hochschule für angewandte wissenschaften
FACHBEREICH ELEKTROTECHNIK UND INFORMATIK **hamburg**
university of applied sciences

Diplomarbeit

Nils Dröge

Realisierung einer handybasierten Kontoabfrage

Studiengang Technische Informatik
Betreuender Prüfer: Prof. Dr. Kai von Luck
Zweitgutachter: Prof. Dr. Gunter Klemke
Abgegeben am 31. Oktober 2002

BERLINER TOR 3 · D - 20099 HAMBURG

Realisierung einer handybasierten Kontoabfrage

Stichworte HBCI, Mobile Kontoauskunft, Mobiltelefon, Java 2 Micro Edition, Kryptographie

Zusammenfassung

Ziel dieser Arbeit ist zu überprüfen, in wie weit sich kleine mobile Geräte, insbesondere Handys, schon heute für Bankanwendungen benutzen lassen. Die entwickelte Software benutzt dabei den in Deutschland etablierten Homebanking Standard HBCI. Die Schwierigkeit war dabei, die umfangreiche Spezifikation auf den schmalen Ressourcen eines Handys umzusetzen. Mit der neuen Generation von Handys mit Java-Unterstützung ergibt sich eine breite Hardware-Plattform. Ich zeige in der Arbeit, dass es für einen Teil der Geräte heute schon möglich wäre, die Bankgeschäfte, wie Kontoabfrage und Überweisungen, durchzuführen. Der Durchbruch wird aber erst für die nächste Handygeneration zu erwarten sein, wenn die beiden Standards besser zusammenarbeiten. Die betrachtete Bankenanbindung lässt sich aber auch als eine spezielle Business-Anwendung sehen. Die Arbeit zeigt deshalb auch, dass man leistungsfähige Anwendungen mit verschlüsselter Kommunikation schon heute umsetzen kann und welche Software sich dafür eignet bzw. welche Module/Pakete man dafür benutzen kann.

Realization of a handy based account inquiry

Keywords HBCI, Mobile Banking, Handy, Java 2 Micro Edition, Cryptographic

Abstract

A goal of this work is to be examined, into as far small mobile devices, in particular Handys, already today for applications of banks be used leaves itself. The developed software used thereby the Homebanking standard HBCI established in Germany. The difficulty was about to convert the extensive specification on narrow resources of a Handys. With the new generation of Handys with Java support a broad hardware platform results. I show the banking transactions in the work that it is already possible for a part of the devices today, as account inquiry and transfers to accomplish. The break-through will however only have to be expected for the next Handygeneration, if the two standards work better together. The regarded bank binding can be seen in addition, as a special Buisiness application, the work shows therefore also that one can convert efficient applications with coded communication already today and which software is suitable for it and/or one can use which module/packages for it.

Danksagung

Am Ende einer solchen Arbeit weiß man gar nicht wo man Anfangen soll. Dieses Dokument stellt nicht nur den Abschluß meiner Diplomarbeit dar, sondern auch des Studiums, deshalb möchte ich allen, die mich dabei Unterstützt haben danken, für die Mühen, Sorgen und Geduld, die Ihr mit mir geteilt habt. Den zahlreichen Entwicklern, welche die freie Software zur Erstellung dieser Arbeit entwickelt haben, möchte ich an dieser Stelle meinen Dank aussprechen. Vielen Dank auch an meinen betreuenden Professor Kai von Luck, für seine Zeit und die Fähigkeit, Hinweise in geschickt formulierte Fragen zu verstecken. Mein größter Dank gilt meiner Familie und vor allem meiner Frau Andrea.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

1.1 Motivation und Beschreibung

Homebanking gab es schon in den Tagen von Btx. Die Vorteile sind schnell gefunden: Unabhängigkeit von den Öffnungszeiten der Filialen, Kontrolle über die Bankgeschäfte. Doch es gibt auch Nachteile: bis heute ist es fast ausschließlich nur über den PC möglich, seine Bankgeschäfte zu erledigen. Für die meisten Privatkunden stellt dies keine besondere Erleichterung dar. Das Starten eines PC liegt wegen des flüchtigen Speichers im Minutenbereich, zu viel, wenn man nur mal eben seinen Kontostand abfragen möchte. Dazu kommt die mangelnde Mobilität, komplizierte Einrichtung, fehlende Treiber für eventuelle Kartenlesegeräte oder die Unsicherheit eines PC, der am Netz hängt.

Die Unzufriedenheit über die PC-Lösung führte zu der Überlegung, ob es nicht einen anderen Weg geben könnte. Die Anforderungen an das Homebankinggerät sind:

- Es muss einfach zu bedienen sein
- Die Bankanwendung und somit auch das Gerät sollen schnell verfügbar sein, mit schneller Einschaltzeit von wenigen Sekunden
- Das Gerät muss entweder günstig, oder sowieso im Haushalt vorhanden sein
- Es muss über einen Internetzugang verfügen
- Es muss technisch in der Lage sein, eine „beliebige“ Anwendungen auszuführen

Aus diesem Katalog ergeben sich folgende potenzielle Gerätegruppen: Web-Tablett, Settop-Box, PDA und Handy.

Ungeachtet der theoretischen Pläne gibt es leider nur sehr wenige Produkte. Die Möglichkeit per Browser bieten mittlerweile viele Banken an. Zudem kann man seinen Kontostand per SMS und PIN abrufen, und es gibt einen Client für den Palm. Die Lösungen erscheinen zum Teil noch sehr unkomfortabel.

Größere Unabhängigkeit dürfte von einem Standard zu erwarten sein, der generell von den Banken unterstützt wird. In Deutschland kommt dafür nur der HBCI-Standard in Frage. Trotz gegenteiliger Nachrichten wird HBCI von vielen, insbesondere von den großen Banken, für das Homebanking angeboten.

Die oben genannten Geräte sind sehr unterschiedlich. Dennoch bieten alle Geräte-Gruppen Unterstützung für den üblichen Verdächtigen an, wenn es um plattformübergreifende Programmierung geht: Java. Das Handy stellt dabei das Gerät mit den geringsten Ressourcen dar und bildet den kleinsten gemeinsamen Nenner. Seit über einem Jahr gibt es mit Java 2 Micro Edition (J2ME) eine Java Variante, die speziell auch auf Handys ausgerichtet ist.

Eines der ersten Handy mit der Unterstützung für J2ME ist das Siemens SL45i, es hat zudem auch einen SD-Card-Slot. Auch viele PDAs und Webtabletts haben Erweiterungskarten. Deshalb war der erste Ansatz, diese Erweiterungskarten als Sicherheitsmedium zu benutzten, wie die HBCI auf dem PC die Smartcart zur Signierung und Verschlüsselung benutzt. Leider stellte sich schnell heraus, dass die Spezifikation von J2ME bzw. die untersuchte Unterform CLDC 1.0 mit MIDP 1.0 keine Schnittstelle zu den Erweiterungskarten hat.

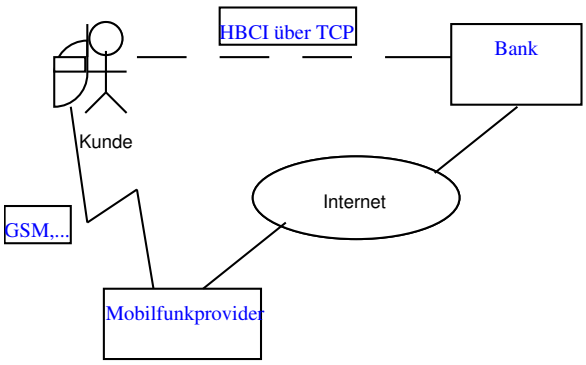

Abbildung 1.1: Kommunikation zwischen dem Kunden mit einem Handy und der Bank

Da auch die Verschlüsselung, welche sehr rechenintensiv ist, nicht nativ unterstützt wird, hätte man zu diesem Zeitpunkt das Projekt aus ökonomischer Sicht beenden müssen, um nach Alternativen zu suchen. Eine wird in den Kapiteln 2.5 und 4.2 angesprochen.

Es ist trotz dieser fehlenden Parameter möglich eine HBCI-Anwendung für ein Java-Handy zu schreiben, allerdings mit weiteren Einschränkungen. Diese Arbeit soll nicht nur die Grenzen von den aktuellen Handys zeigen, sondern im allgemeinen die Möglichkeiten einer kryptographischen Anwendung zur J2ME. Ein Blick in die nächste Zukunft lößt fast alle heutigen Einschränkungen auf, denn die Handys werden immer leistungsfähiger und die Nachfolger der behandelten Standards stehen schon in den Startlöchern.

„Mobile Banking“ wird als Schlüsseltechnologie für den Erfolg des mCommerce angesehen [46]. Auch die Bescheibung von J2ME sieht neben dem Entertainment den mCommerce als eines der Ziele [5]. In der Spezifikation von HBCI steht seit jeher Mobilität, bzw. Handys – wenn auch nur als Transportmedium – drinnen. Jetzt müssen diese nur weiter zusammenwachsen. Schon heute sind mobile Bankanwendungen möglich.

Nach Einschätzung von Gardner [1] wird Java den Handymarkt dominieren: „40% of PDAs and 68% of mobile phones will be Java™technology-enabled by 2006!“

1.2 Überblick über das Dokument

In folgenden Kapiteln werde ich zeigen, dass es schon heute möglich wäre, Homebanking auf einem Handy über HBCI zu benutzen. Außerdem, wie Anwendungen mit vertraulichen Informationen auf einem javafähigen Handy realisiert werden können und welche Grenzen dabei entstehen.

Dazu werde ich im Kapitel 2 Vorbereitung die notwendigen Grundlagen für die Realisierung der Anwendung erläutern und im Kapitel 3 Design und Realisierung beschreiben, wie ich vorgegangen bin und was ich dabei berücksichtigt habe. Im letzten Kapitel 4 Nachbereitung werde ich meine Arbeit zusammenfassen und kritisch beurteilen.

Ich habe die Kapitelüberschriften bewußt so benannt, um den prozesshaften Charakter hervorzuheben, nach dem ich vorgegangen bin.

Im Kapitel 2 werde ich als erstes kryptographische Verfahren beschreiben, da diese vom

deutschen Bankenstandard zur Sicherung der übertragenen Daten benutzt werden. Ich habe mich auf die für HBCI relevanten Algorithmen und die zum Verständnis dafür notwendigen Verfahren beschränkt. Danach erläutere ich den von HBCI benutzten Transport sowie den Nachrichtenaufbau, und wie die Sicherheitstechniken eingesetzt werden. Ich werde Besonderheiten der Java 2 Micro Edition beschreiben und mich kritisch zu den vorhandenen Standards äußern, bevor ich am Ende des Kapitels die benutzte Test- und Anwendungsumgebung beschreibe.

Im Abschnitt 3 zeige ich, wie man mit modernen Entwicklungstechniken effizient stabile Anwendungen schreiben kann. Dazu beschreibe ich zuerst, welche Entwicklungsmethode ich benutzt habe. Bevor ich dies an einem Beispiel darstelle, gebe ich einen Überblick über die Architektur. Abschließend zeige ich einige Realisierungsbeispiele und schließe das Kapitel mit einem Rückblick ab.

Das letzte Kapitel 4 beginnt mit einer Zusammenfassung. Danach werte ich das Ergebnis aus und gebe einen Ausblick auf die möglichen Erweiterungen und die Entwicklung der zugrunde liegenden Technik.

1.3 Begriffe

Zum besseren Verständnis möchte ich vorab die Equivalenzen einiger Begriffe darstellen. Die Begriffe werden im nächsten Kapitel erläutert und im Abkürzungsverzeichnis können die verwendeten Kurzformen nachgeschlagen werden.

- Mit J2ME, Java2ME, Micro Edition, oder ähnliches ist immer Java2 Micro Edition Version 1.4 mit CLDC Version 1.0 und MIDP Version 1.0 gemeint.
- Mit HBCI ist immer die bei Erstellung des Dokumentes gültige Version 2.2 gemeint.
- HBCI-Server, Institutsrechner, Bankenrechner oder ähnliches stellen die Server-Seite bei HBCI dar. Mit Test-Server ist im allgemeinen der HBCI-Server mit der Adresse `www.hbci-kernel.de` auf Port 3000 gemeint.
- Clientsoftware, HBCI-Client, Kundensoftware, o.ä. ist die Client-Software auf Kundenseite.

2 Vorbereitung

Der Umgang mit Geld ist ein sensibles Thema. Der Benutzer von Bankanwendungen möchte seine Daten vor Dritten schützen. Auch die Bank hat Interesse an der Vertraulichkeit. Zudem soll sichergestellt werden, dass die getätigten Aufträge von einer autorisierten Person vorgenommen wurden. Diese Ziele werden durch folgende Techniken erreicht:

- Vertraulichkeit -> Verschlüsselung
- Integrität -> Hash
- Authentizität -> Signatur

Im nächsten Abschnitt werden diese Techniken erklärt und Schwierigkeiten, bzw. Sicherheitslücken diskutiert.

2.1 Sicherheitstechniken

In der Vergangenheit hatten Staaten und vornehmlich militärische Stellen ein Interesse daran, Botschaften, die abgefangen werden konnten, für Unberechtigte unleserlich zu gestaltet, aber trotzdem den Inhalt zu erhalten. Neben der Verschleierung der Nachricht ist es auch wichtig die Identität des Versenders sicherzustellen bzw. das der Inhalt nicht verändert wurde. Im Mittelalter wurden dafür Siegel benutzt. Die Wissenschaft, die sich damit beschäftigt, nennt man Kryptographie. Nachfolgend werden nur die Verfahren und Algorithmen beschrieben, die von HBCI benutzt werden oder zum Verständnis dieser beitragen. Als Ergänzung und zur Vertiefung in diese Materie sei auf die einschlägige Literatur verwiesen.

Eine mehr formale Beschreibung, aber mit vielen Beispielen und Hintergründen, liefert das Script von Völler [44]. Unterhaltsamer, aber umfassend ist das Buch von Schmeh [38]. Einige Beispiele sind aus der kurzen und verständlichen Beschreibung von Fischer u. a. [12] entnommen.

2.1.1 Begriffe

Zum Verständnis der Verfahren ist es notwendig, einige Begriffe zu erklären. Bei den kryptographischen Verfahren geht es immer um mindestens drei Parteien. Es hat sich eingebürgert, statt von Person A und Person B, die Namen Alice und Bob zu benutzen. Alice und Bob sind die „Guten“, die Nachrichten austauschen wollen. Sollen zwischen drei Personen Nachrichten ausgetauscht werden, kommt Carol hinzu. Einen „Bösen“ gibt es auch, in der Literatur wird er/sie unterschiedlich benannt, hier heißt er Paul und ist Hacker. Paul ist technisch auf dem

neuesten Stand und will die Nachrichten abfangen, manipulieren oder sonst wie mißbrauchen. In 2.2 HBCI werden Alice und Bob durch die Bank und den Benutzer ersetzt.

Wenn die Nachricht so verändert wird, um den Inhalt vor Paul zu verbergen, nennt man das verschlüsseln oder chiffrieren. Der umgekehrte Weg, die Nachricht wieder lesbar machen, heißt entschlüsseln oder dechiffrieren. Um zwischen verschlüsselter und nicht verschlüsselter Nachricht unterscheiden zu können, werden die folgenden Begriffe benutzt:

Klartext ist der unverschlüsselte Text. Ein Klartext kann auch in binärer Form sein.

Chiffretext ist der Text nach der Verschlüsselung.

Chiffre (Cipher) ist das Verfahren, der Algorithmus, der zur Verschlüsselung eingesetzt wird.

Schlüssel (Key) Bei den heute benutzten Verfahren wird ein Schlüssel als zusätzliche Information für den Text verwendet, um die Daten zu ver- oder zu entschlüsseln.

Wenn vertrauliche Daten zwischen Alice und Bob ausgetauscht werden, gibt es Anforderungen an das benutzte Verfahren:

Vertraulichkeit Es soll sichergestellt werden, dass die Nachrichten zwischen Alice und Bob nicht von einem Dritten, wie z.B. Paul gelesen werden können. Dies wird durch das Verschlüsseln der Nachricht erreicht.

Integrität Die Integrität gewährleistet, dass die Nachricht nicht verändert wurde. Dies wird z.B. durch das Verwenden eines Hashwertes erreicht, diese Herkunft sollte aber nachweisbar sein. Die Integrität wird oft zusammen mit der Authentizität erreicht.

Authentizität Die Authentizität einer Nachricht garantiert die Herkunft. Dies wird mit Hilfe der asymmetrischen Verschlüsselung erreicht.

Durch die Kombination dieser drei Punkte ist es nicht möglich, dass Paul die Kommunikation zwischen Alice und Bob abhören, manipulieren oder gefälschte Nachrichten einschleusen kann.

Angriffsmethoden Wenn man über Kryptographie redet, muss man sich auch Gedanken zur Abwehr möglicher Attacken machen. Es wird im allgemeinen zwischen den folgenden Angriffen auf die Verfahren unterschieden:

brute-force-attack Bei diesem Angriff werden alle Schlüssel durchprobiert. Dies ist die einfachste aber auch zeit- oder rechenintensivste Methode. März 1998 22.000 Systeme 39 Tage zum entschlüsseln einer mit DES (56Bit) chiffrierten Nachricht.

known-plaintext-attack Hier ist ein Teil der Nachricht oder die vollständige Nachricht bekannt. Man nutzt die Eigenheit mancher Chiffre aus, dass sie bei gleichem Schlüssel und gleichem Klartext immer den selben Chiffretext ausgeben. Bei vielen Nachrichtenformaten und auch bei HBCI ist z.B. der Nachrichtenkopf bzw. Segmentkopf immer ähnlich.

choosen-plaintext-attack Eine Variante ist der known-plaintext-attack. Hier wird der Klartext durch den Angreifer vorgegeben, um z.B. gezielt wiederkehrende Muster zu provozieren.

replay-attack Bei dieser Attacke werden abgefangene Chiffretexte zu einem späteren Zeitpunkt wieder versendet, um z.B. Überweisungen auf das Konto des Angreifers zu wiederholen.

man-in-the-middle-attack Auch bei diesem Angriff wird Paul aktiv. Den Aufbau einer verschlüsselten Verbindung von Alice fängt Paul ab und tut so, als wäre er Bob. Danach baut Paul eine verschlüsselte Verbindung zu Bob auf und behauptet, Alice zu sein. Alice und Bob fühlen sich durch die verschlüsselte Verbindung in Sicherheit, sind es aber nicht, da Paul alle Nachrichten mitlesen, verändern, oder löschen kann.

Die Qualität der Schlüsselsuche läßt sich häufig durch stochastische Verfahren, wie z.B. die Häufigkeitsanalyse erhöhen. Für mehr Informationen zu diesem Thema möchte ich auf die Literatur verweisen.

Bei den Verschlüsselungsmethoden unterscheidet man zwischen symmetrischer Verschlüsselung, auch als private key cipher gekannt und der asymmetrischen Verschlüsselung oder public key cipher.

2.1.2 Symmetrische Verschlüsselung

Bei der symmetrischen Verschlüsselung wird sowohl für die Verschlüsselung als auch für die Entschlüsselung derselbe Schlüssel benutzt. Das heißt, dass beide Parteien, also Alice und Bob, den gleichen Schlüssel haben. Wenn Alice auch Nachrichten mit Carol austauschen will, die Bob nicht lesen soll, müssen Alice und Carol einen anderen gemeinsamen, geheimen Schlüssel haben. Deshalb spricht man auch vom private key Verfahren.

Einer der ältesten bekannten Verschlüsselungen ist der Caesar-Chiffre. Weist man jedem Buchstaben im Alphabet eine Zahl zu und fängt beim Überschreiten des letzten Buchstaben wieder bei A an, dann kann man bei einem Wort jedem Buchstaben eine Zahl hinzu addieren und erhält ein neues, so wird z.B. aus Hallo bei der Addition mit 8:

Listing 2.1: Beispiel für den Caesar-Chiffre

```
Hallo -> Pittw {Key:=+8}
```

Die 8 ist hierbei der Schlüssel. Zur Entschlüsselung der Daten wird der umgekehrte Weg, die Subtraktion der einzelnen Buchstaben mit dem Schlüssel, verwendet. Kommt ein Wert außerhalb der Buchstabenmenge, in dem Fall ein negativer Wert heraus, wird dieser von Z abgezogen. Oder anders ausgedrückt: Ist das Ergebnis außerhalb der Menge der Buchstaben, wird das Ergebnis modulo der Mengengröße genommen.

Der Chiffre gehört, wie auch die beiden folgenden, zu den Substitutionsverfahren. Es wird ein Zeichen durch ein anderes ersetzt.

Eine Abwandlung ergibt sich bei variabler Addition. Es wird jedem Buchstaben ein neuer Wert hinzu addiert:

Listing 2.2: Beispiel für den modifizierten Caesar-Chiffre

```
H+1=G, a+2=c, l+3=o, l+4=p, o+5=t
Hallo -> Gcopt  {Key:=1,2,3,4,5}
```

Ist der Schlüssel kürzer als der Klartext, nimmt man wieder den ersten Teil des Schlüssels.

Eine binäre Form ist der Vernam-Chiffre. Die binäre Addition ist XOR[1]. Nimmt man den 8-Bit ASCII-Code für die Codierung des Wortes, mit 41_{16}[2] (64 Dezimal) für A und 61_{16} für das kleine a ergibt Hallo den Wert

Listing 2.3: Beispiel für den Vernam-Chiffre

```
Hallo := hex 0x48616C6C6F

     01001000 01100001 01101100 01101100 01101111
XOR  00000001 00000010 00000011 00000100 00000101
-------------------------------------------------
     01001001 01100011 01101111 01101000 01101010

Gcohj := hex 0x49636F686A

Hallo -> Gcohj  {Key:=hex 0x0102030405}
```

Man beachte, dass trotz des gleichen Schlüssels zum vorherigen Chiffre, sich die letzten beiden Zeichen unterscheiden. Es können auch Zeichen entstehen, die nicht zum Alphabet gehören, da sich die Zeichenmenge vergrößert hat. Zur Entschlüsselung wird wieder eine XOR-Verknüpfung mit dem Schlüssel benutzt. Mit dem Verman-Chiffre lässt sich nicht nur Schrift chiffrieren, sondern beliebige binäre Daten.

Es ist der einzige Chiffre der mathematisch als sicher bewiesen wurde, wenn folgende Merkmale gegeben sind: der Schlüssel muss zufällig sein und dieselbe Länge wie der Klartext haben. Der Verman-Chiffre wird deshalb auch als „One-Time-Pad“ bezeichnet.

Eine andere Methode ist die Transposition, das Vertauschen der Buchstabenpositionen. Der Schlüssel dient als Tabelle, um die neuen Positionen der Buchstaben anzugeben:

Listing 2.4: Beispiel für die Transposition

```
Hallo -> laoHl {Key:=4,2,5,1,3}
```

Für die Transposition lassen auch größere Blöcke vertauschen. Es gibt Spalten- und Zeilentranspositionen. Ist der Schlüssel größer als die Eingabemenge, spricht man von einer Expansionstransposition, da der Chiffretext gegenüber dem Klartext expandiert wird. Die Permutation ist eine Sonderform, bei der zwei Blöcke vertauscht werden. Auch dieses Verfahren lässt sich in eine binäre Form überführen.

Diese beiden Grundverfahren werden heute nicht mehr separat verwendet, da sie zu einfach zu knacken sind. Sie lassen sich aber auch kombinieren und mehrfach hintereinander ausführen. Damit haben wir die Legobausteine der modernen symmetrischen Verschlüsselungsver-

[1] Dies ist nicht ganz korrekt, XOR ist eine bitweise Addition ohne Übertrag

[2] 48_{16} heißt 48 mit der Basis 16, auch als Hexadezimale Schreibweise bekannt. In C/C++ und Java wird es als 0x48 geschrieben.

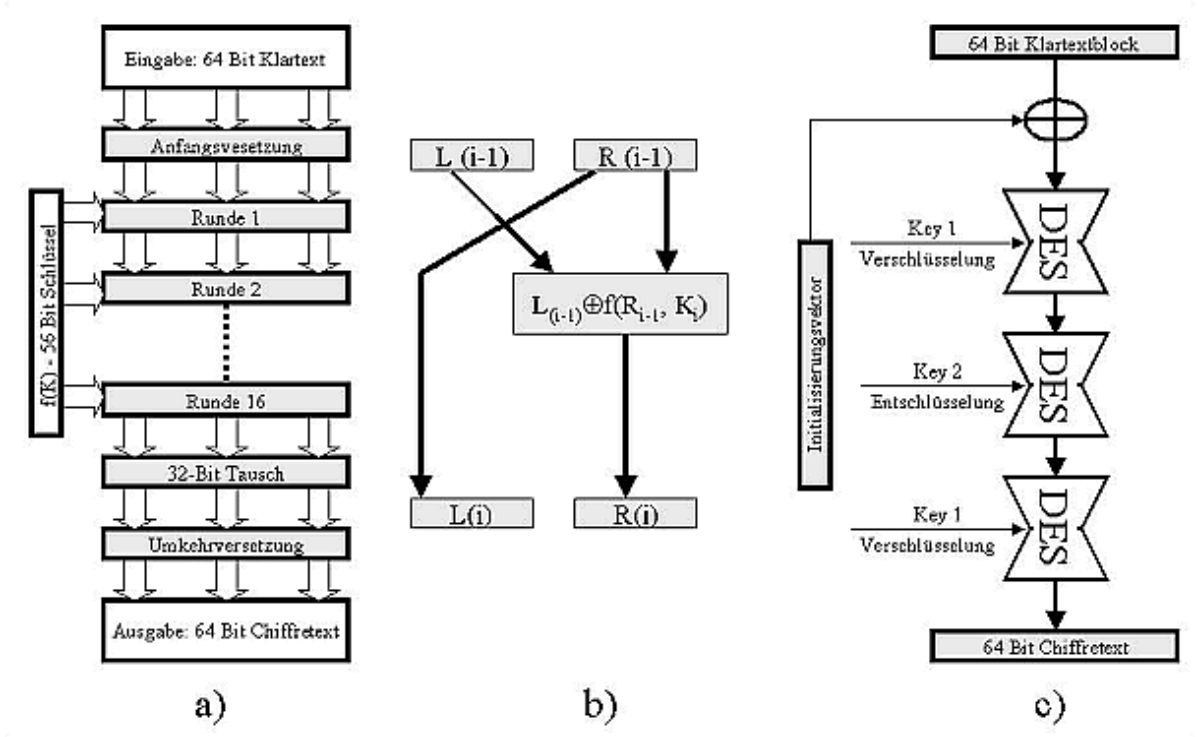

Abbildung 2.1: a) DES-Algorithmus allgemeiner Ablauf b) Detailansicht einer DES-Runde c) 2-Key-Triple-DES [49]

fahren. Durch Permutation, Vertauschen von ganzen Blöcken, lässt sich zusätzlich erreichen, dass möglichst viele Bits von vielen anderen Bits bzw. vom Schlüssel verändert werden. Durch die richtige Kombination lassen sich sehr robuste Algorithmen bilden, die sich gegen stochastische Analyseverfahren resistent zeigen. Man spricht vom Konfusions-Diffusions-Prinzip. Wer einen spielerischen Einstieg in die Analysetechniken sucht, dem sei ein c't-Artikel [24] empfohlen.

Eine wichtige Unterscheidung macht man zwischen stromorientierten und blockbasierten Verfahren. Stromarientierte Verfahren können den Klartext bitweise verschlüsseln. Ein Vertreter ist der Vermon-Chiffre, ein anderer ist der von Roland Rivest entwickelte RC4, er wird neben anderen bei SSL verwendet. Zu den blockorientierten Verfahren gehören der DES und somit auch das von HBCI verwendete TripleDES, aber auch der Nachfolger AES.

Blockorientierte Verfahren

Die blockorientierten Verfahren verschlüsseln immer Klartexte mit fester Länge. Das Auffüllen der nicht benutzten Daten nennt man „padding“. Auch beim padding versucht man es dem Angreifer möglichst schwierig zu machen. Damit zwischen den Daten und dem Aufgefüllten unterschieden werden kann, muss man sich natürlich auch hier einigen, wie aufgefüllt wird. Beim HBCI Version 2.2 wird die Norm ISO9796 benutzt.

Blockorientierte Verfahren haben die Eigenheit bei gleichem Klartext und gleichem Schlüs-

sel den gleichen Chiffretext auszugeben. Dies kann zum einen für eine stochastische Analyse ausnutzen und außerdem für eine replay-attacke, bzw. Cut-And-Paste-Attacke. Eine Möglichkeit das zu verhindern, ist das Cipher Block Chaining (CBC). Beim CBC werden vorher verschlüsselte Blöcke mit einbezogen.

DES

Der Data Encryption Standard (DES) wurde 1976 offizieller Standard der amerikanischen Regierungsbehörde. Er basiert auf dem ursprünglich von IBM entwickelten Verschlüsselungsalgorithmus Luzifer. Nach Änderungen durch die National Security Agency (NSA) konnte DES den Verdacht einer Hintertür bis heute nicht loswerden. Der Algorithmus ist ein blockorientiertes Verfahren. Es verschlüsselt 64 Bit große Blöcke und benutzt dazu einen 64-Bit-Schlüssel. Der effektive Schlüssel ist aber nur 56 Bit lang, die Bits 8,16,24,32,48,56 und 64 sind Paritätsbits.

DES ist wie die meisten Blockchiffren ein so genannter „iterierter Chiffre“, d.h. er besteht aus einer Wiederholung von 16 fast gleichen Runden, wie in Abb. 2.1 a). Die Runden unterscheiden sich nur durch den 48 Rundenschlüssel, . Das Prinzip einer Runde in in Abb. 2.1 b) dargestellt. In jeder Runde finden Substitutionen und Transpositionen statt, man spricht auch vom Konfusions-Diffusions-Prinzip. Es gibt 8 Substitutions-Boxen, S-Boxen S_i genannt, sie sind völlig irregulär („konfusion“) und sorgen so dafür, dass man z.B. bei einem Known-Plaintext-Angriff die Gesamtfunktion nicht nach Schlüsseln auflösen kann. Die Permutation sorgt dafür, dass sich Bits verteilen („diffusion“). Das führt zu einer Veränderung der Bits im Chiffretext gegenüber dem Klartext um rund 50%, Zitat nach [Smil97] in [12].

Die Entschlüsselung funktioniert sehr ähnlich wie die Verschlüsselung. Mit dem „Feistelprinzip“ kann sogar dieselbe Funktion zur Ver- und Entschlüsselung verwendet werden. Dadurch spart man die Hälfte der Implementierung ein, was für kleine Geräte vorteilhaft ist.

Zu beachten ist, beim DES-Algorithmus gibt es 4 so genannte schwache und 12 halbschwache Schlüssel. Benutzt man diese Schlüssel, dann bildet sich im Chiffretext ein charakteristisches Muster, welches Rückschlüsse auf den verwendeten Schlüssel zulässt. Insbesondere bei der Schlüsselgenerierung muss der Programmierer darauf achten, dass diese dokumentierten Schlüssel nicht verwendet werden. DES sollte wegen des kurzen Schlüssels nur in der Variante Triple-DES benutzt werden.

Triple-DES

Um die Sicherheit zu erhöhen durchläuft der Triple-DES-Algorithmus, wie der Name schon sagt, das DES-Verfahren dreimal mit einem anderen Schlüssel. Dadurch erhöht sich die effektive Schlüssellänge auf 168 Bit. Mit Paritätsbits ist der Schlüssel 192 Bit lang. Die Blocklänge bleibt bei 64 Bit. Der Algorithmus wird auch 3DESede abgekürzt. Ede steht für encryption decryption encryption und soll bedeuten, dass mit dem ersten Schlüsselteil verschlüsselt wird, mit dem zweiten entschlüsselt und mit dem dritten wieder verschlüsselt, siehe Abb. 2.1 c).

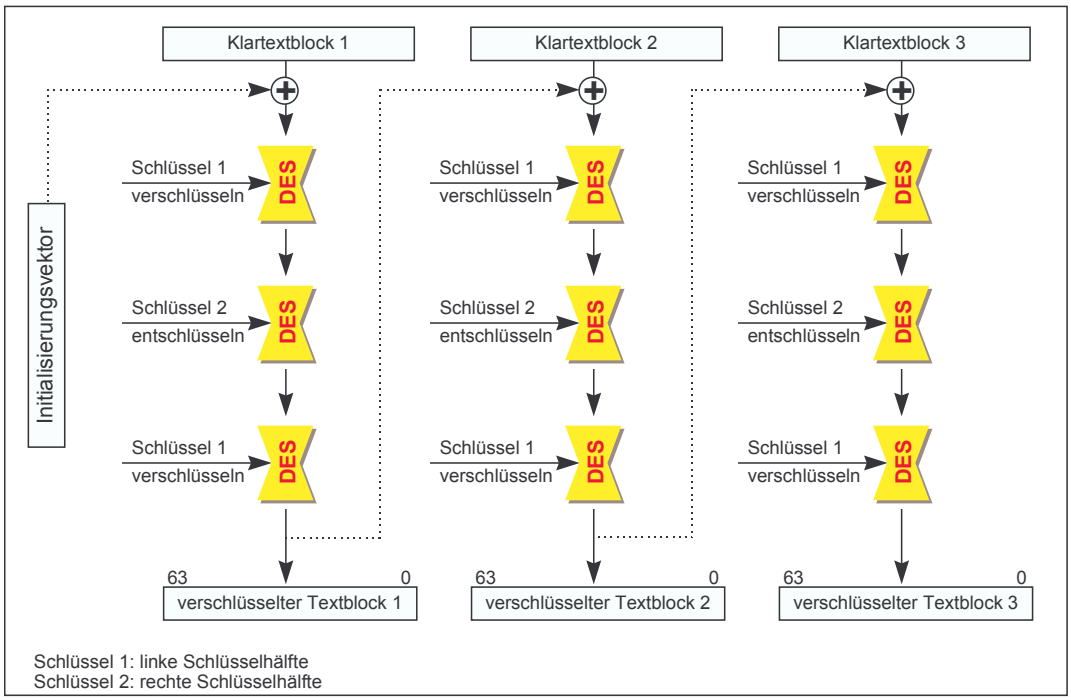

Abbildung 2.2: Beim CBC werden vor der Verschlüsselung der Klartext mit dem zuletzt verschlüsselten Block per xor verknüpt, um stochastischen Methoden weniger Angriffsfläche zu liefern. [49]

2-Key-Trible-DES

Der 2-Key-Trible-DES-Algorithmus arbeitet wie der TripleDES, benutzt aber bei der Verschlüsselung jeweils den gleichen Schlüssel, s. Abb. 2.1 c). Dadurch ist die effektive Schlüssellänge 112 Bit, bzw. 128 Bit mit Paritätsbits.

2.1.3 Asymmetrische Verschlüsselung

Eine der Schwächen der symmetrischen Verschlüsselung ist, dass man für jeden Kommunikationspartner einen Schlüssel braucht. Wenn Alice,Bob und Carol jeweils miteinander verschlüsselte Nachrichten austauschen wollen, brauchen sie für jede Paarung einen Schlüssel, d.h. Alice mit Bob, Alice mit Carol und Bob mit Carol, macht drei Schlüssel. Kommt ein vierter hinzu sind es schon sechs Schlüssel. Es werden $(n-1)!$, bzw. $\frac{(n)\times(n-1)}{2}$ Schlüssel für n Partner benötigt. Zudem müssen die Schlüssel über ein sicheres Medium übertragen werden.

In den 70er Jahren wurde mit der asymmetrischen Verschlüsselung ein neues Verfahren entwickelt, um die Schwachstellen der symmetrischen Verschlüsselung zu beheben. Es beruht im Prinzip auf mathematischen Problemen, die sich nur in einer Richtung elegant lösen lassen. Z.B ist es einfach das Produkt zweier Primzahlen zu bilden, zum anderen sehr schwierig eine Zahl in ihre Primfaktoren zu zerlegen.

Das erste veröffentlichte Verfahren, in dem vertrauliche Nachrichten zwischen Alice und Bob ausgetauscht werden können, wurde 1975 von Whitfield Diffie und Martin Hellman vorgestellt, nach denen es auch benannt wurde: Diffie-Hellman-Verfahren. Bei diesem Algorithmus müssen beide Kommunikationspartner aktiv an der Schlüsselgenerierung mitarbeiten, dieser Umstand macht es für viele Aufgaben unpraktikabel.

RSA

RSA wurde 1977 von den Namensgebern Ronald Rivest, Adi Shamir und Len Adleman entwickelt. Die Sicherheit von RSA beruht auf dem Faktorisierungsproblem, d.h. die Multiplikation zweier großer Primzahlen p und q stellt eine Einwegfunktion dar. Während die Multiplikation einfach ist, ist die Zerlegung einer Zahl in ihre Primfaktoren schwierig.

Zwei Schlüssel, public und private bilden ein Paar. Um ein RSA-Schlüsselpaar zu erzeugen, werden zwei zufällig ausgewählte große Primzahlen p und q (100 bis 200 Dezimalstellen und mehr) benötigt. Nun wird das Produkt n der beiden Primzahlen gebildet. Anschließend wird der zufälliger Wert e so gewählt, dass er teilerfremd mit $(p-1)\cdot(q-1)$ ist. Der Wert d (decrypt) wird nun mit Hilfe des erweiterten euklidschen Algorithmus berechnet:

$$d = e^{-1}\ mod((p-1)(q-1))$$

Die Zahlen e und n bilden den öffentliche Schlüssel und die Zahlen d und n den geheimen Schlüssel.

Zu verschlüsselnde Nachrichten, die größer als n sind, werden in kleinere Blöcke zerlegt. Die Nachrichtenblöcke m_i werden mit der Formel verschlüsselt:

$$c_i = m_i^e\ mod\, n$$

c_i sind die verschlüsselten Nachrichtenblöcke. Die Entschlüsselung erfolgt analog:

$$m_i = c_i^d\ mod\, n$$

Im HBCI-Standard ist die Länge des Produkts n, der beiden großen Primzahlen p und q, von 708 Bit bis 768 Bit festgelegt worden. Der Chiffrierschlüssel e wurde auf die 4. Fermatsche Primzahl festgelegt: $e = 2^{16} + 1 = 65537$. Die beiden großen Primzahlen dürfen sich in ihrer Länge um höchstens 12 Bit unterscheiden. Weiterhin darf $(p-1)$ und $(q-1)$ nicht durch den Chiffrierschlüssel e teilbar sein.

2.1.4 Hash-Algorithmen

Eine Hashfunktion verarbeitet eine beliebig lange Nachricht und erzeugt daraus einen Hashwert fester Länge. Für die Kryptographie sind Einweg-Hashfunktionen von besonderer Bedeutung. Die Anforderungen sind:

- Einwegfunktion,

- Die Veränderungen eines einzelnen Bits im Eingabetext muss zu einem anderen Hashwert führen.
- Wenige Kollisionen, d.h. die berechneten Hashwerte sollen sich gleichmäßig über gesamte mögliche Hashwertmenge verteilen.

Eine der bekanntesten Hash-Algorithmen ist der MD4 (Message Digest) von Ronald Rivest. Dieser gilt nicht mehr als sicher genug.Eine Weiterentwicklung ist der SHA-1 (Secure Hash Algorithmus). Eine andere europäische Entwicklung ist RIPEMD-160 (RACE Integrity Primitives Evaluation Message Digest) [3]. Wie der Name andeutet, erzeugt dieser Algorithmus einen 160 Bit großen Wert. RIPEMD ist Runden orientiert, siehe Abb. 2.1.4 , bei RIPEMD-160 sind es 5 Runden, die doppelte Ausführung soll interne Kollisionen vermeiden.

2.1.5 Hybridverfahren

Sowohl die symmetrische als auch die asymmetrische Verschlüsselung haben ihre Vor- und Nachteile. Um die Vorteile beider Algorithmen auszunutzen, kombiniert man diese in einem hybriden Verfahren. Man nutzt die Geschwindigkeit der symmetrischen Verschlüsselung gegenüber der asymmetrischen. Die asymmetrische Verschlüsselung hat dagegen den Vorteil, dass man weniger Schlüssel braucht.

Zum Verschlüsseln der Daten benutzt man das symmetrische Verfahren, z.B. TripleDES. Der Schlüssel wird vorher mit einem Zufallsgenerator erzeugt. Üblicherweise vor jeder Verschlüsselung. Nach dem Verschlüsseln, wird der so genannte Nachrichtenschlüssel mit dem asymmetrischen Verfahren, z.B. RSA mit dem öffentlichen Schlüssel des Partners, verschlüsselt. Die verschlüsselten Daten werden zusammen mit dem verschlüsselten Nachrichten-Schlüssel übertragen.

2.1.6 Signatur

Die elektronische Signatur, auch als digitale Unterschrift bezeichnet, soll die Herkunft und die Integrität der Nachricht sichern. Dazu wird über die Nachricht ein Hashwert, z.B. mit RIPEMD-160, gebildet. Dieser Wert wird mit dem eigenen privaten Schlüssel per asymmetrischer Verschlüsselung, z.B. RSA, verschlüsselt. Die Signatur wird an die Nachricht angehängt.

2.2 HBCI

Fast alles Beschriebene ist der 768 Seiten langen Spezifikation [49] zu entnehmen. Bei einigen Abläufen muss man aber mehrere Abschnitte lesen, auf die nicht immer verwiesen wird. So übersieht man leicht eine wichtige Kleinigkeit. Ich versuche in diesem Kapitel einen Überblick zu verschaffen. Was übrig bleibt sind die so genannten Geschäftsvorfälle, die sich aber vom Aufbau ähneln. Wer nicht auf einem exotischen System entwickelt, dem empfehle ich einen Blick auf die API vom [23]. Seine Bibliotheken sind lizenzfrei in eigene Programme zu integrieren und man ist auf dem neusten Stand.

Sowohl vom Kunden als auch von den Banken bestand schon sehr lange der Wunsch, dass Kunden ihre Bankgeschäfte von zu Hause aus erledigen können. Die Banken erhoffen sich durch die Selbständigkeit der Kunden nicht nur einen besseren Service, vielmehr geht es um Kostenreduktion. Eine Filiale mit seinen Mitarbeitern verursacht höhere Kosten, als ein Callcenter oder ein Server inklusive Administrator, wenn man die gesamten Kosten pro Kunde zugrunde legt. Der Kunde dagegen hat den Vorteil, nicht mehr von den Öffnungszeiten der Filialen abhängig zu sein oder sich eine günstigere Bank zu suchen, die aber nicht in seiner Nähe ist. Das so genannte Homebanking ist als Ergänzung zu sehen, es gibt kaum eine Bank, die sich nur auf das Homebanking beschränkt.

Im August 1995 veröffentlichte der Zentrale Kreditausschuss (ZKA) die erste Version des Homebanking Computer Interface (HBCI). HBCI ist eine standardisierte Schnittstelle zwischen Kunde und Bank. Das Ziel war die damals verbreiteten inkompatiblen Homebankingsysteme abzulösen. Es wurden bis dahin fast alle Homebankinggeschäfte über BTX getätigt. Aktuell ist die Version 2.2, im April sollte Version 3.0 erscheinen, an dieser wird aber wie auch schon an der Version 4.0 gearbeitet [11]. Mittlerweile ist die Vorversion 3.0 erschienen, in ihr sind aber keine wichtigen Neuerungen in Bezug auf diese Arbeit enthalten. Am Ende des Kapitels werde ich auf einige Neuheiten eingehen. Da der Standard viele Interessen vereinen sollte und aus den Kritikpunkten der damaligen Systeme, ergaben sich folgende Anforderungen, die HBCI erfüllen sollte:

Multibankfähigkeit: Jede Zahlungsverkehrsoftware mit HBCI-Kernel soll in der Lage sein, mit jedem Kreditinstitut bzw. HBCI-Server zu kommunizieren. Erreicht werden soll dies über ein einheitliches Protokoll, fest definierte Kommunikationsdialoge und -ports.

Transportdienstunabhängigkeit: Das HBCI-Protokoll kann und soll mit jedem Transportdienst (TCP/IP, CEPT, etc.) zurechtkommen und somit dem Kunden eine freie Wahl des Zugangs (Providerwahl) ermöglichen.

Endgeräteunabhängigkeit: Neben dem PC und Notebook sollen auch mobile Geräte wie Handy und PDA als mögliche HBCI-Kommunikationsplattformen berücksichtigt werden.

Offenheit: Durch frei zugängliche Spezifikationen und Entwicklungstools (APIs) soll nicht nur Vertrauen geschaffen, sondern auch die einfache Entwicklung von HBCI-Anwendungen vorangetrieben werden.

Sicherheit: Die Sicherheitsmechanismen zum Schutz der übertragenen Daten, sowie die Authentifizierung soll auf den derzeit anerkannten und als sicher geltenden Standard beruhen und gegebenenfalls erweiterbar sein.

Flexibilität: Die HBCI-Spezifikation soll möglichst modular (objektorientiert) aufgebaut sein, um eigene Geschäftsvorfälle hinzufügen und parametrisieren zu können und um die Eigenentwicklung von HBCI-Anwendungen zu vereinfachen.

Präsentationsdienstunabhängigkeit: Der HBCI-Kernel soll nur Rohdatenströme zur Verfügung stellen und die „Kunde ↔ Bank“-Kommunikation abwickeln. Die Visua-

lisierung und Interaktion mit dem Kunden soll alleine Aufgabe der Homebanking-Anwendung sein.

Anwendungssystem-Unabhängigkeit: Es soll möglich sein, HBCI-Anwendungen sowohl über fest installierte Software, als auch über dynamisch geladene Software-Module (JavaApplets, ActiveX, etc.) zu realisieren.

Plattformunabhängigkeit: Die HBCI-Spezifikation soll keine Einschränkungen bezüglich des verwendeten Betriebssystems machen und als Entwicklungsumgebung (HBCI-Kernel) möglichst auch für alle Plattformen verfügbar sein.

Online und Offline-Arbeiten: soll möglich sein, um Übertragungskapazitäten effektiver zu nutzen, und um dem Kunden mehr Komfort zu bieten.

In der Spezifikation von HBCI wird zwischen dem Benutzer und dem Kunden unterschieden. Der Kunde ist der Inhaber des Kontos, während der Benutzer die Berechtigung über den Zugriff auf das Konto über HBCI hat. Wichtig ist diese Unterscheidung bei Firmen-Konten. Es gibt zwei Szenarien, zum einen kann ein Benutzer sowohl auf sein Privatkonto zugreifen und auf das Firmenkonto, d.h. es gibt zwei Kunden und einen Benutzer. Die zweite Möglichkeit ist, dass mehrere Angestellte einer Firma auf das Firmenkonto zugreifen. Dann gibt es einen Kunden und mehrere Benutzer. Die Spezifikation sieht hierfür die Unterscheidung zwischen Kunden- und Benutzer-ID vor, wenn eines oder beide Szenarien zutreffen. Der Einfachheit halber wird Zukünftig davon ausgegangen, das Kunde und Benutzer identisch sind, was üblicherweise bei einem Privatkonto der Fall ist.

2.2.1 Dialogablauf

Die Übertragung der Nachrichten ist zwar unabhängig vom Übertragungsmedium, aber in der Praxis wird überwiegend TCP/IP über Port 3000 benutzt, weshalb ich mich darauf beschränke. Das ISO/OSI-Schichten-Modell ist in 7 Schichten eingeteilt. Das HBCI-Protokoll ist nach dieser Einteilung in der obersten, der Applikationsebene einzuordnen. Auch wenn innerhalb der Norm Verschlüsselung und Transformation von Zeichensätzen beschrieben sind, lassen sie sich nicht in niedrigere Schichten einteilen, sondern gehören mit in die siebte Ebene.

Die Kommunikation zwischen Kunde und Bank wird Dialog genannt. Der Verbindungsaufbau geht immer vom Kunden auf. Zuerst wird die logische Verbindung, z.B. TCP-Verbindung auf Port 3000 aufgebaut. Der Dialog beginnt mit einer speziellen Nachricht, der Dialoginitialisierung und endet mit der Nachricht Dialogende. Der Nachrichtenaustausch läuft synchron, jede Kundennachricht muß erst vom Banken-Server beantwortet werden, bevor eine neue Nachricht gesendet werden kann. Zwischen der Dialoginitialisierung und dem Ende können beliebige Auftragsnachrichten geschickt werden. Nach dem Dialogende kann ein neuer Dialog, z.B. von einem anderen Kunden aufgebaut werden, ohne die logische Verbindung zu trennen.

Es ist auch die asynchrone Bearbeitung von Aufträgen vorgesehen. Dies kann bei Sammelüberweisungen sinnvoll sein. Dann wird nach dem Auftrag der Dialog beendet. Der Erfolg der Aufträge kann dann über ein Statusprotokoll abgefragt werden.

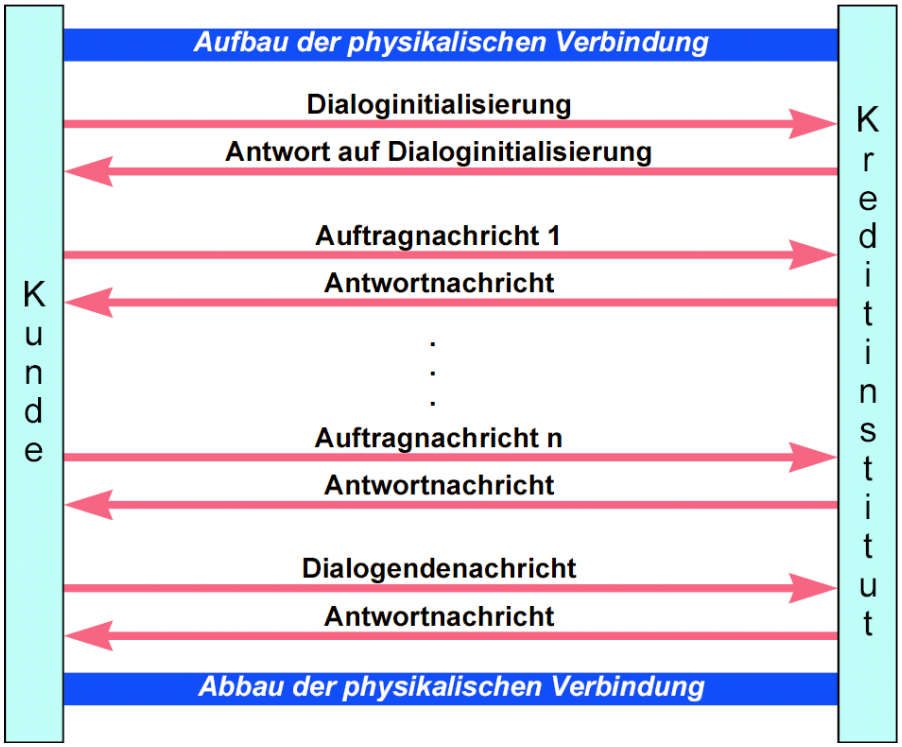

Abbildung 2.3: HBCI Dialogablauf [49]

Der Datenstrom wird nicht verschlüsselt, sondern die Vertraulichkeit und Authentizität der Nachrichten wird dadurch erreicht, dass jede Nachricht separat signiert und verschlüsselt wird. Ausnahmen bestätigen bekanntlich die Regel. Die Anforderung der öffentlichen Schlüssel des Kreditinstitutes wird durch einen anonymen Dialog angefordert. Der anonyme Zugang ist unverschlüsselt und unsigniert durchzuführen, über ihn können auch die Parameterdaten, s. Kap. 2.2.3 erneuert werden.

2.2.2 Nachrichtenaufbau

Alle Nachrichten, inklusive der Dialoginitialisierungsnachricht und der Dialogendenachricht haben den selben Aufbau. Sie bestehen aus drei oder mehr Segmenten. Die Segmente aus Datenelementen DE oder Datenelementegruppen DEG. Und die DEG bestehen wiederum aus Datenelementen, vor HBCI 3.0 Gruppendatenelement GD genannt.

Syntax Beispiel eines Segmentes

```
HIDAB:6:1:5+1234567:280:10020030+7654321:280:20030040+MEIER
    FRANZ++1000,:DEM+52+000+MIETE:UND NEBENKOSTEN
    +19960901+00001+19960701:M:1:1:19970601+N:::3'
```

Es ergeben sich bei HBCI drei hierarchische Ebenen:

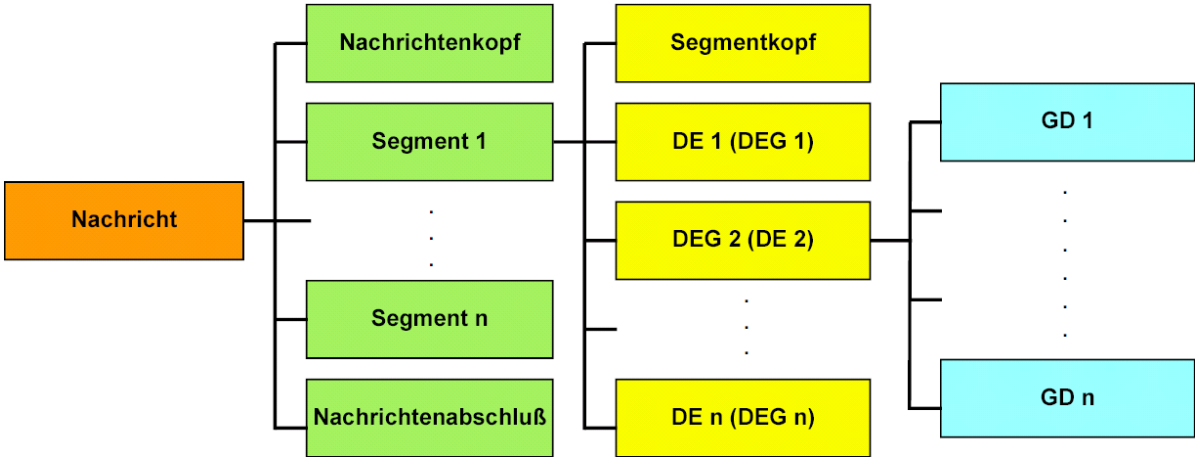

Abbildung 2.4: Nachrichtenaufbau [49]

- Datenelemente DE/Datenelementgruppen DEG
  Dies sind die einfachsten Elemente einer Nachricht, ein Datenelement kann aus einer einfachen Zahl bestehen, z.B. Bankleitzahl. Ein Datenelement kann aber auch ein transparent eingestelltes S.W.I.F.T.-Format sein. Logisch zusammengehörige Datenelemente werden zu Datenelementgruppen zusammengefasst.

- Segmente
  HBCI-Segmente enthalten alle logisch zusammengehörigen Datenelemente und Datenelementgruppen, welche einen Geschäftsvorfall beschreiben.

- Nachricht
  Innerhalb einer Nachricht können sich mehrere Geschäftsvorfall-Segmente befinden, so können z.B. 3 Überweisung und eine Umsatzanfrage übertragen werden.

Wie in Abb. 2.4 dargestellt, enthält jede Nachricht zwei spezielle Segmente, einen Nachrichtenkopf und ein Nachrichtenende. Im Nachrichtenende steht die gleiche Nachrichtennummer wie im Kopf. Im Nachrichtenkopf stehen zusätzlich Angaben zur verwendeten DialogNummer, der HBCI-Version und die Gesamtgröße der Nachricht.

Jedes Segment hat einen Segmentkopf, dieser ist ein DEG. Über den Segmentkopf wird der Inhalt definiert. Er enthält eine Kennung, eine laufende Nummer, eine Version, sowie eventuell eine Bezugsnummer, wenn es sich um die Antwort auf ein anderes Segment handelt.

Zur Trennung der Elemente verwendet HBCI eine Trennzeichensyntax. Diese ist an UN-/EDIFACT angelehnt. „Aufgrund der Komplexität von UN/EDIFACT und der zahlreichen Geschäftsvorfälle, werden die Formate nach eigenen Regeln aufgebaut. Die Inhalte im Sicherheitsbereich werden aus UN/EDIFACT Version 3 übernommen.“ Dazu kommen mit DTAUS und S.W.I.F.T., Fremdformate aus dem Bankenbereich, die in den HBCI Nachrichten gekapselt werden.

Beispiel anonyme Dialoginitialisierung

+	Datenelement Ende
:	Gruppendatenelement Ende
’	Segment Ende
?	Freigabezeichen (bei Steuerzeichen im Text)
@	Kennzeichen für binäre Daten

Tabelle 2.1: Die Sonderzeichen der Trennzeichensyntax im Überblick

```
HNHBK:1:2+000000000109+220+0+1’
HKIDN:2:2+280:09950003+9999999999+0+0’
HKVVB:3:2+0+0+1+xpresso+0.1’
HNHBS:4:1+1’
```

Die Grundatenelemente GD werden durch einen Doppelpunkt : voneinander getrennt. Datenelemente und Datenelementegruppen werden innerhalb eines Segmentes durch ein Pluszeichen + getrennt. Das Segment endet mit einem Apostroph ’ Im Beispiel 2.2.2 beginnt das Segment mit dem Segmentkopf, das erste DEG: `HIDAB:6:1:5`, darauf folgt das nächste DEG: `1234567:280:10020030`. Nach den Nebenkosten folgen zwei DE: `+19960901+00001+`, erstes DE ist `19960901`.

Allgemeiner dargestellt ergibt sich:

Segmente: `SEG:=Segmentkopf+DE+DEG+DE+...DEG’`
→ |SEG:=...+DE+GD:GD:GD+DE+...’|

Datenelementegruppen: `DEG:=GD:GD...:GD`

Binärdateien: `SEG:=...+@Länge@Binärdaten+DEG+...’`

Es wurde auch gleich dargestellt, wie transparente Daten übertragen werden. Ein vorgestelltes At-Zeichen @ signalisiert, dass es sich im Folgenden um Binärdaten handelt, in diesen dürfen Plus und Doppelpunkt nicht interpretiert werden. Binärdaten können ein DE oder ein GD sein. Fremdformate, wie DTAUS oder S.W.I.F.T., werden als transparente Daten, wie binäre Daten behandelt. Das letzte Spezialzeichen ist das Fragezeichen ? es maskiert alle Steuerzeichen aus.

2.2.3 Banken- und Userdatenparameter

Da beim HBCI so viele unterschiedliche Interessen berücksichtigt werden müssen, hat man mit den Parameterdaten ein Verfahren geschaffen, mit dem sich die Kundensoftware anpassen lässt. Es gibt zwei Arten, die Userparameterdaten UPD und die Bankenparameterdaten BPD. Beide werden vom Banken-Server geliefert. In den Parameterdaten stehen z.B. Informationen über die unterstützten Geschäftsvorfälle oder wie viele Zeilen das Betreffeld bei Überweisungen hat. Leider geben diese Informationen keine Garantie über seine Einhaltung, so kann ein Geschäftsvorfall trotzdem abgelehnt werden. Mit dem Parameterdaten wird eine Versionsnummer mitgesendet. In der Dialog tialisierung gibt man die gespeicherte Version mit an. Liegt eine neuere Version vor, so wird diese in der Antwort mitgeliefert.

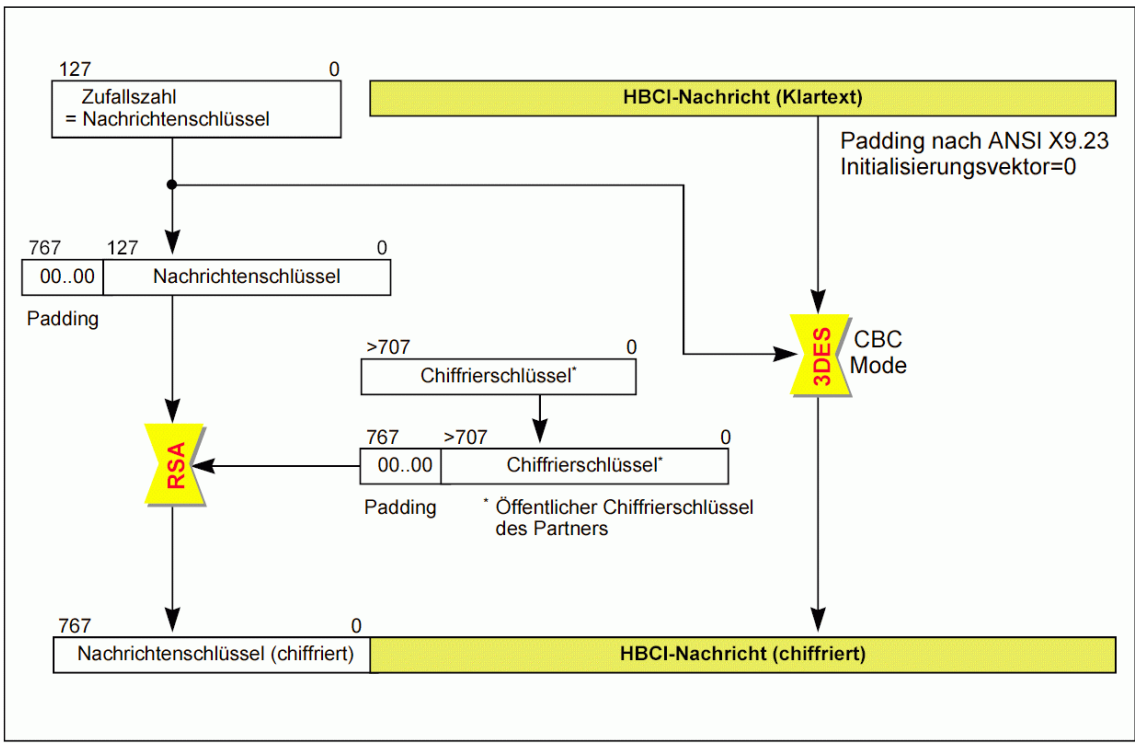

Abbildung 2.5: Verschlüsselung der Nachrichten und des Nachrichtenschlüssels. Der Schlüssel kommt in der Verschlüsselungskopf (HNVSK), die Nachricht in das Segment verschlüsselte Daten (HNVSD) [49]

2.2.4 Signatur und Verschlüsselung

Wie bereits erwähnt, werden bei HBCI jede Nachricht einzeln signiert und verschlüsselt. Wie nicht anders zu erwarten, werden auch diese Daten in Segmente verpackt. Die Signatur, verschlüsselte Daten und die Schlüssel werden im binären Format gekapselt. Zur Erinnerung: Binärdaten beginnen mit einem @, dann kommt die Länge gefolgt von einem weiteren @, anschließend kommen die Daten.

HBCI unterscheidet zwischen drei Schlüsseln, dem Signierschlüssel, dem Chiffrierschlüssel und dem Nachrichtenschlüssel.

Im Signaturkopf stehen Informationen zum verwendeten Verfahren und dem Schlüssel. Das Segment Signaturabschluß enthält die Signatur. Die Signatur wird nicht über die gesamte Nachricht gebildet, sondern beginnt mit (einschließlich) dem Signaturkopf und endet vor dem Signaturabschluß. Es sind bis zu drei verschachtelte Signaturen erlaubt. Da es für Privatkunden nicht relevant ist, wird dies nicht weiter erläutert.

Für jede Nachricht wird ein neuer Nachrichtenschlüssel gebildet, dieser wird verschlüsselt in den Chiffrierkopf eingetragen. Auch hier stehen Informationen zum verwendeten Chiffrierschlüssel und dem benutzten Verfahren. Nun werden alle Segmente zwischen dem Nachrichtenkopf und dem Nachrichtenabschluß mit dem Chiffrierschlüssel verschlüsselt, und in das Segment Chiffrierabschluß gepackt. Jede verschlüsselte Nachricht besteht somit aus vier Seg-

Vor Verschlüsselung:		Nach Verschlüsselung:	
Nr.	Segmentname	Nr.	Segmentname
1	Nachrichtenkopf	1	Nachrichtenkopf
2	Signaturkopf	998	Verschlüsselungskopf
3	Auftrag 1	999	Verschlüsselte Daten (enthält:
4	Auftrag 2		| 2 Signaturkopf
			| 3 Auftrag 1
			| 4 Auftrag 2)
5	Signaturabschluß		| 5 Signaturabschluss)
6	Nachrichtenabschluss	6	Nachrichtenabschluss

Tabelle 2.2: Stufen einer verschlüsselten Nachricht

menten.

Die Spezifikation empfiehlt für den Aufbau von verschlüsselten Nachrichten folgendes Vorgehen einzuhalten:

1. Die Nachricht ist zunächst unverschlüsselt aufzubauen, inklusive Signatur.
2. Das Segment „Verschlüsselungskopf“ ist direkt hinter dem Nachrichtenkopf einzustellen.
3. Die verschlüsselten Signatur- und Auftragssegmente sind in das Segment „Verschlüsselte Daten“ einzustellen.

Ich empfehle aber den Nachrichtenkopf erst am Schluss zu erstellen. In ihm wird die Gesammtgröße der Nachricht eingetragen und diese ist als letztes bekannt.

Vor der Verschlüsselung weisen die Segmente eine kontinuierliche Nummerierung auf (s. Abb. links). Um die Eindeutigkeit der Segmentnummern zu gewährleisten, erhält das Segment „Verschlüsselungskopf“ die Segmentnummer 998 und das Segment „Verschlüsselte Daten“ die Segmentnummer 999, siehe Tabelle 2.2.

Es handelt sich um eine der zentralen Mechanismen bei HBCI und unterscheidet sich von mir bekannten. Deshalb möchte ich die Phasen, die eine Nachricht bis zur Verschlüsselung durchläuft, noch einmal anders darstellen. Die Segmente werden im Folgenden durch ihre Kennung im Segmentkopf dargestellt. Nachrichtenkopf = HNHBK, Signierkopf = HNSHK,

HNHBK	HBCI-Nutzdaten	HNHBS		
HNHBK	HNSHK	HBCI-Nutzdaten	HNSHA	HNHBS
	Diese Segmente gehen in die Signatur ein			
	Diese Segmente werden verschlüsselt			
HNHBK	HNVHK	HNVHD *crypt(HNSHK | HBCI-Nutzdaten | HNSHA)*		HNHBS

In der HBCI-Spezifikation werden zwei Verfahren zur Verschlüsselung und zum Signieren genannt. Alle bisher besprochenen Mechanismen gelten für beide Varianten. Die Verschlüsselung der Nachricht ist bei beiden gleich, es wird der so genannte 2-Key-Triple-DES-Algorithmus im CBC-Modus mit ISO-10126 Padding angewendet. Die beiden Verfahren sind:

Verfahren	Signatur	Schlüssel-Verschlüsselung	Nachrichten-Verschl.
DDV	MAC	2-Key-Triple-DES	2-Key-Triple-DES
RDH	RSA-EU	RSA	2-Key-Triple-DES

Tabelle 2.3: Signatur und Verschlüsselsverfahren

DDV steht für DES-DES-Verfahren. Es wird nur 2-Key-Triple-DES benutzt. Zur Verschlüsselung der Nachrichtenschlüssel wird der DES-Algorithmus im ECB-Modus angewendet. Für die Signatur wird ein Message Authentic Cods MAC benutzt.

RDH steht für RSA-DES-Hybrid-Verfahren. Es wird der RSA-Algorithmus für den Nachrichtenschlüssel verwendet und die Signatur.

Das DDV ist nur für Chipkartenapplikationen vorgesehen. Softwarelösungen müssen das RDH benutzen, deshalb beschränke ich mich auf dieses. Der Hashwert wird sowohl bei DDV und beim RDH durch RIPEMD-160 für die Signatur gebildet. Der Hashwert wird nach ISO-9697-1 codiert, bevor er mit RSA, beim RDH, verschlüsselt wird.

2.2.5 Sicherheit von HBCI

DDV ist optional, RDH ist verpfichtend, mit der Ausnahme [49]: „Ausgenommen hiervon sind Endgeräte, die eine RSA-EU-Lösung oder RDH-Verschlüsselung noch nicht erlauben (z.B. Smartphones mit MAC-Chipkarte erlauben ggf. keine RSA-EU, ...“

Die Bildung der elektronischen Signatur erfolgt wie schon im Abschnitt 2.1.6 angegeben:

1. Bildung des Hashwertes
2. Ergänzen des Hashwertes auf eine vorgegebene Länge und
3. Berechnung der elektronischen Signatur über den Hashwert.

Das Hashing ist in den beiden Verfahren DDV und RDH identisch, es wird der RIPEMD-160-Algorithmus verwendet. Die beiden anderen Verarbeitungsschritte sind jeweils verschieden, siehe Tabelle 2.3.

Grundsätzlich können Kunde und Kreditinstitut beim asymmetrischen Verfahren (RDH) über zwei Schlüsselpaare verfügen: Ein Signierschlüsselpaar und ein Chiffrierschlüsselpaar. Der Signierschlüssel wird zum Unterzeichnen von Nachrichten verwendet, während der Chiffrierschlüssel zum Verschlüsseln von Nachrichten dient. Falls ein Kreditinstitut seine Nachrichten nicht signiert, kann es auf das Signier- schlüsselpaar verzichten.

2.2.6 Angriffsmethoden

Untersucht man die Angriffsmöglichkeiten auf HBCI, muss man zwischen zwei Sachen unterscheiden, zwischen einem Angriff auf das Verschlüsselungs-Verfahren und einem auf die Software.

Knacken der Verschlüsselung

Um es vorweg zu nehmen, sowohl DES als auf RSA gelten bis heute als sicher, wenn man die Schlüssel groß genug wählt. Bei RSA wird für zukünftige Anwendungen zur Zeit 1024 Bit empfohlen, 768 Bit sollten nicht unterschritten werden. HBCI benutzt Schlüssellängen von 708 bis 768 Bit und liegt damit unter der Empfehlung. Es besteht aber kein Grund zur Panik, da auch Schlüssel dieser Länge zur Zeit noch einen unvertretbaren Aufwand erfordern. Der ZKA hat aber schon reagiert und für die nächste Version Schlüssellängen von 1024 bis 2048 Bit angekündigt. Für DES bzw. Triple-DES sind 112 Bit ausreichend.

Wie schon gesagt, gibt es kaum einen Verschlüsselungsalgorithmus, bei dem die Sicherheit mathematisch bewiesen wurde. Der Algorithmus wird anhand bekannter Analysemethoden untersucht. Das ist auch der Grund, warum neue Algorithmen so lange brauchen, bis sie eingesetzt werden. Man möchte vielen Forschern die Möglichkeit bieten, Schwachstellen zu finden. Bei den von HBCI benutzten Verfahren, sind bis heute keine Schwachstellen bekannt. Der einzige Weg eine verschlüsselte Nachricht zu entschlüsseln besteht in der brute-force-attack. Die empfohlenen Schlüssellängen beziehen sich daher auf den Aufwand, den man zum Durchprobieren der Schlüssel benötigt.

Um die Überlegenheit der eigenen Verschlüsselungsverfahren zu demonstrieren, stiftete die Firma RSA 1998 einen Preis für denjenigen, der es als erstes schafft eine Nachricht, die mit DES (56 Bit) verschlüsselt wurde, zu knacken. Im März 1998 brauchten 22.000 Systeme 39 Tage zum Entschlüsseln der chiffrierten Nachricht. Die Key wurde nach der brute-force-methode gesucht und der Schlüssel erst fast am Ende gefunden.

Nimmt man nun an, dass sich nach dieser Methode der Schlüssel, im Durchschnitt bei 50% der getesteten Schlüssel ermitteln lässt, sollten diese 22.000 Systeme den Schlüssel im durchschnitt am 20. Tag finden. Wenn man zudem vom Moore'schen Gesetz ausgeht, verdoppelt sich die CPU-Leistung jedes Jahr (bzw. all $11/2$ Jahre). Das heißt im Jahre 2005 brauchen 22.000 Systeme nur noch einen $1/4$ Tag.

HBCI benutzt mit 2-Key-Triple-DES 112 Bit. Mit jedem Bit verdoppelt sich die Anzahl der möglichen Bits und somit auch die benötigte Zeit bei gleicher Hardware. Wenn wir den $1/4$ Tag mit 2^56 multiplizieren, ergeben sich 2^54 Tage oder 2^45 Jahre. Selbst wenn man bei Specialhardware und neuen Algorithmen von einer Beschleunigung von 10 pro System ausgeht, muss man immer noch mehr länger warten, als das geschätzte alter der Erde.

Um die Sicherheit gegen stochastische Methoden oder Angriffe, wie z.B. die known-plaintext-attack zu erhöhen, wendet HBCI weitere Maßnahmen an:

- ISO 9697-1 Encoding, ISO 10126, CBC gegen known-plaintext-attack
- IDs gegen replay-attack
- Signaturen gegen man-in-the-middle-attack

Mit anderen Worten, nach heutigem Wissen und der vorhandenen Technologie, ist die von HBCI benutzte Verschlüsselung sicher.

Die wirkliche Schwachstelle befindet sich oft an anderer Stelle. Es sind gespeicherte Schlüssel leicht zugänglich[3]. Das ist auch die Methode mit dem der Stern erfolgreich war. Angriff mit Trojanern im Stern-Artikel [32] und Stellungnahme des ZKA [40].

Trojaner auf dem Handy sind bisher nicht möglich, aber auf dem Palm durch so genannte Betriebssystemhacks[4]. Dies scheint aber die Volks- und Raiffeisenbanken nicht zu stören, denn sie bieten bereits einen HBCI-Client für den Palm an [34], der seine Schlüssel im RAM speichert.

HBCI gilt als das sicherste Verfahren für Online-Banking.

2.3 Java™2 Micro Edition (J2ME)

2.3.1 Java's coming home

Die Sprache Java wurde ursprünglich von der Firma SUN für „embedded Systeme“ vorgestellt, um in diesem Bereich plattformunabhängige Software zu entwickeln (da diese kleinen Geräte in der Regel keinen Compiler haben, benutzt Java einen Bytecode). Doch es kam zuerst anders. Die Idee ein Programm „überall“ starten zu können, ohne erneutes Übersetzen der Anwendung, wurde von der heterogenen Netzgemeinschaft begeistert aufgenommen. Java wurde für Applets und einfache Serverprogramme benutzt. Diesen Trend unterstützte SUN mit der verstärkten Weiterentwicklung in diesem Bereich. Trotz des Erfolges als „Internet-Sprache“ wurde auch die Version für Kleinstgeräte weiter entwickelt. Auf Basis von Java 1.1 präsentierte SUN picoJava und die JavaCard. Mit der Herausgabe von Java 2 wurden alle Java-Zweige unter einem Dach vereint (s. Abb. 2.6).

Es stellte sich heraus, dass picoJava nicht allen Anforderungen gerecht werden konnte, da die Geräte zu unterschiedlich sind. Zum einen soll Java klein und schnell sein und zum anderen alle Schnittstellen unterstützen, die auf kleinen Geräten vorkommen, vom Netzwerk, über verschiedene Funksysteme bis zur grafischen Darstellung bei einzeiligen Displays oder Webtablets.

Mit Java 2 veröffentlichte Sun die Version für embedded Systems, Java 2 Micro Edition, kurz J2ME genannt. J2ME enthält nicht mehr eine API, sondern ist unterteilt. Es gibt zwei Gerätegruppen (s. Abb. 2.7): Die Connected Devices Configuration CDC für kleine Laptops bis PDAs, und die Connected Limited Devices Configuration CLDC für PDAs, Handys und Waschmaschinen. Zusätzlich gibt es noch die Profiles, welche die Anforderungen einzelner Geräteklassen unterstützen. Ein Gerät gehört entweder zu CDC oder zu CLDC, kann aber mehrere Profiles unterstützen. Veröffentlicht ist bisher das Mobile Information Device Profile MIDP Version 1.0.2 und das Personal Profile. Das Personal Profile soll das bisherige personalJava ablösen. CDC und CLDC existieren in der Version 1.0. Wenn nicht anders erwähnt, bezieht sich die folgende Beschreibung auf diese Versionen. Am Ende dieses Abschnittes wird ein Ausblick auf die kommenden Erweiterungen gegeben. Mit Schnittstellen sind Interfaces gemeint.

[3] Wenn z.B. die Diskette im Laufwerk gelassen wird, kann sie sowohl vom PC ausgelesen werden oder sie wird entwendet.

[4] dies sind nützliche Erweiterungen

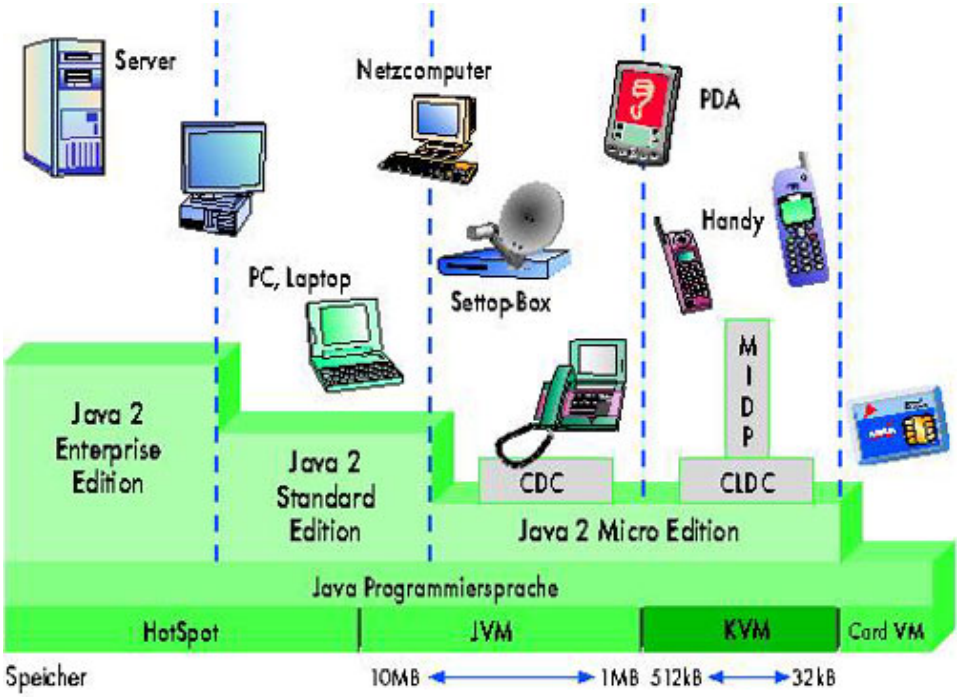

Abbildung 2.6: Java 2 Platformen

Um dem geringen Speicher und CPU-Leistung gerecht zu werden, wurde eine neue optimierte virtuelle Maschine zuerst für den Palm entwickelt. Diese sollte möglichst wenig Ressourcen verbrauchen und Plattformunabhängig sein. Wegen des geringen Speicherverbrauches wurde sie „kilobyte virtuall machine“ KVM genannt. Die KVM ist eine Implementierung der Java Virtuellen Maschine mit einigen Zugeständnissen an die beschränkten Ressourcen. Die KVM

- bietet nur eine Möglichkeit Klassen nachzuladen, man kann diesen Mechanismus nicht überschreiben, um z.B. Klassen aus dem Netz zu laden.
- hat keinen automatischen garbage collector, kein finalize() und keine Serialisierung.
- hat keine Schnittstelle für eigene Erweiterungen in einer anderen Sprache, kein Java Nativ Interface JNI.
- bietet keine Serialisierung, das heißt die persitente Speicherung von Instanzen muss von Hand implentiert werden.
- unterstützt keine „floating points“.
- hat nur simple Threads, keine Thread-Gruppen und keine „Daemon Threads“

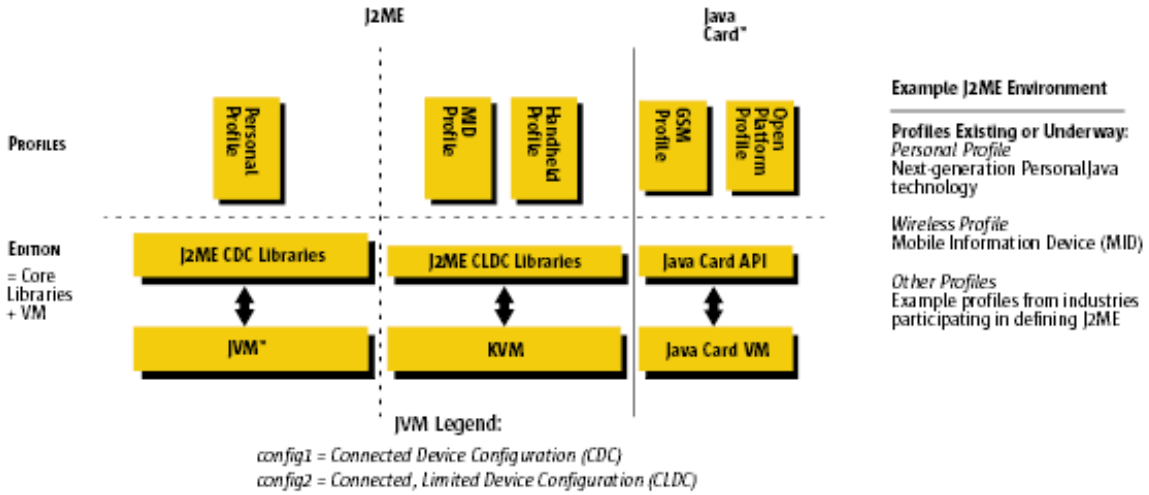

Abbildung 2.7: Java 2 Micro Edition (Quelle: Sun)

2.3.2 CLDC und MIDP

CLDC mit MIDP stellt zusammen eine API für Mobile Geräte dar. Diese API ist gegenüber der Standard Edition von Java J2ME sehr stark abgespeckt. CDLC stellt die Klassenbereiche `java.io`, `java.lang` und `java.util`, sowie `javax.microedition.io` zur Verfügung. MIDP erweitert die Klassen um die Bereiche `javax.microedtion.lcdui`, `javax.microedtion.midlet` und `javax.microedtion.rms`. Auch Klassen aus den von der J2SE bekannten `java.*` stellen nur ein subset, sowohl was die Klassen anbelangt, als auch deren Implementierung bzw. Methodenumfang.

In Anlehnung an die Applets nennt sich die zentrale Einstiegsklasse Midlet aus dem Packet `javax.microedtion.midlet`. Man braucht mindestens ein Klasse, die von Midlet abgeleitet ist. Sie kann sich in drei Zuständen befinden: aktiv, zerstört oder wartend. Eine Änderung der Zustände wird vom Gerät durch den Aufruf der entsprechenden Methoden: startApp(), destroyApp() und pauseApp() angezeigt. Beim Aufruf einer J2ME-Anwendung wird eine Instanz des Midlets erzeugt und befindet sich im Zustand ‘wartend´. Erst mit dem Aufruf von startApp() sollten entsprechend Aktionen durchgeführt werden.

2.3.3 Netzwerk

Die Klassen und Schnittstellen aus `javax.microedtion.io` stellen das Generctic Connection Framework dar. Sie sind für die gesamte Netzwerkkommunikation zuständig. Es gibt nur eine Klasse: `Connector` mit der, über zahlreiche open-Methoden, alle Verbindungen aufgebaut werden. Die Methode `Connection = Connector.open( String url)` liefert eine Klasse mit der Schnittstelle Connection zurück. Es handelt sich um eine Fabrikmethode [17], die je nach Art der Verbindung eine Klasse mit von Connection abgeleiteter Schnittstelle zurück liefert. `Connector.open("http://localhost")` liefert

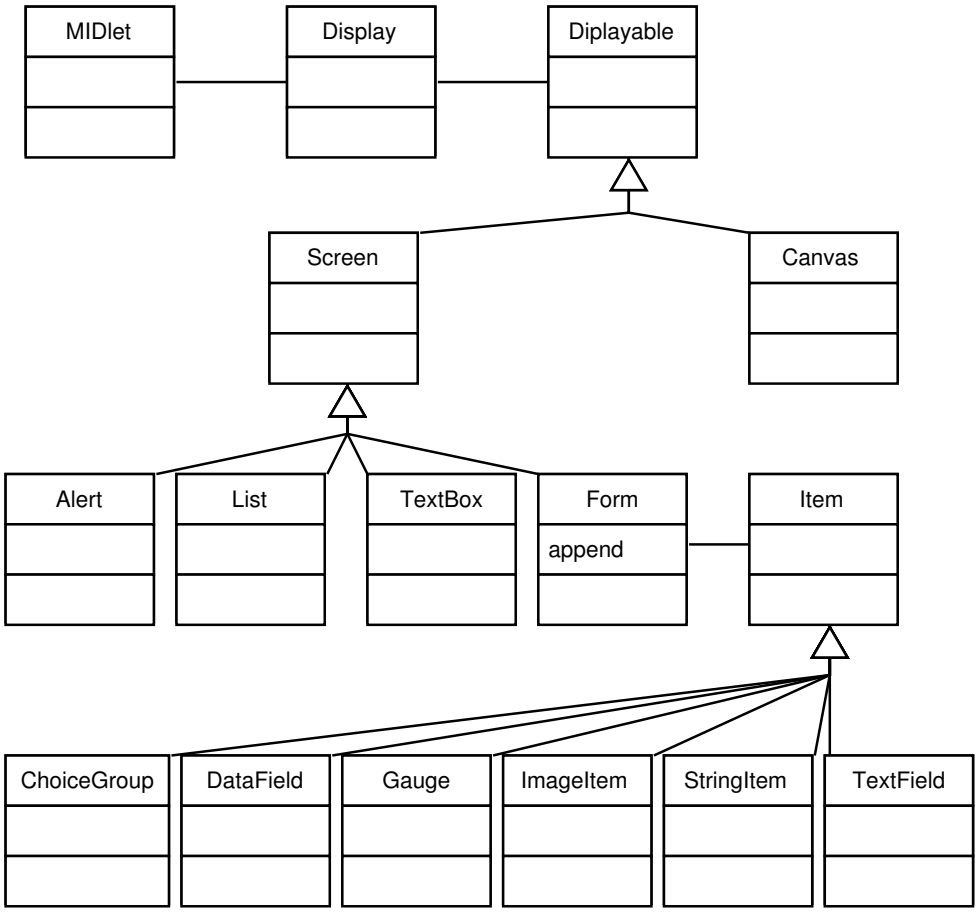

Abbildung 2.9: javax.microedition.lcdui

z.B. eine Klasse mit der Schnittstelle `HttpConnection`. Die Hierarchie ist in Abb. 2.8 dargestellt. Soll eine Anwendung auf einem Port horchen, dann steht in der url nur das Schema und der Port, z.B. `socket://:3000`. Es müssen zwar alle Klassen und Schnittstellen von den Herstellern implementiert werden, aber es wird als unterstützte Verbindungsart nur HTTP vorgeschrieben.

2.3.4 GUI

Auch für die grafische Ausgabe hat sich gegenüber der Java 2 Standard Edition fast alles geändert. Das Zusammenspiel der einzelnen Klassen ist in Abb. 2.9 dargestellt. Die Klassen für die GUI sind im Paket `javax.microedtion.lcdui` zusammengefasst. Es gibt zwei unterschiedliche Möglichkeiten den Bildschirm zu beschreiben. Zum einen über die Klasse `Canvas`, mit ihr ist es möglich den Bildschirm direkt über Pixel anzusprechen. Zum zweiten

über die Klasse `Screen`, sie bietet eine strukturierte Ausgabe, z.B. über Forms oder Listen an. Ein Mischen der beiden Bearbeitungsformen ist nicht möglich! Die Klasse `Display` übernimmt die Verwaltung, mit der statischen Methode getDisplay() erhält man die Instance der Klasse und kann dann mittelt `setCurrent(Displayble show)` das Canvas oder den Screen anzeigen. Eine Besonderheit stellen die Alerts dar, sie sind nicht von `Displayable` abgeleitet und sollen wie der Name andeutet Meldungen darstellen. Bis auf die Canvas, ist die Darstellung auf dem Bildschirm der Implementierung des Gerätes überlassen. Das bedeutet, das ein Formular auf zwei unterschiedlichen Fabrikaten zu verschiedenen Darstellungen führen kann. Diese abstrakte Möglichkeit, die Anzeige zu gestalten, soll den unterschiedlichen Displaygrößen und Farbtiefen Rechnung tragen. Wählt man für die Anzeige den Screen, so kann man diesem eine Liste,ein Textfeld oder ein Form zu weisen. Dem Form kann man dann die üblichen Felder wie Labels, Text, Listen, Bilder oder Auswahlfelder ein- oder hinzufügen siehe unterste Zeile in Abb. 2.9.

2.3.5 Speicherung

Das letzte Paket `javax.microedtion.rms` bringt die Schnittstelle zur Persistens von Daten. Auch hier ist die Schnittstelle sehr einfach gehalten und bringt wieder Neuheiten bei der Benutzung. Jeder Applikation wird ein eigener Bereich zum Speichern gegeben, der von den anderen Anwendungen nicht zugänglich ist. Aufgrund der einfachen Struktur ist nicht zu erwarten, dass diese Einschränkungen gebrochen werden können. Problematischer ist, dass nicht sichergestellt werden kann, dass die gespeicherten Daten durch Synchronisierung auf den PC gelangen oder auf einem Wechselmedium gespeichert werden, z.B. MMC (Multi Media Card). Deshalb sollte man bei sensiblen Inhalten entsprechende Maßnahmen ergreifen, z.B. durch vorheriges Verschlüsseln.

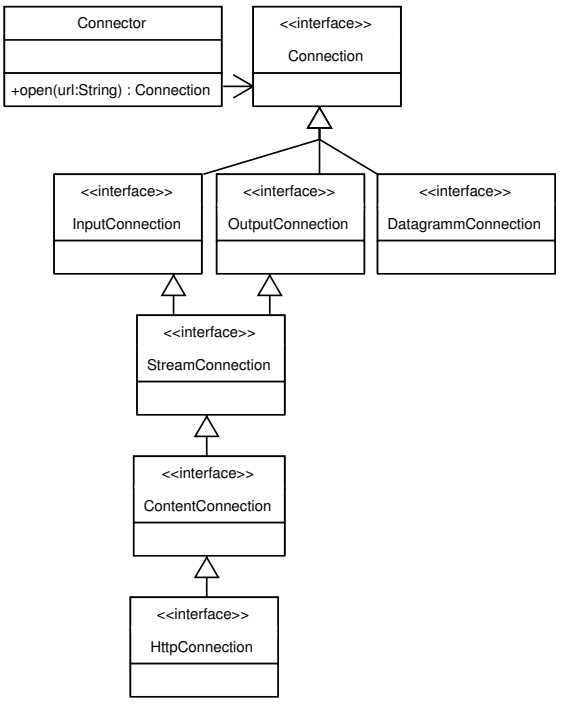

Abbildung 2.8: javax.microedition.io

2.3.6 Verschlüsselung

Sun hat mit JCA und JCE einen Standard geschaffen, der ab Java 2 fester Bestandteil der API ist. Leider steht diese Schnittstelle nur in der Standard und Enterprise Edition zur Verfügung. Es steht somit zwar keine einheitliche Schnittstelle bereit, aber es gibt aus der Zeit von Java 1.1 noch zahlreiche Kryptographie-Bibliotheken. Die Auswahl und die Benutzung von kryptographischen Paketen in der J2ME werden in Kapitel 3.6 besprochen.

Zukunft `http://www.jugs.ch/html/events/2002/J2ME_overview.pdf`
Symbian: J2ME ab 7.0, JavaPhone, PersonalJava ab 6.0

2.4 Hardware

Die erste Implementierung von J2ME mit CLDC und MIDP wurde im Auftrag von SUN für den Palm Pilot vorgenommen. Mittlerweile gibt es auch zahlreiche Handys, die Java unterstützen. Auch wenn es in der Werbung nicht explizit gesagt wird, handelt es sich ausschließlich um die Micro Edition von Java 2. Für den Motorola Communicator 9200 gibt es zusätzlich Picojava. Einige Hersteller sind mit den Möglichkeiten von J2ME noch nicht ganz zufrieden und haben eigene Erweiterungen eingebaut, die natürlich nicht zueinander kompatibel sind. Die erste Generation von Java-Handys wurde in [30] getestet.

In der Spezifikation wird für die reale Implementierung in einem Gerät unterteilt zwischen „muss“, „kann“ und „sollte“. Es müssen alle Klassen implementiert werden, aber nicht die komplette Funktionalität. Ein Beispiel: Es müssen von jedem Gerät HTTP-Verbindungen unterstützt werden, aber alle anderen Verbindungen, z.B. reine TCP-Verbindungen, auch Socket genannt, müssen nicht unterstützt werden. So führt dann folgendes Beispiel auf einem Gerät, welches keine Socket-Verbindung unterstützt, zu einer Exception:

```
1  try {
2    StreamConnection sc = Connector.open("socket://localhost
       :80");
3  } catch (ConnectionNotFoundException e) {
4    showAlert("Verbindung konnte nicht ögeffnet werden oder
       Verbindungsart wird nicht üunterstttzt!");
5  }
```

Eines der ersten Handys mit Java Unterstützung ist das Siemens SL45i, und sein kleiner Bruder M50 ist das erste lowcost Handy. Beide zeichnen sich durch die folgenden Eigenschaften aus:

- Max. 120 kB Speicher (Heap) pro Midlet. (150 kB beim M50)
- in Segmente zu 16 kB aufgeteilt, wichtig für Array und große Objekte
- zusätzliche API

Die zusätzliche Siemens-API ist für die geforderte Aufgabe nicht hilfreich. Einzig die Unterstützung von HTTPS-Verbindungen, welche in MIDP 1.0.2 hinzugefügt wurden, sind interessant für nicht Spieleentwickler.

In Zukunft sollte man Ausschau nach Handys mit Symbian OS halten, welche Java2ME ab der Version 7.0 anbieten [33]. Auch für Pocket PCs mit Windows CE gibt es Java2ME, z.B. von IBM [27] und alle PDAs mit Linux, z.B. der Zaurus von Sharp oder Yopy (http://www.invair.de/filewalker.html).

PDAs mit PalmOS und den Zaurus gibt es auch als Variante mit integriertem GSM-Modem. Den Z100 gibt es mit Microsofts Windows CE Abkömmling Stinger und Java Unterstützung. Für alle anderen mit Compact Flash-Slot gibt es GSM-Module. Dem Wunsch, von überall seine Bankgeschäfte zu erledigen, steht also nichts mehr entgegen.

Damit gibt es eine große Anzahl von mobilen Geräten, auf welchen die Anwendung ausgeführt werden könnte.

2.5 Verbesserungsmöglichkeiten

2.5.1 HBCI

Es macht in der Regel keinen Sinn, von einem etablierten und weitgehend getesteten System, auch wenn es ein wenig angestaubt ist, abzuweichen oder es ganz zu ersetzen. Auf der anderen Seite gibt es gute Gründe, zumindest über eine Ergänzung nachzudenken.

HBCI hat sich bis auf den deutschen Markt, trotz einiger Anstrengungen und der internationalen Ausrichtung, noch nicht einmal in Europa ausbreiten können. Auf der anderen Seite gibt es mit dem OFX [35] einen Standardisierungsversuch in den USA, der aber seine Schwächen vor allem in der Benutzerauthentisierung hat. OFX ist ein Bestreben, die bestehenden konkurrierenden Systeme IFX [25] und Gold zu vereinen. Mit einer weiteren Verbreitung dieser Standards, wird sich auch die Unterstützung für die zugrunde liegende Technik weiter verbreiten. In der aktuellen Version ist OFX und IFX von SGML auf XML umgestiegen. Das bisher verwendete Codierungsverfahren von HBCI ließe sich in XML überführen, unter Beibehaltung der Geschäftsvorfälle und Sicherheitsverfahren.

2.5.2 J2ME

Betrachtet man das Ganze aus Sicht von J2ME, ergeben sich weit mehr Argumente. Schon in den frühen Anfängen von HBCI wurde immer auch die Mobilität angesprochen. Und nicht nur der Werkstattbericht [46], sondern auch das ZKA, haben die Notwendigkeit einer mobilen Lösung zum Homebanking erkannt. Mit J2ME gibt es erstmals die Möglichkeit Handyübergreifend Software zu entwickeln, um Bankgeschäfte durchzuführen. Die Einschränkungen dieser Geräte in Bezug auf Speicher und Rechenleistung erfordern, möglichst viele Systemfunktionen zu benutzen, um die Programme kompakt und leistungsstark zu machen.

Die derzeitige Version MIDP 1.0, welche in den aktuellen Handys benutzt wird, enthält außer HTTPS noch keine Unterstützung für business-Anwendungen. Schaut man sich aber die zukünftigen Entwicklungen für J2ME auf dem Java Community Process [28] an, soll die nächste Version von MIDP 2.0 [36] auch einen XML-Parser beinhalten. Zudem findet man sowohl eine API für Kryptographie [5], als auch eine API für Webservices bzw. SOAP[5] [10].

Die ersten Firmen implentieren schon eigene Lösungen [16].

Aber auch wenn es viele Bestrebungen gibt J2ME zu erweitern und zu ergänzen, bleiben Wünsche übrig. Zu den Zielen von MIDP 2.0 zählen auch den mCommerce zu unterstützen. Darum wurden neben einem XML-Parser auch eine HTTPS Unterstützung eingebracht. Bei HTTPS-Verbindungen muss sich zwar der Server authentisieren, aber der Benutzer bleibt unbekannt. Ein direkter Zugriff auf die Verschlüsselungs-API bleibt dem Programmierer unter MIDP 2.0 leider verwehrt, es muss also wieder eine eigene lib benutzt werden.

Wünschenswert wäre auch ein universeller Zugriff auf eventuell vorhandene Erweiterungskarten, um dort Verschlüsselung, Signatur, etc. in Hardware durchführen zu können. Das Siemens SL45i hat z.B. einen SD-Card-Slot, welcher sich durchaus dazu benutzen ließe. Es gibt auch andere Erweiterungskarten, die unter Umständen in Handys Einzug halten können. Diese

[5] SOAP setzt auf XML auf.

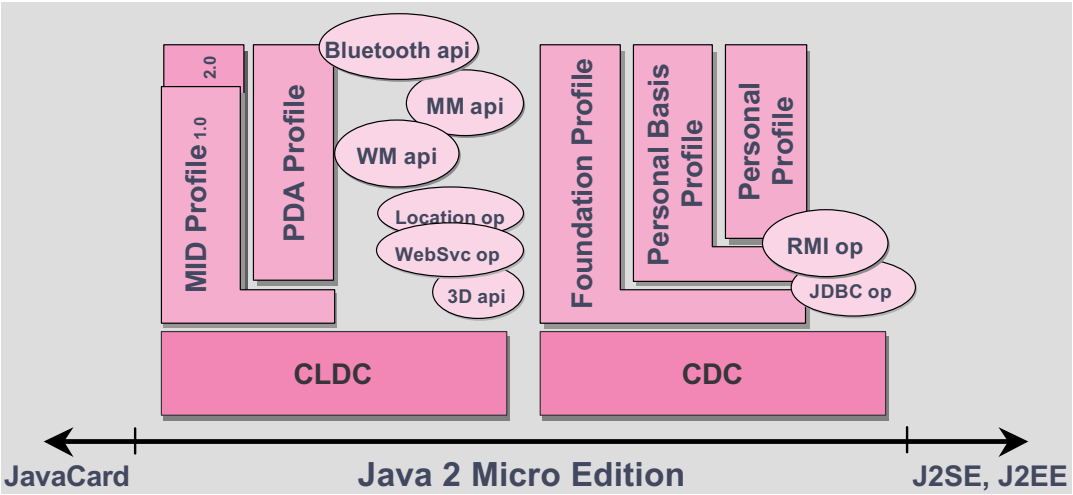

Abbildung 2.10: Die Zukunft von Java 2 Micro Edition

ließen sich zur Verbesserung der Sicherheit einsetzen. Einen Überblick der Karten liefert ein Artikel in der c't [20] [19].

Die beste Möglichkeit den Benutzer zu identifizieren ist allerdings in jedem Handy schon vorhanden. Über die SIM-Karte werden die Benutzer identifiziert. Mit dem SIM Applikation Toolkit (SAT) [45] [41] gibt es sogar schon eine genormte Schnittstelle, für die schon ganze Bankanwendungen geschrieben wurden [39]. Leider gibt es kein Bestreben, diese unter Java 2 Micro Edition zugänglich zu machen.

Der letzte Wunsch auf der Liste ist neben den gravierenden Mängeln, nur Kosmetik. J2ME hat in seinem Bestreben Ballast abzuwerfen, leider vergessen, dass es zur Zeit einen weiteren Mangel bei Handys gibt, nämlich die Bandbreite. Obwohl JAR-Archive den ZIP-Algorithmus benutzen und somit jede Implementierung diesen unterstützen muss, wird keine Schnittstelle zur Verfügung gestellt. XML lässt sich wegen seines in der Praxis kleinen Zeichensatzes hervorragend komprimieren.

Zusammenfassend

Für Java wären folgende Erweiterungen wünschenswert:

- Krypto-API soll ein Bestandteil von MIDP 2.0 sein, und keine zusätzliche API, um eine hohe Marktdurchdringung zu erhalten.

- Eine Klasse ZIPStream ist hilfreich und stellt dem gegenüber keinen hohen Aufwand dar, da der Algorithmus schon für das auspacken des JARs vorhanden ist.
- Eine Verbindung zu bekannten Schnittstellen der Handys oder SIM-Karten, wie z.B. SAT-API und Card-Kontroller-API, können die Sicherheit und Geschwindigkeit erhöhen.

Für HBCI sind zumindest optional, folgende Veränderungen zu bedenken:

- Umstieg auf XML/SOAP mit optionaler ZIP-Kompremierung
- Transport über ssl oder Verschlüsselung mit Triple-DES und Signierung RSA/SHA

Die Suche nach der gewünschten Verbesserung für HBCI brachten unter anderem diese beiden Quellen zum Vorschein:

1. XML sei ursprünglich für HBCI Version 4.0 geplant [11].
2. Aber die Fragen und Antwort-Seite spricht eventuell schon von Version 3.0[6] [15], das würde auch die Verzögerung erklären:

   „Thema: XML-basierte HBCI-Nachrichten
   Warum wird das Nachrichtenformat XML nicht durch HBCI unterstützt? - Sind für die Zukunft auch XML-basierte HBCI-Nachrichten geplant?
   Als HBCI entwickelt wurde, stand XML noch nicht zur Diskussion. Im Rahmen der HBCI-Version 3.0 wird diskutiert, ob man von der EDIFACT-angelehnten Syntax auf XML übergehen sollte. Konkrete Aussagen sind heute aber leider noch nicht möglich."

PIN/TAN

Die Wiedereinführung von dem PIN/TAN-Verfahren halte ich nicht für ein gute Idee. Der Verwaltungsaufwand scheint vielleicht für die Banken ähnlich zu sein, der Kunde muss sich aber immer rechtzeitig um neue TANs bemühen bzw. ihm werden diese zugeschickt. Und diese Zettel muss er griffbereit, aber für andere unzugänglich halten. Insbesondere bei mobilen Geräten ist die Aufbewahrung der TANs in der Brieftasche ein Sicherheitsrisiko. Bei einer alternativen Verwaltung der TANs im Endgerät selbst, halte ich die Lösung von Zertifikaten für zweckmäßiger.

Auch wenn es nicht direkt zu diesem Thema gehört, ist es doch ein Verbesserungsvorschlag, der die Sicherheit erhöhen würde. Die Initiative zur Einrichtung einer zentralen Rufnummer, bei der man alle verloren gegangenen Karten, etc. melden könnte [42]. Das würde dazu führen, das schneller reagiert werden kann, wenn wirklich mal das Handy oder der PDA geklaut wird. Wer kann sich schon alle Rufnummern merken, die im Verlustfall angerufen werden müssen.

[6] Die Vorversion 3.0 ist jetzt öffentlich zugänglich und benutzt weiterhin ausschließlich die bisherige Trennzeichensyntax

2.6 Anwendungs- und Test-Umgebung

Bei HBCI ist die Version 2.2 zur Zeit aktuell, sie wird für die Entwicklung zu Grunde gelegt. Bei Java ist es die Java 2 Micro Edition mit dem CLDC 1.0 und dem MIDP 1.0. Es werden keine weiteren Profile angenommen. Das Referenzsystem ist das J2ME Wireles Toolkit JWTK, es diente als Testplattform in der Version 1.0.3 und später in der Version 1.0.4. Es lassen sich auch Emulatoren von anderen Herstellern einbinden. Nach einer kostenlosen Registrierung erhält man Emulatoren für alle javafähigen Handys von Siemens und Nokia und auch den Palm-Emulator.

Als Entwicklungsumgebung habe ich hauptsächlich Sun ONE Studio Mobile Edition[7] benutzt, es bietet eine sehr gute Unterstützung des JWTK an. Die Refaktorisierung habe ich, wegen der besseren Unterstützung, unter Eclipse[8] gemacht, dafür wird das JWTK nur mangelhaft angebunden.

Als Gegenstelle diente der Test-Server von HBCI-Kernel. Nach einer kostenlosen Registrierung erhält man die Zugangsdaten per Post zugeschickt. Der Test-Server bietet eine vollständige Implementierung von HBCI 2.01 bis 2.2 für die gängigsten Geschäftsvorfälle an. Es werden Überweisungen sowie Salden und Umsatzanfragen simuliert. Damit bietet er alles notwendige zum Überprüfen des HBCI-Dialoges.

[7] Forte for Java ist der alte Name

[8] Entwicklungsumbegung von IBM

3 Design und Realisierung

Warum ist Design und Realisierung in einem Kapitel? - Darauf gibt es zwei Antworten: Zum einen ist ein Teil des Designs während der Implementierung entstanden bzw. hat sich verändert. Zum zweiten haben sich Design und Implementierung abgewechselt. Was damit gemeint ist, werde ich im nächsten Abschnitt vertiefen. Die Abschnitte laufen nicht immer in einer chronologischen Reihenfolge. Der Abschnitt Netzwerk war z.B. als erstes fertig, er befindet sich aber am Ende, da er sich mit der Programmierung der Netzwerk-Schnittstelle befasst.

3.1 Vorgehen

Für die Entwicklung der Software habe ich mich verschiedener Techniken bedient. Zum einen bei den Prototypen von Floyd [13], bei den Entwurfsmuster [18] und von XP habe ich KISS[1], UnitTests und Refaktorisierung[2] genommen. Für die Beschreibung der Vorgehensmodelle möchte ich auf meine Studienarbeit und die dort angegeben Quellen verweisen [9].

Ungewöhnlich erscheint, das mischen von englischen und deutschen Begriffen bei Konzeption und Programmierung. Bei HBCI handelt es sich hauptsächlich um einen deutschen Standard, bei selbsterklärendem Sourcecode erschien es mir sinnvoll, die Klassen und Methoden entsprechen zu benennen. Auf der anderen Seite, haben sich in der Entwicklung viele englische Begriffe etabliert, zusätzlich kommen Richtlinien durch, z.B. „setter“ und „getter“-Methoden. Dass führt manchmal zu Namen, wie `getSignaturId()`. Ich denke, dass dies trotzdem dem besseren Verständnis beträgt.

Nachdem feststand, dass das Programm die HBCI-Spezifikation Version 2.2 und Java2ME benutzt, habe ich mich mit drei Fragen beschäftigt:

1. Was sind die Anforderungen?
2. Welche Einschränkungen kann ich machen?
3. Welche Risiken gibt es?

Die Anforderungen und die Einschränkungen werden im übernächsten Abschnitt 3.3 behandelt. Finanzielle und politische Risiken kann ich vorerst vernachlässigen. Es bleibt, dass es nicht funktionieren kann oder nicht innerhalb der Zeit zu realisieren ist. Während der Voruntersuchung (Recherche) zeigten sich zwei Probleme. Einige Handys sind nicht in der Lage

[1] keep it stupidly (and) simple

[2] Englisch: refactoring, wird normalerweise mit Umstrukturierung übersetzt. Ich benutzte lieber das weniger geläufige Wort Refaktorisierung, um die Philosophie auszudrücken, die dahinter steckt.

eine TCP-Verbindung, ohne HTTP, aufzubauen. Zum zweiten ist bei MIDP keine Verschlüsselung enthalten. Die Suche nach möglichen alternativen kryptographischen Bibliotheken wird im Abschnitt 3.6 dargestellt. Begleitend zur Voruntersuchung sollten drei Prototypen mehr Klarheit bringen, welche im Abschnitt 3.2 beschrieben werden.

Nachdem alle kritischen Aspekte, bis auf HBCI, geklärt waren, ging es mit einer detaillierteren Analyse weiter. Dabei ging es um die Identifizierung und das Zusammenspiel der teilnehmenden Paketen und Klassen. Dabei war mir wichtig, dass die Kryptographie austauschbar ist, oder sogar das HBCI-Protokoll durch ein anderes ersetzbar wäre. Das Ergebnis ist im Abschnitt 3.4 zu sehen.

Auch wollte ich weitere Erfahrungen mit den Techniken der evolutionären Softwareentwicklung machen. Diese ist dem bekannten „eXtreme Programming“ ähnlich, aber nicht so restriktiv was die Praktiken angeht. Zusätzlich wollte ich einige Techniken benutzten, welche XP so effizent machen. Kent Beck, der als der Vater von XP gilt, hat einmal gesagt, dass ein Vorgehensmodell nicht dem Selbstzweck gilt, sondern das Projekt fördern soll.

Ein Leitsatz von Kent Beck hatte es mir noch angetan: „make it work, make it right, make it fast“[3], der für mich durch einige „Core Practices“ erklärt. Ich möchte herausstellen, dass ich nicht nach XP vorgegangen bin, sondern nur einige von XP bekannte Sprachbegriffe benutze. Ich habe die folgenden Praktiken eingesetzt:

- Design für heute, es wird nur die gebrauchte Funktionalität implementiert.
- Kurze Iterationen.
- Fortlaufende Integration, d.h. ist ein Teil fertig, wird er eingebaut.
- Refaktorisierung.
- Testgetriebene Programmierung. Benutzen wollte ich UnitTests. Diese sind leider im Keim gescheitert, dazu wird in 3.9 eingegangen. Es gab zwar kein Framework, ich habe aber trotzdem zahlreiche Testfälle geschrieben.

Die Testfälle tragen zum Verständnis der Programme bei. Sie ergänzen die Dokumentation und Diagramme ebenso wie selbsterklärender Quelltext oder die mit javadoc generierten Beschreibungen.

Wie schon erwähnt, dienten die „Design Pattern“ [18] als Vorlage. Ein Beispiel für das Zusammenwirken der Techniken ist die Implementierung des Generators der HBCI-Nachrichten. Das erste Modell ist durch die Vorlage eines Entwurfsmusters entstanden. Das Muster wurde an die geringe Komplexität angepasst. Und bei der Implementierung, führte ein neueres Verständnis zur erneuten Anpassung des Designs.

Die HBCI-Spezifikation umfasst 768 Seiten. Die evolutionäre Entwicklung ist eine Methode, sich einzelnen Teilmengen zu nähern. Eine Redensart sagt „Learning by doing“. Nachdem das Ziel feststand und die Komponenten mit ihren Aufgaben, nicht mit den Methoden[4], identifiziert waren. Habe ich eine Liste der Iterationen gemacht. Die Iterationen sind so ausgelegt, dass wenn eine neue zentrale Klasse hinzukommt, geringe neue Funktionalität hinzukommt.

[3] Dieser Satz ist oft zitiert, die urprüngliche Quelle ist mir bisher verborgen geblieben.

[4] Eine Möglichkeit, wie man die Klassen findet, wird in [14, Kap. 4.5 Klassenkarten] mit CRC-Karten beschrieben.

3.2 Drei Protypen (mit neuen Erkenntnissen)

Um die Machbarkeit zu untersuchen und um mich mit der Technik vertraut zu machen, habe ich vor Beginn zwei experimentelle und einen explorativen Prototyp erstellt. Ich habe keine Mühe in das Design gesteckt, da es sich um Wegwerfprodukte handelte. Als Vorlage dienten die Demoanwendungen, welche mit dem J2ME 2 Wireless Toolkit mitgeliefert werden.

HttpDemo wurde zum Netzwerk-Prototypen. Gesendet wurde eine statisch, vorher zusammengestellte Dialoginitialisierung. Im Gegensatz zu Http, darf die Verbindung nicht nach dem Erhalt der Antwort abgebaut werden. Das Programm ist einfach, es lässt sich aber mit seiner Hilfe feststellen, ob ein Handy Socket-Verbindungen zulässt und wie dies auf der Handyoberfläche signalisiert wird.

Beim zweiten Prototyp wollte ich mich weiter mit den Klassen von J2ME vertraut machen. Er stellt zudem eine Studie über eine mögliche GUI der Software dar, auf deren Basis über Bedienbarkeit und Vereinfachungen diskutiert werden kann. Einige definierte Kommandos werden nicht dargestellt und werden erst über einen Menü-Button sichtbar. Man kann sich nun überlegen, ob man einige Funktionen nicht über eine Auswahlliste anbietet statt über diese Kommandos.

Der dritte Prototyp ging um das Thema Verschlüsselung. Basis waren die Examples von Bouncy Castle und die DemoSuite von Suns JWTK. Ergebnis war, dass schon auf den Emulatoren[5] insbesondere die RSA-Verschlüsselung zu langsam ist. Auch nur die Größe dieses Prototyps ließ vermuten, dass es nicht auf alle Handys passen würde. Die Gründe, warum es sinnvoll ist die Entwicklung fortzusetzen, werden in der Vorbereitung und Nachbereitung erläutert.

3.3 Anforderungen

Die Anwendung ist für das Handy ausgelegt, deshalb wird, wenn nicht anders erwähnt, von einem Mobiltelefon als Gerät ausgegangen.

Das Ziel der Diplomarbeit ist es, einen HBCI-Client zu entwickeln und zu realisieren, der auf jeder beliebigen „J2ME-Virtuell Machine“ läuft. Da die Geräte sehr unterschiedlich sind, müssen einige Einschränkungen gemacht werden:

- Es werden ausschließlich HBCI-Server, welche das HBCI-Protokoll Version 2.2 verstehen, unterstützt. Referenz ist der Test-Server von PPI.
- Der Client unterstützt nur die notwendigsten HBCI-Nachrichten.
- Das Gerät muss die J2ME-API des CLDC 1.0 und MIDP 1.0 unterstützen. Außerdem muss es Socket-Connections zulassen, obwohl dies nicht zu den geforderten Merkmalen von MIDP (1.0 und 2.0) gehört.
- Das Gerät muss der VM genügend Speicher zur Verfügung stellen. Der benötige Speicher wird aufgrund der umfangreichen Krypto- und HBCI-API die Mindestanforderungen von MIDP 1.0 überschreiten.

[5] Welche nach Herstellerangaben schneller als die realen Geräte sind

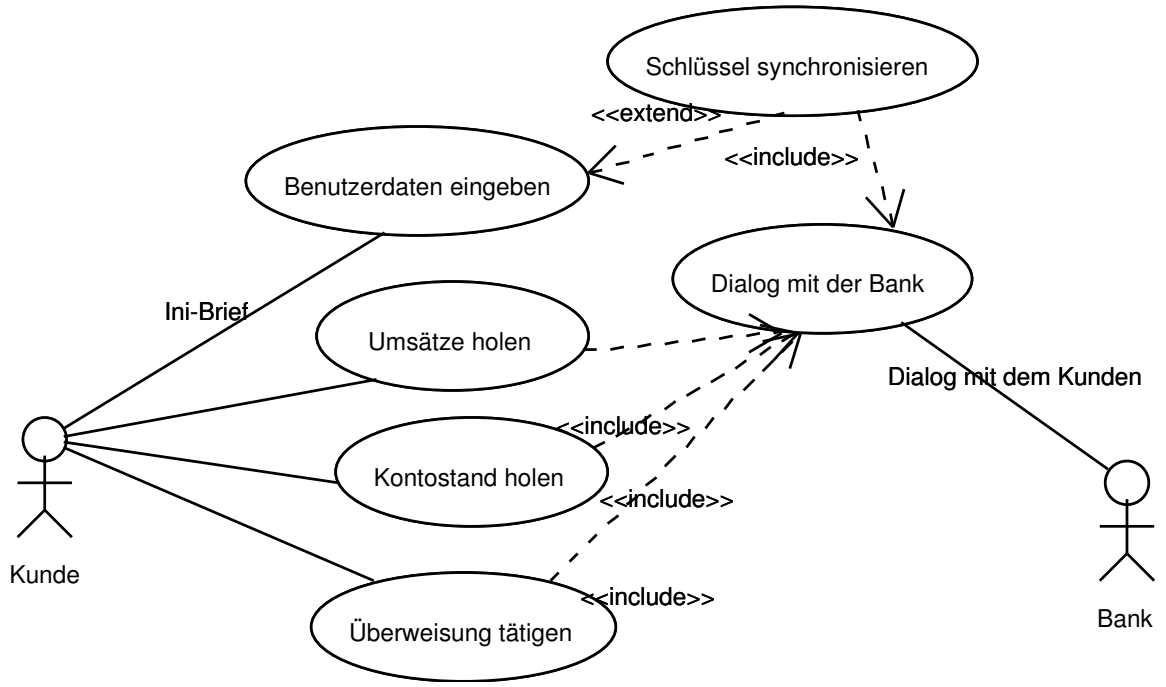

Abbildung 3.1: Anwendungsfälle für den HbciDialog

Die Anforderungen an das System aus technischer Sicht ergeben sich zum Teil schon aus den Vorgaben, die in dem Kapitel Anwendungs- und Test-Umgebung diskutiert wurden. Aus Benutzersicht sind diese aufgrund der einfachen Zielsetzung schnell beschrieben.

Aus Benutzersicht

- KISS (keep it stupid simple), in diesem Fall das Programm soll vor allem leicht zu bedienen sein.

Am besten ist es, der Kunde erhält seinen aktuellen Kontostand, nachdem er sich identifiziert hat, z.B. durch Eingabe seiner PIN. Leider sind dafür noch einige Daten nötig, die man z.B. mit einem „Wizard“ Schritt für Schritt abfragen könnte, wenn diese noch nicht vorhanden sind. Zusätzlich sollte es möglich sein, nach der Identifizierung die Kontodaten zu ändern. Zum Zweiten wird die Bedienung durch die Möglichkeit mit HBCI mehrerer Konten und Banken zu verwalten, komplizierter. In dieser ersten Version wird nur ein Konto unterstützt. Wenn mehrere Konten verwaltet werden, sollte die Auswahl direkt nach der Identifizierung erfolgen.

Aus technischer Sicht Wie oben schon erwähnt ist ein Teil der Anforderungen schon durch die Vorgaben gesetzt:

- Die Anwendung soll auf möglichst jedem Handy laufen, welches J2ME mit CLDC 1.0 und MIDP 1.0 erfüllt. Wie sich später zeigen wird, ist dies leider nicht ganz zu erfüllen,

da die Verschlüsselung alleine schon sehr viel Speicher verbraucht und somit das Limit von MIDP 1.0 Geräten übersteigt. Allerdings bieten die meisten Geräte der Anwendung mehr Speicher zur Verfügung an.

- Die Anwendung soll den HBCI-Standard 2.2 soweit vollständig implementieren, wie es zur Erfüllung der gestellten Aufgabe notwendig ist. Das heißt, dass DTAUS nicht unterstützt wird, da es für Sammel-Überweisungsaufträge benutzt wird, welche üblicherweise nicht im mobilen privaten Bereich gebraucht werden. Außerdem wird nur ein Sicherheitsverfahren, das RDH-Verfahren, unterstützt.

- Verschlüsselte Speicherung der Daten, s. 3.3.1.

3.3.1 Zugangsschutz

Um die Sicherheit der auf dem Handy gespeicherten Informationen und Schlüssel zu gewährleisten, sind einige Maßnahmen zu ergreifen. Der Zugang zur Software wird erst nach der Identifizierung gewährt. Bei den aktuell technischen Möglichkeiten ist dies mit einem Passwort oder einer PIN zu erreichen.

Aufgrund der auf Ziffern ausgelegten Tastatur, ist es bei Handys üblich eine PIN statt eines Passwortes zu verwenden. Es ist zwar auch möglich Buchstaben einzugeben, da aber die Zeichen aus Sicherheitsaspekten nicht angezeigt werden sollten, führt dies bei der Eingabe zu Problemen. Insbesondere, wenn der Benutzer Eingabehilfen[6] für SMS gewohnt ist, ist nicht sichergestellt, welches Wort eingetippt wurde. Bei PDAs ist ein Passwort vorzuziehen.

Dieses Passwort wird dann zur Verschlüsselung der Daten, der Informationen und der Schlüssel benutzt, bevor diese gespeichert werden. Ein Verlust des Passwortes, führt damit auch zum Verlust der Zugangsdaten. Um ein Erraten der Schlüssel zu verhindern, könnten die nachfolgenden Maßnahmen die Sicherheit erhöhen:

- Wird das Passwort zweimal falsch eingegeben, beendet sich die Software.

- Wird das Passwort insgesamt zehn mal hintereinander falsch eingegeben, werden alle Daten gelöscht.

- Beim Aufruf wird angezeigt, wie oft das Passwort falsch eingegeben wurde.

In Zukunft könnte die Identifizierung über biometrische Verfahren erfolgen. Siemens hat bisher nur eine Studie eines UMTS-Handy mit Fingerabdruck-System durchgeführt[7]. HP ist mit einer Produktankündigung für dieses Jahr schon einen Schritt weiter. Der iPAQ H5400 soll einen Fingerabdruck-Scanner eingebaut haben[8].

3.4 Überblick

Auf der Suche nach den ersten Klassen kamen zuerst: Kunde, Bank und Konto. Eine GUI gibt es auch, sie repräsentiert aber keine Klassen. Im Hinblick auf eine HBCI-Version auf

[6] Bei Nokia heißt sie z.B. T9 (automatische Worterkennung)

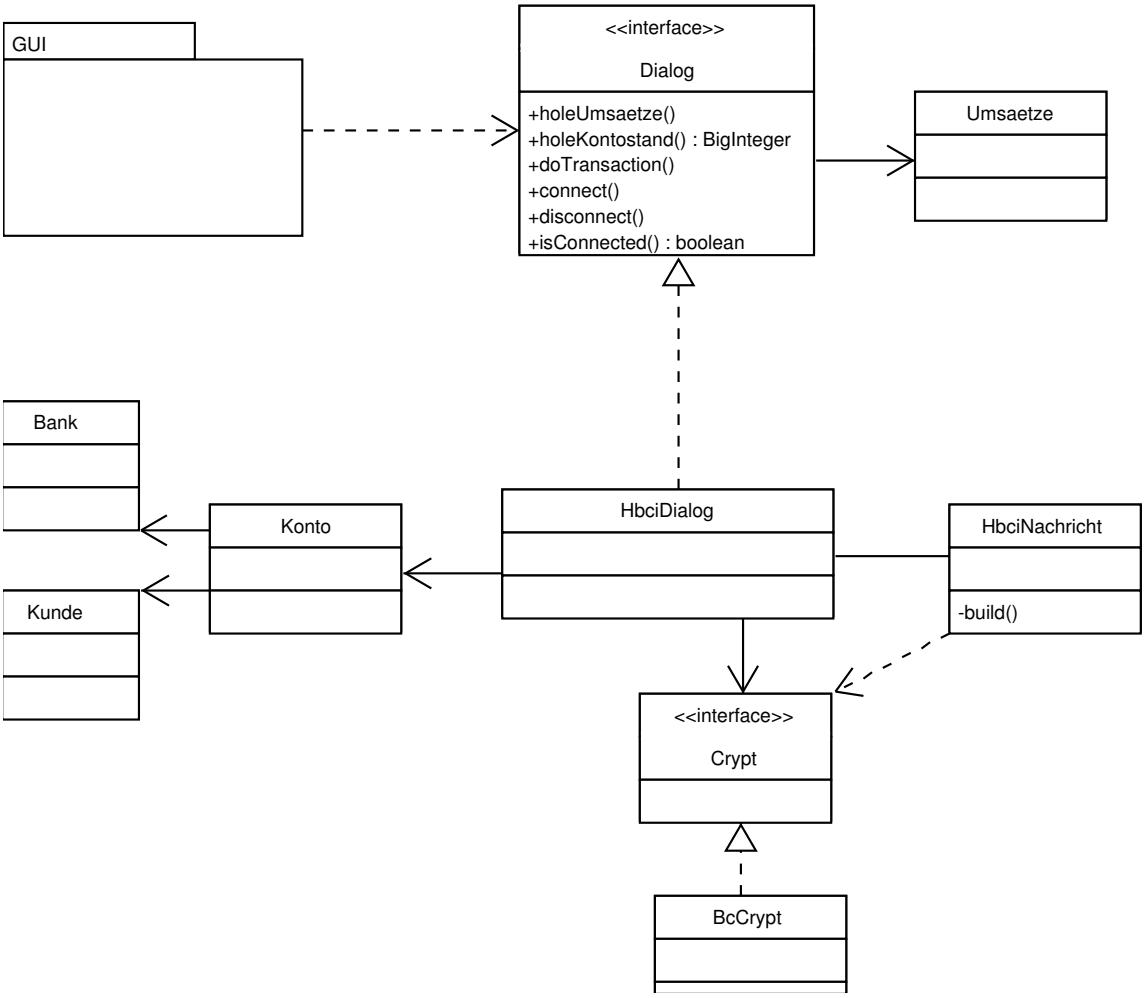

Abbildung 3.2: Die Übersicht der beteiligten Klassen. Über HbciDialog wird das Zusammenspiel koodiniert.

XML Basis, wollte ich für alles, was mit dem Protokoll zu tun hat, eine lose Kopplung, um es leicht austauschen zu können. Für diese Schnittstelle habe ich das Interface Dialog eingefügt. Die Anforderungen ergeben sich aus denen des Kunden. Die GUI soll möglichst einfach zu bedienen sein, weshalb sich für die Schnittstelle Dialog Methoden ergeben, wie `holeKontostand()`. Als Realisierung folgt dann die Klasse HbciDialog. Sie übernimmt auch die zentrale Verwaltung, wodurch die anderen Teile unabhängig voneinander und damit flexibel bleiben. Auch die Kryptographie sollte austauschbar sein, weshalb die Schnittstelle Crypt mit der Realisierung BcCrypt hinzukam.

Das Generieren einer Nachricht ist einer der ersten Iterationspunkte und wird deshalb mit aufgeführt. Darüber, wie die Nachrichten interpretiert werden, war ich mir zu diesem Zeitpunkt noch nicht schlüssig. Es spielt aber auch keine Rolle, da nur die Anpassung der Klasse HbciDialog notwendig ist. Alle Rückmeldungen werden in Container-Klassen geliefert, z.B.

Umsätze.

Um eine geringere Komplexität zu erhalten, habe ich mich entschlossen, die Datenstrukturen für den Generator und den Parser vorerst getrennt zu entwickeln.

3.5 Iterationen

Die evolutionäre Entwicklung erweitert das Programm in Intervallen um zusätzliche Funktionalität. Bei der Reihenfolge der Implementierungsschritte habe ich darauf geachtet, dass bei größeren Umstellungen in der Struktur, die Erweiterung der Funktionalität geringer ist. Wenn z.B. durch Refaktorisierung Veränderungen vorgenommen wurden, wird dies durch die Tests überprüft. Wird dagegen das Programm erweitert, dann werden zuerst neue Testfälle implementiert, bevor neue Funktionen hinzukommen. Es wechseln sich Erweiterung und Refaktorisierung ab, was einen gleichbleibenden Aufwand darstellt. Entsprechend den Zielen habe ich diesen Iterationsplan aufgestellt:

1. Unverschlüsselte Nachrichten (Version 0.1)
   a) Erzeugen Dialoginitialisierung <Nachrichten-Generator entwickeln>
   b) Erzeugen Dialogbeendigung
   c) Auslesen und Anzeigen der Institutsnachrichten (HIKIM)
   d) Speichern der Institutsnachrichten ()
   e) Anzeigen und Löschen gespeicherter Institutsnachrichten
   f) Lesen und Speichern der BDP <Nachrichten-Parser entwickeln>
   g) Lesen und Speichern der UDP
2. Verschlüsselte Nachrichten (Version 0.2) <BcCrypt entwickeln>
   a) Ini-Brief Eingabe, umwandeln in ein Zertifikat und mit dem Passwort verschlüsselt speichern. <Konto und Bank verfeinern>
   b) Salden-Abfrage
3. Fehlerbehandlung vervollständigen
4. Umsatzabfrage (Version 0.3)
5. Einzelüberweisung (Version 0.4)
6. Mögliche weitere Iterationen nach Bedarf:
   - Daueraufträge/Zeitgebundene Einzelaufträge bearbeiten
   - Überarbeiten der Oberfläche, Benutzung von kAWT oder MWT
   - Aktienhandel

- Client auf Basis von XML (evtl. SOAP) und HTTPS (dies ist ein neues Projekt, Teile dessen können für die Middleware benutzt werden.)
- DDV

Innerhalb der Diplomarbeit soll die Version 0.2 implementiert werden. In den spitzen Klammern stehen die Hauptklassen, die hinzukommen.

3.6 Auswahl der Kryptographie-API

3.6.1 Kriterien

Die Kriterien für die Wahl einer Bibliothek ergeben sich natürlich hauptsächlich durch die von HBCI geforderten Algorithmen. Es sollten möglichst alle unterstützt werden. Die Kodierungen und Modi haben dabei die niedrigste Priorität. Sie sollte mit wenig Aufwand unter J2ME zu benutzen sein. Das Lizenzmodell spielt auch eine Rolle, wichtiger ist aber das Vertrauen in die Korrektheit der Implementierung. Dabei kann ein offener Sourcecode helfen. Hier die Anforderungen im Überblick:

- Benutzbar ohne Lizenzgebühren
- Symmetrische Verschlüsselung: 2-Triple-DES, CBC-Block-Mode, ISO 26129-2 Padding, Secure Key Generation[7]
- Signatur für RDH: RSA, RIPEMD-160, ISO 9796-1 Padding
- Schlüsselübermittlung für RDH: RSA, ISO-10126
- Signatur DDV: 2-Key-Triple-DES, CFC-Block-Mode, MAC, RIPEMD-160
- Schlüsselübermittlung DDV: 2-Key-Triple-DES, ECB-Block-Mode.
- Java2ME: direkt für J2ME geschrieben oder leicht gewichtige API (light weigth api), für eine schnelle Anpassungsfähigkeit an J2ME. Geringe Größe bzw. modular, man sollte nur benötigte Klassen benutzen können. Und wie immer schneller und kleiner heap[8].
- Saubere Implementierung: Wenn die Bibliothek fehlerhaft arbeitet, nützen die obigen Kriterien nichts, da dann eine zeitaufwendige Nachbereitung notwendig ist.

[7] Beinhaltet in erster Linie einen Zufallsgenerator, welcher von außen nicht berechenbare Zufallszahlen liefert, siehe 2.2.6.

[8] Speicherverbrauch zur Laufzeit

3.6.2 Teilnehmer

Man findet bereits zahlreiche Libraries. Die nachfolgende Auswahl erhebt keinen Anspruch auf Vollständigkeit. Es ist eine Liste derer, die mir aufgefallen sind[9]:

- jHBCI™[21] von Uwe Günther [22]: Entstanden für seine Diplomarbeit, enthält alle notwendigen Algorithmen sind implementiert, JCE Schnittstelle. Lizenzmodell: LGPL.
- Bouncy Castle Crypto API [31]: Sehr umfangreiche Bibliothek aber schlecht dokumentiert, sie beinhaltet auch alle notwendigen Algorithmen zusätzlich AES und ElGamal, sowohl JCE als auch eine leicht gewichtige API für J2ME geeignet. Lizenzmodell: MIT.
- Die Technische Universität Graz (TUG) entwickelte im Institute for applied information processing and communications (IAIK) eine Reihe von Java Kryptographie-Bibliotheken. Unter anderem auch zwei Produkte für die Java 2 Micro Edition:
  - Das Produkt IAIK-JCE ME, welche eine schnelle Portierung von JCE basierten Programmen verspricht.
  - Und das Produkt IAIK-iSaSiLk ME, sie ist auf https, mit TLS 1.0 and SSL 3.0, ausgelegt. Damit hat sie TRiple-DES und RSA implementiert.

  Mit den Entwicklerlizenzen ab 500 Euro kam sie für die Diplomarbeit nicht in die weitere Betrachtung.
- logi.crypto [37]: Ist für Java 1.1, enthält alles bis auf RIPEMD-160, die letzte stabile Version ist vom 10/2000, ein Entwickler. Lizenzmodell: GPL.
- Die Firma Cryptix hat auch ein Produkt, mit dem gleich lautenden Namen: Cryptix 3 ist eine JCE 1.1 Implementierung. Die Benutzung unterliegt der an der Berkeley License angelehnten Cryptix General License: „... Our code is covered under the Cryptix General License unless stated otherwise. The Cryptix General License is a straight copy of the widely known Berkeley License (with a different copyright holder obviously). [...]“.
- Eine Bibliothek möchte ich hier außer Konkurrenz vorstellen, da ich sie erst am Ende der Diplomarbeit gefunden habe, und sie deshalb nicht in die Bewertung mit einfließt: BeeCrypt [4] „BeeCrypt for Java is a JCE 1.2 compatible Cryptographic Service Provider (CSP)“. BeeCrypt ist eine aktiv, kommerziell gepflegte Bibliothek.

3.6.3 And the winner is: Bouncy Castle

Einige der Pakete unterstützen nur JCA (Java Cryptography Architecture) und JCE (Java Cryptography Extension), welche unter MIDP nachgebildet werden müssen. Das hätte zwar den Vorteil, dass die „Cryptographic Service Provider“ leicht auszutauschen sind, es stellt aber einen erheblichen Programmier- und Speicheraufwand dar. Meine erste Wahl wäre in diesem Fall jHBCI gewesen.

[9] Produkte mit zu hohen Preisen sind gleich durch das Raster gefallen

„Bouncy Castle Crypto API“ gibt es auch für J2ME. Die API wird aktiv von einer Entwicklergemeinde gepflegt. Die Entwicklung der benötigten Algorithmen ist seit einiger Zeit abgeschlossen. Eine rege Diskussion, offener Quellcode und die Benutzung in mehreren Projekten, haben zu einer stabilen, vermutlich fehlerfreien Bibliothek geführt. Die mangelnde Dokumentation wird durch Beispielprogramme ein wenig kompensiert. Auch die klare Struktur der API hilft bei der Einarbeitung. Ein weiterer Vorteil ist das offene Lizenzmodell. Es erlaubt mir nur wirklich benötigte Klassen zu verwenden, dadurch wird die Anwendung kleiner, und es lassen sich benötigte Anpassungen unproblematisch vornehmen.

3.7 HBCI

Auch beim Generieren und Interpretieren der HBCI-Nachrichten war der erste Ansatz, eine vorhandene Bibliothek zu benutzen. Die Chancen standen auch nicht schlecht, obwohl HBCI im Gegensatz zur Verschlüsselungstechnik ein lokales Problem ist. Denn der HBCI-Kernel stellt auch eine Java-Variante kostenlos zur Verfügung [23]. Der HBCI-Kernel stellt eine komplette Implementierung des HBCI-Standards Version 2.2 dar. Inklusive der Parser für DTAUS und S.W.I.F.T. und der Abwärtskompatibilität zu 2.01. Doch leider ist das Paket für die gestellte Aufgabe zu aufgebläht. Entscheidener ist aber der Umstand, dass der HBCI-Kernel nicht im Quellcode geliefert wird, so dass ein Anpassung und Extraktion der benötigten Teile nicht möglich war. Neben des begrenzten Speicherplatzes, unterscheidet sich J2ME erheblich vom unterstützten J2SE. Ein Beispiel ist die Netzwerk-Schnittstelle oder die Kryptographische API.

Andere Quellen, die auch nur eine Teillösung für HBCI in Java bieten, waren nicht auffindbar. Also war eine Neuentwicklung notwendig. Wie schon in 3.4 gesagt, habe ich den Nachrichten-Generator und Parser getrennt entwickelt.

3.7.1 HBCI-Generator

Der Entwicklungsprozeß des Nachrichten-Generators ist ein Beispiel, wie sich das Design durch Refaktorisierung verändert hat. Dieser Prozeß soll hier beispielhaft beschrieben werden.

Für das erste Design bediente ich mich des Decorator aus [17], mit der Einschränkung, die zu simplen DE und DEG nicht zu implementieren. Das führte zu dem Design in Abb. 3.3. Der Name Builder ist zugegebenermaßen schlecht gewählt, zeigt aber, dass es sich im Grunde um eine Kombination aus Decorator und Builder-Muster handelt. Der HbciDialog ist demnach der Director.

Als erstes begann ich nach dem Iterationsplan mit der Klasse Dialoginitialisierung. Nach der oben erläuterten Technik: „Design für heute“, habe ich außer den Interfaces Builder, Segment und der Klasse HbciNachricht vorerst nichts weiter implementiert. Deshalb fing ich mit einer statischen Nachricht an, um diese segmentweise aufzuteilen. Während der Implementierung von der Klasse Dialogbeendigung zeigte sich, dass einige Methoden redundant waren. Alle Segmente, welche mit „HN“ beginnen, kommen in jeder Nachricht vor. Sie sind in der Klasse HbciNachricht besser aufgehoben. Eine typische Situation für Refaktorisierung. Die weitere Recherche in der Spezifikation von HBCI zeigte, dass die meisten Segmente sich den

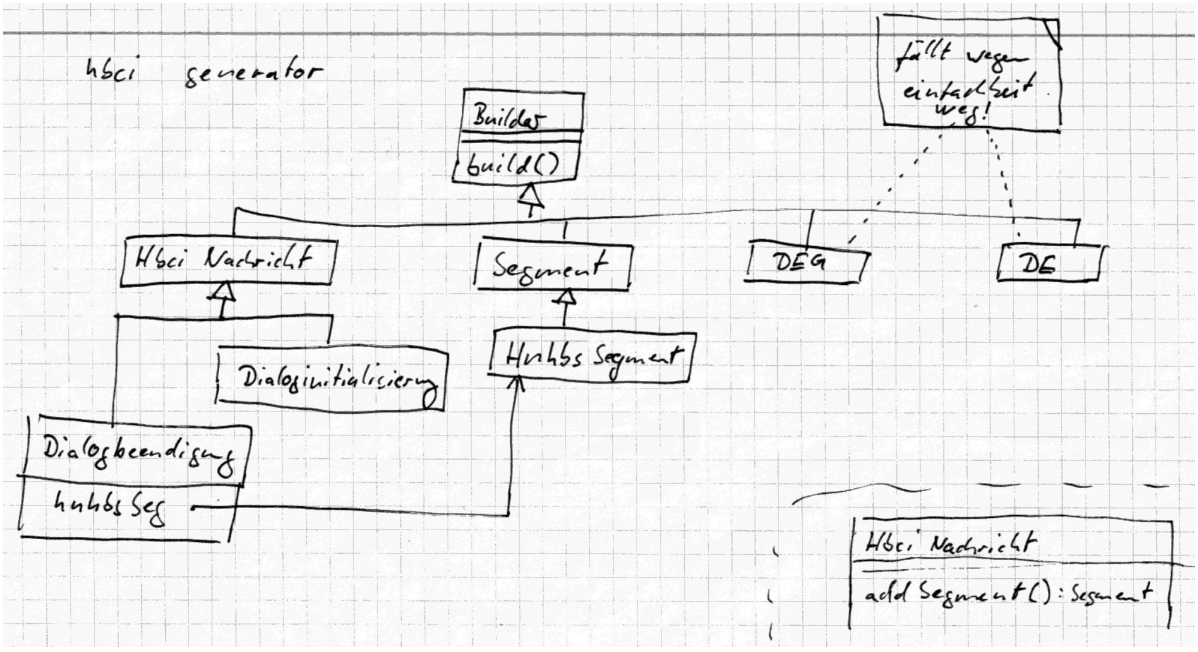

Abbildung 3.3: Erster Entwurf des Generator für HbciNachrichten, nach dem Entwurfsmuster Decorator

speziellen Nachrichten oder ihren Generalisierungen zuordnen lassen. Der Overhead mit der Verwaltung der Segmente war nicht notwendig. Er wurde durch geschützte Methodenaufrufe ersetzt. Das Ergebnis der Umstrukturierung zeigt Abb. 3.4 auf Designebene.

Die Klasse HbciNachricht übernimmt jetzt das Signieren und das Verschlüsseln der Nachricht. Die Spezialisierungen liefern ihre Segmente.

Nachdem die Dialoginitialisierung mehr als zwei Varianten über einen internen Status zu verwalten hatte, entstanden durch Refaktorisierung drei Spezialisierungen, was dem State-Muster entspricht.

Wrapping

Bei einer Realisierung in Java ist eine generelle Einigung darüber, wie HBCI-Entities abgebildet werden notwendig, ausgehend von den HBCI-Grundelementen, bwz. -Datenelementen, habe ich folgendes festgelegt:

- Alphanumerisch (an), Text (txt) -> String
- Numerisch (num) -> long oder int oder Klasse BigInteger, abhängig von der Länge
- Numerisch mit führenden Nullen (dig) -> HBCINumber
- Fließkommadarstellung (float), Wert/Beträge bis zu 15 Stellen (wrt) -> Klasse HBCI-Fixkomma mit BigInteger(long),int
- Datum (dat,vdat) -> Klasse HBCIDate

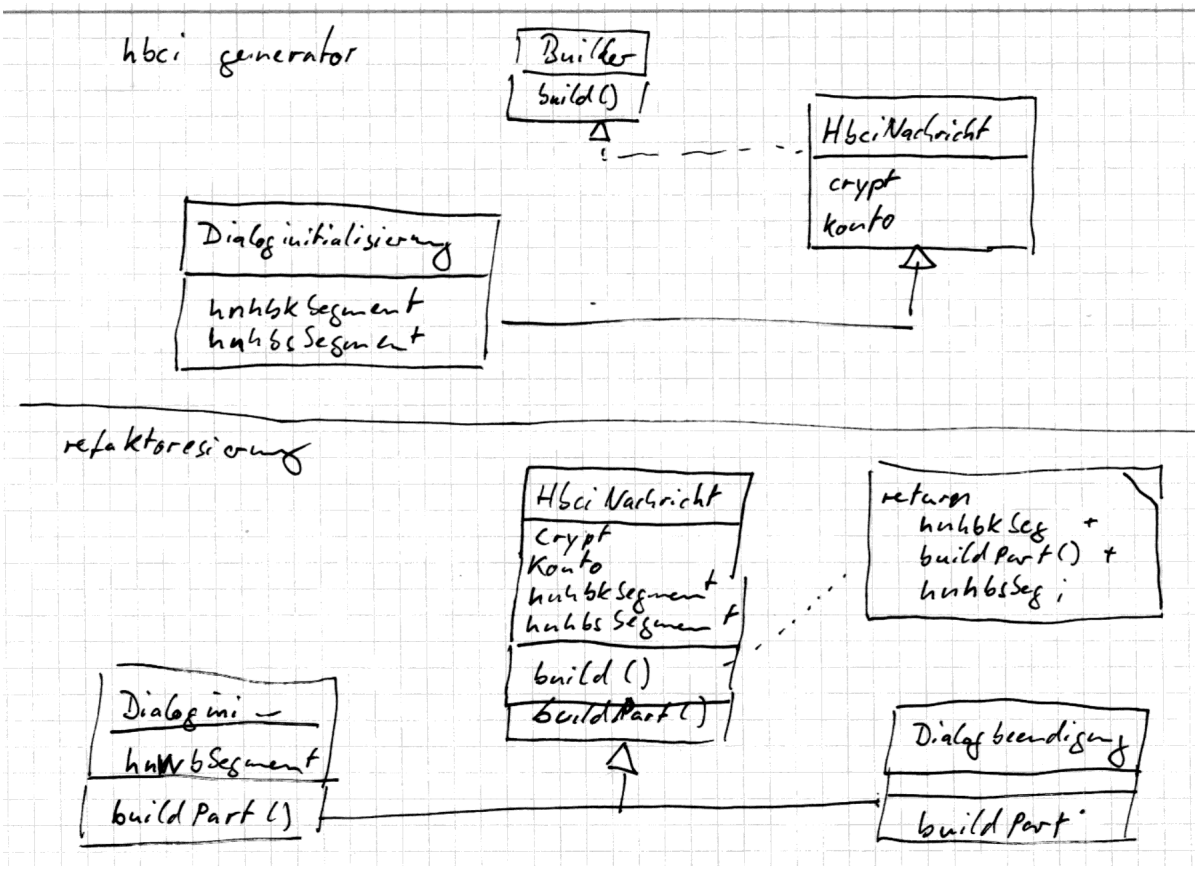

Abbildung 3.4: Refaktorisierung des Hbci-Nachrichten-Generator

- Uhrzeit (tim) -> Klasse HBCITime
- binary: -> Klasse Bin
  - S.W.I.F.T. -> SwiftUmsatz
  - Signatur -> Klasse Signatur mit byte[96].

Es hat sich gezeigt, das String besser geeignet ist vom Zahlen aufzunehmen, welche die 32 Bit Grenze überschreiten. BigInteger liefert eine nicht gewünschte hexadezimale Darstellung zurück. Da auf den Zahlen keine mathematischen Operationen ausgeführt werden müssen, reicht String als Container aus.

3.7.2 HBCI-Parser

Das Generieren von Nachrichten gestaltet sich recht einfach. Schwieriger ist oft das Parsen der erhaltenen Nachrichten, wenn sie dynamische Felder und variable Längen enthalten. Ich wollte mich bei der Entwicklung nicht allein auf meine Erfahrungen verlassen und habe mich vorher ein bißchen umgesehen.

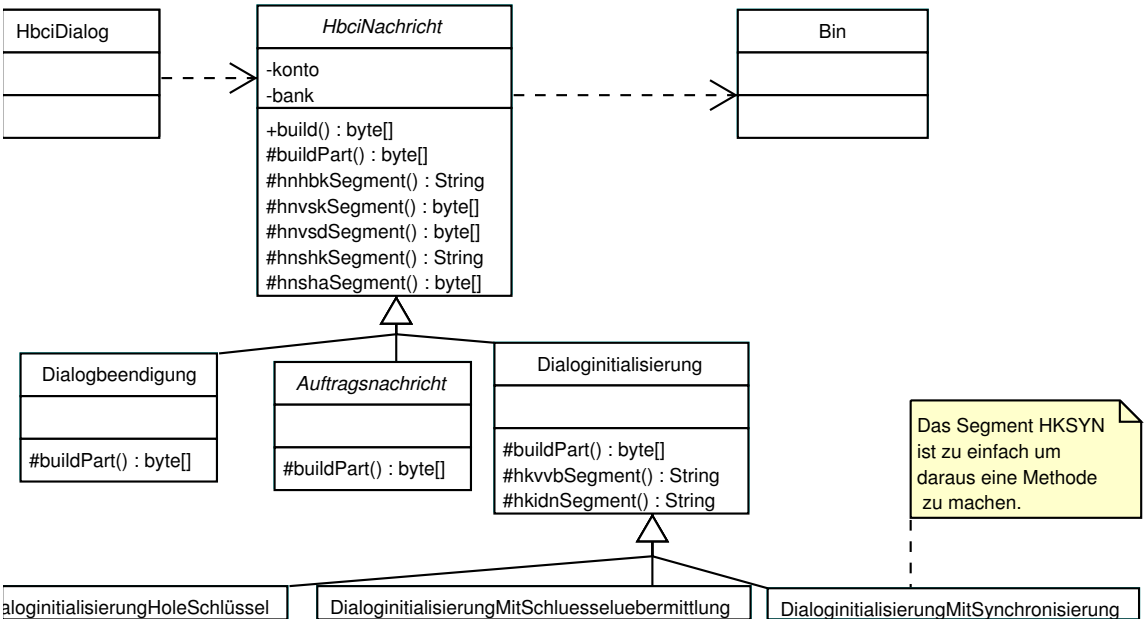

Abbildung 3.5: Entgültige Form von HbciNachricht

Die von HBCI benutzten Formate (die Trennzeichensyntax und SWIFT) lassen sich als Sprache auffassen. Neben der Eigenkonstruktion, bietet die moderne Programmiermethodik verschiedene Ansätze zur Verarbeitung an.

Zum Glück haben sich schon einige Entwickler vorher damit auseinandergesetzt und die verschiedenen Lösungen zugänglich gemacht. Auch die Entwurfsmuster von [18] haben mit dem Interpreter eine Lösung parat. Bei den Interpreter-Muster wird angemerkt, dass es sich nur für einfache Sprachen eignet und dass man für komplexere Sprachen auf einen Parsergenerator zugreifen sollte.

In der Literatur halfen mir drei Werke: [47], [2] und [43]. Bildet man die Syntax als EBNF, zeigt sich, dass es sich um eine LLR1-Sprache handelt, welche sich besonders leicht interpretieren lässt.

Für Java gibt es mittlerweile zahlreiche ParserGeneratoren und Compiler-Compiler, so dass man nicht auf zusätzliche Programme angewiesen ist. Es gibt Portierungen, die in der Unix-Welt bekannten Lex bzw. Flex, Yacc und Bison. Unter Java mehr verbreitet sind Varianten, welche die Sprachmöglichkeiten direkt ausnutzen und unterstützen. Einer der bekanntesten ist der von Javasoft bereitgestellte JavaCC (Java Compiler Compiler). Die Programmierer nennen als Vorzüge, dass nur eine Datei für Syntax und Interpretation benutzt wird, und dass er sehr anpassbar sei. Andere trennen zwischen der Syntax und dem Programmcode, was die Übersichtlichkeit erhöht.

Syntax

Bevor man sich aber für eine Variante entscheidet, muss erst mal analysiert werden, wie groß der interpretierbare Sprachraum ist. Es gibt zwei unterschiedliche Formate, wenn auch eine Umsatzanfrage möglich sein soll.

Die Syntax ist relativ einfach:

- jede Nachricht enthält drei oder mehr Segmente
- jedes Segment enthält durch einen Doppelpunkt getrennte Elemente: DE und DEG genannt.
- jedes DEG enthält durch ein Plus-Zeichen getrennte Elemente: GD genannt.
- DE und GD können Zeichen des HBCI-Zeichensatzes mit fester oder dynamischer Länge enthalten.
- DEs können auch binäre Daten enthalten. Diese fangen mit einem @-Zeichen an, gefolgt von der Anzahl der Datenbytes. Danach wieder ein @-Zeichen, gefolgt von den Binärdaten.
- Daten in Fremdformaten, wie S.W.I.F.T., werden wie binäre Daten behandelt.

Damit lässt sich die HBCI-Nachricht verifizieren und parsen. Nachrichten und Segmente sind aber formatiert, d.h. jede Nachricht enthält spezielle Segmente und jedes Segment definierte Elemente. Um den Inhalt der Nachrichten validieren zu können, müssen die Nachrichten und Segmente identifiziert werden. Etwas analoges wie die DTD oder Schema bei XML gibt es für HBCI nicht, deshalb muss jeder Entwickler sein eigenes Verfahren entwickeln. Anhand des ersten Segments, wird der Nachrichtentyp ermittelt. Damit steht fest, welche Segmente sie enthalten muss und welche sie zusätzlich enthalten kann. Jedes Segment beginnt mit dem so genannten Segment-Header. Dieser besteht aus fünf Großbuchstaben, welche das Segment angeben.

Listing 3.1: EBNF von HBCI

```
Nachricht := Seg Seg {Seg}+
Seg       := SegHead ':' (DE|DEG)+ '''
SegHead   := an
DEG       := GD '+' {GD}+
GD        := B | E
DE        := B | E
B         := an | txt | dta | num | dig | float | bin
E         := jn | datum | vdat | time | id | ctr | cur | wrt
jn        := 'J' | 'N'
dig       := '0' | '1' ... '9'
```

Listing 3.2: EBNF von HBCI (time)

```
time        := hh mm ss
hh          := '00' | '01' ... '23'
mm          := ss
ss          := '00' | '01' ... '59'
```

Parser-Schnittstelle

Wofür man sich entscheidet, ist auch eine Frage nach der Schnittstelle, die man benutzen will. Oft auch als Parser-Modell bezeichnet. Diese Bezeichnung führt zur Verwechselung mit einem der Schnittstellen für Parser. Das Prizip läßt sich in drei Gruppen einteilen:

Push Schrittweises Durchlaufen der TAGs, z.B. mit `next()`, `nextToken()`.

Pull Benutzung eines Listeners bzw. Callback-Methode, bei definierten TAG. `new Parser(Listener myListener).read(Reader in);`

Modell Komplette Überführung der Daten. Repräsentation der Daten im Speicher, meist als Baumstruktur. Speicherintensiv.

Letztendlich habe ich mich entschieden, keinen Parsergenerator zu benutzten, da mir erstens die Syntax einfach genug für eine schnelle, fehlerfreie Implementierung erschien. Und zum zweiten ich die Kontrolle über den Sourcecode behalten wollte, um ihn für die begrenzten Ressourcen optimieren zu können.

Bei der Schnittstelle habe ich einen Kompromiss gewählt. Ich wollte die Flexibilität des modellbasierten Parsers, da ich nicht alle Elemente der HBCI-Nachrichten auslesen wollte, und ich noch nicht sagen konnte, worauf es ankam. Um den Speichergebrauch zu begrenzen, habe ich immer nur ein Segment abgebildet und dieses dann zur Verarbeitung weitergereicht. Für die Baumstruktur habe ich den Vector benutzt. Alle DEs (und GDs) bis auf binaries sind als Strings abgebildet. Für die Binaries habe ich die Wrapper-Klasse Bin vom Generator benutzt.

Die Umsätze werden in zwei S.W.I.F.T.-Formaten gekapselt. Die gebuchten Umsätze kommen im Format MT 940, und die nicht gebuchten im Format MT 942, diese sind im Anhang C.1 angegeben. Dafür ist ein eigener Parser notwendig.

3.7.3 Verschlüsselte HCBI-Nachrichten

Nachdem im Kapitel 2.2 schon die Verschlüsselung von HBCI erklärt wurde, soll hier noch einmal sequenzieller Form die Schritte zum Erzeugen einer verschlüsselten Nachricht aufgezeigt werden:

1. Generieren einer signierten Nachricht.
2. Generierung einer Zufallszahl als Nachrichtenschlüssel (2-Triple-DES 128 Bit)
   - mit ungrader Parität

- ist kein schwacher oder halbschwacher Schlüssel

3. Verschlüsseln der Daten (Segmente) im CBC-Modus gemäß ISO 10116 (ANSI X3.106).
   - Das Padding der Nachricht erfolgt oktettorientiert gemäß ISO 10126 (ANSI X9.23), der Initialisierungsvektor ist 0x 00 00 00 00 00 00 00 00
4. Die höchstwertigen Bits des Nachrichtenschlüssels werden mit Nullen aufgefüllt, bis er eine Länge von 768 Bit hat. Dann wird dieser mit dem öffentlichen Schlüssel der Bank verschlüsselt.
5. Die erhaltenden 96 Bytes werden an den Anfang der verschlüsselten Nachricht gestellt.
6. Die verschlüsselten Daten mit dem verschlüsselten Nachrichtenschlüssel, werden als binary in das „hnvsd“-Segment mit der Segmentnummer 999 gestellt!
7. Es wird ein Verschlüsselungskopf erstellt.
   - mit der Segmentnummer 998
   - in der DE-Gruppe Algorithmusparameter wird zusätzlich der verschlüsselte Nachrichtenschlüssel eingetragen.

3.7.4 Bevor es losgehen kann

Bevor eine gesicherte Verbindung zwischen Kunde und Bank stattfinden kann, müssen die RSA-Schlüssel erzeugt werden. Dann muß der öffentliche Signierschlüssel auf einem sicheren Medium ausgetauscht werden. Dies geschied auf dem Postweg oder persönlich.

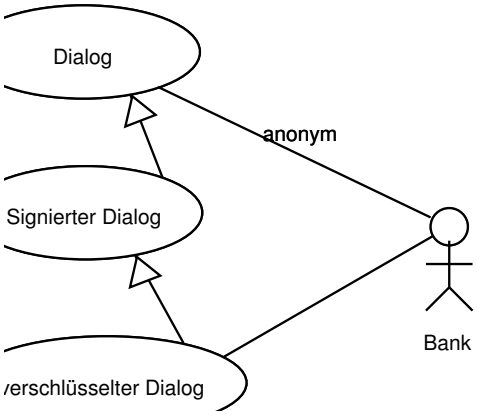

Abbildung 3.6: Anonyme, signierte und verschlüsselte Dialoge

Der Ini-Brief

Es gibt zwei Ini-Briefe, einen von der Bank und einen vom Kunden. Auf ihm ist der öffentliche Signier-Schlüssel abgedruckt, abgesichert durch einen Hashwert. Da das Generieren der RSA-Schlüssel zu rechenintensiv ist und der verwendete Zufallsgenerator nicht sicher genug ist, müssen die Schlüssel auf einem anderen Rechner, z.B. dem PC des Kunden generiert werden. Da die Handysoftware nur eine Ergänzung darstellen soll, ist dies auch sinnvoll.

Das Problem ist, es muss auch der private Schlüssel eingegeben werden. Eine schon vorhandene HBCI-Software auf den PC gibt in der Regel die privaten Schlüssel nicht heraus. Aus Gründen der Sicherheit darf sie den Schlüssel auch nicht Preis geben. Im Idealfall verbleibt der Schlüssel auf einer Karte, welche die Kryptographie

durchführt. Das der private Schlüssel manchmal doch zugänglich ist, zeigt der in 2.2.6 angesprochene Artikel.

Es gibt auch eine „saubere“ Lösung. HBCI sieht die Verwendung von mehreren Benutzern für ein Konto vor. Jeder Benutzer hat eine eigene Benutzer-ID und eigene Schlüssel. Deshalb muss ein zweiter Zugang bei der Bank beantragt werden. Das bedeutet, es gibt ein Konto mit zwei Benutzern. Eine zusätzliche Software erzeugt danach die Schlüssel und den Ini-Brief.

Nun muss noch geklärt werden, wie die RSA-Schlüssel aus dem Handy kommen. Als Vorlage für Software dient das CryptoTool von [21], es muss noch um eine Benutzerschnittstelle ergänzt werden. Ein Hashwert sorgt für die Korrektheit bei der Übertragung. Das automatische Erzeugen einer HTML-Seite als Ausdruck eines Ini-Briefes ist auch kein großer Aufwand. Alternativ zur manuellen Übertragung, ist eine Übermittlung der Schlüssel über eine TCP-Verbindung möglich. Die verschlüsselte Übertragung wird mit einem selbst definierten Passwort geschützt. Dies ist eine sehr pragmatische Lösung, welche mit allen PC, Handy-Kombinationen möglich sein sollte. Es sind auch andere Übertragungsmöglichkeiten denkbar, z.B. durch die Übertragung auf eine MMC für das Siemens SL 45 oder durch eine Synchronisation. Dies ist aber vom Handy abhängig.

Synchronisierung

Nachdem die Schlüssel erzeugt und die Ini-Briefe mit der Bank ausgetauscht wurden, müssen die Benutzerdaten, die Zugangsparameter und die Schlüssel der Bank müssen eingeben werden. Danach werden die verblieben Schlüssel, die Kundensystem-ID und die Signatur-ID übertragen. Vorher ist keine Kommunikation mit dem Bankenrechner möglich.

Zur Übertragung sind vier Dialoge notwendig, in denen in der Dialoginitialierungnachricht spezielle Segmente vorhanden sind, und die sofort mit einer Dialogendenachricht beendet werden. Die Reihenfolge ist dabei einzuhalten:

1. Die öffentlichen Schlüssel der Bank anfordern. Dies geschied über einen anonymen Zugang
2. Synchronisation der Signatur-ID. Dazu wird Kundensystem-ID auf 0 gesetzt, solange sie nicht bekannt ist, und die Signatur-ID auf „999999999999“
3. Der Kunde übermittelt seine beiden öffentlichen Schlüssel. Dabei ist zu beachten:
   - Die Nachricht ist mit dem eigenen privaten Signierschlüssel zu unterzeichnen.
   - Diese Nachricht muss sowohl signiert als auch chiffriert sein.
   - Der Signierschlüssel muss durch einen Ini-Brief bestätigt sein!
4. Synchronisation der Kundensystem-ID

3.8 Benutzerinterface

Das Benutzerinterface wurde mit Hilfe von explorativen Prototypen [13] entwickelt. Da später bei der Darstellung auf die unterschiedlichen Bildschirmgrößen und Eigenschaften der Geräte

mit angepassten Ansichten reagiert werden sollte, wurde bei dem Design das Konzept des Modell/View/Controller MVC benutzt.

Zur Realisierung der Ansicht stehen unter J2ME mehrere Bibliotheken mit unterschiedlichen Ansätzen zur Verfügung. Die Klassen im Paket |java.microedition.lcdui| sind darauf ausgelegt, möglichst unabhängig von der Bildschirmform zu sein und bieten sich daher an, um eine allgemeine Darstellung zu implementieren. Deshalb wurde diese Bibliothek für das primäre Darstellungsmodul benutzt.

Die darzustellenden Informationen sind zum Teil recht umfangreich, z.B. Umsätze oder Überweisung. Es erleichtert die Eingabe, wenn bekannte Elemente benutzt werden. Möchte man eine Überweisung in der Bank tätigen, füllt man dort einen Überweisungsträger aus. Der Benutzer erwartet die selbe Aufteilung der Eingabefelder bei der Onlineüberweisung. Die einzelnen Elemente so genau zu positionieren, ist mit dem lcdui nicht möglich. Ein PDA unterstützt mit seiner Displaygröße durchaus die Möglichkeit so eine Aufteilung darzustellen. Mit den Bibliotheken OMT und kAWT ist es möglich, die Darstellung mehr zu steuern.

3.9 Test

UnitTests lassen sich mit mit JUnit durchführen.
Für die Micro Edition von Java 2 gibt es kein Framework zum Testen von Oberflächen. Chris Collins von RoleModel Software hat zwar mit J2MEUnit [6] das JUnit-Framework auf J2ME portiert, dies hilft aber nur um Performancetests oder Kompatibilitätstests für die einzelnen Geräte durchzuführen.

Auch fehlt bisher Test-Framework für die Benutzerschnittstelle von J2ME.

3.10 Realisierung

3.10.1 Benutzung der Bouncy Castle Crypto API

Die API von Bouncy Castle ist nach dem Baukastenprinzip aufgebaut. Z.B. benutzt eine Instance der Klasse `DESedeEngine()` für eine TripleDES-Verschlüsselung. Benötigt man den CBC-Modus, dann generiert man die Instance mit `engine = new CBCBlockCipher(new DESedeEngine())`. Entsprechend, wenn man ein spezielles Padding -Verfahren einsetzen möchte, siehe Listing 3.3 Zeile 12. In der Regel müssen alle Instanzen vor der Benutzung initialisiert werden, bei der Verschlüsselung übergibt man dazu den Schlüssel, siehe Listing 3.3 Zeile 18. Ist der erste Parameter `false` wird eine entschlüsselung durchgeführt.

Listing 3.3: TripleDES-Verschlüsseln mit Bouncy Castle

```
1 // Initialisierungvector fuer CBC–Modus
2 byte[] IV = {0,0,0,0, 0,0,0,0};
3
4 byte[] keyBytes = {7,6,5,4,3,2,1};
5
```

```
6  String msg = "Dieser text wird verschluesselt";
7
8  // Parameter: Schluessel inclusive Initialisierungsvector
9  ParametersWithIV cryptKey = new ParametersWithIV( new
       DESedeParameters(keyBytes), IV);
10
11 // TripleDES Engine im CBC-Modus mit ISO-10126 Padding
12 PaddedBufferedBlockCipher cipher = new
       PaddedBufferedBlockCipher( new CBCBlockCipher(new
       DESedeEngine()), new ISO10126d2Padding());
13
14 byte[] rv = null;
15 int oLen = 0;
16
17 // Schluessel üfr Verschluesselung setzen (false :=
       Entschluesselung)
18 cipher.init( true, desKey);
19
20 // Groesse üfr Ausgangsdaten ermitteln
21 cryptData = new byte[cipher.getOutputSize(msg.length())];
22
23 // Die ersten vollstaendigen Bloecke verschluesseln und nach
       cryptData kopieren
24 oLen = cipher.processBytes( msg.getBytes(), 0, msg.length(),
       cryptData, 0);
25
26 try {
27     // Letzten Block verschluesseln
28     cipher.doFinal(cryptData, oLen);
29 } catch (CryptoException e) {
30     e.printStackTrace();
31     alert("Ooops, encrypt/decrypt exception");
32 }
```

Der Schlüsselgenerator überprueft, ob es sich um einen schwachen oder semischwachen Schlüssel handelt und setzt die Paritätsbits.

Listing 3.4: Schlüsselgenerierung mit Bouncy Castle

```
1 DESedeKeyGenerator desKG = new DESedeKeyGenerator();
2 KeyGenerationParameters kgp = new KeyGenerationParameters( new
      SecureRandom(), 112);
3 desKG.init(kgp);
4 byte[] desKey = desKG.generateKey();
```

Beim Signieren ist zu beachten, das man für HBCI 2.2 nicht die Klasse `ISO9796d2Signer` benutzt, da noch der Standard ISO-9697-1 verwendet wird, kleine Zahl großer Unterschied.

Listing 3.5: Signieren mit Bouncy Castle

```
1  static BigInteger  mod = new BigInteger("b259d2...", 16);
2  static BigInteger  pubExp = new BigInteger("11", 16);
3  static BigInteger  privExp = new BigInteger("92e08f...", 16);
4
5  String msg = "Dieser text wird signiert";
6
7  RSAKeyParameters publicKey  = new RSAKeyParameters(false, mod
       , pubExp);
8  RSAKeyParameters privateKey = new RSAKeyParameters(true, mod,
       privExp);
9
10 Digest digest = new RIPEMD160Digest();
11
12 AsymmetricBlockCipher engine = new ISO9796d1Encoding( new
       RSAEngine());
13
14 byte[] sign = null;
15
16 // 160 Bit reservieren
17 byte[]  hash = new byte[digest.getDigestSize()];
18
19 // Mit Daten fuettern und Hashwert bilden
20 digest.update( msg.getBytes(), 0, msg.length());
21 digest.doFinal( hash, 0);
22
23 // Initialisieren üfr Verschluesselung
24 engine.init( true, privateKey);
25 try {
26     // Hash-Wert verschluesseln: Signatur fertig!
27     sign = engine.processBlock( hash,  0, hash.length);
28
29     // Initialisieren üfr Entschluesselung
30     engine.init( false, publicKey);
31
32     // Hash-Wert entschluesseln
33     byte[] hash2 = engine.processBlock( sign,  0, sign.length)
           ;
34
35     // und vergleichen
```

```
36      arrayEquals( hash, hash2);
37  } catch( InvalidCipherTextException e) {
38      e.printStackTrace();
39  }
```

3.10.2 Netzwerk

Wie üblich bei Java gestaltet sich die TCP/IP-Schnittstelle als sehr einfach nutzbar.

Listing 3.6: Benutzung der Netzwerk-Schnittstelle

```
1  StreamConnection sc = Connector.open( "socket://www.hbci-
       kernel.de:3000");
2
3  OutputStream os = sc.getOutputStream();
4  os.write("Hallo welt".getBytes());
5
6  InputStream is = sc.getInputStream();
7  byte[] data = new byte[256];
8
9  while( is.read(data) != -1 )
10 {
11   // process data[] ...
12 }
```

Da `read(data)` solange blockiert, bis der Puffer voll ist, wenn die Verbindung noch steht. Und im Gegensatz zu Http wird die Verbindung nach der Übertragung[10] nicht durch den Server unterbrochen, daher sollte man die folgende Lösung benutzen:

```
10 while( is.available() > 0 )
11 {
12   is.read(data)
13   // process data[] ...
14 }
```

3.11 Fazit

Trotz aller Vorbereitungen gibt es immer einige Unbekannte, die bei der Entwicklung zu Schwierigkeiten und Verzögerungen führen.

Ein Problem gab es mit der Signatur. Die vom Server gelieferte Signatur lies sich nicht verifizieren. Eine lokal erstellte und signierte Nachricht hingegen schon. Dadurch vermutete

[10] Bei HTTP 1.0 wird für jeden Content eine eigene Verbindung zum Server aufgebaut und nach erfolgreicher Übertragung beendet.

ich eine Inkompatibilität der relativ selten benutzten ISO-9796-1 Encoding. Die Überprüfung verbrauchte viel Zeit. Der Fehler lag aber in der Klasse String, welche ich als Behälter für die binären Daten benutzt habe. Aus mir unerfindlichen Gründen wandelt die Klasse sechs Zeichen um: -99, -112, -113, -115, -127 werden in eine 63 gewandelt. Beim lokalen Test mit signierten Nachrichten ist nur durch Zufall kein Fehler aufgetreten, da weder in den Schlüsseln noch in der Signatur diese Zahlen vorhanden waren.

Ein weiteres Problem war ein wenig langwieriger, da einige Tage durch E-Mail Korrespondenz mit den freundlichen Mitarbeitern vom PPI notwendig waren. Es zeigte sich, dass der Instituts-Server nicht richtig auf die Anforderung einer für die Initialisierung des Kundenkontos notwendigen „Synchronisierung der Signatur-ID“ antwortete. Die letzte E-Mail befindet sich im Anhang. Ob dies ein generelles Problem auch bei anderen Servern darstellt, ist noch nicht abschließend geklärt.

Nachdem die Schwierigkeiten mit dem Signieren und der Verschlüsselung behoben sind, kann es richtig losgehen. Davor ist aber wieder die Refaktorisierung angebracht, um den „riechenden“ Programmcode[11] zu beseitigen.

Z.B. durch eine einfachere Behandlung der Binärdaten, `byte[]` ist umständlich und fehleranfällig. Eine Klasse zur Verwaltung würde helfen. In Anlehnung an den `StringBuffer` könnte diese Klasse `ByteBuffer` heißen. Der Name ist aber schon in `java.nio.*` belegt, deshalb wird sie vorerst `BinBuffer` heißen. Die Methodennamen werden sich an `StringBuffer` und nicht an `ByteBuffer` orientieren, da auch der dynamische Aspekt von `StringBuffer` übernommen werden soll. Man sollte nichts selber machen, was es schon gibt. Es scheint ein bekanntes Problem zu sein, wenn man sich im Internet umsieht[12]. Umso erstaunlicher, dass eine solche Klasse nicht in der Standard Edition vorhanden ist (geschweige in der Micro Edition):

- `com.ibm.wbi.util.ByteBuffer`
- `nl.justobjects.toolkit.collection.ByteBuffer`
  |http://www.justobjects.org/xbook/api/nl/justobjects/toolkit/collection/ByteBuffer.html|

Auch wenn der Parser sehr gut funktioniert, kann man den Speicherverbrauch durch eine Umstrukturierung verringern. Wichtiger ist im Moment die RSA-Verschlüsselung. Sie ließe sich noch optimieren, wenn man ausnutzt, dass der öffentliche Exponent immer den Wert $2^{16} + 1$ hat[13], weitere Optimierungen wird der Profiler vom JWTK 1.0.4 ergeben.

[11] „it smelled“ ist der geläufige Ausdruck dafür, das etwas nicht in Ordnung ist. Es werden auch Richtlinien gegeben, wann man etwas ändern sollte, aber keine exakte (deterministische) Vorgehensweise

[12] Google suche: ByteBuffer -java.nio

[13] eine Multiplikation lässt sich günstig durch 16-faches Schifften mit anschließender Addition erreichen.

4 Nachbereitung

4.1 Zusammenfassung

Ich habe mich in meiner Diplomarbeit mit Verschlüsselungstechniken, der Umsetzung einer Spezifikation sowie der Programmierung von Handys beschäftigt. Dabei habe ich mich kritisch mit den Standards auseinander gesetzt. Der realisierte Prototyp wurde mit UML-Methoden entwickelt, für die Modellierung habe ich auf Entwurfsmuster zugegriffen. Da ich ausgewählte Praktiken des leicht gewichtigen Vorgehensmodell „extreme programming“ genutzt habe, haben sich Modellierung und Implementierung in einem evolutionären Entwicklungsprozess abgewechselt bzw. das vorhandene Entwurfsmodell wurde mit den neuen Erkenntnissen verfeinert.

Ich habe mich wegen der zunehmenden Verbreitung auf Handys für die Programmiersprache Java 2 Micro Edition entschieden, und mich in die Schnittstellen von CLDC und MIDP sowie der verwendeten „Bouncy Castle Crypto-API“ eingearbeitet.

Für die umgesetzte Spezifikation HBCI 2.2 habe ich einen Parser und einen Generator für den HBCI-Dialog mit den benötigten Nachrichtenformaten entwickelt. Dazu war neben dem Wissen über die von HBCI verwendete Trennzeichensyntax und das S.W.I.F.T.-Format eine Vertiefung in Kryptographie sowie deren Implementierung notwendig.

Das dabei entstandene Dokument (Diplomarbeit) beschreibt im Kap. 2 Java2ME sowie HBCI und die relevante Kryptographie. Im Kap. 3 ist der Entwicklungsprozess der Hauptmodule und einige Implementierungsdetails dargestellt.

4.2 Auswertung

4.2.1 Ergebnis

Der erstellte Software ist in der Lage, gemäß der HBCI-Spezifikation, verschlüsselte und signierte Nachrichten auszutauschen. Damit sind die Grundlagen für alle gängigen Geschäftsvorfälle[1] geschaffen. Sie ist leicht erweiterbar, um z.B. Umsätze oder Überweisungen abzufragen bzw. zu tätigen.

Dieser Prototyp implementiert alle notwendigen Komponenten für einen Dialog mit dem Bankenrechner. Er beherrscht den Grundnachrichtenaufbau und die Dialoginitalisierung mit Beendigung. Somit kann der Kontostand mit einem geeigneten Handy, von der Bank abgefragt werden.

[1] Sammelüberweisungen sind nicht enthalten, entsprechen aber auch nicht der normalen Tätigkeit eines mobilen Benutzers

Der Speicherverbrauch für die meisten Handys noch zu groß. Zum zweiten ist die RSA-Ver- und Entschlüsselung sehr rechenintensiv, weshalb das Handy einen entsprechend leistungsfähigen Prozessor[2] benötigt.

4.2.2 Durchführung

Die verwendeten Praktiken, insbesondere die UnitTests im Zusammenspiel mit der Refaktorisierung haben mit ihrer Leistungsfähigkeit überzeugt. Nur durch dieses Gespann hat die Umstellung der Binärdaten von `String` auf `byte[]` nicht zu noch längeren Verzögerungen geführt. Das zeigt, dass die Tests nicht nur während der Implementierung die Motivation erhöhen, weil sofort Entwicklungen sichtbar werden, sondern auch strukturelle Fehler zu einem späten Zeitpunkt sich schnell beheben lassen. Im nächsten Projekt werde ich noch konsequenter UnitTests benutzen. und nur die benötigten Methoden implementieren. Dieses Beispiel hat mir gezeigt, dass man bei diszipliniertem Einsatz der Tests keine Änderungen scheuen muss.

Ich würde aber empfehlen an geeigneten Stellen Schnittstellen mit loser Kopplung einzufügen. Das verringert die Komplexität und macht die zu ändernden Bereiche kleiner. Und wenn man einen leicht gewichtigen Entwicklungsprozess benutzen möchte, dies auch konsequent mit allen Regeln durchführt. Man sollte auch beachten ob er sich für das Projekt oder Teilprojekt eignet.

Die Benutzbarkeit der Anwendung ist auch abhängig von der Geschwindigkeit, mit der die Benutzeranfragen behandelt werden. Kryptographische Algorithmen benötigen viel Rechenkapazität. Zu Beginn erstellte ich einen explorativen Prototypen, um mich mit der Benutzung der Krypto-Bibliothek vertraut zu machen und um die Geschwindigkeit der einzelnen Verschlüsselungsalgorithmen auf realen Geräten überprüfen zu können. Nachdem fast alle (DES, DESede, DES Key Generation, RSA) Tests implementiert waren, wollte ich noch heraus finden, wie lange die komplette Herstellung einer verschlüsselten und signierten Nachricht dauern würde. Es hatte sich bereits gezeigt, dass die Zeit für das symmetrische Verschlüsseln und der Hashwert zu vernachlässigen waren, gegenüber dem RSA-Algorithmus. Zu diesem Zeitpunkt, des Krypto-Prototypen, entdeckte ich einen Artikel [48], der meine Ergebnisse bestätigte. Der Autor kommt in dem Artikel zu dem selben Schluss, dass es für zukünftige Versionen von MIDP notwendig sei, kryptographische Schnittstellen zur Verfügung zu stellen, wenn auf den „Java-Handy“ Buisness oder Commerzanwendungen geschrieben werden sollen, s.2.7.

4.3 Ausblick

Für Java auf dem Handy sieht auch in nächster Zeit sehr gut aus. Die Betriebssystem-Allianz „Symbian“, der Nokia, SonyEricsson, Motorola, Siemens und Samsung angehören, hat in ihrem OS bereits Java2ME integriert.

Die Software ist so ausgelegt, dass sie sich leicht erweitern lässt. Ein geeignetes Endgerät vorausgesetzt, kann man beliebige Geschäftsvorfälle implementieren. Das reicht von einfachen Überweisungen und der Umsatzabfrage, über das Erstellen, Ändern und Löschen von

[2] Ein DragenBall wie im Palm ist ausreichend.

Daueraufträgen, bis zur Verwaltung des Aktiendepots. Für die Benutzerschnittstelle kann man andere GUI-Bibliotheken, wie z.B. kAWT benutzen. Damit wäre der Sprung vom MIDP zum „Personal Profile“ kleiner.

Die Bedeutung der Kryptographie, auch für kleine Geräte, wurde vom „Java Community Process“ JCP erkannt. Der Nachfolger von MIDP, die Version 2.0, bringt Verschlüsselungen mit, aber nur in der Form einer SSL-Verbindung, ohne eine API zur eigenen Kryptographie. Für PDAs ist mittlerweile das „Personal Profile“ mit JCE freigegeben. Eine zusätzliche API auch als Ergänzung zu MIDP ist mit dem [5] in Entwicklung. Diese bringt eine Schnittstelle zur Hardware mit. Das wiederum läßt den Einsatz von kryptographischen Spezialchips oder Smartcards zu, aus denen ein Geschwindigkeitsgewinn und eine höhere Sicherheit resultiert. Nicht zuletzt werden die Mobiltelefone immer leistungsfähiger, um verbesserte Multimedia Fähigkeiten und UMTS beherrschen zu können.

Mit der Version 4 von HBCI soll das Protokoll auf XML umgestellt werden. Wenn man sich schon heute mit SOAP und SSL unter Java2ME auseinander setzt, braucht man bei der Veröffentlichung nicht mehr viel Aufwand zu betreiben. Zur Zeit stellt das Gespann XML mit HTTPS die beste Wahl dar, wenn man sichere Client/Server-Anwendungen benötigt.

Literatur

[1] : *Zitat Gartner*. online. 05.04.2002 11:31. – URL `http://www.qualcomm.com/brew/brewtimes/ppt/clohessy.pdf`. – Zugriffsdatum: 20.10.2002

[2] AHO, Alfred V. ; SETHI, Ravi ; ULLMANN, Jeffrey D.: *Compilerbau Teil 1*. 2. Oldenburg, 1999. – ISBN 3-486-25294-1

[3] BOSSELAERS, Antoon: *The hash function RIPEMD-160*. online. 17 August 1999. – URL `http://www.esat.kuleuven.ac.be/~bosselae/ripemd160.html`. – Zugriffsdatum: 20.8.2002

[4] BV., Virtual U.: *BeeCrypt*. online. – URL `http://www.virtualunlimited.com/products/beecrypt/`. – Zugriffsdatum: 20.10.2002

[5] CHEN, Zhiqun: *JSR 177 – Security and Trust Services API for J2ME*. online. – URL `http://jcp.org/jsr/detail/177.jsp`. – Zugriffsdatum: 20.8.2002

[6] COLLINS, Chris: *J2ME Unit*. 2000

[7] DAL: *Heise News-Ticker: PDA mit WLAN und Fingerabdruck-Scanner von HP*. online. 10.10.2002 15:07. – URL `http://www.heise.de/newsticker/data/dal-10.10.02-000/default.shtml`. – Zugriffsdatum: 20.10.2002

[8] DPA: *Heise News-Ticker: Die Rückkehr der Bildtelefonie mit UMTS*. online. 20.02.2001 17:07. – URL `http://www.heise.de/newsticker/data/ll-20.02.01-002/default.shtml`. – Zugriffsdatum: 20.10.2002

[9] DRÖGE, Nils: *Studienarbeit - Partizipative, evolutionäre Entwicklung im Robot-Art Umfeld*, Hochschule für angewandte Wissenschaften Hamburg, Fachbereich Technische Informatik, Studienarbeit, 2002

[10] ELLIS, Jon ; YOUNG, Mark: *JSR 172 – J2ME Web Services Specification*. online. – URL `http://jcp.org/jsr/detail/172.jsp`. – Zugriffsdatum: 20.8.2002

[11] : *Finance*. online. – URL `http://www.datadesignag.com/download/docs/PD_FinanceBrowser3.pdf`. – Zugriffsdatum: 2002

[12] FISCHER, Stephan ; STEINACKER, Achim ; BERTRAM, Reinhard ; STEINMETZ, Ralf: *Von Den Grundlagen Zu Den Anwendungen*. Berlin, Heidelberg : Springer, 1998. – ISBN 3-540-64654-X

[13] FLOYD, Christiane: *STEPS*. 1987

[14] FOWER, Martin ; SCOTT, Kendall: *UML konzentriert (deutsche Übersetzung von „UML Distillert“ - ISBN 0-201-32563-2)*. Addison-Wesley, 1998. – ISBN 3-8273-1329-5

[15] : *Fragen und Antworten zum Thema HBCI*. online. – URL `http://www.hbci.de/fragen/4.html`. – Zugriffsdatum: 20.10.2002

[16] FUN COMMUNICATIONS GMBH: *fun Produkte - ebanking - MobileBanking*. online. – URL `http://www.fun.de/deutsch/produkte/onlinebanking/mobilebanking.htm`. – Zugriffsdatum: 2002

[17] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralpf ; VLISSIDES, John: *Desgin Pattern*. Addison-Wesley, 1996

[18] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralpf ; VLISSIDES, John: *Entwurfsmuster*. Addison-Wesley, 1996. – ISBN 0-201-63361-2

[19] GLEICH, Clemens: Blitzgescheit, Mobile Flash-Speichermedien im Vergleich. In: *c't* 8 (2002), S. 168

[20] GLEICH, Clemens: Das Gigabyte im Geldbeutel, Mini-Massenspeicher in Flash-Technologie. In: *c't* 8 (2002), S. 164

[21] GÜNTHER, Uwe: *JHBCI, OpenSource HBCI Toolkit for Java*. online. – URL `http://www.jhbci.de/`. – Zugriffsdatum: 2002

[22] GÜNTHER, Uwe: *Entwicklung eines JCA/JCE-API konformen Kryptographischen Service Providers für HBCI unter Java*, Fachhochschule Schmalkalden, Fachbereich Informatik, Diplomarbeit, März 2002. – URL `http://www.cscc.de/diplom/diplom.web.lin.pdf`

[23] : *HBCI Kernel*. online. – URL `http://www.hbci-kernel.de`. – Zugriffsdatum: 20.8.2002

[24] HENDRIK KOY, Jörg S.: Selbst geknackt – Spielerisches Erforschen der Kryptographie. In: *c't* 14 (2001), S. 204

[25] : *International Financial Exchange, IFX Forum Inc.* online. – URL `http://www.ifxforum.org`. – Zugriffsdatum: 20.10.2002

[26] : *Java 2 Platform, Micro Edition*. online. – URL `http://java.sun.com/j2me/`. – Zugriffsdatum: 20.10.2002

[27] : *Java2ME for Windows CE*. online. – URL `http://www-3.ibm.com/software/pervasive/products/wsdd/`. – Zugriffsdatum: 24.10.2002

[28] : *Java Community Process*. online. – URL `http://jcp.org/jsr/tech/j2me.jsp`. – Zugriffsdatum: 20.10.2002

[29] KAISER, Fabian: *HBCI Annoyances*. O'Reilly, August 2001. – ISBN 1-201269-23

[30] KROLL, Michael: Käffchen, J2ME in der Praxis. In: *iX* 4 (2002), April

[31] : *Legion of the Bouncy Castle*. online. – URL `http://www.bouncycastle.org`. – Zugriffsdatum: 20.8.2002

[32] LIEDTKE, Dirk: Hacker kapern Online-Konto. In: *Stern* 21 (2002), Mai. – URL `http://www.stern.de/computer-netze/readme/pflichtlektuere/artikel/?id=201756`

[33] : *Java2ME for Symbian OS*. online. – URL `http://www.symbian.com/technology/standard-java.html`. – Zugriffsdatum: 24.10.2002

[34] : *MobileBanking (PDA) - Banking mit Ihrem Organizer*. online

[35] : *Open Financial Exchange*. online. – URL `http://www.ofx.net`. – Zugriffsdatum: 20.10.2002

[36] PEURSEM, Jim V.: *JSR 118 – Mobile Information Device Profile 2.0*. online. – URL `http://jcp.org/jsr/detail/118.jsp`. – Zugriffsdatum: 20.8.2002

[37] RAGNARSSON, Logi: *logi.crypto*. online. – URL `http://www.logi.org/logi.crypto/stable/`. – Zugriffsdatum: 20.8.2002

[38] SCHMEH, Klaus: *Kryptografie und Public-Key-Infrastrukturen im Internet*. ix Edition. dpunkt, 2001. – 2. Auflage 2001. ca. 600 Seiten. – ISBN 3-932588-90-8

[39] : *STARSIM Applications*. online. – URL `http://www.gdm.de/eng/products/04/index.php4?product_id=386`. – Zugriffsdatum: 2002

[40] : *Stellungnahme zu neuen Hackerangriffen auf HBCI*. online. 15.05.2002. – URL `http://www.hbci.de/aktuell/1.html`. – Zugriffsdatum: 20.10.2002

[41] : *Technical Information, SIM/USIM Application Toolkit (SAT/USAT)*. online. – URL `http://www.etsi.org/frameset/home.htm?/plugtests/02upcomingevents/smartcard/smart%5Ftechnical.htm`. – Zugriffsdatum: 2002

[42] TOL: *Heise News-Ticker: Verein für Notrufnummer 114 zur Karten-Sperrung*. online. 05.04.2002 11:31. – URL `http://www.heise.de/newsticker/data/tol-05.04.02-000/default.shtml`. – Zugriffsdatum: 7.8.2002

[43] VÖLLER, Reinhard: *Formale Sprachen und Compiler*. URL `http://users.informatik.fh-hamburg.de/~voeller/comp.pdf`. – Zugriffsdatum: 20.8.2002, März 1999

[44] VÖLLER, Reinhard: *Kryptologie - Eine Einführung*. URL `http://users.informatik.fh-hamburg.de/~voeller/cryptopdf.zip`. – Zugriffsdatum: 20.10.2002, Februar 2001

[45] : *Vortragsfolien, SIM Application Toolkit*. – URL `http://www.ti.fhg.de/trierer_symposien/symposium_-_smart_cards/vortrage/dr._ulrich_sporn/sld008.html`. – Zugriffsdatum: 2002

[46] ITZ (Hrsg.) ; SFZ (Hrsg.) ; IAT (Hrsg.): WerkstattBericht Nr. 49, Entwicklung und zukünftige Bedeutung mobiler Multimediadienste / IZT – Institut für Zukunftsstudien und Technologiebewertung, SFZ – Sekretariat für Zukunftsforschung, IAT – Institut Arbeit und Technik. dec 2001. – Forschungsbericht. – ISBN 3-929173-49-2

[47] WIRTH, Niklaus: *Grundlagen und Techniken des Compilerbaus*. Addison-Wesley, 1996. – ISBN 3-89319-931-4

[48] YUAN, Michael J.: How to digitally sign and verify XML documents on wireless devices using the Bouncy Castle Crypto APIs. In: *IBM developerWorks* (2002), June. – URL `http://www-106.ibm.com/developerworks/java/library/j-midpds.html`. – Zugriffsdatum: 20.8.2002

[49] ZENTRALER KREDITAUSSCHUSS (ZKA): *HBCI Spezifikation Version 2.2*. 2.2. Test: , may 2000. – URL `http://www.hbci.de/download/HBCI220D/HBCI22.pdf`

A Abkürzungsverzeichnis

AES Advanced Encryption Standard

API Application Programming Interface

Applet Programm-Komponente die vom Browser ausgeführt wird.

BPD Bankparameterdaten

Btx Bildschirmtext

CBC Cipher Block Chaining

CDC Connected Devices Configuration

CEPT Conference Européennes des Adminstrations des Postes et Télécommunications

CLDC Connected Limited Devices Configuration

DDV DES DES Verfahren

DTAUS „Datensatzformat für den Inlandszahlungsverkehr (veröffentlicht in den Bedingungen für die Beteiligung von Kunden am beleglosen Datenträgeraustausch mittels Disketten)“ [49]

DE Datenelement

DEG Datenelementgruppe

DES Data Encryption Standard

GD Gruppendatenelement

GDG Gruppendatenelementgruppe

EDIFACT Electronic Data Interchange for Administration, Commerce and Transport

ETSI European Telecommunications Standards Institute

HBCI Home Banking Computer Interface

ISO International Organisation for Standardisation

J2EE Java 2 Enterprise Edition

J2ME Java 2 Micro Edition

J2SE Java 2 Standard Edition

JCA Java Cryptography Architekture

JCE Java Cryptography Extension

KISS keep it stupid simple

MD5 Message Digist Version 5

MAC Message Authentication Code

Midlet In Anlehnung an das Applet.

MIDP Mobile Information Device Profile

OSI

PDA Personal Digital Assistent

PIN Personal Identification Number

RACE Research and Development in Advanced Communications Technologies in Europe

RC Ron's Code oder Rivest Cipher

RDH RSA DES Hybridverfahren

RFC Request for Comment

RIPEMD RACE Integrity Primitives Evaluation Message Digist

RSA benannt nach Ronald Rivest, Adi Shamir und Len Adleman

SEG Segment

SHA Secure Hash Algorithmus

SIZ Informatikzentrum der Sparkassenorganisation GmbH

SOAP Simple Object Aplication Protocol

SSL Secure Socket Layer

S.W.I.F.T. Society for Worldwide Interbanking Financial Communication

TAN Transaktionsnummer

UN/EDIFACT s. EDIFACTA

UPD Userparameterdaten

JWTK J2ME 2 Wireless Toolkit

XML Extended Makeup Language

ZKA Zentraler Kreditausschuss

B E-Mail vom PPI

> Sehr geehrter Marcussen–Wulff,
>
> ich werde in zukunft wieder genauer lesen. Die Spezifikation ist zwar sehr
> gut und genau, aber zum teil etwas üunbersichtich.
>
> das setzen der signatur–id auf '9999999999999999' hat leider nicht den
> ügewnschten erfolg gebracht.
> Mir ist nicht klar warum die signatur–id nicht synchronisiert werden muss,
> ist das nur bei ihrem test–server der fall? Wo bekomme ich sie sonst her?
>
> Ich habe folgende Nachricht gesendet:
>
> überschlsselter teil:
> HNSHK
:2:3+1+6790+1+1+1::0+9999999999999999+1:20021022:152949+
1:999:1+6:10:16
> +280:09950003:Benutzer–000152–03:S:0:1'
> HKIDN:3:2+280:09950003+Benutzer–000152–03+0+0'
> HKVVB:4:2+54+0+1+xpresso+0.1'
> HKSYN:5:2+2'
> ...
>
> Die Antwort:
> HNHBK:1:3+000000000402+220+1444918192512201+1+1444918192512201:1'

> HNSHK:2:3+1+4321+1+1+2+1+1:20021022:152918+1:999:1+6:10:16+
280:09950003:0995
> 0003:S:0:1'
> HIRMG:3:2+9010::Nachricht ist komplett nicht bearbeitet .+9800::Dialog
> abgebrochen.'
> HIRMS:4:2:2+9210::Inhaltlich ung?ltig.'

```
> HNSHA:5:1+4321+@96@<1c 3f dd 2c d7 74 e7 2c cb db f3 42 2e
    66
> d2 50 c7 e c4 6f 10 38 85 8d b1 23 41 db 25 1c f5 c7 47 df
    82
> f 41 5a 95 1d f9 f2 1f fa d5 dd 7b df 2 75 f1 67 fd fc a6 3f
> 2b 76 8b bc 58 f7 c7 bb e 9c 5 4f 1a f0 86 57 4 95 b3 e4
    75 52
> c6 c1 1d 27 29 81 10 20 5a 7 7d f2 fc 4b 39 f7 58 ba>'
> HNHBS:6:1+1'
>
> Wenn ich hkidn durch nachfolgendes Segment ersetze (letzte
   ziffer auf 1)
> HKIDN:3:2+280:09950003+Benutzer-000152-03+0+1'
>
> Bekomme ich die logische meldung falsche kundensystem-id.
> HIRMS:4:2:2+9210:6,3:Inhaltlich ung?ltig.'
>
> Benutzerkennung ist mit der Kunden-ID identisch!?
```

Sehr geehrter Herr öDrge,

so wie es im Moment aussieht, handelt es sich um ein serverseitiges Problem bzw. eine fehlende Dokumentation in der HBCI-Spec.. Ich werde mich in den änchsten Tagen mit der HBCI-Leitstelle in Verbindung setzten, um weiteres zu äklren.
Die HBCI-Spec. sieht zwar eine Synchronisation der Signatur-ID vor, allerdings ist diese Signatur-ID dann dem Kundensystem 0 zugeordnet (da ja noch keine Kundensystem-ID existiert). üFr die üüSchlsselbermittlung wird aber keine üPrfung der Signatur-IDü
durchgefhrt, da keine Doppeleinrichungskontrolle erforderlich ist, so dass die Synchronisation an dieser Stelleü überflssig ist.
Nach Einreichung der eigenen üSchlssel wird die Signatur-ID beginnend mit 1 verwendet.

Ich üwrde Ihnen empfehlen, äzunchst auf die Synchronisation zu verzichten und mit der üSchlsseleinreichung fortzufahren.

MfG.

Nico Marcussen-Wulff

C HBCI

HBCI Kreditinstitutsnachricht:

Dialoginitialisierung

Nr.	Name	Typ	Kennung	Status	Anzahl	Anmerkungen
1	Nachrichtenkopf	SEG	HNHBK	M	1	s. Kap. II.6.2
2	Signaturkopf	SEG	HNSHK	K	1	s. Kap. VI.5.2
3	Rückmeldungen zur Gesamtnachricht	SEG	HIRMG	M	1	s. Kap. II.8.2
4	Rückmeldungen zu Segmenten	SEG	HIRMS	K	n	s. Kap. II.8.3
5	Bankparameterdaten	SF		K	1	s. Kap. III.3.2.2
6	Userparameterdaten	SF		K	1	s. Kap. III.3.2.3
7	Übermittlung eines öffentlichen Schlüssels	SEG	HIISA	K	2	s. Kap. III.3.2.4
8	Kreditinstitutsmeldung	SEG	HIKIM	K	n	s. Kap. III.3.2.5
9	Signaturabschluss	SEG	HNSHA	K	1	s. Kap. VI.5.3
10	Nachrichtenabschluss	SEG	HNHBS	M	1	s. Kap. II.6.3

Synchronisierung

Nr.	Name	Typ	Kennung	Status	Anzahl	HBCI Kap.[1]
1	Nachrichtenkopf	SEG	HNHBK	M	1	II.6.2
2	Signaturkopf	SEG	HNSHK	K	1	VI.5.2
3	Rückmeldungen zur Gesamtnachricht	SEG	HIRMG	M	1	II.8.2
4	Rückmeldungen zu Segmenten	SEG	HIRMS	K	n	II.8.3
5	Bankparameterdaten	SF		K	1	III.3.2.2
6	Userparameterdaten	SF		K	1	III.3.2.3
7	Übermittlung eines öffentlichen Schlüssels	SEG	HIISA	K	2	III.3.2.4
8	Synchronisierungsantwort	SEG	HISYN	M	1	III.8.2.2
9	Kreditinstitutsmeldung	SEG	HIKIM	K	n	III.3.2.5
10	Signaturabschluß	SEG	HNSHA	K	1	VI.5.3
11	Nachrichtenabschluss	SEG	HNHBS	M	1	II.6.3

BDP

Nr.	Name	Typ	Kennung	Status	Anzahl	HBCI Kap.
1	Bankparameter allgemein	SEG	HIBPA	M	1	IV.2
2	Kommunikationszugang rückmelden	SEG	HIKOM	K	1	IV.3
3	Sicherheitsverfahren	SEG	HISHV	K	1	IV.4
4	Komprimierungsverfahren	SEG	HIKPV	K	1	IV.5
5	Parameterdaten	SF		K	1	IV.7

UDP

Nr.	Name	Typ	Kennung	Status	Anzahl	Anmerkungen
1	Userparameter allgemein	SEG	HIUPA	M	1	
2	Kontoinformation	SEG	HIUPD	K	n	

Parameterdaten : HIUEBS, HIKAZS, HISALS

Segmentgenerierung

Dialoginitialisierung

Nr.	Name	Typ	Kennung	Status	Anz.	HBCI Kap.
1	Nachrichtenkopf	SEG	HNHBK	M	1	II.6.2
2	Signaturkopf	SEG	HNSHK	M	1	VI.5.2
3	Identifikation	SEG	HKIDN	M	1	III.3.1.2
4	Verarbeitungsvorbereitung	SEG	HKVVB	M	1	III.3.1.3
5	Anforderung eines öffentlichen Schlüssels	SEG	HKISA	K	2	III.3.1.4
6	Signaturabschluss	SEG	HNSHA	M	1	VI.5.3
7	Nachrichtenabschluss	SEG	HNHBS	M	1	II.6.3

Dialogbeendigung

C.1 S.W.I.F.T.-Format

ISO 8859-Subset Zeichensatz, 7 Bit, <LF><CR><SP>'()+,-./<0-9>:?<A-Z><a-z>{}

```
SWIFT          := (Zeile <CR><LF>)+
Zeile          := Tag Feld ("/" Feld)+
Tag            := ":" Nummer ":"
Feld           := FreierText | Mehrzweckfeld | ...
Mehrzweckfeld := Text ("?" Text)+
```

MT940 S.W.I.F.T.-Format (K=Kannfeld,M=Muss)

id	K/M	Feld	
:21:	K	Bezugsreferenznummer	
:25:	M	Kontobezeichnung	
:28C:	M	Auszugsnummer	
:60a:	M	Anfangssaldo	
:61:	K	Umsatz	Wiederholungszyklus
:86:	K	Mehrzweckfeld	
:62a:	M	Schlusssaldo	
:64:	K	Aktueller Valutensaldo	
:65:	K	Zukünftige Valutensalden	
:86:	K	Mehrzweckfeld	

Beispiel:

```
:20:1234567
:21:9876543210
:25:10020030/1234567
:28C:5/1
:60F:C991101DEM2187,95
:61:9911011102DM800,NSTONONREF//55555/OCMT/EUR409,03/
:86:008?00DAUERAUFTRAG?100599?20Miete November
   ?3010020030?31234567?32MUELLER?34339
:61:9911021102CM3000,NTRFNONREF//55555/OCMT/EUR1533,88/
:86:051?00UEBERWEISUNG?100599?20Gehalt Oktober?21Firma
   Mustermann GmbH?3050060400?310847564700?32MUELLER?34339
:62F:C991131DEM4387,95
-
```

MT942 S.W.I.F.T.-Format

```
:20: M Auftragsreferenznummer
:21: K Bezugsreferenznummer
:25: M Kontobezeichnung
:28C: M Auszugsnummer
:34F: M Mindestbetrag (Kleinster Betrag der gemeldeten
   äUmstze)
:34F: K Mindestbetrag (Kleinster Betrag der gemeldeten Haben
   -äUmstze)
:13: M Erstellungszeitpunkt
|      K Wiederholungszyklus |
| :61: K Umsatz              |
| :86: K Mehrzweckfeld       |
:90D: K Anzahl und Summe der Soll-Buchungen
:90C: K Anzahl und Summe der Haben-Buchungen
```

D Inhalt der beigefügten CD

Die CD ist in drei Bereiche aufgeteilt. Im Verzeichnis midlet befindet sich das lauffähige Programm zur Kontoabfrage. Das Unterverzeichnis `midlet\prototypen` enthält die benutzten explorativen und experimentellen Prototype.

Im Verzeichnis src befinden sich der zugehörige Quelltext. Die Eigenentwicklungen sind in `src\xpresso`, die entwickelten Komponenten sind in `src\xpresso\xshbci`.

Das Verzeichnis `addon` enthält die benutzte Krypto-API von Bouncy Castle, sowie jHBCI, welches sich zur Erzeugung neuer Schlüssel einsetzen lässt.

Im Hauptverzeichnis ist die Diplomarbeit abgelegt.

Das zum Testen benötigte J2ME Wireless Toolkit, darf aus rechtlichen Gründen nicht mit auf die CD, man findet es unter [26].

Versicherung über Selbständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Ort, Datum Unterschrift des Studenten