepartmentDescriptor.class,
IfcGlobalProperties.class, IfcItemEntry.class,
        IfcMessage.class, IfcPOSStatus.class, IfcPriceReduction.class,
IfcPriceReductionEntry.class, IfcResult.class,
        IfcSubtotalEntry.class, IfcTaxEntry.class, IfcTenderDescriptor.class,
IfcTenderEntry.class, IfcTextEntry.class,
        IfcTotalOverride.class, IfcTransaction.class })
@XmlTransient //hide AbstractObject in xsd
public abstract class IfcAbstractObject implements IfcObject, Serializable
{

    @XmlTransient //hide classId in wsdl for all child classes
    private final String classId;

    protected IfcAbstractObject()
    {
        classId = classIdForClass(this.getClass());
    }

    //JAXB can not handle Interfaces like Map/ @XmlElement(nillable = true)
    static private Map classIdMap = null;
```

```
    static
    {
        classIdMap = new HashMap();

        Class[] allDataObjectClasses =
                new Class[] { …
```

…

In @XmlSeeAlso müssen alle Klassen aufgelistet werden die IfcAbstractObject erweitern. @XmlTransient ermöglicht das von der dieser Basis-Klasse in der WSDL nichts zu sehen ist.

7.2 Realisierung des WS-Client als Android App für „mobile shopping“

beschreiben der Lösung

KSOAP2 scheint ein gutes Framework, um Webservice unter Android ansprechen zu können, zu sein. Leider stellte sich während der Erstellung der Arbeit heraus, dass es nur schlecht dokumentiert ist. Die Arbeitsergebnisse zu der Umsetzung der Webservice-Klients sollen hier erläutert werden.

Zu aller erst war ist es notwendig die ksoap2 Bibliothek herunter zu laden z.B. direkt aus dem Subversion Repository mit dem Konsolenbefehl wget.

```
wget http://ksoap2-
android.googlecode.com/svn/m2repo/com/google/code/ksoap2-android/ksoap2-
android-assembly/2.6.5/ksoap2-android-assembly-2.6.5-jar-with-
dependencies.jar
```

Danach wird die Bibliothek ins Eclipse Projekt der Android-App eingebunden.

Die Realisierung lässt sich durch die Codelistings von IscanDroid.java und IfcAuthenticationData.java IfcGetGlobalPropertiesInput.java, als auch das von BasicInput.java nachvollziehen. Dabei war wichtig gewesen das die Daten-Typen, für das erzeugen eines envelope, das Interface knmSerializable implementieren. (`org.ksoap2.serialization.KvmSerializable`)

7.3 Codegenerierung

7.3.1 Erzeugen des Codes für den Webservice

Das entworfene Template zur erzeugung aller Datentypen mit den richtigen Annotationen befindet sich im Anhang als Listing(DataObjects.itf). Für die Wrapper-Klassen der Inputparameter wurde das Params Template erstellt. Ein generatives Themplate für die Webservice-Klassen wurde noch nicht erstellt. Hier müssten für die Syncronisationsgruppen die entsprechende Webservice Interfaces und deren implementierungen erzeugt werden in denen dann die Webservicemethoden nach dem oben beschriebenen Prinzip aufgelistet sind.

7.3.2 Erzeugen des Codes für den AndroidClient

Ein generatives Themplate für die Android-Klassen wurde leider noch nicht erstellt.

8 Test

Zum Testen wurde das Programm SoapUI (siehe [23])verwendet. Mit dem Programm ist es möglich den Webservice gezielt zu testen. Als Eingang bekommt es die WSDL-Datel die der entwickelte Webservice-Implementierung zur Laufzeit veröffentlicht. Das Programm analysiert die WSDL und unterstützt den Entwickler bzw. Tester einen Envelope für eine Webmethode zu erstellen und abzusenden. Eine entsprechende Respone wird dann empfangen und dargestellt. Automatisierte Tests werden dabei auch unterstützt.

9 Zusammenfassung und Ausblick

Zusammenfassend kann man sagen, dass die Komplexität des Themas für jemanden der sich zuvor nicht mit Webservices auseinander gesetzt hat sehr hoch ist. Hinzu kommt das das Umfeld in das sich die Arbeit bewegt ebenfalls eine hohe Komplexität mitbringt (POS-Schnittstelle und Selfcheckout) Auch bei Wincor Nixdorf wurde der Aufwand für Einarbeitung und Umsetzung der Lösung für das Thema unterschätzt. Der Autor bedauert sehr, dass es ihm nicht möglich war, mehr auf die einzelnen Teilbereiche tiefer einzugehen. Wenn noch mehr Zeit und sich im Team mit dem Thema beschäftigt wird ist es jedoch möglich die Visionen von Wincor Nixdorf vollständig umzusetzen. Das Thema bietet genügend Inhalte um als Gruppenarbeit abgehandelt zu werden. Diese Entscheidung hätte sicherlich zu einem besseren Ergebnis mit mehr Tiefgang geführt.

Überschattet wurde diese Arbeit zudem durch tief greifende emotionale Ereignisse im privaten Umfeld die hier nicht weiter ausgeführt werden.

Zur der erwarteten Performance können leider keine adäquaten Aussagen getroffen werden. Dies muss noch einmal gesondert betrachtet werden. Für den Fall das die Performance ein Problem darstellen sollte, könnten noch andere RPC ähnliche nicht so populäre Verfahren wie JSON-RPC (siehe [24],[25]) untersucht werden. Auch generelle Veränderungen an der POS Schnittstelle sollten in hinblick auf Performance vielleicht verändert werden. Beispielsweise wird der PO-Sstatus bei fast jeden Aufruf in der Antwort (Return) geschickt. Man könnte hierfür eine Kosten-Nutzen-Rechnung dafür machen und den POS-Status nur an wirklich brisanten Stellen aktualisieren. Man könnte auch zukünftig mit einer Art selbst verwalteten Zustandsmodell im Client arbeiten. Wenn Auf einen Return im Allgemeinen verzichtet werden könnte ließe sich auch schon viel besser über eine Umsetzung mit REST nachdenken.

10 Verzeichnisse

10.1 Programmlistings

WebMainGroup.java:

```
package wn.retail.mobile.server;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.xml.bind.annotation.XmlSeeAlso;

import com.wn.retail.iscan.ifccommon_2_0.dataobjects.IfcAuthenticationData;
import com.wn.retail.iscan.ifccommon_2_0.dataobjects.IfcPOSStatus;
import com.wn.retail.iscan.ifccommon_2_0.dataobjects.ObjectFactory;
import com.wn.retail.iscan.ifccommon_2_0.params.IfcGetGlobalPropertiesInput;
import com.wn.retail.iscan.ifccommon_2_0.results.alt.IfcGetGlobalPropertiesReturn;

@WebService(name = "WebMainGroup", targetNamespace =
"http://dataobjects.ifccommon_2_0.iscan.retail.wn.com/")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface WebMainGroup
{


    @WebMethod
    @WebResult(partName = "GetGlobalPropertiesReturn")
    public IfcGetGlobalPropertiesReturn getGlobalProperties(
        @WebParam(name = "IfcGetGlobalPropertiesInput") IfcGetGlobalPropertiesInput
ifcGetGlobalPropertiesInput);


  // die restlichen webmethoden.. (wegen der Übersichtlichkeit hier entfernt)



    /*
        //TODO move to PosEventGroup
         // Hilfs Webmethoden ermöglichen Callbacks!!
        @WebMethod
        @WebResult(partName = "return")
        public BasicMethod callPosEventGroupBack();

        //TODO move to PosEventGroup
        @WebMethod
        @WebResult(partName = "return")
        public IfcResult sendPosEventResponseBack(BasicReturn basicReturn);
        */
}
```

IfcAuthenticationData (nicht für android):

```
/*
 * File: IfcAuthenticationData.java
 *
 * Interface version 2.0; generated 06/10/31 10:19:57
 *
 * Copyright (c) 2004, 2005 Wincor Nixdorf International GmbH,
 * Heinz-Nixdorf-Ring 1, 33106 Paderborn, Germany
 * All Rights Reserved.
 *
 * This software is the confidential and proprietary information
 * of Wincor Nixdorf ("Confidential Information"). You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered
 * into with Wincor Nixdorf.
 */

package com.wn.retail.iscan.ifccommon_2_0.dataobjects;

import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

import com.wn.retail.iscan.ifcbase.methods.ProtocolException;
import
com.wn.retail.iscan.ifccommon_2_0.abstracts.IfcAbstractObject;

/**
 * Java version of the iSCAN interface data type AuthenticationData.
 *
 * THIS FILE IS GENERATED. DO NOT CHANGE IT MANUALLY!
 *
 * @author Sï¿½nke K., Wincor Nixdorf International GmbH
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "AuthenticationData", propOrder = { "user",
"password", })
//@XmlRootElement
//this class represent a top-level XML node(extends
IfcAbstractObject)
//but Inheritance of IfcAbstractObject is not needed in wsdl
representation
public class IfcAuthenticationData extends IfcAbstractObject
{

    private String user;
    private String password;

    public IfcAuthenticationData()
    {
    }
```

```
    public String getUser()
    {
        return user;
    }

    public void setUser(String user)
    {
        this.user = user;
    }

    public String getPassword()
    {
        return password;
    }

    public void setPassword(String password)
    {
        this.password = password;
    }

    protected void appendPropertiesTo(DataOutput output)
        throws IOException
    {
        super.appendPropertiesTo(output);
        appendStringTo(getUser(), output);
        appendStringTo(getPassword(), output);
    }

    protected void initPropertiesFrom(DataInput input)
        throws IOException,
        ProtocolException
    {
        super.initPropertiesFrom(input);
        setUser(readStringFrom(input));
        setPassword(readStringFrom(input));
    }

    public Object clone()
    {
        IfcAuthenticationData clone = new IfcAuthenticationData();
        clone.setUser(getUser());
        clone.setPassword(getPassword());
        return clone;
    }
}
```

10.2 Literaturverzeichnis

[1]

[2] http://en.wikipedia.org/wiki/Point_of_sale

[3] http://de.wikipedia.org/wiki/Verkaufsort

[4] http://www.sparxsystems.de/uml/ea-function/

[5] http://de.wikipedia.org/wiki/Android_(Betriebssystem)

[6] http://www.openhandsetalliance.com/oha_faq.html

[7] Android 2 Grundlagen und Programmierung,2.Auflage unter Mitarbeit von David Müller, Arno Becker und Marcus Pant, dpunkt.verlag GmbH, ISBN 978-3-89864-677-2

[8] http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels

[9] http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#uses

[10] http://developer.android.com/guide/topics/manifest/manifest-intro.html

[11] http://developer.android.com/about/dashboards/index.html#Platform (Current Distribution)

[12] http://jax-ws.java.net/

[13] http://metro.java.net/

[14] http://jax-ws.java.net/2.2.7/docs/ch02.html#running-on-top-of-jdk-6

[15] http://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html

[16] http://axis.apache.org/axis/

[17] TPiSCAN_Interface_2.0.pdf (Ausdruck im Anhang)

[18] Java Webservices in der Praxis Realisierung einer SOA mit WSIT, Metro und Policies,Oliver Heuser und Andreas Holubek, dpunkt.verlag, ISBN 978-3-89864-596-6

[19] http://www.w3.org/TR/soap/

[20] http://www.w3.org/TR/wsdl#_introduction

[21] http://www.w3.org/TR/#tr_WSDL

[22] http://dbis.eprints.uni-ulm.de/458/1/Klei08.pdf

[23] http://www.soapui.org/About-SoapUI/what-is-soapui.html

[24] http://json-rpc.org/

[25] http://code.google.com/p/android-json-rpc/

10.3 Abbildungsverzeichnis