IMPLEMENTASI APLIKASI *SMART HEALTH* PADA *SMART CARD* UI BERBASIS *JAVACARD*

SKRIPSI

oleh:

<u>DONNY</u>

04 03 03 0357

DEPARTEMEN TEKNIK ELEKTRO

FAKULTAS TEKNIK UNIVERSITAS INDONESIA

DEPOK

JANUARI, 2008

IMPLEMENTASI APLIKASI *SMART HEALTH* PADA *SMART CARD* UI BERBASIS *JAVACARD*

SKRIPSI

oleh:

<u>DONNY</u>
04 03 03 0357

SKRIPSI INI DIAJUKAN UNTUK MELENGKAPI SEBAGIAN PERSYARATAN MENJADI SARJANA TEKNIK

DEPARTEMEN TEKNIK ELEKTRO
FAKULTAS TEKNIK UNIVERSITAS INDONESIA
DEPOK
JANUARI, 2008

PERNYATAAN KEASLIAN SKRIPSI

Saya menyatakan dengan sesungguhnya bahwa Skripsi dengan judul:

IMPLEMENTASI APLIKASI *SMART HEALTH*
PADA *SMART CARD* UI BERBASIS *JAVACARD*

yang dibuat untuk melengkapi sebagian persyaratan menjadi Sarjana Teknik pada program studi Teknik Elektro Departemen Teknik Elektro Fakultas Teknik Universitas Indonesia, sejauh yang saya ketahui bukan merupakan tiruan atau duplikasi dari Skripsi yang sudah dipublikasikan dan atau pernah dipakai untuk mendapatkan gelar kesarjanaan di lingkungan Universitas Indonesia maupun di Perguruan Tinggi atau instansi manapun, kecuali bagian yang sumber informasinya dicantumkan sebagaimana mestinya.

Depok, 7 Januari 2008

(Donny)
0403030357

Implementasi aplikasi..., Donny, FT UI, 2008

PENGESAHAN

Skripsi dengan judul:

IMPLEMENTASI APLIKASI *SMART HEALTH*
PADA *SMART CARD* UI BERBASIS *JAVACARD*

dibuat untuk melengkapi sebagian persyaratan menjadi Sarjana Teknik pada Program Studi Teknik Elektro Departemen Teknik Elektro Fakultas Teknik Universitas Indonesia dan telah disidangkan pada tanggal 4 Januari 2008 dalam sidang ujian skripsi.

Depok, 7 Januari 2008
Dosen Pembimbing,

Dr. Ir. Riri Fitri Sari, MM, M.Sc
NIP. 132.127.785

Implementasi aplikasi..., Donny, FT UI, 2008

PERPUSTAKAAN FAKULTAS TEKNIK * UNIVERSITAS INDONESIA *

UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada:

Dr. Ir. Riri Fitri Sari, MM, M.Sc

Selaku dosen pembimbing yang telah bersedia meluangkan waktu untuk memberi pengarahan, diskusi dan bimbingan serta persetujuan sehingga skripsi ini dapat selesai dengan baik.

Donny
NPM 04 03 03 0357
Departemen Teknik Elektro

Dosen Pembimbing
Dr. Ir. Riri Fitri Sari, MM, M.Sc

IMPLEMENTASI APLIKASI *SMART HEALTH*
PADA *SMART CARD* UI BERBASIS *JAVACARD*

ABSTRAK

Smart Card adalah teknologi kartu yang didalamnya terdapat sebuah *chip* komputer. *Smart Card* dapat diprogram untuk menciptakan berbagai macam aplikasi dan dapat digunakan sebagai tempat penyimpanan data. *Smart Card* memiliki protokol yang disebut *Application Protocol Data Unit* (APDU) untuk mengatur proses komunikasi data. Salah satu jenis *Smart Card* adalah *JavaCard*. Sesuai namanya, *JavaCard* memiliki basis Java sehingga pemrograman pada *JavaCard* menggunakan bahasa pemrograman Java.

Dalam Skripsi ini dilakukan pembangunan sebuah aplikasi *JavaCard* yang berkaitan dengan bidang medis dan disebut *Smart Health*. Aplikasi ini berfungsi untuk membaca dan menulis data medis pada *JavaCard*. Data medis yang disimpan pada *JavaCard* dibagi menjadi enam kategori. Proses pembangunan aplikasi terdiri dari tiga bagian yaitu pembuatan applet *JavaCard*, applet *connector*, dan modul terminal. Applet *JavaCard* adalah program untuk mengatur proses pembacaan dan penulisan data medis pada *JavaCard*, serta untuk mengatur alokasi memori *JavaCard*. Applet *connector* berfungsi sebagai penghubung applet *JavaCard* dengan modul terminal. Modul terminal adalah aplikasi *Graphical User Interface* (GUI) yang diakses oleh pengguna.

Analisa terhadap kinerja aplikasi dilakukan dengan mempertimbangkan alokasi memori *JavaCard*, peran APDU dalam lalu lintas data, pemrosesan data, efektitifitas penggunaan memori, dan kecepatan kerja aplikasi. Selain itu, juga dilakukan pembahasan mengenai pengembangan aplikasi di masa mendatang. Analisa ini memberi hasil bahwa kunci dari pembuatan aplikasi *JavaCard* terletak pada pemrograman applet *JavaCard*, pengaturan APDU untuk memproses operasi-operasi aplikasi, dan alokasi memori.

Kata kunci: *Smart Card*, *JavaCard*, APDU, *Smart Health*, data medis.

Donny NPM 04 03 03 0357 Electrical Engineering	Supervisor Dr. Ir. Riri Fitri Sari, MM, M.Sc

SMART HEALTH APLICATION IMPLEMENTATION ON UI _SMART CARD JAVACARD_ BASED

ABSTRACT

Smart Card is a card technology that has an embedded microprocessor inside the card. It can be programmed to create application, perform task, and store information. Smart Card has a protocol called *Application Protocol Data Unit* (APDU), used to control data communication process. *JavaCard* is a kind of Smart Card. In conjuction with its name, *JavaCard* is a Java based *Smart Card* and uses Java programming language to create *JavaCard* application.

In this work, a *JavaCard* application called *Smart Health* has been developed. This application has two functions. First, to read medical records from *JavaCard* and second, to write medical records into the *JavaCard*. Medical record which will be saved in the *JavaCard* was split into six categories. *Smart Health* application has three main parts, *JavaCard* applet, connector applet, and terminal module. *JavaCard* applet is a program to control the reading and writing process on *JavaCard* and also controls *JavaCard* memory allocation. Connector applet is an interface program to connect the *JavaCard* applet with the terminal module. Terminal module is a *Graphical User Interface* (GUI) module. User interacts with the *Smart Card* application using this module.

Performance analysis of this application focuses on the *JavaCard* memory allocation, APDU role on data traffic, data process, memory optimalization, and application speed. In addition we also analyze further development of this *Smart Health* application. The result of the analysis shows that *JavaCard* applet programming, APDU arrangement to process operation, and memory allocation are the keys to build a *JavaCard* application.

Keywords: _Smart Card_, _JavaCard_, APDU, _Smart Health_, medical record.

DAFTAR ISI

PERNYATAAN KEASLIAN SKRIPSI ii
PENGESAHAN iii
UCAPAN TERIMA KASIH iv
ABSTRAK v
ABSTRACT vi
DAFTAR ISI vii
DAFTAR GAMBAR ix
DAFTAR TABEL x
DAFTAR LAMPIRAN xi
DAFTAR SINGKATAN xii
BAB 1 PENDAHULUAN 1
1.1 LATAR BELAKANG 1
1.2 PERUMUSAN MASALAH 2
1.3 TUJUAN 3
1.4 PEMBATASAN MASALAH 3
1.5 METODOLOGI PENELITIAN 3
1.6 SISTEMATIKA PENELITIAN 3
1.7 SISTEMATIKA PENULISAN 4
BAB 2 DASAR TEORI *SMART CARD* 5
2.1 DEFINISI SMART CARD 5
2.2 STANDAR SMART CARD 5
2.3 ARSITEKTUR SMART CARD 6
2.3.1 Karakteristik Fisik 6
2.3.2 Elemen Smart Card 8
2.4 KONEKSI SMART CARD 9
2.4.1 Smart Card Reader 9
2.4.2 Application Protocol Data Unit (APDU) 10
2.5 JAVACARD 12
2.5.1 Arsitektur Java Card 12
2.5.2 Java Card Virtual Machine (JCVM) 13
2.5.3 Java Card Runtime Environment (JCRE) 14
2.6 PROGRAMMING JAVACARD 15
2.6.1 JavaCard Framework 15

Implementasi aplikasi..., Donny, FT UI, 2008

2.6.2 OpenCard Framework (OCF) 16
2.7 UI CARD BUNDLE LIBRARY 17

BAB 3 PERANCANGAN APLIKASI *SMART HEALTH* 20

3.1 APPLET JAVACARD 20
3.2 APPLET CONNECTOR 26
3.3 PEMBUATAN MODUL TERMINAL 29
3.3.1 Modul InsertMedicalRecord 30
3.3.2 Modul ViewMedicalRecord 32
3.4 INTEGRASI SISTEM SMART HEALTH 34

BAB 4 ANALISA KINERJA APLIKASI *SMART HEALTH* 39

4.1 INSTALASI APLIKASI SMART HEALTH 39
4.2 ANALISA PROGRAM APLIKASI SMART HEALTH 39
4.3 HAL-HAL YANG MEMBATASI APLIKASI 50
4.3.1 Alokasi Memori JavaCard 50
4.3.2 Kompresi Data 52
4.3.3 Kecepatan Proses JavaCard 53
4.4 TINGKAT KEMUDAHAN PENGGUNAAN APLIKASI 54
4.5 PENGEMBANGAN APLIKASI MENDATANG 54

BAB 5 KESIMPULAN 56

DAFTAR ACUAN 57

LAMPIRAN 58

DAFTAR GAMBAR

Gambar 2.1 Ukuran standar dan komponen *Smart Card* [1] 7
Gambar 2.2 Karakteristik Fisik sebuah *Smart Card* [2] 7
Gambar 2.3 Delapan titik kontak [2] 7
Gambar 2.4 Elemen *Smart Card* [3] 8
Gambar 2.5 *Command* APDU [2] 11
Gambar 2.6 *Response* APDU [2] 11
Gambar 2.7 Arsitektur *JavaCard* [2] 12
Gambar 2.8 *JavaCard* Virtual Machine [4] 14
Gambar 2.9 *JavaCard* Runtime Environment [4] 14
Gambar 2.10 Arsitektur *OpenCard Framework* [5] 16
Gambar 2.11 Infrastruktur *Smart Card* UI [6] 17
Gambar 2.12 *Smart Card* UI 18
Gambar 3.1 *Class Diagram* Applet *JavaCard* 25
Gambar 3.2 *Activity Diagram* Applet *JavaCard* 26
Gambar 3.3 *Class Diagram* Applet *Connector* 28
Gambar 3.4 *Activity Diagram* Applet *Connector* 29
Gambar 3.5 *Class Diagram* Modul Terminal *Smart Health* 30
Gambar 3.6 Tampilan modul InsertMedicalRecord 31
Gambar 3.7 Tampilan modul ViewMedicalRecord 33
Gambar 3.8 *Use Case Diagram* sistem *Smart Health* 35
Gambar 3.9 *Sequence Diagram* pengambilan data pribadi 36
Gambar 3.10 *Sequence Diagram* pengambilan data medis 37
Gambar 3.11 *Sequence Diagram* penulisan data medis 38
Gambar 4.1 Perangkat keras aplikasi Smart Health 39
Gambar 4.2 Tampilan awal menu penulisan data medis 42
Gambar 4.3 Tampilan awal menu pembacaan data medis 43
Gambar 4.4 Tampilan aktif menu pembacaan data medis 44
Gambar 4.5 Tampilan saat penulisan data medis 45
Gambar 4.6 Tampilan saat pembacaan data medis 46
Gambar 4.7 Data Flow Diagram penulisan data medis 47
Gambar 4.8 Data Flow Diagram pembacaan data medis 48
Gambar 4.9 Diagram lingkaran penggunaan memori dengan data 5600 byte 51
Gambar 4.10 Diagram lingkaran penggunaan memori dengan data 5000 byte 52
Gambar 4.11 Diagram lingkaran penggunaan memori dengan data 4400 byte 52
Gambar 4.12 Diagram batang perbandingan data sebelum dan sesudah kompresi 53

Implementasi aplikasi..., Donny, FT UI, 2008

DAFTAR TABEL

TABEL 2.1 SUBSET PEMROGRAMAN *JAVACARD* [4] 15
TABEL 3.1 *COMMAND* APDU PENYIMPANAN DATA MEDIS 22
TABEL 3.2 *COMMAND* APDU PEMBACAAN DATA MEDIS 23
TABEL 3.3 *RESPONSE* APDU PEMBACAAN DATA MEDIS 24
TABEL 4.1 RINCIAN MEMORI SATU DATA MEDIS DAN DATA PRIBADI TAMBAHAN 41
TABEL 4.2 PESAN KESALAHAN APLIKASI *SMART HEALTH* 49
TABEL 4.3 PEMBOROSAN MEMORI *JAVACARD* 51
TABEL 4.4 PERBANDINGAN DATA SEBELUM DAN SESUDAH KOMPRESI 53
TABEL 4.5 TANGGAPAN RESPONDEN TERHADAP APLIKASI *SMART HEALTH* 54

Implementasi aplikasi..., Donny, FT UI, 2008

DAFTAR LAMPIRAN

LAMPIRAN 1 *LISTING* PROGRAM APPLET *JAVACARD* 59
LAMPIRAN 2 *LISTING* PROGRAM APPLET *CONNECTOR* 67
LAMPIRAN 3 *LISTING* PROGRAM MODUL TERMINAL 73

PERPUSTAKAAN FAKULTAS TEKNIK * UNIVERSITAS INDONESIA *

DAFTAR SINGKATAN

AID	Application Identifier
APDU	Application Protocol Data Unit
API	Application Programming Interface
CAD	Card Acceptance Device
CAP	Converted Applet
CPU	Central Processing Unit
EEPROM	Electrically Erasable Programmable Read Only Memory
GUI	Graphical User Interface
IC	Integrated Circuit
IFD	Interface Device
JAR	Java Archive
JCRE	JavaCard Runtime Environment
JCVM	JavaCard Virtual Machine
JDK	Java Development Kit
MIPS	Mega Instruction Per Second
OCF	OpenCard Framework
PC	Personal Computer
RAM	Random Access Memory
ROM	Read Only Memory

Implementasi aplikasi..., Donny, FT UI, 2008

BAB 1
PENDAHULUAN

1.1 *LATAR BELAKANG*

Salah satu tujuan dari perkembangan teknologi adalah bagaimana untuk membuat hidup manusia menjadi semakin mudah. Berbagai teknologi telah dan terus dikembangkan dalam mencapai tujuan ini, seperti teknologi transportasi, teknologi komunikasi, teknologi informasi dan perbankan, dan teknologi industri dan otomatisasi. Namun, teknologi yang telah disebutkan tadi tidak dapat memenuhi seluruh aspek kehidupan manusia.

Kehidupan manusia modern dipenuhi dengan berbagai sistem dan informasi. Informasi tersebut disimpan dan diakses manakala dibutuhkan. Dengan semakin kompleksnya informasi-informasi tersebut, dibutuhkan cara agar kegiatan-kegiatan yang melibatkan informasi tersebut berjalan dengan efisien dari segi waktu dan biaya, serta terjaminnya keamanan dari informasi-informasi tersebut.

Untuk mencapai tujuan tersebut, sedang dikembangkan sebuah teknologi baru dan coba diimplementasikan dalam memenuhi kebutuhan hidup manusia, diantaranya adalah teknologi *Smart Card*. Gagasan awal dari munculnya teknologi ini adalah bagaimana untuk membuat fungsi-fungsi dalam kartu pengenal, kartu ATM, kartu kredit, serta kartu telepon seluler, yang kesemuanya berisi informasi-informasi penting, tergabung menjadi satu dan manusia tidak akan kesulitan dengan banyaknya kartu yang ada dalam dompet mereka. Hal inilah yang coba diwujudkan dengan teknologi *Smart Card*.

Teknologi *Smart Card* muncul setelah kartu *magnetic stripe* yang diciptakan untuk tujuan yang kurang lebih sama. Namun data yang disimpan dalam *magnetic stripe* lebih sedikit kapasitasnya, mudah terhapus dan tidak aman untuk menyimpan informasi-informasi yang penting di dalamnya. Sementara Amerika Serikat mengembangkan jaringan komputer *online mainframe-based* sebagai solusi untuk

Implementasi aplikasi..., Donny, FT UI, 2008

processing dan verifikasi, Eropa mengembangkan inovasi yang lebih personal dengan menciptakan *Smart Card* yang membawa informasi yang diperlukan tersebut.

Smart Card adalah kartu berbahan plastik yang didalamnya terdapat sebuah *embedded microchip* kecil dalam *contact pad* keemasan pada sebelah sisi kartu, yang dapat diisikan dengan data dan diprogram untuk menangani berbagai macam aplikasi dalam kehidupan manusia. Untuk menggunakannya, *Smart Card* dimasukkan ke dalam sebuah slot pembaca namun bisa juga dibaca dari jarak jauh sehingga lebih praktis dan menghemat waktu. Tidak seperti *magnetic card*, *Smart Card* membawa semua fungsi dan informasi di dalamnya sehingga tidak memerlukan akses ke *remote database* pada saat penggunaannya.

Smart Card dapat diaplikasikan sebagai alat pembayaran elektronik atau *electronic wallet*, kartu telepon seluler, kartu mahasiswa, reservasi hotel, pemesanan tiket pesawat dan kendaraan penjemput, kartu pengenal untuk menggunakan jasa transportasi massa, kartu ijin mengemudi, kartu perpustakaan, dan kartu rumah sakit yang berisi catatan medis. Hal terakhir inilah yang dicoba untuk diwujudkan sebagai salah satu contoh aplikasi *Smart Card* yang dapat diimplementasikan dalam kehidupan sehari-hari. Data kesehatan adalah data yang bersifat sangat pribadi dan menjadi salah satu informasi penting yang wajib menyertai seseorang kemanapun dia pergi. Kepemilikan informasi tersebut merupakan kepentingan dasar seorang pasien dan tidak boleh dirahasiakan oleh sebuah institusi kesehatan manapun, karena informasi tersebut adalah hak milik pasien.

1.2 *PERUMUSAN MASALAH*

Pada Skripsi ini, aplikasi yang dibuat adalah aplikasi *Smart Card* berbasis *JavaCard* yang diimplementasikan dalam bidang medis sebagai sarana untuk menyimpan dan membaca catatan medis dari pemilik *Smart Card*. Aplikasi ini disebut dengan aplikasi *Smart Health*. Aplikasi *Smart Health* ini mencakup personalisasi pemilik *Smart Card* dan berisi data-data yang berhubungan dengan riwayat kesehatan pemilik *Smart Card*. Dengan demikian, setiap rumah sakit, klinik, dan Puskesmas dapat berbagi catatan medis pasien dengan menggunakan *Smart Card*.

Implementasi aplikasi..., Donny, FT UI, 2008

1.3 *TUJUAN*

Tujuan dari Skripsi ini, antara lain:

1. Mengimplementasikan Java dalam pembuatan aplikasi *Smart Health* dalam sistem *Smart Card* Universitas Indonesia.
2. Memahami arsitektur *JavaCard* dan mengerti bagaimana membuat suatu aplikasi dalam *JavaCard*.
3. Mengimplementasikan aplikasi *Smart Health* yang telah dibangun dalam bidang medis sebagai sarana penyimpanan catatan medis.
4. Melakukan evaluasi terhadap kinerja aplikasi *Smart Health* yang dibuat.

1.4 *PEMBATASAN MASALAH*

Pekerjaan yang didefinisikan dalam skripsi ini dibatasi pada pembuatan aplikasi *Smart Card* jenis *Java Card*. *Java Card* adalah *Smart Card* berbasis *Java environment*. Identifikasi pemilik *Smart Card* atau data personal dalam *Smart Card* dibatasi pada nama, NPM, kode organisasi, tanggal lahir, jenis kelamin, kewarganegaraan, status pernikahan, dan golongan darah. Catatan medis pada aplikasi *Smart Health* terdiri dari kode institusi, ID dokter, tanggal pemeriksaan, anamnesa (keluhan pasien), diagnosa, dan terapi (pengobatan yang diberikan).

1.5 *METODOLOGI PENELITIAN*

Penelitian meliputi tahap analisis *requirement* aplikasi, perancangan aplikasi, implementasi, pengujian, dan *maintenance* aplikasi. Hasil penelitian ini akan dianalisa, juga dilakukan evaluasi kelebihan dan kekurangan dari kinerja aplikasi, dan sejauh mana aplikasi *Smart Card* yang dibangun dapat dikembangkan dan mencakup bidang-bidang lain yang lebih luas.

1.6 *SISTEMATIKA PENELITIAN*

Penelitian dilakukan mulai dari proses mengumpulkan informasi mengenai kategori medis yang dibutuhkan, pembelajaran mengenai *Smart Card*, khususnya *JavaCard*, pembangunan program aplikasi, dan mengevaluasi serta menganalisa hasil kerja aplikasi yang telah dibuat.

Implementasi aplikasi..., Donny, FT UI, 2008

1.7 *SISTEMATIKA PENULISAN*

Agar pembahasan masalah menjadi lebih sistematis, Skripsi ini dibagi dalam beberapa bab, masing-masing membahas pokok bahasan tertentu dalam mencapai tujuan akhir dari Skripsi ini.

Bab pertama, Pendahuluan, berisi latar belakang permasalahan, tujuan, perumusan masalah, pembatasan masalah, metodologi penelitian, sistematika penelitian, dan sistematika penulisan.

Bab kedua menjelaskan landasan dan dasar pengetahuan dari penelitian yang dilakukan dalam Skripsi ini, yaitu dasar pengetahuan *Smart Card*, arsitektur *Smart Card*, *JavaCard*, pemrograman *JavaCard*, dan UI *Card Bundle Library*.

Bab ketiga menggambarkan langkah-langkah pembuatan aplikasi *Smart Health* mulai dari pembuatan applet *JavaCard*, applet *connector*, modul terminal, hingga integrasi sistem *Smart Health*.

Bab keempat berisi analisa dan pembahasan mengenai evaluasi dari kinerja aplikasi *Smart Health* yang dibuat serta pengembangan aplikasi mendatang.

Bab kelima berisi kesimpulan mengenai pembuatan aplikasi *JavaCard* secara umum dan aplikasi *Smart Health* secara khusus.

Implementasi aplikasi..., Donny, FT UI, 2008

BAB 2
DASAR TEORI *SMART CARD*

2.1 *DEFINISI SMART CARD*

Smart Card adalah kartu yang berukuran seperti kartu kredit tetapi memiliki sebuah *chip* computer atau mikroprosesor di dalamnya. Kartu ini dapat diprogram untuk melakukan suatu tugas dan menyimpan informasi, tetapi harus diingat bahwa mikroprosesor pada kartu tidak sekuat mikroprosesor *Personal Computer* (PC), sehingga kemampuannya tidak dapat disamakan dengan PC. Untuk mengkoneksikan kartu dan komputer diperlukan *Card Acceptance Device* (CAD), seperti *card reader*, yang bertindak sebagai terminal penghubung atau *interface device*.

Terdapat dua tipe dasar dari *Smart Card*, yaitu *intelligent Smart Card* dan *memory card*. Pada *Intelligent Smart Card* terdapat mikroprosesor dan memiliki kemampuan untuk membaca, menulis, dan menghitung, seperti komputer kecil. Sebaliknya, *memory card*, tidak memiliki mikroprosesor di dalamnya dan hanya diperuntukkan bagi penyimpanan informasi.

Penggunaan *Smart Card* memiliki beberapa keuntungan jika dibandingkan dengan *magnetic card*. Keuntungannya adalah lebih dapat diandalkan, dapat menyimpan informasi ratusan kali lebih banyak, lebih sulit untuk dirusak atau dipalsukan, cukup mudah dalam pemrogramannya, mampu melakukan berbagai fungsi dalam jangkauan industri yang luas, kompatibel dengan perangkat elektronik yang mudah dibawa seperti ponsel, *notebook*, PC dan merupakan teknologi yang terus berkembang.

2.2 *STANDAR SMART CARD*

Secara umum, terdapat tiga standar dari *Smart Card* yang dikembangkan dalam industri di dunia saat ini, yaitu :

Implementasi aplikasi..., Donny, FT UI, 2008

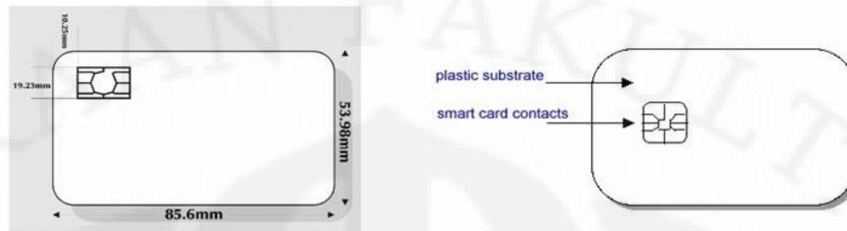
1. **PC/SC:** Model ini diperkenalkan oleh Microsoft dan beberapa perusahaan lainnya. *PC/SC* adalah antarmuka aplikasi *Smart Card* untuk berkomunikasi dengan *Smart Card* dari PC yang berbasis Win32.
2. ***OpenCard Framework*****:** adalah sebuah standar terbuka yang menyediakan operasi antar aplikasi *Smart Card* lewat NC, POS, *desktop*, *notebook*, *set top box*, dan sebagainya. *OpenCard* biasanya harus digunakan dalam komunikasi dengan perangkat eksternal dan/atau menggunakan *library* yang ada di *client*. Selain itu, *OpenCard* juga menyediakan antarmuka ke PC/SC agar dapat digunakan untuk perangkat yang berbasis Win32.
3. ***JavaCard***: diperkenalkan pertama kali oleh *Schlumberger* dan hanya menggunakan *JavaCard* sebagai satu-satunya kartu yang dipasarkannya saat ini, sekaligus sebagai perusahaan pertama yang memiliki lisensi *JavaCard* yang berstandar ISO 7816.

Smart Card dapat dipergunakan atau diaplikasikan dalam kehidupan manusia, antara lain dalam pelayanan perbankan, *electronic wallet*, transportasi, catatan medis, pelayanan reservasi, universitas, dan lain sebagainya. Aplikasi *Smart Card* juga dapat diperluas dengan menggunakan layanan Internet sehingga dengan demikian akan semakin mempermudah pengguna dalam bertransaksi.

2.3 *ARSITEKTUR SMART CARD*

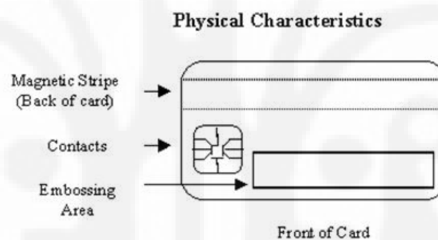
2.3.1 Karakteristik Fisik

Ada dua bentuk dari *Smart Card*, yaitu *contact Smart Card* dan *contactless Smart Card*. Pada *contact Smart Card* koneksi dibuat pada saat *reader* dari *Smart Card* berhubungan dengan *chip* emas kecil pada bagian depan kartu. Pada *contactless Smart Card*, komunikasi dapat dilakukan menggunakan antena. Dengan hanya mendekatkan kartu ke *receiver* dari antena maka kartu terssebut dapat berkomunikasi. *Contactless Smart Card* dapat digunakan pada aplikasi dimana proses *insertion* atau *removal* kartu dirasa tidak praktis atau situasi dimana kecepatan proses itu penting.



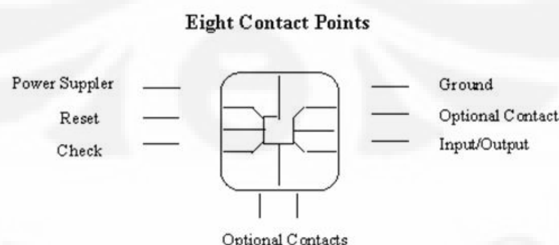
Gambar 2.1 Ukuran standar dan komponen *Smart Card* [1]

Ukuran standar sebuah *Smart Card* ditentukan oleh ISO 7816. Jika memperhatikan kartu telepon seluler, maka akan terlihat bahwa terdapat dua buah komponen yaitu *Smart Card* itu sendiri dan substrat plastiknya yang menjaga *Smart Card* dari kerusakan sebelum digunakan. Karakter fisik dari *Smart Card* berdasar standar ISO 7816 dapat dilihat pada beberapa aplikasi seperti kartu kredit, kartu ATM dan sebagainya seperti Gambar 2.2.



Gambar 2.2 Karakteristik Fisik sebuah *Smart Card* [2]

Normalnya, sebuah *Smart Card* tidak memiliki catu daya, layar, atau keyboard. *Smart Card* berinteraksi dengan dunia luar menggunakan antarmuka komunikasi serial lewat delapan titik kontak atau *contact point* yang ada. Berikut adalah gambar delapan titik kontak *Smart Card*.

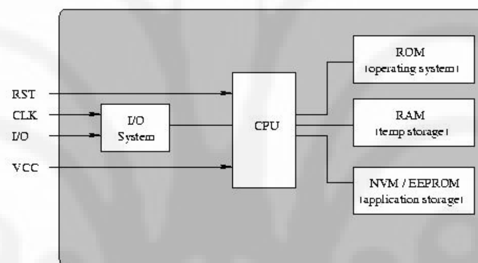


Gambar 2.3 Delapan titik kontak [2]

2.3.2 Elemen Smart Card

2.3.2.1 Central Processing Unit (CPU)

Central Processing Unit dari *Smart Card* pada umumnya merupakan mikrokontroler 8-bit tetapi ada juga yang menggunakan mikrokontroler 16 bit dan 32 bit. Mikrokontroler pada *Smart Card* tidak memiliki kemampuan *multi-threading* dan fitur–fitur yang kuat lain seperti pada komputer standar. CPU *Smart Card* menjalankan instruksi pada kecepatan lebih kurang 1 MIPS. Beberapa *Smart Card* juga dilengkapi dengan koprosesor untuk meningkatkan kecepatan komputasi enkripsi.



Gambar 2.4 Elemen *Smart Card* [3]

2.3.2.2 Memori

Smart Card memiliki tiga jenis memori di dalamnya :

1. *Persistent Non-mutable Memory*

   Memori ini merupakan *Read Only Memory* (ROM) dan memiliki kapasitas antara 8 sampai 24 Kilobyte. Sistem operasi dan beberapa perangkat lunak dasar, seperti algoritma enkripsi disimpan pada memori ini.

2. *Persistent Mutable Memory*

   Memori ini merupakan *Electrically Erasable Programmable Read Only Memory* (EEPROM) dan digunakan untuk menjalankan aplikasi dan untuk menulis aplikasi ke dalam *Smart Card*. Akan tetapi, kerja memori ini sangat lambat dan memiliki keterbatasan dalam penulisan, hanya dapat

ditulisi sebanyak 100.000 kali. Ukuran dari memori ini sebesar 1 sampai 24 Kilobyte.

3. *Non-persistent Mutable Memory*

   Memori ini merupakan *Random Access Memory* (RAM) dan dipergunakan untuk keperluan komputasi dan respons yang cepat. Memori ini berukuran kecil, hanya sebesar 1 Kilobyte.

2.3.2.3 Input/Output

Proses *input/output* dalam *Smart Card* dilakukan melalui port I/O tunggal yang dikontrol oleh mikroprosesor untuk menjamin bahwa komunikasi kartu telah distandarisasi, dalam bentuk *Application Protocol Data Unit* (APDU).

2.3.2.4 Interface Device (IFD)

Smart Card memerlukan listrik dan sinyal *clock* untuk menjalankan program tetapi tidak terdapat dalam *Smart Car*d itu sendiri. Hal ini disediakan oleh *Interface Device*, biasanya adalah *Smart Card reader*, pada *contact* dari kartu. *Smart Card reader* bertanggung jawab untuk memulai jalur komunikasi antara perangkat lunak aplikasi pada komputer dengan sistem operasi pada *Smart Card*. *Smart Card reader* memiliki kemampuan membaca dan menulis aplikasi pada *Smart Card*.

Jalur komunikasi pada *Smart Card* adalah *half-duplex*. Hal ini berarti bahwa data dapat dikirim dari IFD ke *Smart Card* atau dari *Smart Card* ke IFD, tetapi data tersebut tidak dapat dikirimkan pada waktu yang sama. Data yang dikirim atau diterima oleh *Smart Card* disimpan dahulu di dalam *buffer* pada RAM. Karena RAM dari *Smart Card* tidak besar, maka dalam pemindahan atau pengiriman data, hanya 10–100 byte data yang dapat diproses.

2.4 ***KONEKSI SMART CARD***

2.4.1 ***Smart Card Reader***

Smart Card reader digunakan untuk berkomunikasi dengan *Smart Card* dan membuat serta mengembangkan aplikasi berbasis *Smart Card*. *Smart Card reader*

Implementasi aplikasi..., Donny, FT UI, 2008

menyediakan jalur bagi aplikasi yang telah dibuat untuk mengirim dan menerima perintah dari *Smart Card*. Aplikasi berkomunikasi dengan *Smart Card reader*, yang kemudian akan berkomunikasi dengan *Smart Card*, menggunakan protokol standar, yaitu ISO 7816. ISO 7816 adalah standar yang mendeskripsikan antarmuka level paling bawah dengan *Smart Card*, yang pada level ini byte–byte data ditransfer antara *card reader* dan *Smart Card*. Standar ini juga mendefinisikan karakteristik mekanik dan elektrik selain protokol komunikasi dengan *Smart Card*.

Alur komunikasi dengan *Smart Card* adalah half-duplex. Hal ini berarti bahwa komunikasi dengan *Smart Card* dapat berlangsung secara dua arah namun bergantian satu dengan yang lainnya. Hubungan antara *Smart Card* dengan *reader* adalah hubungan *master–slave*. Dengan kata lain, *Smart Card* memainkan peran pasif, selalu menunggu perintah dari terminal *reader*. *Smart Card reader* melakukan koneksi dengan *contact point* dan mentransfer data. *Smart Card* akan memproses data dan mengembalikan data kepada *reader*, yang akan mengembalikan ke aplikasi.

Untuk mengirim perintah ke *Smart Card*, pertama harus memilih perintah yang didukung oleh *Smart Card*, dikemas ke dalam paket perintah ISO, dan mengemas kembali perintah baru ini ke dalam bentuk yang dibutuhkan oleh *Smart Card reader*. Pertukaran data dan perintah antara *Smart Card* dengan *Smart Card reader* dilakukan dalam bentuk paket, yang disebut dengan paket *Application Protocol Data Unit* (APDU).

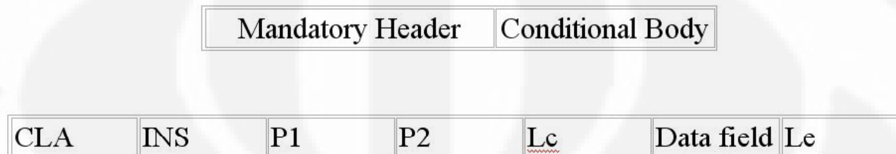
2.4.2 Application Protocol Data Unit (APDU)

Application Protocol Data Unit (APDU) adalah pesan perintah yang dikirimkan dari *application layer*, dan pesan respons yang dikembalikan oleh *Smart Card* ke *application layer*. Komunikasi dengan *Smart Card* dan *card reader* dilakukan menggunakan APDU. Sebuah APDU dapat dipertimbangkan sebagai paket data yang berisi instruksi lengkap atau respons lengkap dari *Smart Card*. Terdapat dua macam APDU, yaitu *Command* APDU dan *Response* APDU. *Smart Card* selalu menunggu *Command* APDU dari terminal, kemudian mengeksekusi perintah di

Implementasi aplikasi..., Donny, FT UI, 2008

dalamnya dan menjawab kepada terminal dengan *Response* APDU. *Command* APDU dan *Response* APDU saling bertukar satu sama lain antara *Smart Card* dan terminal.

Command APDU



Mandatory Header				Conditional Body		
CLA	INS	P1	P2	Lc	Data field	Le

Gambar 2.5 *Command* APDU [2]

Mandatory Header mengkodekan perintah yang dipilih. Terdiri dari empat bagian: *Class* (CLA), *Instruction* (INS), parameter 1 (P1), dan parameter 2 (P2). Setiap bagian berukuran 1 byte.

- CLA: *Class* byte. Pada kebanyakan *Smart Card*, byte ini dipergunakan untuk mengidentifikasi sebuah aplikasi.
- INS: *Instruction* byte. Byte ini mengindikasikan kode instruksi.
- P1-P2: Parameter byte. Byte ini digunakan untuk memberikan informasi kualifikasi selanjutnya kepada perintah dalam APDU.

Lc berisi banyaknya byte pada *data field* dari *Command* APDU. *Le* berisi jumlah maksimum byte yang diharapkan pada *data field* dari *Response* APDU sebagai jawaban.

Response APDU

Conditional Body	Mandatory Trailer	
Data field	SW1	SW2

Gambar 2.6 *Response* APDU [2]

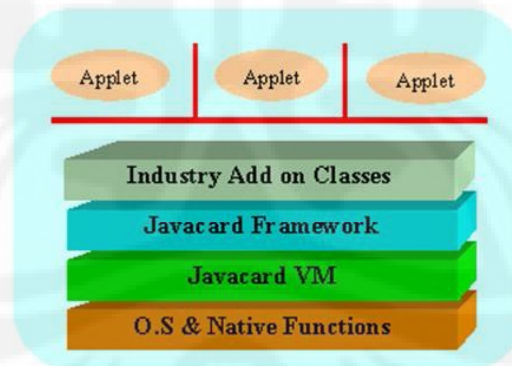
Pada *Response* APDU, *status byte* SW1 dan SW2 mendefinisikan status pemrosesan dari *Command* APDU di dalam *Smart Card*.

Terdapat beberapa kelas yang disediakan untuk mentransportasikan APDU, yaitu `Command`, `Response`, `ISOCardReader`, dan `ISOCommand`. Kelas `Command` berfungsi mengenkapsulasi *Command* APDU. `Response` berfungsi mengenkapsulasi *Response* APDU. `ISOCardReader` membentuk sebuah antarmuka dan harus diimplementasikan oleh setiap perangkat. `ISOCommand` membangun sebuah perintah `ISOCommand` dan dieksekusi melalui antarmuka yang dibentuk oleh `ISOCardReader`.

2.5 *JAVACARD*

JavaCard adalah *Smart Card* yang mampu untuk menjalankan program-program yang berbasis Java. Dengan menggunakan Java, maka pembuatan aplikasi untuk *Smart Card* akan semakin mudah. *Programmer* dapat membuat *Java Card Virtual Machine* (JCVM) dan *Application Programming Interface* (API) dalam *Smart Card*. Sistem minimum yang diperlukan adalah 16 Kilobyte ROM, 8 Kilobyte EEPROM, dan 256 byte RAM.

2.5.1 Arsitektur Java Card



Gambar 2.7 Arsitektur *JavaCard* [2]

Java Card Virtual Machine (JCVM) dibangun diatas rangkaian *Integrated Circuit* (IC) dan implementasi sistem operasi dan fungsi–fungsi awal. Lapisan JCVM menyembunyikan teknologi *proprietary* fabrikasi dengan bahasa yang umum dan antarmuka sistem. *Java Card Framework* mendefinisikan seperangkat kelas

Implementasi aplikasi..., Donny, FT UI, 2008

Application Programming Interface (API) untuk mengembangkan aplikasi *JavaCard* dan menyediakan layanan sistem untuk aplikasi tersebut. Bidang industri atau bisnis dapat menambahkan *add-on library* untuk membentuk sebuah layanan atau memperbaiki keamanan dan model dari system. Aplikasi *JavaCard* disebut dengan applet. Berbagai applet dapat saling berdampingan dalam satu kartu. Setiap applet diidentifikasikan secara unik oleh *Application Identifier* (AID).

2.5.2 Java Card Virtual Machine (JCVM)

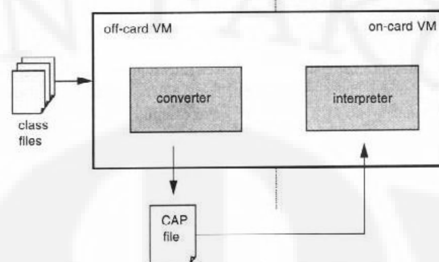
JCVM terbagi menjadi dua bagian, *converter* yang bekerja di luar *JavaCard* dan *interpreter* yang bekerja di dalam *JavaCard*. Saat sebuah *class file* diberikan dan *export file* yang dibutuhkan telah dibuat oleh Java *compiler*, *converter* melakukan hal-hal berikut :

1. Memeriksa pelanggaran *JavaCard language subset*
2. Memecahkan referensi simbolik ke bentuk yang lebih *compact* sehingga dapat ditangani dengan lebih efisien pada *JavaCard*
3. Mengoptimisasi *byte code*
4. Mengalokasikan penyimpanan dan membentuk *virtual machine data structures* untuk mewakili kelas-kelas
5. Menghasilkan *file* biner *Converted Applet* (CAP) yang dapat dieksekusi dan bekerja pada *interpreter* dan *export file*

Setelah menerima *file* CAP, interpreter bertanggung jawab untuk:

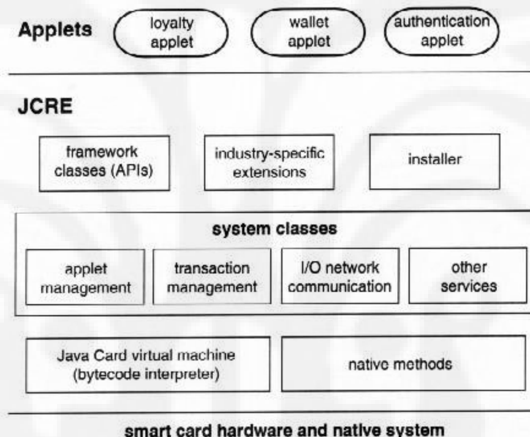
1. Mengeksekusi *byte code* yang diterima
2. Mengontrol alokasi memori dan penciptaan objek
3. Menjamin keamanan dari waktu kerja (*runtime*)

Implementasi aplikasi..., Donny, FT UI, 2008



Gambar 2.8 *JavaCard* Virtual Machine [4]

2.5.3 Java Card Runtime Environment (JCRE)



Gambar 2.9 *JavaCard* Runtime Environment [4]

JVCM menangani alokasi memori, eksekusi *byte code*, manajemen objek, dan keamanan. *Native methods* menangani protokol komunikasi *low-level*, manajemen memori, dan mendukung kriptografi. *System classes* menangani tugas – tugas yang biasanya ditangani oleh pusat sistem operasi dan menginstruksikan *native methods* untuk menjalankannya. API *classes* berisi kumpulan instruksi Java (Java library) yang mempermudah untuk membuat applet. *Installer* bertugas memuat applet ke dalam *JavaCard*. Perluasan dari JCRE dapat berupa *industry-specific library*. Sebagai contoh, *Open Platform* memperluas layanan JCRE untuk memenuhi kebutuhan keamanan khusus dalam aplikasi seperti bank.

JCRE aktif dan mulai menyalin data atau program dari EEPROM dan ROM ke RAM, yang lebih cepat, saat *JavaCard* dimasukkan ke *Card Acceptance Device*

Implementasi aplikasi..., Donny, FT UI, 2008

(CAD). Pada waktu transaksi, data dan objek yang harus dilindungi disalin oleh EEPROM dari RAM. EEPROM tetap menyimpan data tersebut meski tidak ada listrik.

2.6 *PROGRAMMING JAVACARD*

JavaCard adalah *Smart Card* yang berbasis Java. Oleh karena itu, program *JavaCard* ditulis menggunakan bahasa pemrograman Java. Program Java tersebut di-*compile* menggunakan *compiler* Java pada umumnya. Akibat keterbatasan memori dan kemampuan komputasi, tidak semua fitur–fitur yang terdefinisi dalam *Java Language Specification* didukung oleh *JavaCard*. Berikut adalah fitur–fitur yang didukung dan tidak didukung dalam *JavaCard*.

Tabel 2.1 Subset pemrograman *JavaCard* [4]

Fitur Java yang didukung	Fitur Java yang tidak didukung
Boolean, byte, short, integer	Double, float, long, character, string
Array satu dimensi	Array multi dimensi
Kelas, paket, antarmuka, *exception* Java	*Dynamic Class Loading*
Fitur *object-oriented* Java: *inheritance*, *virtual method*, *overloading*, dan *dynamic object creation*.	*Security Manager*
	Garbage Collection
	Thread

2.6.1 JavaCard Framework

JavaCard Framework dirancang untuk memudahkan serta mendukung pembuatan sistem dan aplikasi S*mart Card*. *JavaCard Framework* menyembunyikan infrastruktur S*mart Card* dan memberikan antarmuka pemrograman yang langsung dan relatif mudah bagi pengembang aplikasi. *Java Card Framework* terdiri dari tiga paket:

1. `javacard.framework`: merupakan paket inti pada *JavaCard*. Mendefinisikan kelas – kelas seperti `Applet` dan `PIN`, yang merupakan kelas fundamental dalam membangun blok–blok program *JavaCard*. Kelas `APDU`,

Implementasi aplikasi..., Donny, FT UI, 2008

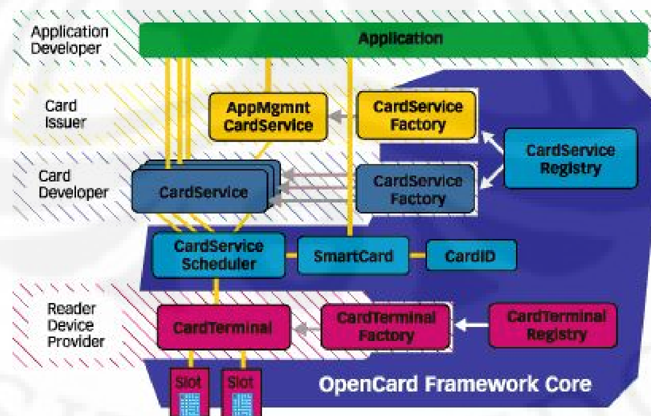
`System` dan `Util` menyediakan *runtime* dan layanan sistem untuk program *JavaCard*, seperti penanganan APDU dan pembagian objek.

2. `javacardx.crypto` dan `javacardx.cryptoEnc`: Kedua paket ini mendukung fungsi–fungsi kriptografi yang diperlukan di dalam *Smart Card*.

2.6.2 OpenCard Framework (OCF)

OpenCard Framework merupakan sebuah standar terbuka dalam mengembangkan solusi *end-to-end* menggunakan *Smart Card* yang tidak terikat hanya pada satu *platform*, jenis kartu, atau aplikasi. *OpenCard* memungkinkan hal tersebut dengan sebuah arsitektur yang terdiri dari dua subsistem primer, subsistem untuk *card terminal* dan untuk *card service*. *Card terminal* adalah perangkat dimana *Smart Card* dimasukkan ke dalamnya, seperti *Smart Card reader*. *Card service* adalah layanan yang digunakan oleh sebuah aplikasi luar untuk berkomunikasi dengan aplikasi di dalam *Smart Card*, yang dimasukkan ke dalam terminal. Berikut adalah beberapa *card service* yang terdapat dalam OCF:

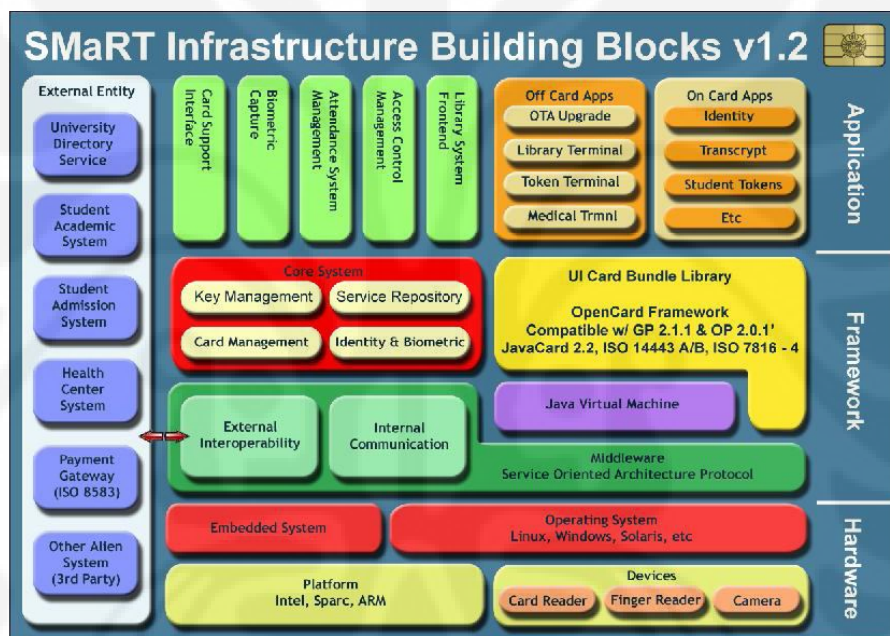
1. `FileAccessCardService`: *card service* yang digunakan untuk berinteraksi dengan file–file yang berbasis *Smart Card*.
2. `SignatureCardService`: abstraksi sederhana untuk mengimpor, verifikasi, dan mengekspor informasi *key*, baik *public key* maupun *private key*.



Gambar 2.10 Arsitektur *OpenCard Framework* [5]

2.7 *UI CARD BUNDLE LIBRARY*

UI Card Bundle Library adalah sebuah bundel yang dibuat khusus oleh Direktorat Pengembangan dan Pelayanan Sistem Informasi (PPSI)-Universitas Indonesia sebagai *platform* pengembangan bagi aplikasi-aplikasi *on-card* atau berhubungan dengan kartu. *Library* ini berlisensi GPL atau *Open Source* dan dapat digunakan oleh siapa saja untuk pengembangan aplikasi *Smart Card*. Berikut ini adalah gambar dari infrastruktur *Smart Card* UI yang telah dibangun.



Gambar 2.11 Infrastruktur *Smart Card* UI [6]

UI Card Bundle Library terdiri dari 5 bagian, yaitu:

1. *smartcard-common*: adalah library yang berisi program-program untuk mengatasi *JavaCard Exception*. *JavaCard Exception* adalah error atau kesalahan yang timbul saat proses komunikasi dengan *JavaCard*, seperti `Invalid PIN Exception`.
2. *smartcard-connector*: adalah library yang berisi program yang mendukung proses pembangunan applet pada *JavaCard*, seperti *Card Event*, *Command* APDU, dan *Response* APDU.

3. *smartcard-data*: adalah library yang berisi applet-applet *JavaCard* yang telah dibuat seperti *Smart SIAK* dan *Digital Transcript*.
4. *smartcard-masterkey*: adalah library yang berisi *Master Key* dari *Smart Card* yang dipergunakan oleh UI. *Master Key* adalah sebuah istilah untuk menyebut sebuah kunci kriptografi untuk mengakses *Smart Card* secara aman, baik dari segi otentikasi, integritas data, maupun keamanan data. *Master Key* ini digunakan untuk membuka *secure channel* untuk komunikasi dengan applet pada *Smart Card*.
5. *smartcard-perso-service-client*: adalah library yang berisi program-program yang ditujukan untuk personalisasi *Smart Card* dan applet-applet yang terdapat di dalamnya seperti *Applet Manager* dan *Update Master Key*.

Universitas Indonesia sudah menyelesaikan beberapa aplikasi yang siap diterapkan sebagai layanan tambahan bagi sivitas akademika UI, antara lain adalah aplikasi *Smart SIAK* dan *Digital Transcript*. *Smart SIAK* adalah aplikasi khusus untuk mengakses sistem *SIAK-NG* dimana proses otentikasi pengguna akan dilakukan secara aman dengan menggunakan teknologi *Smart Card* dan pengenalan data biometrik sidik jari. *Digital Transcript* adalah aplikasi penyimpanan transkrip akademik mahasiswa UI secara digital dengan menggunakan *Smart Card* dengan sifat otentik dan legalitas yang tetap menyatu dengan institusi UI.

Gambar 2.12 *Smart Card* UI

Selain dua aplikasi di atas, UI juga mengembangkan beberapa aplikasi lain, diantaranya aplikasi absensi, kuota *printing*, dan kuota parkir. Aplikasi *Smart Health* adalah aplikasi berikutnya yang akan dibangun oleh UI. Tujuan aplikasi *Smart Health* adalah agar setiap warga UI memiliki catatan medis yang tersimpan dalam *Smart*

Implementasi aplikasi..., Donny, FT UI, 2008

Card dan agar institusi-institusi kesehatan dapat berbagi data medis pasien melalui *Smart Card*. Penelitian ini dilakukan untuk mencapai tujuan tersebut. Penelitian dilakukan mulai dari penentuan spesifikasi, perancangan, implementasi, hingga evaluasi dan *maintenance* aplikasi. UI adalah institusi pertama di Indonesia dalam pembangunan aplikasi kesehatan menggunakan *Smart Card* dan diharapkan menjadi standar bagi pengembangan aplikasi kesehatan berbasis *Smart Card*. Spesifikasi aplikasi akan dibuka bagi pihak-pihak lain yang ingin mengembangkan aplikasi serupa.

BAB 3
PERANCANGAN APLIKASI *SMART HEALTH*

Pada pembuatan aplikasi *Smart Health* menggunakan *JavaCard* ini, penulis melewati beberapa tahapan. Tahapan pertama adalah membuat applet pada *JavaCard*, dilanjutkan dengan pembuatan aplikasi *connector* yang dikoneksikan dengan aplikasi modul terminal pada pengguna. Pembuatan aplikasi *connector* ini memiliki tujuan untuk menghubungkan applet *JavaCard* dengan aplikasi modul terminal sehingga operasi pembacaan dan penulisan data pada *JavaCard* dapat dilakukan dalam satu tahapan proses. Dengan kata lain, aplikasi *connector* adalah antarmuka antara *JavaCard* dan modul terminal. Sedangkan aplikasi modul terminal adalah sebuah aplikasi *end user* atau aplikasi yang dipergunakan oleh pengguna untuk menjalankan aplikasi *Smart Health* secara keseluruhan.

3.1 *APPLET JAVACARD*

Pembuatan applet *JavaCard* dilakukan dengan menggunakan bantuan perangkat lunak *Smart Cafe Professional 2.0* yang berbasis Java. *Smart Cafe Professional* adalah perangkat lunak yang ditujukan untuk mempermudah pembuatan applet pada *JavaCard* dan dilengkapi dengan perangkat simulasi applet. Dalam simulasi applet *JavaCard*, dapat dilihat bagaimana kerja dari applet, apakah fungsi–fungsi pada applet sudah berjalan sesuai *requirement* dan memeriksa adakah kekurangan atau *bug* pada applet. Pembuatan applet *JavaCard* menggunakan library *javacard.framework* yang berisi *Application Programming Interface* (API) pendukung pembuatan applet *JavaCard*.

Applet *Javacard* terdiri dari satu kelas yang dinamakan `PKMApplet` dengan beberapa method yaitu `InitializePIN()`, `install()`, `process()`, `setData()`, `sendData()`, `resetData()`, `getRecordLength()`, `sendAddData()`, dan `setAddData()`. Prinsip utama dalam pembuatan applet *JavaCard* adalah mengatur

lalu lintas *Application Protocol Data Unit* (APDU) untuk proses pembacaan dan penulisan data dalam *JavaCard*. Semua operasi pada *JavaCard* dilakukan dengan menggunakan APDU.

Method `InitializePIN()` digunakan untuk otentifikasi pemilik *JavaCard* sehingga menjamin keamanan informasi yang terdapat dalam *JavaCard*. Method `install()` digunakan untuk melakukan instalasi applet yang telah dibuat ke dalam *JavaCard* dan merupakan bagian dari *javacard.framework*. Dalam method ini ada beberapa parameter yang diinstalasi, yaitu *Application Identifier* (AID) dan PIN untuk mengakses informasi dalam *JavaCard*. Method `process()` adalah method yang digunakan untuk menspesifikasi APDU yang akan digunakan dalam applet dan mengatur proses lalu lintas data pada *JavaCard*. Setiap parameter dari APDU diinisialisasi dan instruksi-instruksi yang akan dilakukan pada applet diatur pada method ini.

Method `setData()` adalah method untuk menulis data pada *JavaCard*. Pada method ini digunakan sebuah array `apduBuffer[]` yang berfungsi sebagai *buffer* untuk menyimpan data yang ingin ditulis untuk kemudian di simpan dalam array `medicalRecord[]` pada memori *JavaCard*. Array `medicalRecord[]` dibagi menjadi blok-blok sehingga data yang ditulis tidak tercampur satu sama lain. Method `sendData()` adalah method untuk membaca data yang tersimpan dalam array `medicalRecord[]` pada *JavaCard*. Data yang tersimpan dalam array tersebut dibawa oleh *Response* APDU.

Method `resetData()` adalah method yang digunakan untuk mengulang penulisan data karena adanya *error* dalam penulisan data. Method `getRecordLength()` adalah method yang digunakan untuk mengetahui berapa banyak data medis yang telah tersimpan dalam *JavaCard*.

Method `sendAddData()` adalah method yang digunakan untuk membaca data pribadi tambahan, yaitu data kewarganegaraan dan status pernikahan, yang tersimpan dalam *JavaCard*. Data pribadi selain kedua data tersebut diambil dari applet *JavaCard* lain, yaitu applet `MahasiswaUIApplet`. Method `setAddData()`

adalah method yang digunakan untuk menulis data pribadi tambahan yang telah disebutkan di atas ke dalam memori *JavaCard*. Penulisan data pribadi tambahan dilakukan pada saat personalisasi *JavaCard*. Jadi, applet `PKMApplet` hanya menyimpan data catatan medis, data kewarganegaraan dan data status pernikahan karena data pribadi lainnya sudah tersimpan dalam applet `MahasiswaUIApplet`.

Pembacaan data pribadi lainnya dilakukan dengan berinteraksi dengan applet `MahasiswaUIApplet` dan data pribadi yang dibaca ditampilkan pada modul terminal aplikasi *Smart Health*. Dengan demikian, tidak terjadi duplikasi data sehingga penggunaan memori *JavaCard* dapat dioptimalkan. Berikut adalah mekanisme penulisan dan pembacaan data pada applet `PKMApplet`.

Organisasi penulisan data ke dalam *JavaCard* dilakukan sebagai berikut. Setiap data medis dibagi menjadi 6 kategori yaitu institusi, ID dokter, tanggal pemeriksaan, anamnesa, diagnosa, dan terapi. Dengan demikian, penulisan data ke dalam *JavaCard* dilakukan per bagian, dari kategori pertama hingga kategori keenam. Proses ini diatur dengan menggunakan parameter-parameter di dalam APDU. *Command* APDU untuk menyimpan data adalah seperti di bawah ini.

Tabel 3.1 *Command* APDU penyimpanan data medis

CLA	INS	P1	P2	LC	Data
90	31	00	00	0X	00 0Y data institusi
90	31	00	00	0X	01 0Y data ID Dokter
90	31	00	00	0X	02 0Y data tanggal
90	31	00	00	0X	03 0Y data anamnesa
90	31	00	00	0X	04 0Y data diagnosa
90	31	00	80	0X	05 0Y data terapi

Dengan `CLA` adalah byte yang menunjukkan aplikasi *Smart Health*, `INS` menunjukkan byte instruksi penulisan data, `P2` adalah byte parameter dimana jika bernilai `0x80` maka penulisan satu data medis telah selesai. `LC` berisi panjang byte yang terdapat pada bagian APDU Data. Byte APDU Data yang pertama menunjukkan

Implementasi aplikasi..., Donny, FT UI, 2008

sequence penyimpanan data, byte kedua berisi panjang data yang akan disimpan pada *JavaCard*.

Response APDU, SW1 dan SW2, yang dikirim oleh *JavaCard* akan bernilai `90 00` jika proses penulisan data berhasil. Jika ada kondisi *Command* APDU yang tidak sesuai dengan format diatas maka Response APDU akan bernilai `54 00` (*Error State Corrupted*) dan `55 00` (*Error Set Data Failed*).

Untuk organisasi pembacaan data dari *JavaCard*, pembacaan data dilakukan dengan membaca satu persatu data medis yang telah tersimpan di dalam *JavaCard* atau dengan memilih data medis ke berapa yang ingin dibaca. Kemudian data medis tersebut akan dibaca perbagian, mulai dari kategori pertama hingga kategori keenam. Format *Command* APDU untuk proses pembacaan data medis adalah :

Tabel 3.2 *Command* APDU pembacaan data medis

CLA	INS	P1	P2	LC	Data
90	30	00	XY	02	00 0Y
90	30	00	XY	02	01 0Y
90	30	00	XY	02	02 0Y
90	30	00	XY	02	03 0Y
90	30	00	XY	02	04 0Y
90	30	00	XY	02	05 0Y

Dengan `CLA` adalah byte yang menunjukkan aplikasi *Smart Health*, `INS` menunjukkan byte instruksi pembacaan data, `P2` berisi urutan data medis yang ingin dibaca, `LC` berisi panjang byte yang terdapat pada bagian APDU Data. Byte APDU Data yang pertama menunjukkan *sequence* atau kategori dari urutan data medis yang akan dibaca, byte kedua berisi panjang data yang akan dibaca dari *JavaCard*.

Data medis yang dibaca dari *JavaCard* terdapat pada *Response* APDU yang dikirim dari *JavaCard*. Berikut adalah format *Response* APDU yang dikirim oleh *JavaCard*.

Implementasi aplikasi..., Donny, FT UI, 2008

Tabel 3.3 *Response* APDU pembacaan data medis

SW1	SW2	Data
90	00	XY XX 00 YY data institusi
90	00	XY XX 01 YY data ID Dokter
90	00	XY XX 02 YY data tanggal
90	00	XY XX 03 YY data anamnesa
90	00	XY XX 04 YY data diagnosa
90	00	XY XX 05 YY data terapi

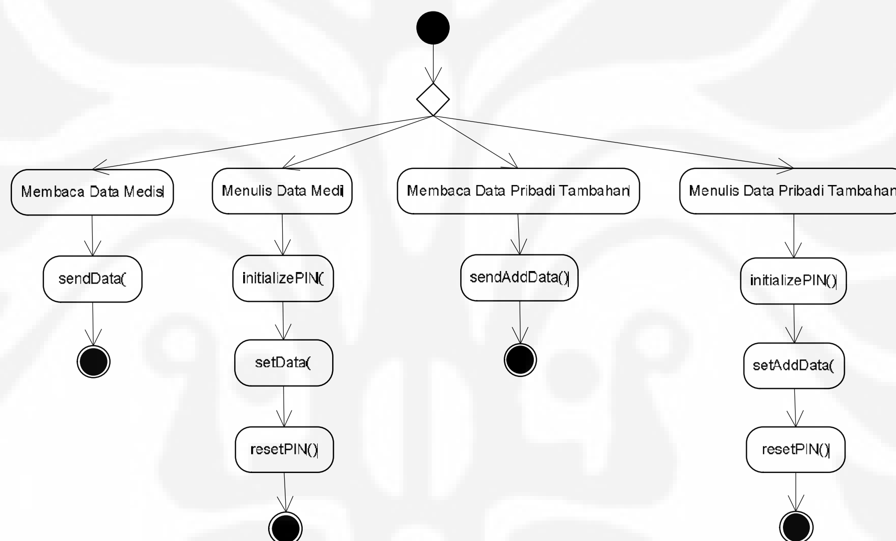
Byte SW1 dan SW2 yang bernilai `90 00` menunjukkan bahwa pembacaan data berhasil. Byte pertama pada bagian Data akan bernilai `0x00` jika data medis yang dibaca bukan merupakan data medis yang terakhir dan akan bernilai `0x80` jika data medis yang dibaca merupakan data medis yang terakhir. Byte kedua menunjukkan urutan data medis yang dibaca. Byte ketiga menunjukkan *sequence* atau urutan kategori dari data medis yang dibaca. Byte keempat menunjukkan panjang data medis yang dibaca dari dalam *JavaCard*. Byte-byte seterusnya merupakan data medis yang dibaca dari memori *JavaCard*.

Berikut adalah *Class Diagram* dari applet *JavaCard* `PKMApplet`, seperti telah dijelaskan di atas.

Implementasi aplikasi..., Donny, FT UI, 2008

Gambar 3.1 *Class Diagram* Applet *JavaCard*

Pada applet *JavaCard* `PKMApplet` ini, proses pertama adalah applet di-*install* pada *JavaCard* dan dipilih. Jika pengguna ingin membaca data medis yang tersimpan maka applet *JavaCard* akan membaca data medis dari dalam memori. Jika user ingin menulis data medis maka PIN harus diinisialisasi untuk memulai sesi penulisan data medis pada *JavaCard*. Setelah operasi penulisan data medis telah selesai maka applet akan me-*reset* PIN. Untuk pembacaan dan penulisan data pribadi tambahan, mekanisme yang digunakan sama seperti mekanisme pembacaan dan penulisan data medis. Berikut adalah *Activity Diagram* dari applet *JavaCard* `PKMApplet`.



Gambar 3.2 *Activity Diagram* Applet *JavaCard*

3.2 *APPLET CONNECTOR*

Setelah applet pada *JavaCard* dibuat, tahap berikutnya adalah pembuatan applet *connector*, yang berfungsi sebagai penghubung applet *JavaCard* dengan modul terminal. Pembuatan applet *connector* ini menggunakan bantuan *UI Card Bundle Library*, s*martcard-connector* dan s*martcard-data* untuk pengaturan *error* atau *exception* yang dapat timbul dalam proses input dan output antara applet *JavaCard* dengan modul terminal.

Applet *connector* terdiri dari satu kelas yang dinamakan `PKMAppletConnector` dan terdiri dari method `openWriteSession()`,

closeWriteSession(), getMedicalRecord(), setMedicalRecord(), getRecordLength(), getAddData(), setAddData(), closeApplet(), dan getAID(). Method–method pada applet *connector* inilah yang akan menjadi konektor lalu lintas *input* dan *output* antara terminal dan *JavaCard*.

Method openWriteSession() adalah method untuk memulai sesi penulisan data medis pada *JavaCard*. Method ini akan memeriksa validasi PIN yang dimasukkan oleh pengguna. Jika PIN sesuai maka *flag* otentifikasi akan bernilai *true* dan proses penulisan data medis dapat dilakukan. Method closeWriteSession() adalah method untuk mengakhiri sesi penulisan data medis. Pada method ini PIN akan di-*reset* kembali dan *flag* otentifikasi akan bernilai *false*.

Method getMedicalRecord() merupakan method yang mengatur pembacaan data dari *JavaCard*. Method ini akan mengirimkan APDU dengan instruksi untuk menyalin data dari memori *JavaCard*. Data yang diambil disalin dari *Response* APDU ke array chunk[], dipilah-pilah dari *delimiter* dan kemudian dimasukkan ke array *ByteArrayOutputStream* bernama baos[]. Array inilah yang kemudian akan ditampilkan pada modul terminal sehingga data pada *JavaCard* dapat dibaca. Method setMedicalRecord() digunakan untuk menulis data ke memori *JavaCard* dari modul terminal. Data dari modul terminal yang berupa array data[] dipindahkan ke array dataBuffer[]. Array dataBuffer[] inilah yang akan disalin ke *Command* APDU untuk kemudian disimpan dalam *JavaCard*.

Method getRecordLength() adalah method penghubung applet *JavaCard* dan modul terminal yang digunakan untuk mengetahui berapa banyak data medis yang telah tersimpan dalam *JavaCard*. Method getAddData() dan setAddData() adalah method penghubung pembacaan dan penulisan data pribadi tambahan pada *JavaCard*. Mekanisme pembacaan dan penulisan data pribadi tambahan ini menggunakan algoritma yang sama seperti method getMedicalRecord() dan setMedicalRecord().

Method closeApplet() adalah method untuk menutup atau me-nonaktifkan applet PKMConnector dan method yang digunakan untuk memeriksa AID yang

Implementasi aplikasi..., Donny, FT UI, 2008

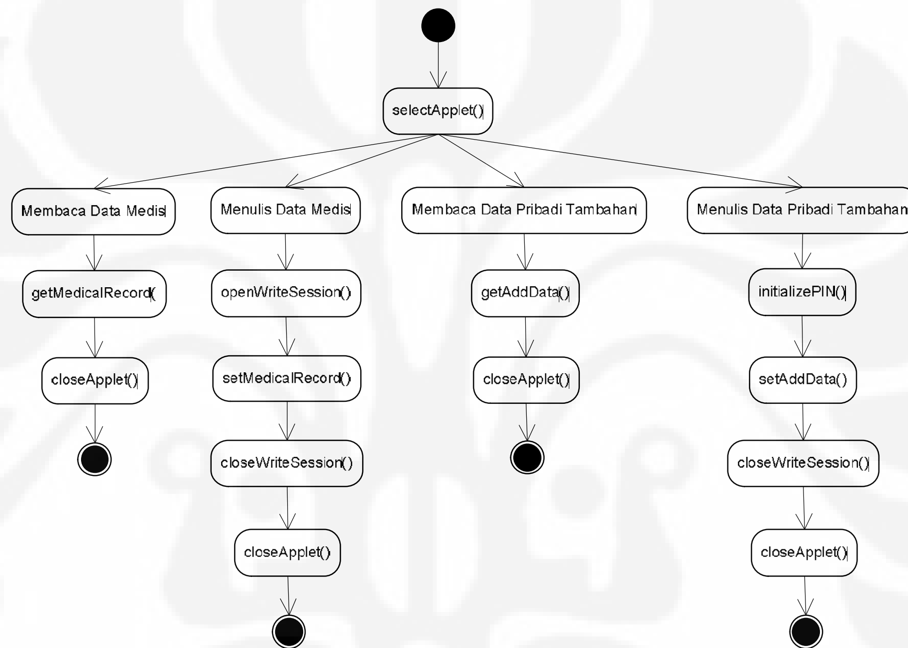
dimasukkan oleh user untuk memilih applet `PKMApplet`. Jika AID sudah sesuai maka applet `PKMApplet` akan diaktifkan.

Berikut adalah *Class Diagram* dari applet `PKMConnector`.

Gambar 3.3 *Class Diagram* Applet *Connector*

Pada applet *connector* `PKMAppletConnector`, proses dimulai dengan memilih applet `PKMApplet` pada *JavaCard* dengan terlebih dahulu memeriksa AID applet yang di-*input* oleh pengguna. Jika pengguna ingin membaca data medis yang tersimpan maka applet *connector* akan mengirimkan APDU pembacaan data medis dan data medis akan diambil dari *JavaCard*. Jika user ingin menulis data medis maka PIN harus divalidasi untuk memulai sesi penulisan data medis pada *JavaCard*.

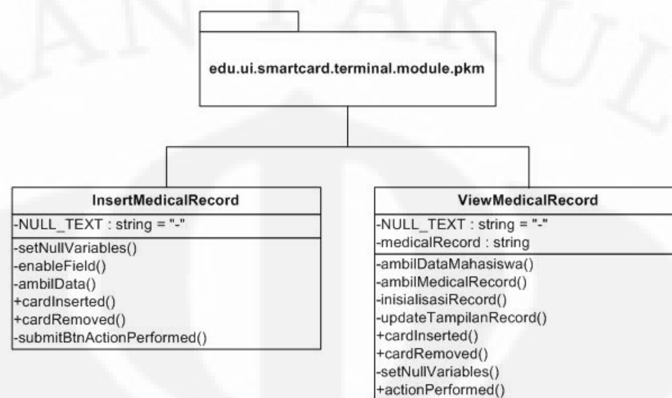
Setelah operasi penulisan data medis telah selesai maka sesi penulisan data medis akan ditutup dengan me-*reset* PIN. Kemudian applet `PKMApplet` pada *JavaCard* akan di-nonaktifkan. Proses pembacaan dan penulisan data pribadi tambahan menggunakan mekanisme yang sama seperti mekanisme pembacaan dan penulisan data medis. Berikut adalah *Activity Diagram* dari applet `PKMConnector`.



Gambar 3.4 *Activity Diagram* Applet *Connector*

3.3 *PEMBUATAN MODUL TERMINAL*

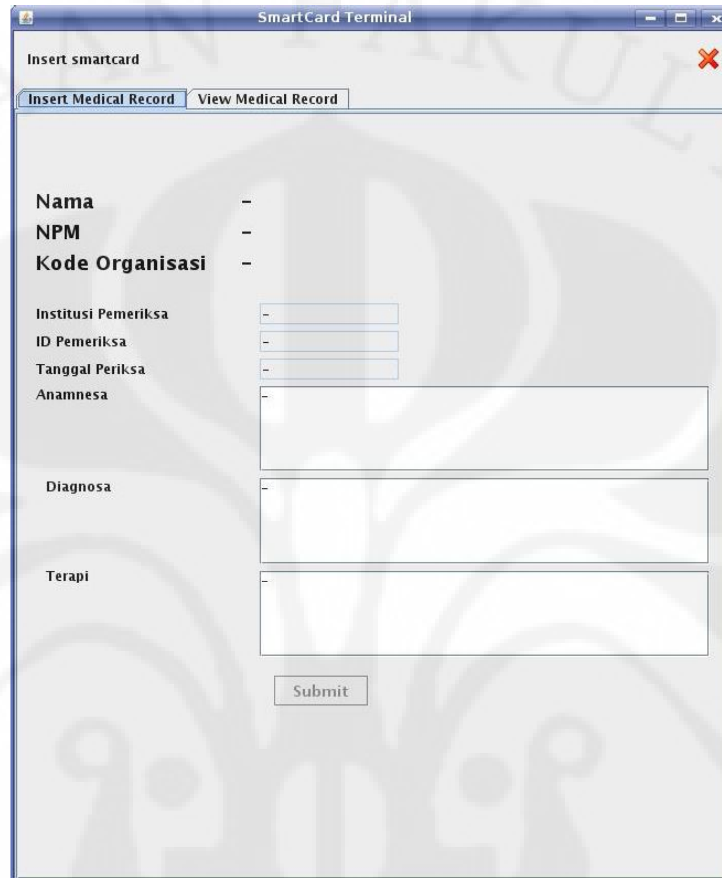
Modul terminal adalah sebuah aplikasi *end user* atau aplikasi yang dipergunakan oleh pengguna untuk menjalankan aplikasi *Smart Health* secara keseluruhan. Modul terminal aplikasi *Smart Health* adalah aplikasi *Graphical User Interface* (GUI) yang terdiri dari dua bagian, yaitu modul `InsertMedicalRecord` dan modul `ViewMedicalRecord`. Dengan kata lain, modul terminal *Smart Health* merupakan satu *package* Java dengan dua kelas di dalamnya, yaitu kelas `InsertMedicalRecord` dan `ViewMedicalRecord`. Berikut adalah *Class Diagram* dari modul terminal *Smart Health*.



Gambar 3.5 *Class Diagram* Modul Terminal *Smart Health*

3.3.1 Modul InsertMedicalRecord

Modul `InsertMedicalRecord` adalah modul yang digunakan untuk menulis data medis dan disimpan dalam *JavaCard*. Cara kerja dari modul ini adalah jika *JavaCard* dimasukkan ke dalam *reader* maka pertama–tama modul ini akan menampilkan data pribadi pemilik *JavaCard* seperti nama, NPM, dan kode organisasi. Setelah data medis ditulis pada *text area* dan di-*submit*, maka modul ini akan mengirimkan data tersebut ke applet *connector* yang kemudian dilanjutkan ke applet *JavaCard* sehingga data dapat tersimpan dalam *JavaCard*. Berikut adalah tampilan dari modul `InsertMedicalRecord`.



Gambar 3.6 Tampilan modul `InsertMedicalRecord`

Modul `InsertMedicalRecord` dibangun dengan satu kelas yang terdiri dari beberapa method, yaitu method `setNullVariables()`, `enableField()`, `ambilData()`, `cardInserted()`, `cardRemoved()`, dan `submitBtnActionPerformed()`.

Method `setNullVariables()` adalah method untuk menonaktifkan seluruh variabel pada modul, baik *JLabel*, *JButton*, *JTextField*, maupun *JTextArea*. Method `enableField()` adalah method untuk mengaktifkan *field JTextField*, *JTextArea*, dan *JButton* pada modul. Method `ambilData()`merupakan method yang digunakan untuk mengambil data pribadi pemilik kartu yang berada pada applet `MahasiswaUIApplet`. Applet `MahasiswaUIApplet` diaktifkan, data pribadi yang dibutuhkan diambil, dan kemudian applet dinonaktifkan kembali. Method

Implementasi aplikasi..., Donny, FT UI, 2008

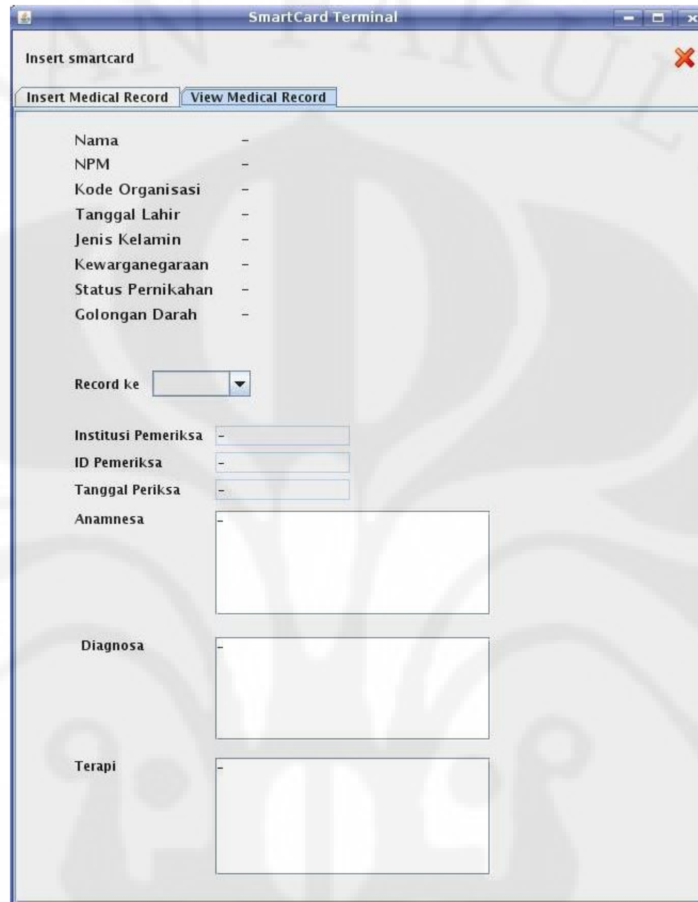
`enableField()`dipanggil dan data pribadi ditampilkan pada modul terminal. Dengan demikian, aplikasi *Smart Health* juga berinteraksi dengan applet lain pada *JavaCard* dengan tujuan untuk optimalisasi memori pada kartu.

Method `cardInserted()` menginstruksikan modul untuk menjalankan method `ambilData()` setelah *JavaCard* dimasukkan pada *reader*. Method `cardRemoved()` menginstruksikan modul untuk menjalankan method `setNullVariables()` setelah *JavaCard* dicabut dari *reader*. Method `submitBtnActionPerformed()` merupakan method yang menghubungkan modul dengan applet *connector*. Setelah *submit button* ditekan, maka pertama – tama method ini akan mengaktifkan applet *connector*, memvalidasi PIN, dan kemudian menulis data medis ke dalam *JavaCard*. Kemudian applet *connector* dinonaktifkan, dan memanggil method `setNullVariables()`.

3.3.2 Modul ViewMedicalRecord

Modul `ViewMedicalRecord` adalah modul untuk menampilkan data medis yang tersimpan dalam *JavaCard*. Cara kerja modul ini adalah sebagai berikut. Setelah *JavaCard* dimasukkan ke *reader*, maka modul akan menampilkan data pribadi pemilik *JavaCard* yaitu nama, NPM, kode organisasi, tanggal lahir, jenis kelamin, kewarganegaraan, status pernikahan, dan golongan darah. Kemudian pengguna dapat memilih data medis keberapa yang ingin dibaca pada *JComboBox* dan modul akan menampilkan data medis tersebut pada *JTextField* dan *JTextArea*. Dibawah ini adalah tampilan dari modul `ViewMedicalRecord`.

Implementasi aplikasi..., Donny, FT UI, 2008



Gambar 3.7 Tampilan modul `ViewMedicalRecord`

Modul `ViewMedicalRecord` dibangun dengan satu kelas yang terdiri dari beberapa method, yaitu method `ambilDataMahasiswa()`, `ambilMedicalRecord()`, `inisialisasiRecord()`, `updateTampilanRecord()`, `cardInserted()`, `cardRemoved()`, `setNullVariables()`, dan `actionPerformed()`.

Method `ambilDataMahasiswa()` merupakan method yang digunakan untuk mengambil data pribadi pemilik kartu yang berada pada applet `MahasiswaUIApplet`. Applet `MahasiswaUIApplet` diaktifkan, data pribadi yang dibutuhkan diambil, dan kemudian applet dinonaktifkan kembali. Method `ambilMedicalRecord()` adalah method untuk membaca seluruh data medis yang tersimpan pada *JavaCard*. Method ini mengaktifkan applet *connector* dan

menjalankan method `getMedicalRecord()` pada applet *connector* sehingga data dari *JavaCard* dapat diambil.

Method `inisialisasiRecord()` digunakan untuk mengatur *JComboBox* yang menunjukkan indeks dari data medis yang tersimpan dalam *JavaCard*. Method `updateTampilanRecord()` adalah method untuk menampilkan data medis pada *field* modul sesuai dengan indeks yang dipilih oleh pengguna.

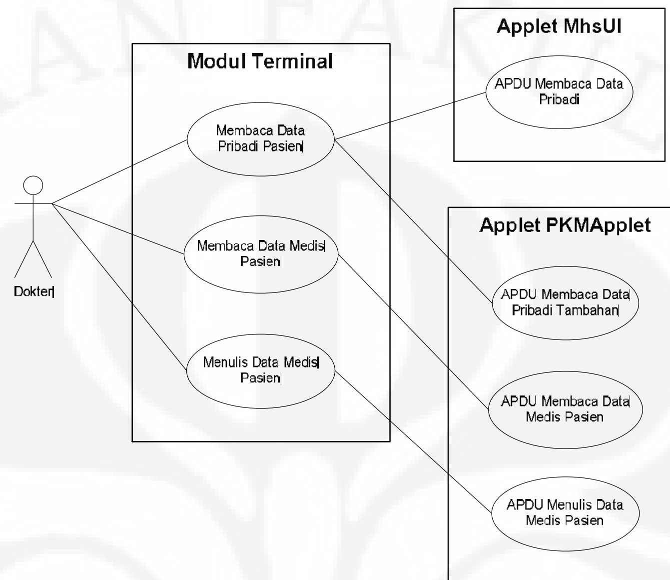
Method `cardInserted()` menginstruksikan modul untuk menjalankan method `ambilDataMahasiswa()`, `ambilMedicalRecord()`, `inisialisasiRecord()`, dan `updateTampilanRecord()` setelah *JavaCard* dimasukkan pada *reader*. Method `setNullVariables()` adalah method untuk menonaktifkan seluruh variabel pada modul, baik *JLabel*, *JComboBox*, *JTextField*, maupun *JTextArea*. Method `cardRemoved()` menginstruksikan modul untuk menjalankan method `setNullVariables()` setelah *JavaCard* dicabut dari *reader*.

Method `actionPerformed()` adalah method yang aktif setelah pengguna memilih indeks data medis yang ingin dibaca dari *JavaCard* melalui *JComboBox*. Method ini kemudian akan mengaktifkan method `updateTampilanRecord()` dan data medis yang dipilih akan ditampilkan pada *field* modul.

3.4 *INTEGRASI SISTEM SMART HEALTH*

Sistem *Smart Health* memiliki tiga komponen utama, yaitu applet *JavaCard*, applet *connector*, dan modul terminal. Ketiga komponen ini saling berinteraksi satu sama lain dan merupakan *requirement* utama dalam pembuatan sistem *Smart Health*. Tanpa salah satu komponen di atas maka sistem tidak dapat berjalan. Integrasi sistem *Smart Health* dilakukan dengan menggabungkan tiga komponen tersebut dalam satu *Java Project*, yang didukung oleh *UI Card Bundle Library*, untuk pengaturan *error* atau *exception* dan pembentukan koneksi antara applet *JavaCard* dengan modul terminal. Berikut adalah *Use Case Diagram* dari sistem *Smart Health*.

Implementasi aplikasi..., Donny, FT UI, 2008



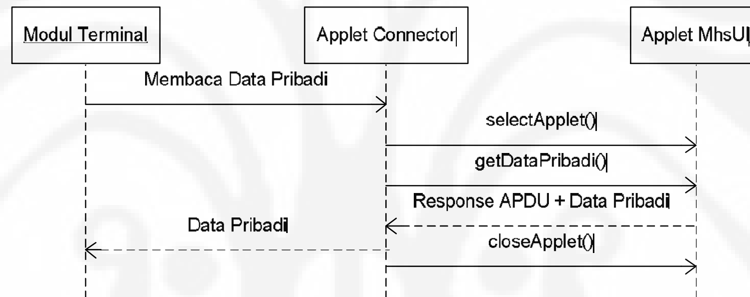
Gambar 3.8 *Use Case Diagram* sistem *Smart Health*

Interaksi antara aplikasi *Smart Health* dan pengguna adalah pengguna dapat membaca data pribadi, membaca data medis, dan menulis data medis dari pemilik *JavaCard*. Dari sisi pengguna, aplikasi *Smart Health* adalah aplikasi yang mudah digunakan karena hanya memerlukan sedikit operasi dari pengguna. Pengguna hanya perlu untuk memasukkan *JavaCard* ke dalam *card reader* dan kemudian pengguna dapat membaca data pribadi, membaca data medis, dan menulis data medis melalui aplikasi *Smart Health*. Pengguna melakukan proses pembacaan dan penulisan melalui modul terminal, dikoneksikan dengan applet *connector*, dan kemudian akan diteruskan kepada applet *JavaCard*, seperti terlihat pada Gambar 3.8.

Aplikasi *Smart Health* terdiri dari dua bagian, yaitu pembacaan data medis dari *JavaCard* dan penulisan data medis pada *JavaCard*. Untuk menjalankan kedua operasi ini harus melalui tiga tahapan kerja, yaitu pembacaan data pribadi dari applet `MahasiswaUIApplet`, pembacaan data pribadi tambahan dan data medis dari applet `PKMApplet`, serta penulisan data medis pada applet `PKMApplet`. Berikut adalah urutan kerja dari aplikasi Smart Health dalam menjalankan operasi pembacaan data medis dari *JavaCard* dan penulisan data medis pada *JavaCard*.

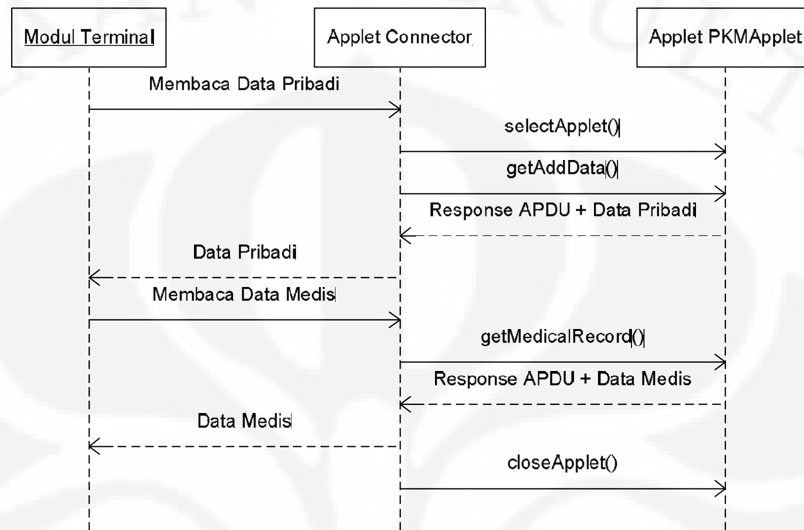
Implementasi aplikasi..., Donny, FT UI, 2008

Proses pembacaan data medis dari aplikasi *Smart Health* memiliki urutan kerja atau *sequence* sebagai berikut. Setelah *JavaCard* dimasukkan ke *card reader*, pertama-tama modul terminal akan mengirimkan perintah kepada applet *connector* untuk mengambil data pribadi yang tersimpan pada *JavaCard*, yaitu pada applet `PKMApplet` dan `MahasiswaUIApplet`. Data pribadi berupa nama, NPM, kode organisasi, tanggal lahir, jenis kelamin, dan golongan darah diambil dari applet `MahasiswaUIApplet`. Applet `MahasiswaUIApplet` diaktifkan, memanggil method untuk membaca data pribadi, ditampilkan pada modul terminal, dan applet di-nonaktifkan.



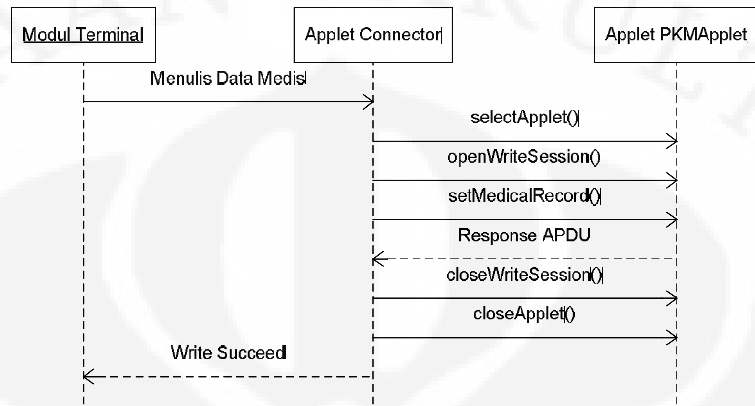
Gambar 3.9 *Sequence Diagram* pengambilan data pribadi

Proses selanjutnya adalah pengambilan data kewarganegaraan dan status pernikahan dari applet `PKMApplet`. Applet `PKMApplet` diaktifkan, memanggil method untuk membaca data pribadi tambahan, dan ditampilkan pada modul terminal. Applet `PKMApplet` tidak di-nonaktifkan karena masih akan mengambil data medis dari *JavaCard*. Setelah data medis diambil dan ditampilkan pada modul terminal, baru kemudian applet `PKMApplet` di-nonaktifkan.



Gambar 3.10 *Sequence Diagram* pengambilan data medis

Proses penulisan data medis dari aplikasi *Smart Health* memiliki urutan kerja atau *sequence* sebagai berikut. Setelah *JavaCard* dimasukkan ke *card reader*, modul terminal akan mengirimkan perintah kepada applet *connector* untuk mengambil data pribadi berupa nama, NPM, dan kode organisasi, yang tersimpan pada *JavaCard*, yaitu pada applet `MahasiswaUIApplet`. Seperti telah ditunjukkan pada Gambar 3.9, applet `MahasiswaUIApplet` diaktifkan, memanggil method untuk membaca data pribadi, ditampilkan pada modul terminal, dan applet di-nonaktifkan. Kemudian setelah data medis ditulis dan di-*submit*, modul terminal akan memberi perintah applet *connector* untuk mengaktifkan applet `PKMApplet`, memulai sesi penulisan data medis dan memanggil method untuk menulis data medis. Setelah applet menerima respon dari *JavaCard* bahwa data medis berhasil disimpan, sesi penulisan dan applet `PKMApplet` akan ditutup. Proses terakhir, modul terminal akan memberitahu pengguna melalui *dialog box*.



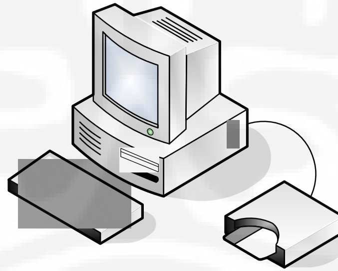
Gambar 3.11 *Sequence Diagram* penulisan data medis

Implementasi aplikasi..., Donny, FT UI, 2008

BAB 4
ANALISA KINERJA APLIKASI *SMART HEALTH*

4.1 *INSTALASI APLIKASI SMART HEALTH*

Instalasi aplikasi *Smart Health* memiliki tahapan sebagai berikut. Tahap pertama adalah mengintegrasikan komponen-komponen sistem dalam satu *Java Project*. Kemudian *Java Project* tersebut di-build menggunakan *Java Virtual Machine* (JVM) membentuk *file Java Archive* (JAR), yaitu *file* yang dapat dieksekusi oleh sistem operasi untuk menjalankan aplikasi *Smart Health*. Perangkat keras yang dibutuhkan untuk mengoperasikan aplikasi *Smart Health* adalah *Personal Computer* (PC), *JavaCard*, dan *JavaCard Reader*. Kemudian dilanjutkan dengan melakukan instalasi *Java Development Kit* (JDK), *JavaCard Reader driver*, dan aplikasi *Smart Health* pada sistem operasi PC. Setelah proses di atas selesai, maka aplikasi *Smart Health* siap dipergunakan.



Gambar 4.1 Perangkat keras aplikasi Smart Health

4.2 *ANALISA PROGRAM APLIKASI SMART HEALTH*

Setelah melakukan pemrograman aplikasi *Smart Health*, didapatkan beberapa pelajaran berharga dalam membuat suatu aplikasi *Smart Card* yang berbasis *JavaCard*. Kunci dari pembuatan aplikasi *JavaCard* terletak pada pemrograman applet *JavaCard* dan bagaimana pengaturan APDU untuk memproses dan

menjalankan operasi-operasi dari aplikasi *JavaCard*. Applet *JavaCard* merupakan bagian integral dalam pembuatan aplikasi karena pada tujuan akhirnya adalah pembacaan dan penyimpanan data pada memori *JavaCard*. Oleh karena itu, dibutuhkan suatu ketentuan dalam penyusunan APDU untuk mengatur lalu lintas pembacaan dan penyimpanan data sehingga proses pembacaan dan penulisan data berjalan sesuai dengan yang diinginkan.

Pada pemrograman applet *JavaCard*, peran dari APDU adalah untuk memindahkan data yang dimasukkan dari modul terminal ke memori *JavaCard* dan memindahkan data dari memori *JavaCard* untuk ditampilkan pada modul terminal. Oleh sebab itu, pembagian memori untuk setiap data juga menjadi bagian yang penting dalam pembuatan aplikasi. Program yang dibuat harus mampu mengetahui dimana letak setiap data yang disimpan dalam memori *JavaCard* sehingga pada proses pengambilan data tidak terjadi kekeliruan seperti data yang diambil tidak lengkap atau tercampur dengan yang lain. Demikian juga pada proses penulisan data, data yang akan disimpan dalam memori *JavaCard* tidak boleh menumpuk data yang sudah ada kecuali untuk keadaan tertentu seperti memori untuk penyimpanan data telah terisi seluruhnya. Dengan kata lain, program yang dibuat harus mampu untuk mengetahui posisi setiap data dalam memori *JavaCard* dengan tepat.

Aplikasi *Smart Health* memiliki spesifikasi sebagai berikut. Aplikasi memiliki fungsi utama untuk menulis dan membaca data medis serta membaca data pribadi dari pemilik *JavaCard*. Fungsi pendukung dari aplikasi *Smart Health* adalah menulis data pribadi tambahan, yaitu data kewarganegaraan dan status pernikahan, yang belum tersimpan dalam *JavaCard*. Aplikasi *Smart Health* mampu menyimpan dua puluh data medis dan dua data pribadi tambahan. Setiap data medis memiliki enam bagian, yaitu institusi, ID Dokter, tanggal, anamnesa, diagnosa, dan terapi. Memori yang dibutuhkan sebagai tempat penyimpanan adalah 340 byte untuk setiap data medis dan 16 byte untuk data pribadi tambahan. Rincian penggunaan memori untuk satu data medis dan dua data pribadi tambahan dapat dilihat pada Tabel 4.1. Keseluruhan memori yang dibutuhkan untuk penyimpanan data pada aplikasi *Smart Health* adalah 6816 byte. *JavaCard* memiliki 64 Kb memori untuk penyimpanan data

Implementasi aplikasi..., Donny, FT UI, 2008

sehingga dapat dilihat bahwa aplikasi *Smart Health* tidak memerlukan memori yang besar, hanya memerlukan 11% dari memori *JavaCard*.

Tabel 4.1 Rincian memori satu data medis dan data pribadi tambahan

Kategori Data	Alokasi Memori *JavaCard*
Institusi	20 byte
ID Dokter	12 byte
Tanggal	8 byte
Anamnesa	100 byte
Diagnosa	100 byte
Terapi	100 byte
Kewarganegaraan	15 byte
Status pernikahan	1 byte

Aplikasi *Smart Health* memiliki tampilan awal berupa *Graphical User Interface* (GUI) yang merupakan modul terminal dari aplikasi *Smart Health* dengan dua menu yaitu menu untuk menulis data medis dan menu untuk membaca data medis. Gambar 4.2 adalah tampilan menu penulisan data medis dan Gambar 4.3 adalah tampilan menu penulisan data medis.

Implementasi aplikasi..., Donny, FT UI, 2008

Gambar 4.2 Tampilan awal menu penulisan data medis

Pada saat aplikasi *Smart Health* dijalankan, kedua menu aplikasi, baik menu penulisan maupun pembacaan data medis, akan berada dalam keadaan tidak aktif. Pengguna tidak dapat mengakses seluruh *field* yang terdapat pada modul terminal. Menu aplikasi akan aktif jika modul terminal mendeteksi adanya *JavaCard* yang dimasukkan ke dalam *card reader*. Setelah menu aplikasi aktif maka pengguna dapat menjalankan operasi pembacaan dan penulisan data medis.

Implementasi aplikasi..., Donny, FT UI, 2008

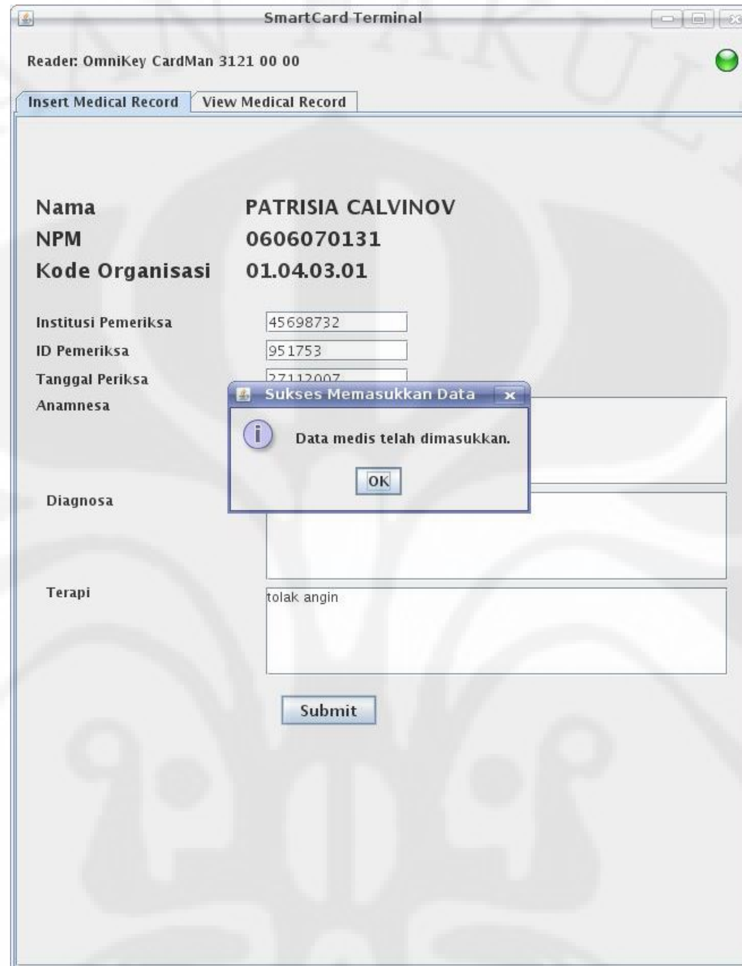
Gambar 4.3 Tampilan awal menu pembacaan data medis

Jika pengguna ingin menulis data medis dan setelah *JavaCard* dimasukkan ke *card reader*, pertama aplikasi *Smart Health* akan membaca data pribadi dari pemilik *JavaCard*. Data pribadi ini diambil dari applet `MahasiswaUIApplet` dan kemudian ditampilkan pada modul terminal.

Gambar 4.4 Tampilan aktif menu pembacaan data medis

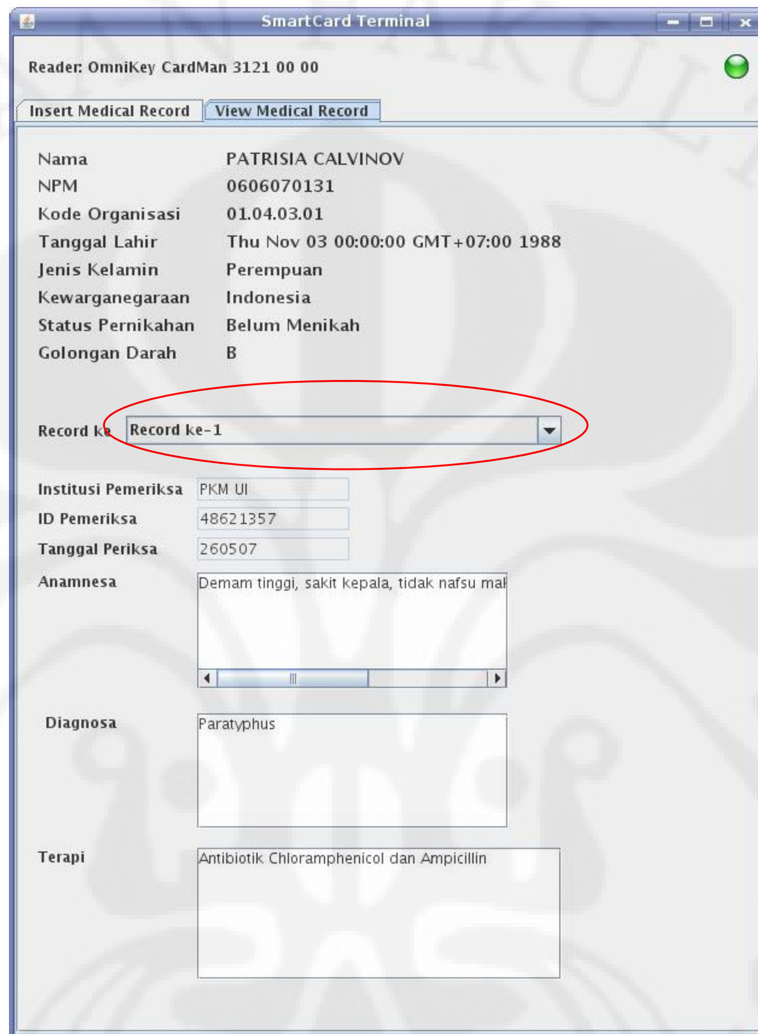
Selanjutnya, pengguna dapat menulis data medis pada *text area*. Setelah tombol "*Submit*" ditekan, maka aplikasi akan mengakses applet `PKMApplet` dan data medis tersebut akan disimpan dalam *JavaCard*. Setelah proses penyimpanan selesai, maka modul terminal akan menampilkan *dialog box* yang menyatakan bahwa data medis telah berhasil disimpan.

PERPUSTAKAAN FAKULTAS TEKNIK UNIVERSITAS INDONESIA



Gambar 4.5 Tampilan saat penulisan data medis

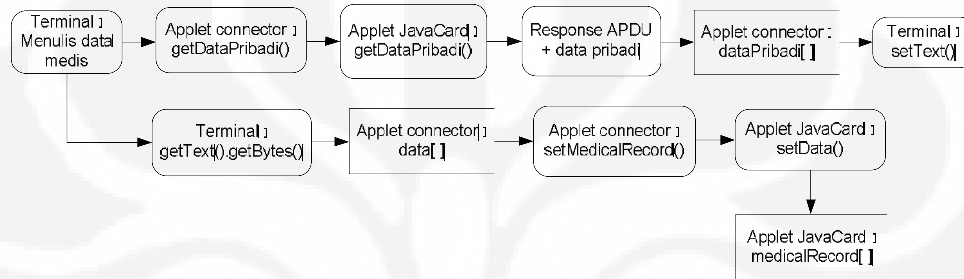
Jika pengguna ingin membaca data medis dalam *JavaCard* dan setelah *JavaCard* dimasukkan ke *card reader*, pertama aplikasi *Smart Health* akan membaca data pribadi dari pemilik *JavaCard* yang tersimpan pada applet `MahasiswaUIApplet` dan `PKMApplet`. Berikutnya aplikasi akan mengambil data medis yang tersimpan pada applet `PKMApplet` satu persatu, dari data pertama hingga data terakhir. Data pribadi dan data medis yang diambil dari *JavaCard* akan ditampilkan pada *field* modul terminal dan pengguna dapat memilih data medis yang ingin dibaca pada *combo box*, yang dilingkari pada Gambar 4.6.



Gambar 4.6 Tampilan saat pembacaan data medis

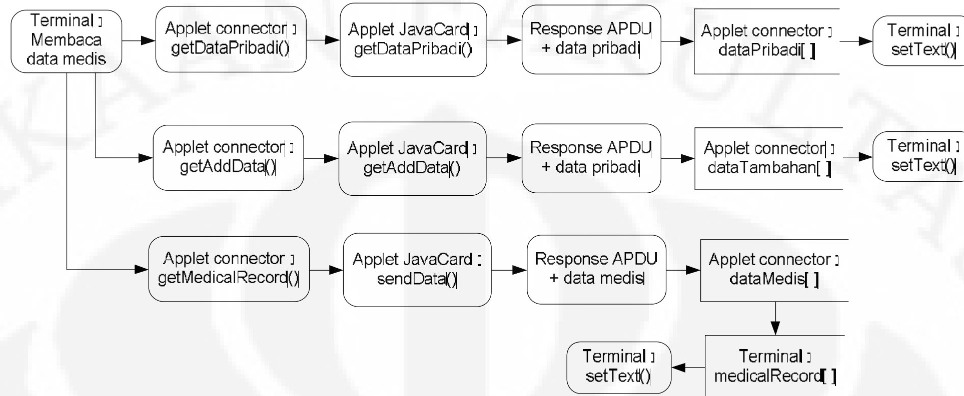
Aliran data pada proses penulisan dan pembacaan data medis adalah sebagai berikut. Proses penulisan data medis memiliki dua buah aliran data, yaitu aliran data pribadi dari applet *JavaCard* ke modul terminal dan aliran data medis dari modul terminal ke applet *JavaCard*. Pada aliran data pribadi, modul terminal menginstruksikan applet *connector* untuk mengakses applet *JavaCard*, kemudian applet *JavaCard* mengambil data dari memori *JavaCard* melalui APDU. Applet *connector* mengambil data ini dari APDU dan disimpan dalam array `dataPribadi[]`. Array inilah yang diakses oleh modul terminal dan kemudian

ditampilkan. Pada aliran data medis, data yang dimasukkan pada modul terminal disimpan oleh applet *connector* dalam array `data[]`. Kemudian applet *connector* menginstruksikan applet *JavaCard* untuk menulis data medis ke dalam *JavaCard*. Data pada array `data[]` disalin dan dimasukkan ke APDU. Melalui APDU, data medis disimpan dalam memori *JavaCard*, pada array `medicalRecord[]`.



Gambar 4.7 Data Flow Diagram penulisan data medis

Proses pembacaan data medis memiliki tiga buah aliran data, yaitu aliran data pribadi, data pribadi tambahan, dan aliran data medis dari applet *JavaCard* ke modul terminal. Pada aliran data pribadi dan data pribadi tambahan, modul terminal menginstruksikan applet *connector* untuk mengakses applet *JavaCard*, kemudian applet *JavaCard* mengambil data dari memori *JavaCard* melalui APDU. Applet *connector* mengambil data ini dari APDU dan disimpan dalam array `dataPribadi[]` dan `dataTambahan[]`. Kedua array inilah yang diakses oleh modul terminal dan kemudian ditampilkan. Pada aliran data medis, modul terminal memberikan instruksi pada applet *connector* untuk mengambil data medis yang tersimpan pada applet *JavaCard*. Applet *connector* mengakses applet *JavaCard*, kemudian applet *JavaCard* mengambil data medis dari memori *JavaCard* melalui APDU. Applet *connector* mengambil data ini dari APDU dan disimpan dalam array `dataMedis[]`. Array ini diakses oleh modul terminal, dipilah-pilah sesuai dengan kategori data medis yang dimiliki, kemudian disimpan oleh modul terminal dalam array `medicalRecord[]` dan ditampilkan pada *field* modul terminal.



Gambar 4.8 Data Flow Diagram pembacaan data medis

Mekanisme penyimpanan dan pembacaan data medis dilakukan oleh applet *JavaCard* menggunakan APDU dengan parameter-parameter yang telah dijelaskan pada bagian perancangan aplikasi. Proses penyimpanan dan pembacaan ini dilakukan secara berurutan, mulai dari kategori pertama (institusi) hingga kategori keenam (terapi). Mekanisme demikian dipilih dengan alasan karena dalam satu kali proses, APDU hanya mampu menyimpan atau membaca data sebanyak 256 byte. Ukuran satu data medis adalah 340 byte sehingga melebihi kapasitas APDU dan dipilih mekanisme penyimpanan dan pembacaan yang bertahap. Dengan mekanisme seperti ini, proses penyimpanan dan pembacaan data menjadi lebih terorganisir dan lebih terkontrol. Aplikasi *Smart Health* dapat menyimpan sebanyak 20 data medis. Jika memori *JavaCard* telah terpakai untuk menyimpan 20 data medis dan pengguna ingin memasukkan data medis yang berikutnya maka program akan menumpuk data yang ingin disimpan pada data pertama. Dengan kata lain, data medis yang terlama akan diganti dengan data medis yang terbaru.

Data medis disimpan dalam memori *JavaCard* dalam bentuk byte dan sesuai dengan alokasi memori yang telah ditentukan untuk setiap kategori. Hal ini berarti bahwa setiap kategori data medis memiliki panjang maksimal dan disimpan pada lokasi yang telah ditentukan dalam memori *JavaCard*. Jika data medis yang dimasukkan dalam memori masih memiliki tempat kosong atau data kurang dari panjang maksimal, maka tempat-tempat yang kosong tersebut akan berisi byte nol.

Oleh karena itu, pada proses pembacaan data dari memori *JavaCard* perlu dilakukan pemisahan antara byte nol dengan byte data. Proses ini dilakukan pada applet *connector*. Hal inilah yang meyebabkan pada applet *connector* data perlu disimpan kembali dalam suatu array dengan pembatas antara kategori data yang satu dengan lainnya. Pembatas data ini disebut dengan *delimiter* dan berisi string ”++++”. Pada modul terminal, *delimiter* akan dipisahkan dari data dan kemudian data ditampilkan pada *field* modul terminal.

Aplikasi *Smart Health* memiliki beberapa pesan kesalahan dan akan timbul jika terdapat kondisi yang tidak terpenuhi dalam menjalankan operasi aplikasi. Setiap pesan kesalahan diwakili oleh sebuah byte dan akan ditampilkan pada *Response* APDU. Tabel 4.2 menunjukkan pesan-pesan kesalahan yang terdapat pada aplikasi *Smart Health*.

Tabel 4.2 Pesan kesalahan aplikasi *Smart Health*

Pesan Kesalahan	Status Byte
`ERR_VERIFICATION_FAILED`	`0x5100`
`ERR_VERIFICATION_REQUIRED`	`0x5200`
`ERR_DATA_CORRUPTED`	`0x5300`
`ERR_STATE_CORRUPTED`	`0x5400`
`ERR_SET_DATA_FAILED`	`0x5500`
`ERR_GET_ADD_DATA_FAILED`	`0x5600`
`ERR_SET_ADD_DATA_FAILED`	`0x5700`
`ERR_EXCESSIVE_LENGTH`	`0x5800`

Pesan `ERR_VERIFICATION_FAILED` timbul jika terjadi kesalahan memasukkan PIN untuk memulai sesi penulisan data medis. Pesan `ERR_VERIFICATION_REQUIRED` timbul saat akan menulis data medis tetapi PIN tidak dimasukkan sehingga tidak dapat memulai sesi penulisan data medis. Jika terjadi kesalahan dalam mekanisme penyimpanan atau pembacaan data medis maupun data pribadi tambahan, maka muncul pesan `ERR_STATE_CORRUPTED` dan menyebabkan data gagal ditulis atau dibaca. Pesan berikutnya yang muncul jika terjadi kegagalan dalam proses pembacaan atau penyimpanan adalah

Implementasi aplikasi..., Donny, FT UI, 2008

ERR_DATA_CORRUPTED, ERR_SET_DATA_FAILED, ERR_GET_ADD_DATA_FAILED, dan ERR_SET_ADD_DATA_FAILED. Jika proses penulisan dan pembacaan data mengalami kegagalan maka proses harus diulang kembali. Pesan ERR_EXCESSIVE_LENGTH muncul jika data medis yang ingin disimpan atau dibaca dari memori *JavaCard* melebihi panjang maksimal yang ditetapkan.

4.3 *HAL-HAL YANG MEMBATASI APLIKASI*

4.3.1 Alokasi Memori JavaCard

Aplikasi *Smart Health* adalah aplikasi yang dalam pembuatannya menggunakan sistem *Fix Length* dimana setiap data yang akan disimpan dalam memori *JavaCard* telah ditentukan panjang maksimalnya dan tidak dapat menyimpan lebih dari yang telah ditentukan. Alokasi memori untuk penyimpanan data pada *JavaCard*, yang ditunjukkan pada Tabel 4.1, diyakini sudah memiliki kapasitas yang cukup untuk mewakili sebuah data medis. Akan tetapi, penggunaan sistem *Fix Length* ini memiliki beberapa kelemahan.

Pertama, jika data yang dimasukkan dalam *JavaCard* masih kurang dari panjang maksimalnya. Hal ini menyebabkan tempat tersisa yang terdapat pada memori *JavaCard* menjadi tidak terisi. Dengan demikian, telah terjadi pemborosan memori, yang seharusnya dapat dipergunakan untuk menyimpan data lain atau untuk aplikasi lain. Kemungkinan pemborosan memori terjadi pada bagian anamnesa, diagnosa, dan terapi karena pada bagian ini tidak dapat diperkirakan panjangnya data medis yang akan ditulis. Kedua, jika data yang dimasukkan pada *JavaCard* melebihi kapasitas maksimal yang dapat ditampung. Hal ini akan menyebabkan data tidak dapat tersimpan seluruhnya pada *JavaCard* sehingga menyebabkan kurangnya informasi dalam data tersebut pada saat data diambil dari memori *JavaCard*.

Tabel 4.3 menunjukkan pemborosan memori yang dapat terjadi pada *JavaCard* jika data medis yang disimpan dalam *JavaCard* tidak melebihi panjang maksimalnya. Sebagai contoh, jika pada bagian anamnesa, diagnosa, dan terapi masing-masing hanya menggunakan memori 80 byte, maka keseluruhan data medis

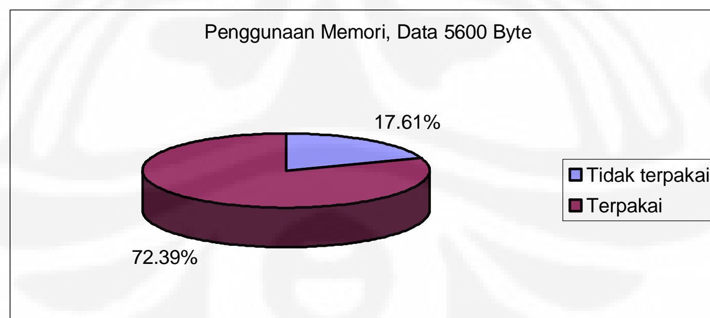
Implementasi aplikasi..., Donny, FT UI, 2008

hanya menggunakan memori 280 byte dari maksimal 340 byte. Dengan demikian, akan terdapat 60 byte yang kosong atau tidak terisi dan jika kondisi tersebut berlaku untuk keseluruhan data medis, akan terjadi pemborosan memori sebesar 1,2 kilobyte atau 17,61% dari keseluruhan 6816 byte memori aplikasi *Smart Health*.

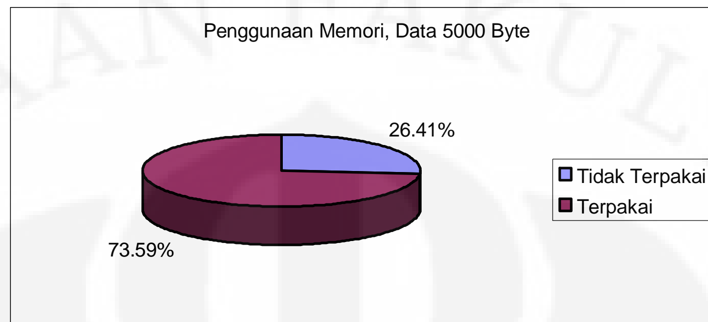
Tabel 4.3 Pemborosan memori *JavaCard*

Penggunaan Memori	Pemborosan Memori
20 data medis @ 280 byte	17,61 %
20 data medis @ 250 byte	26,41 %
20 data medis @ 220 byte	35,21 %

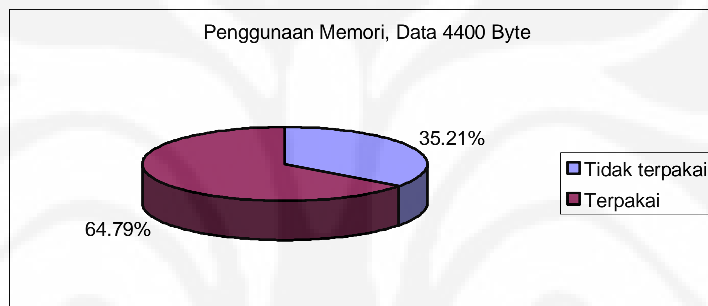
Solusi yang tepat untuk mengatasi kelemahan ini adalah dengan mempergunakan sistem *Dynamic Length* dalam pemrograman aplikasi. Sistem *Dynamic Length* adalah sebuah sistem dimana program yang dibuat harus mampu mendeteksi panjang data yang akan disimpan dalam *JavaCard*. Setiap data tidak ditentukan berapa panjang maksimalnya sehingga seluruh informasi pada data medis dapat tersimpan seluruhnya. Untuk memisahkan data yang satu dengan yang lain digunakan suatu karakter khusus. Pada operasi pembacaan data dari memori, program akan membaca data pada *JavaCard* hingga karakter pemisah terbaca. Dengan demikian, pemborosan memori dapat dihindari, seluruh informasi data medis dapat diambil, dan penggunaan memori *JavaCard* dapat lebih dioptimalkan.



Gambar 4.9 Diagram lingkaran penggunaan memori dengan data 5600 byte



Gambar 4.10 Diagram lingkaran penggunaan memori dengan data 5000 byte



Gambar 4.11 Diagram lingkaran penggunaan memori dengan data 4400 byte

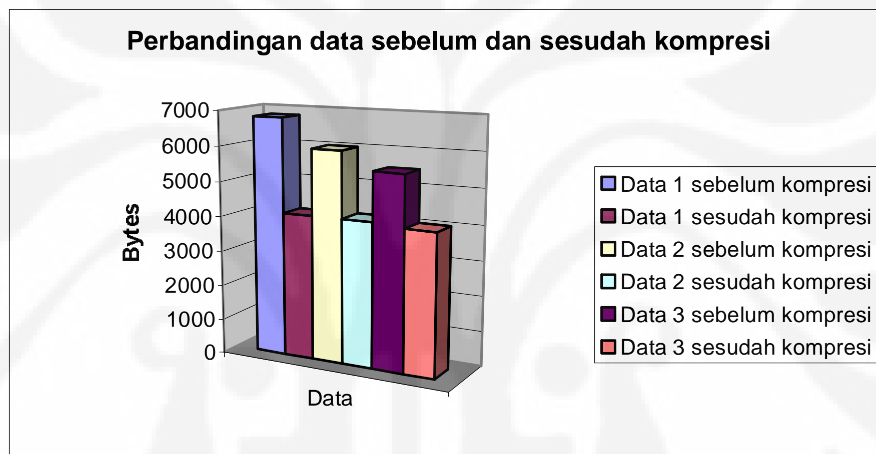
4.3.2 Kompresi Data

Aplikasi Smart Health mempergunakan memori *JavaCard* sebesar 6816 byte untuk menyimpan dua puluh data medis dan data pribadi tambahan. Setiap data medis berukuran 340 byte dan data pribadi tambahan berukuran 16 byte sehingga dibutuhkan 6800 byte untuk data medis dan 16 byte untuk data pribadi tambahan. Dasar perhitungan penggunaan memori di atas adalah bahwa setiap karakter yang dimasukkan bernilai 1 byte. Dengan demikian, setiap data medis memiliki panjang maksimal 340 karakter dan 16 karakter maksimal untuk data pribadi tambahan.

Penggunaan memori pada *JavaCard* dapat lebih dioptimalkan dengan cara melakukan kompresi data. Dengan kompresi data, ukuran data yang akan disimpan menjadi lebih kecil karena setiap karakter akan bernilai lebih kecil dari 1 byte. Dengan demikian, penggunaan memori pada *JavaCard* dapat dihemat. Untuk melakukan kompresi data, yang dibutuhkan hanyalah *library* tambahan sebagai *tools* untuk menjalankan proses kompresi data. Kompresi data dilakukan pada applet

connector, memproses array data dari modul terminal sebelum disimpan dalam *JavaCard*. Pada operasi pembacaan data, data yang diambil dari *JavaCard* oleh applet *connector* akan di de-kompres sebelum ditampilkan pada modul terminal.

Tabel 4.4 menunjukkan hasil dari ujicoba penyimpanan data yang terlebih dahulu melalui proses kompresi data. Proses kompresi data yang dilakukan mampu memperkecil ukuran data hingga menjadi sekitar 60% hingga 70% dari data aslinya. Hal ini berarti bahwa proses kompresi data sangat membantu dalam mengoptimalkan penggunaan memori *JavaCard*.



Gambar 4.12 Diagram batang perbandingan data sebelum dan sesudah kompresi

Tabel 4.4 Perbandingan data sebelum dan sesudah kompresi

Data Sebelum Kompresi	Data Sesudah Kompresi	Persentase
6800 byte	4127 byte	60,7 %
6000 byte	4136 byte	68,93 %
5500 byte	4039 byte	73,44 %

4.3.3 *Kecepatan Proses JavaCard*

Kecepatan kerja dari aplikasi *Smart Health* dipengaruhi oleh faktor kecepatan proses pada *JavaCard*. Pada operasi pembacaan data medis, proses yang dilakukan oleh aplikasi adalah mengakses data pribadi dan data medis yang tersimpan dalam memori *JavaCard*. Seluruh data yang ada dalam memori diambil satu persatu baru kemudian ditampilkan pada modul terminal. Pada proses inilah faktor kecepatan

Implementasi aplikasi..., Donny, FT UI, 2008

proses *JavaCard* sangat berpengaruh. Kecepatan mikrokontroler *JavaCard* sangat menentukan cepat lambatnya proses pengambilan data.

Sebagai contoh, dilakukan ujicoba proses pembacaan data pada kondisi *JavaCard* telah penuh terisi data medis dengan menggunakan *JavaCard* UI 2006 dan 2007. *JavaCard* UI 2006 membutuhkan waktu lebih kurang 20 detik untuk melakukan proses pembacaan data medis. Bandingkan dengan pada saat menggunakan *JavaCard* UI 2007, waktu yang dibutuhkan lebih kurang 12 detik. Hal ini berarti bahwa *JavaCard* UI 2007 memiliki kinerja 40% lebih baik daripada *JavaCard* UI 2006. Dengan demikian, kinerja aplikasi *Smart Health* akan menjadi lebih baik jika didukung oleh kecepatan proses dari *JavaCard*.

4.4 *TINGKAT KEMUDAHAN PENGGUNAAN APLIKASI*

Untuk mengetahui tingkat kemudahan penggunaan dari aplikasi *Smart Health*, aplikasi diujicobakan kepada 15 responden, 12 pria dan 3 wanita, yang bertempat di Direktorat Pelayanan dan Pengembangan Sistem Informasi UI. Responden diminta untuk mempergunakan aplikasi *Smart Health* untuk membaca dan menulis data pada *JavaCard*. Dari 15 responden tersebut, mereka berpendapat bahwa aplikasi *Smart Health* merupakan suatu aplikasi yang mudah dan praktis untuk digunakan. Dengan demikian, dapat disimpulkan bahwa aplikasi *Smart Health* adalah aplikasi dengan tingkat kemudahan penggunaan yang tinggi. Pengguna hanya perlu memasukkan *JavaCard* dalam *card reader* dan operasi penulisan serta pembacaan data dapat dilakukan dengan mengakses *field* modul terminal.

Tabel 4.5 Tanggapan responden terhadap aplikasi *Smart Health*

Jenis Kelamin	Tanggapan	
	Mudah	Sulit
Pria	12	-
Wanita	3	-

4.5 *PENGEMBANGAN APLIKASI MENDATANG*

Beberapa kekurangan dari aplikasi *Smart Health* yang dibangun telah dikemukakan. Pengembangan selanjutnya dapat dimulai dari menelaah ulang

Implementasi aplikasi..., Donny, FT UI, 2008

kekurangan-kekurangan tersebut sehingga kinerja dari aplikasi menjadi semakin baik. Beberapa perbaikan yang diharapkan pada aplikasi masa mendatang adalah terciptanya suatu aplikasi seperti *Smart Health* yang mengadaptasi sistem *Dynamic Length* sehingga tidak terjadi pemborosan memori dan penggunaan memori *JavaCard* dapat dioptimalkan. Selain itu, diharapkan aplikasi mendatang sudah menggunakan proses kompresi data sehingga penggunaan memori dapat ditekan sekecil mungkin.

Aplikasi *Smart Health* cakupannya dapat diperluas dengan menambah menu untuk mencetak data medis sehingga data medis yang terdapat pada *JavaCard* dapat dicetak oleh pengguna. Pengembangan lain yang dapat dilakukan untuk memperluas aplikasi *Smart Health* adalah mengintegrasikan aplikasi dengan sistem *database* institusi kesehatan melalui *Web Service*. Dengan pengembangan ini, dalam satu kali proses maka data medis akan tersimpan pada dua tempat, yaitu pada *JavaCard* dan sistem *database*. Hal ini juga dilakukan untuk mengantisipasi jika terjadi kehilangan *JavaCard*.

Implementasi aplikasi..., Donny, FT UI, 2008

BAB 5
KESIMPULAN

Setelah melakukan perancangan, merealisasikan rancangan, melakukan uji coba terhadap aplikasi, dan menganalisa kinerja aplikasi dapat ditarik kesimpulan:

- *Smart Card* adalah kartu berukuran seperti kartu kredit yang memiliki mikroprosesor dan dapat diprogram untuk melakukan suatu tugas serta menyimpan informasi.
- *JavaCard* adalah *Smart Card* yang berbasis Java. Pemrograman *JavaCard* dilakukan menggunakan bahasa pemrograman Java.
- Pembuatan aplikasi *JavaCard* terdiri dari tiga tahapan yaitu pembuatan applet *JavaCard*, pembuatan applet *connector*, dan pembuatan modul terminal.
- Applet *JavaCard* merupakan bagian integral dalam pembuatan aplikasi dan menentukan kelanjutan dari seluruh operasi aplikasi.
- Applet *connector* berfungsi sebagai penghubung applet *JavaCard* dengan modul terminal.
- Kunci dari pembuatan aplikasi *JavaCard* adalah pemrograman applet *JavaCard* dan pengaturan APDU untuk memproses dan menjalankan operasi-operasi dari aplikasi *JavaCard*.
- APDU berperan untuk mengatur lalu lintas pembacaan dan penyimpanan data.
- Aplikasi *Smart Health* adalah aplikasi yang berfungsi untuk menulis dan membaca data medis serta membaca data pribadi dari pemilik *JavaCard*.
- Alokasi memori *JavaCard* memiliki peranan yang penting dalam pembuatan aplikasi untuk mengoptimalkan penggunaan memori *JavaCard*.
- Kompresi data dapat dipergunakan untuk menghemat penggunaan memori *JavaCard*.
- Kinerja aplikasi *JavaCard* akan menjadi lebih baik jika didukung oleh kecepatan proses *JavaCard*.

Implementasi aplikasi..., Donny, FT UI, 2008

DAFTAR ACUAN

[1] “Teknologi SmartCard dan Impian di Masa Depan”, Antonius Aditya Hartanto.

[2] “Understanding Java Card 2.0, Learn the inner workings of the Java Card architecture, API, and runtime environment”, Zhiqun Chen, Rinaldo Di Giorgio. 1 Maret 1998.

[3] “Elements of Smart Card Architecture”. http://people.cs.uchicago.edu/~dinoj/smartcard/JCarch-1.html. Terakhir diakses pada tanggal 2 Desember 2007.

[4] “Java Cards”. http://people.cs.uchicago.edu/~dinoj/smartcard/JCarch-1.html. Terakhir diakses pada tanggal 2 Desember 2007.

[5] ”JavaCard and OpenCard Framework: A Tutorial”, O.Fodor, V.Hassler. Information Systems Institute, Technical University of Vienna.

[6] http://smartcard.ui.edu. Terakhir diakses pada tanggal 2 Desember 2007.

[7] Chen, Zhiqun. 2000. ” Java Card™ Technology for Smart Cards: Architecture and Programmer's Guide”. Addison Wesley.

[8] ”Applications of Smart Cards”. http://people.cs.uchicago.edu/~dinoj/smartcard/arch-1.html. Terakhir diakses pada tanggal 2 Desember 2007.

[9] ”Smart cards: A primer Develop on the Java platform of the future”, Rinaldo Di Giorgio. 1 Desember 1997.

[10] ”Smart cards and the OpenCard Framework, Learn how to implement a card terminal and use a standard API for interfacing to smart cards from your browser”, Rinaldo Di Giorgio. 1 Januari 1998.

[11] ”How to write a Java Card applet: A developer's guide, Learn the programming concepts and major steps of creating Java Card applets”, Zhiqun Chen. 1 Juli 1999.

Implementasi aplikasi..., Donny, FT UI, 2008

LAMPIRAN

Lampiran 1 *Listing* Program Applet *JavaCard*

```
package edu.ui.smartcard.applet.pkm;

import javacard.framework.*;

public class PKMApplet extends Applet {
  public PKMApplet() {
      register();
   }

   public PKMApplet(byte[] buffer, short offset, byte length) {
      this();
      initializePIN(buffer, offset, length);
      currentState = STATE_NOTHING;
   }

   /** Initialize PIN */
   private void initializePIN(byte[] buffer, short offset, byte length) {
      short aidLength = Util.makeShort( (byte) 0, buffer[offset]);
      short offsetParamLength = (short) ( offset + aidLength + 3 );
      short pinLength = buffer[offsetParamLength];

      pin = new OwnerPIN(PIN_TRY_LIMIT, MAX_PIN_SIZE);
      pin.update(buffer, (short) (offsetParamLength + 1), (byte) pinLength);
   }

   public static void install(byte[] buffer, short offset, byte length) {
      new PKMApplet(buffer, offset, length);
   }

   public void process(APDU apdu) throws ISOException {
      byte[] apduBuffer = apdu.getBuffer();
      byte cla = apduBuffer[ISO7816.OFFSET_CLA];
      byte instruction = apduBuffer[ISO7816.OFFSET_INS];
      byte p2 = apduBuffer[ISO7816.OFFSET_P2];

      /* selecting this applet*/
      if (this.selectingApplet()) {
         resetData();
         return;
      }

      if (cla != CLA) ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

      byte checkCond = (byte) 0;
      /* switch / case prerequisities */
      switch(instruction) {
      case INS_WRITE_DATA:
         checkCond |= CHECK_PIN_VALIDATED;
         break;
      }

      /* mengecek apakah PIN sudah divalidasi */
      if ( (CHECK_PIN_VALIDATED & checkCond) > 0) {
         if (!pin.isValidated())
            ISOException.throwIt(ERR_VERIFICATION_REQUIRED);
      }
```


Implementasi aplikasi..., Donny, FT UI, 2008

```
    switch(instruction) {
    case INS_READ_DATA:
      short blockNumber = Util.makeShort( (byte) 0, apduBuffer[ISO7816.OFFSET_P2]);
      short sequence = Util.makeShort( (byte) 0, apduBuffer[ISO7816.OFFSET_CDATA] );
      short dataLength = Util.makeShort( (byte) 0, apduBuffer[OFFSET_SRC_DATA]);
      sendData(apdu, blockNumber, sequence, dataLength);
        break;
    case INS_WRITE_DATA:
        setData(apduBuffer);
        break;
    case INS_VALIDATE_PIN:
        if (!pin.check (apduBuffer,(short) ISO7816.OFFSET_CDATA,
                    apduBuffer[ISO7816.OFFSET_LC]))
            ISOException.throwIt(ERR_VERIFICATION_FAILED);
        break;
    case INS_RESET_PIN:
        pin.reset();
        break;
    case INS_GET_RECORD_LENGTH:
      getRecordLength(apdu);
        break;
    case INS_READ_ADDITIONAL_DATA:
      short offsetAddData = Util.makeShort( (byte) 0,
            apduBuffer[ISO7816.OFFSET_CDATA] );
        sendAddData(apdu, offsetAddData);
        break;
    case INS_WRITE_ADDITIONAL_DATA:
        setAddData(apduBuffer);
        break;
    default:
        ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        break;
    }
}

/** Mengirim data
  * @param apdu APDU yang dikirim dari user
  * @param sequence Record keberapa yang ingin diambil
  */
private void sendData(APDU apdu, short blockNumber, short sequence, short dataLength) {
  if (overlap == false){
      short finalBlock = (short)(lastPos - 1);
      short readerCounter = (short) ((finalBlock - blockNumber) * BLOCK_SIZE);

      if (currentState == STATE_NOTHING && finalBlock >= blockNumber &&
        expectedReadSequence == sequence && sequence == 0) {
        //awal data
        currentState = STATE_GET_DATA;
        offset2 = readerCounter;
        expectedReadSequence = SEQUENCE_ID_DOKTER;
      } else if (currentState == STATE_GET_DATA && expectedReadSequence == sequence) {
          if (expectedReadSequence == SEQUENCE_ID_DOKTER) {
              //data ke-1 (ID Dokter)
              offset2 += LENGTH_INSTITUSI;
              expectedReadSequence++;
          } else if (expectedReadSequence == SEQUENCE_TGL) {
              //data ke-2 (tanggal)
              offset2 += LENGTH_ID_DOKTER;
              expectedReadSequence++;
```

```
} else if (expectedReadSequence == SEQUENCE_ANAMNESA) {
     //data ke-3 (anamnesa)
     offset2 += LENGTH_TGL;
     expectedReadSequence++;
  }  else if (expectedReadSequence == SEQUENCE_DIAGNOSA) {
     //data ke-4 (diagnosa)
     offset2 += LENGTH_TEXT_AREA;
     expectedReadSequence++;
  } else if (expectedReadSequence == SEQUENCE_TERAPI) {
     //akhir data (therapy)
     offset2 += LENGTH_TEXT_AREA;
     currentState = STATE_NOTHING;
     expectedReadSequence = 0;
} else {
   currentState = STATE_NOTHING;
   expectedReadSequence = 0;
   ISOException.throwIt(ERR_DATA_CORRUPTED);
}
} else {
   currentState = STATE_NOTHING;
   expectedReadSequence = 0;
   ISOException.throwIt(ERR_DATA_CORRUPTED);
}

byte[] returnHeader = new byte[]{(short)(finalBlock - blockNumber) == 0 ? LAST_SEQUENCE :
   (byte) 0x00, (byte) blockNumber, (byte) sequence, (byte) dataLength,(byte) offset2};
 short outgoingLength = (short) (dataLength + returnHeader.length);

 apdu.setOutgoing();
 apdu.setOutgoingLength(outgoingLength);

 apdu.sendBytesLong(returnHeader, (short) 0, (short) returnHeader.length);
 apdu.sendBytesLong(medicalRecord, offset2, dataLength);

 readerCounter = (short) 0;
}

if (overlap == true) {
  short finalBlock = (short) (TOTAL_BLOCKS - 1);

   /* rumus untuk membaca data medis jika terjadi overlap */
   short rumus = (short) ( (short)(finalBlock - blockNumber + lastPos) % (short) TOTAL_BLOCKS );

   short readerCounter = (short) (BLOCK_SIZE * rumus);

  if (currentState == STATE_NOTHING && finalBlock >= blockNumber &&
   expectedReadSequence == sequence && sequence == 0) {
   //awal data
   currentState = STATE_GET_DATA;
   offset2 = readerCounter;
   expectedReadSequence = SEQUENCE_ID_DOKTER;
  } else if (currentState == STATE_GET_DATA && expectedReadSequence == sequence) {
    if (expectedReadSequence == SEQUENCE_ID_DOKTER) {
      //data ke-1 (ID Dokter)
      offset2 += LENGTH_INSTITUSI;
      expectedReadSequence++;
    } else if (expectedReadSequence == SEQUENCE_TGL) {
      //data ke-2 (tanggal)
      offset2 += LENGTH_ID_DOKTER;
      expectedReadSequence++;
```

```
        }  else if (expectedReadSequence == SEQUENCE_ANAMNESA) {
             //data ke-3 (anamnesa)
             offset2 += LENGTH_TGL;
             expectedReadSequence++;
          }   else if (expectedReadSequence == SEQUENCE_DIAGNOSA) {
             //data ke-4 (diagnosa)
             offset2 += LENGTH_TEXT_AREA;
             expectedReadSequence++;
          } else if (expectedReadSequence == SEQUENCE_TERAPI) {
             //akhir data (therapy)
             offset2 += LENGTH_TEXT_AREA;
             currentState = STATE_NOTHING;
             expectedReadSequence = 0;
        } else {
           currentState = STATE_NOTHING;
           expectedReadSequence = 0;
           ISOException.throwIt(ERR_DATA_CORRUPTED);
        }
      } else {
        currentState = STATE_NOTHING;
        expectedReadSequence = 0;
        ISOException.throwIt(ERR_DATA_CORRUPTED);
      }

      byte[] returnHeader = new byte[] {(short)(finalBlock - blockNumber) == 0 ? LAST_SEQUENCE :
          (byte) 0x00, (byte) blockNumber, (byte) sequence, (byte) dataLength, (byte) offset2};
        short outgoingLength = (short) (dataLength + returnHeader.length);

        apdu.setOutgoing();
        apdu.setOutgoingLength(outgoingLength);

        apdu.sendBytesLong(returnHeader, (short) 0, (short) returnHeader.length);
        apdu.sendBytesLong(medicalRecord, offset2, dataLength);

        readerCounter = (short) 0;
    }
}

private void setData(byte[] apduBuffer) {
    /* jika record penuh, counter untuk menulis direset jadi 0 kembali
     * numpuk bow
     */
    if (lastPos > (short)(TOTAL_BLOCKS - 1)){
      lastPos = 0;
      writerCounter = 0;
      overlap = true;
    }

  short offset = writerCounter;
  byte p2 = apduBuffer[ISO7816.OFFSET_P2];
  short sequence = Util.makeShort( (byte) 0, apduBuffer[ISO7816.OFFSET_CDATA]);
  short dataLength = Util.makeShort( (byte) 0, apduBuffer[OFFSET_SRC_DATA]);

  if (dataLength > maxRecordData[sequence]){
    ISOException.throwIt(ERR_EXCESSIVE_LENGTH);
  }

  if (currentState == STATE_NOTHING && expectedWriteSequence == sequence &&
          sequence == 0 && p2 == (byte) 0x00) {
       //awal data (institusi)
```

```
        currentState = STATE_SET_DATA;
        writerCounter += LENGTH_INSTITUSI;
        expectedWriteSequence = SEQUENCE_ID_DOKTER;
    } else if (currentState == STATE_SET_DATA && expectedWriteSequence == sequence) {
        if (p2 == (byte) 0x00 && expectedWriteSequence == SEQUENCE_ID_DOKTER) {
            //data ke-1 (ID Dokter)
            writerCounter += LENGTH_ID_DOKTER;
            expectedWriteSequence++;
        } else if (p2 == (byte) 0x00 && expectedWriteSequence == SEQUENCE_TGL) {
            //data ke-2 (tanggal)
            writerCounter += LENGTH_TGL;
            expectedWriteSequence++;
      }  else if (p2 == (byte) 0x00 && expectedWriteSequence > SEQUENCE_TGL &&
            expectedWriteSequence < SEQUENCE_TERAPI) {
            //data ke-3 sampe 4 (anamnesa, diagnosa)
            writerCounter += LENGTH_TEXT_AREA;
            expectedWriteSequence++;
        } else if (p2 == LAST_SEQUENCE && expectedWriteSequence == SEQUENCE_TERAPI) {
            //akhir data (therapy)
            writerCounter += LENGTH_TEXT_AREA;
            currentState = STATE_NOTHING;
            lastPos ++;
        } else {
            resetData();
            ISOException.throwIt(ERR_STATE_CORRUPTED);
        }
    } else {
        resetData();
        ISOException.throwIt(ERR_STATE_CORRUPTED);
    }

    try {
        JCSystem.beginTransaction();
        Util.arrayCopy(apduBuffer, (short) (OFFSET_SRC_DATA + 1), medicalRecord,
          offset, dataLength);
        if (currentState == STATE_NOTHING) {
            expectedWriteSequence = 0;
        }
        JCSystem.commitTransaction();
    } catch (Exception e) {
        JCSystem.abortTransaction();
        resetData();
        ISOException.throwIt(ERR_SET_DATA_FAILED);
    }
}

private void resetData() {
    if (currentState == STATE_SET_DATA){
      currentState = STATE_NOTHING;
      short position = (short) (lastPos * BLOCK_SIZE);
      writerCounter = position;
      expectedWriteSequence = 0;
  } else if (currentState == STATE_SET_ADD_DATA) {
      currentState = STATE_NOTHING;
      addDataCounter = 0;
      expectedAddDataSequence = 0;
  }
}

/** Mendapatkan banyaknya record yang ada di kartu
```


Implementasi aplikasi..., Donny, FT UI, 2008

```
    * @return Banyaknya record yang ada di kartu
    */
  private void getRecordLength(APDU apdu){
  if (currentState == STATE_NOTHING && overlap == false){
      currentState = STATE_GET_RECORD_LENGTH;
      byte[] recordLength = new byte[1];
      recordLength[0] = (byte) lastPos;

      apdu.setOutgoing();
      apdu.setOutgoingLength((short) recordLength.length);
      apdu.sendBytesLong(recordLength, (short) 0, (short) 1);
      currentState = STATE_NOTHING;
  } else if (currentState == STATE_NOTHING && overlap == true){
      currentState = STATE_GET_RECORD_LENGTH;
      byte[] recordLength = new byte[1];
      recordLength[0] = TOTAL_BLOCKS;

      apdu.setOutgoing();
      apdu.setOutgoingLength((short) recordLength.length);
      apdu.sendBytesLong(recordLength, (short) 0, (short) 1);
      currentState = STATE_NOTHING;
  }
}

/** Mengirim data pribadi tambahan (Warga Negara & Status Kawin)
    * @param apdu APDU yang dikirim dari user
    * @param offsetAddData offset pengambilan data pribadi
    */
private void sendAddData(APDU apdu, short offsetAddData) {
    short dataLength = ADDITIONAL_DATA_LENGTH;
    short expectedOffset = 0;

        if (currentState == STATE_NOTHING && expectedOffset == offsetAddData) {
          currentState = STATE_GET_ADD_DATA;
          byte[] returnHeader = new byte[] {(short)expectedOffset==offsetAddData ? LAST_SEQUENCE:
             (byte) 0x00, (byte) offsetAddData, (byte) dataLength};
          short outgoingLength = (short) (dataLength + returnHeader.length);

            apdu.setOutgoing();
            apdu.setOutgoingLength(outgoingLength);

            apdu.sendBytesLong(returnHeader, (short) 0, (short) returnHeader.length);
            apdu.sendBytesLong(medicalRecord, offsetAddData, dataLength);
            currentState = STATE_NOTHING;
        } else ISOException.throwIt(ERR_GET_ADD_DATA_FAILED);
}

/** Menulis data pribadi tambahan (Warga Negara & Status Kawin)*/
private void setAddData(byte[] apduBuffer) {
      short offset = addDataCounter;
      byte p2 = apduBuffer[ISO7816.OFFSET_P2];
      short sequence = Util.makeShort( (byte) 0, apduBuffer[ISO7816.OFFSET_CDATA]);
      short dataLength = Util.makeShort( (byte) 0, apduBuffer[OFFSET_SRC_DATA]);

      if (dataLength > maxAddData[sequence]){
        ISOException.throwIt(ERR_EXCESSIVE_LENGTH);
      }

    if (currentState == STATE_NOTHING && expectedAddDataSequence == sequence &&
            sequence == SEQUENCE_NEGARA && p2 == (byte) 0x00) {
```

```
        //data Warga Negara
        currentState = STATE_SET_ADD_DATA;
        addDataCounter += LENGTH_NEGARA;
        expectedAddDataSequence = SEQUENCE_STATUS;
    } else if (currentState == STATE_SET_ADD_DATA && expectedAddDataSequence == sequence) {
        if (p2 == (byte) LAST_SEQUENCE && sequence == SEQUENCE_STATUS) {
            //data Status Kawin
            addDataCounter += LENGTH_STATUS;
            currentState = STATE_NOTHING;
        } else {
            resetData();
            ISOException.throwIt(ERR_STATE_CORRUPTED);
        }
    } else {
        resetData();
        ISOException.throwIt(ERR_STATE_CORRUPTED);
    }

    try {
        JCSystem.beginTransaction();
        Util.arrayCopy(apduBuffer, (short) (OFFSET_SRC_DATA + 1), medicalRecord,
          offset, dataLength);
        if (currentState == STATE_NOTHING) {
            expectedAddDataSequence = 0;
            addDataCounter = 0;
        }
        JCSystem.commitTransaction();
    } catch (Exception e) {
        JCSystem.abortTransaction();
        resetData();
        ISOException.throwIt(ERR_SET_ADD_DATA_FAILED);
    }
}

/* Variables */
private OwnerPIN pin;
private static final byte PIN_TRY_LIMIT = (byte) 6;
private static final byte MAX_PIN_SIZE = (byte) 6;
private static final byte CHECK_PIN_VALIDATED = (byte) 1;
private static final byte LAST_SEQUENCE = (byte) 0x80;

/* States */
private static final byte STATE_NOTHING = 0;
private static final byte STATE_GET_DATA = 1;
private static final byte STATE_SET_DATA = 2;
private static final byte STATE_GET_RECORD_LENGTH = 3;
private static final byte STATE_GET_ADD_DATA = 4;
private static final byte STATE_SET_ADD_DATA = 5;

private byte currentState = STATE_NOTHING;

/* Sequences */
private static final byte SEQUENCE_ID_DOKTER = 1;
private static final byte SEQUENCE_TGL = 2;
private static final byte SEQUENCE_ANAMNESA = 3;
private static final byte SEQUENCE_DIAGNOSA = 4;
private static final byte SEQUENCE_TERAPI = 5;
private static final byte SEQUENCE_NEGARA = 0;
private static final byte SEQUENCE_STATUS = 1;
```

```
    private short writerCounter = (short) 0;
    private short offset2 = (short) 0;
    private short addDataCounter = (short) 0;
    private short lastPos = (short) 0; //melihat posisi record terakhir
    private boolean overlap = false;    //menentukan apakah record sudah penuh atau belum
    private short expectedWriteSequence = (short) 0;
    private short expectedReadSequence = (short) 0;
    private short expectedAddDataSequence = (short) 0;

    /* The data */
    private static final short LENGTH_INSTITUSI = (short) 20;
    private static final short LENGTH_ID_DOKTER = (short) 12;
    private static final short LENGTH_TGL = (short) 8;
    private static final short LENGTH_TEXT_AREA = (short) 100;
    private static final short LENGTH_NEGARA = (short) 15;
    private static final short LENGTH_STATUS = (short) 1;

    /* hasil pertambahan dari LENGTH_... (20 + 12 + 8 + 100 + 100 + 100) */
    private static final short BLOCK_SIZE = (short) 340;
    private static final short TOTAL_BLOCKS = (short) 20;

    /* hasil dari BLOCK_SIZE * TOTAL_BLOCKS (190 * 20) */
    private static final short MEDICAL_RECORD_LENGTH = (short) 6800;
    private static final short ADDITIONAL_DATA_LENGTH = (short) 16;

    private byte[] medicalRecord = new byte[MEDICAL_RECORD_LENGTH];
    private byte[] additionalData = new byte[ADDITIONAL_DATA_LENGTH];
    private static final short[] maxAddData = new short[] {15, 1};
    private static final short[] maxRecordData = new short[] {20, 12, 8, 100, 100, 100};

    private static final short OFFSET_SRC_DATA = (short) ( ISO7816.OFFSET_CDATA + 1 );

    /* CLA */
    private static final byte CLA = (byte) 0x90;

    /*Instructions*/
    private static final byte INS_READ_DATA = (byte) 0x30;
    private static final byte INS_WRITE_DATA = (byte) 0x31;
    private static final byte INS_VALIDATE_PIN = (byte) 0x32;
    private static final byte INS_RESET_PIN = (byte) 0x33;
    private static final byte INS_GET_RECORD_LENGTH = (byte) 0x34;
    private static final byte INS_READ_ADDITIONAL_DATA = (byte) 0x35;
    private static final byte INS_WRITE_ADDITIONAL_DATA = (byte) 0x36;

    /* Errors */
    private static final short ERR_VERIFICATION_FAILED = (short) 0x5100;
    private static final short ERR_VERIFICATION_REQUIRED = (short) 0x5200;
    private static final short ERR_DATA_CORRUPTED = (short) 0x5300;
    private static final short ERR_STATE_CORRUPTED = (short) 0x5400;
    private static final short ERR_SET_DATA_FAILED = (short) 0x5500;
    private static final short ERR_GET_ADD_DATA_FAILED = (short) 0x5600;
    private static final short ERR_SET_ADD_DATA_FAILED = (short) 0x5700;
    private static final short ERR_EXCESSIVE_LENGTH = (short) 0x5800;
}
```

Lampiran 2 *Listing* Program Applet *Connector*

```
package edu.ui.smartcard.data.pkm;

import java.io.ByteArrayOutputStream;
import java.io.IOException;

import edu.ui.smartcard.connector.APDUTerminal;
import edu.ui.smartcard.connector.Applet;
import edu.ui.smartcard.connector.CommandAPDU;
import edu.ui.smartcard.connector.ResponseAPDU;
import edu.ui.smartcard.connector.ResponseErrorException;
import edu.ui.smartcard.connector.TerminalException;
import edu.ui.smartcard.data.InvalidPINException;
import edu.ui.smartcard.data.VerifyingRequiredException;

public class PKMAppletConnector extends Applet {

    public PKMAppletConnector(APDUTerminal terminal) {
        super(terminal);
        // TODO Auto-generated constructor stub
    }

    public void openWriteSession(String pin) throws InvalidPINException,
            ResponseErrorException, TerminalException {

        if (!authenticated) {
            byte[] bPin;

            if (pin == null) {
                bPin = new byte[] { 0 };
            } else
                bPin = pin.getBytes();

            byte[] apdu = new byte[6 + bPin.length];

            apdu[0] = CLA;
            apdu[1] = INS_VALIDATE_PIN;
            apdu[2] = 0;
            apdu[3] = 0;
            apdu[4] = (byte) (bPin.length );
            System.arraycopy(bPin, 0, apdu, 5, bPin.length);
            apdu[apdu.length - 1] = END_OF_APDU;

            ResponseAPDU res = sendAPDU(new CommandAPDU(apdu));

            if (!res.isSuccess()) {
                if (res.isEqual(ERR_VERIFICATION_FAILED))
                    throw new InvalidPINException();
                else
                    throw new ResponseErrorException(res);
            }

            authenticated = true;
        }
    }

    public void closeWriteSession() throws ResponseErrorException,
```

```
        TerminalException {
    // Actually resets the PIN number
    ResponseAPDU res = sendAPDU(new CommandAPDU(new byte[] { CLA,
            INS_RESET_PIN, 0, 0, 0 }));

    if (!res.isSuccess()) {
        throw new ResponseErrorException(res,
                "Unable to close applet instance.");
    }

    authenticated = false;
}

public byte[] getMedicalRecord(int record)
    throws ResponseErrorException, TerminalException, IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();

    byte[] maxLength = new byte[] {20, 12, 8, 100, 100, 100};
    System.out.println("record: "+ record);
    for (int sequence = 0; sequence < 6; sequence++) {
        byte[] apdu = new byte[7];// { CLA, INS_READ_DATA, 0, 0, 2, (byte)
        // record,
        // (byte) BLOCK_SIZE }; // nanti diubah jadi B9
        apdu[0] = CLA;
        apdu[1] = INS_READ_DATA;
        apdu[2] = 0;

        /* ganti sama record */
        apdu[3] = (byte) record;// (sequence != 5) ? 0 : (byte) 0x80;

        apdu[4] = (byte) 0x02;
        apdu[5] = (byte) (sequence);
        apdu[6] = maxLength[sequence];

        ResponseAPDU res = sendAPDU(new CommandAPDU(apdu));

        if (!res.isSuccess()) {
            if (res.isEqual(ERR_DATA_CORRUPTED))
                throw new ResponseErrorException(res, "Data Corrupted");
            else {
                System.out.println("error: " + res.getSW1SW2());
                throw new ResponseErrorException(res, "Unable to get data");
            }
        }

        byte[] result = res.getData();
        //System.out.println("result: ");
        //for (int i = 0; i < result.length; i++) {
        //    System.out.print(result[i] + " ");
        //}
        //System.out.println();

        byte[] chunk = new byte[maxLength[sequence]];
        System.arraycopy(result, 5, chunk, 0, chunk.length);
        short x1 = (short) (chunk.length - 1);
        while (chunk[x1] == 0 && x1 > 0) {
            x1--;
        }
        byte[] dataMedis = new byte[x1 + 1];
        System.arraycopy(chunk, 0, dataMedis, 0, x1 + 1);
```

```
        baos.write(dataMedis);
        baos.write(DELIMITER.getBytes());

    }

    return baos.toByteArray();
}

public void setMedicalRecord(int sequence, byte[] data)
        throws VerifyingRequiredException, ResponseErrorException,
        TerminalException {

    int maxLength = -1;

    switch (sequence) {
    case 0:
        maxLength = LENGTH_INSTITUSI;
        break;
    case 1:
        maxLength = LENGTH_ID_DOKTER;
        break;
    case 2:
        maxLength = LENGTH_TGL;
        break;
    case 3:
        maxLength = LENGTH_TEXT_AREA;
    case 4:
    case 5:
        maxLength = LENGTH_TEXT_AREA;
        break;
    default:
        return;
    }

    byte[] dataBuffer = new byte[maxLength];
    System.arraycopy(data, 0, dataBuffer, 0,
            maxLength > data.length ? data.length : maxLength);

    byte[] apdu = new byte[7 + dataBuffer.length];
    apdu[0] = CLA;
    apdu[1] = INS_WRITE_DATA;
    apdu[2] = 0;
    apdu[3] = (sequence != 5) ? 0 : (byte) 0x80;
    apdu[4] = (byte) (dataBuffer.length + 2);
    apdu[5] = (byte) (sequence);
    apdu[6] = (byte) dataBuffer.length;

    System.arraycopy(dataBuffer, 0, apdu, 7, dataBuffer.length);
    ResponseAPDU res = sendAPDU(new CommandAPDU(apdu));

    if (!res.isSuccess()) {
        if (res.isEqual(ERR_VERIFICATION_REQUIRED))
            throw new VerifyingRequiredException();
        else if (res.isEqual(ERR_STATE_CORRUPTED))
            throw new ResponseErrorException(res,
                    "State medical record corrupted");
        else if (res.isEqual(ERR_SET_DATA_FAILED))
            throw new ResponseErrorException(res, "Set data failed");
        else {
```

```
            System.out.println("Error: " + res.getSW1SW2());
            throw new ResponseErrorException(res, "Unable to write data");
        }
    }
}

public int getRecordLength() throws ResponseErrorException,
        TerminalException {
    byte[] apdu = new byte[] { CLA, INS_GET_RECORD_LENGTH, 0, 0, 0 };

    ResponseAPDU res = sendAPDU(new CommandAPDU(apdu));

    if (!res.isSuccess()) {
        throw new ResponseErrorException(res, "Unable get record length");
    }

    byte[] arrayB = res.getData();
    return (int) (arrayB[0]);
}

public byte[] getAddData() throws ResponseErrorException,
        TerminalException, IOException {

    ByteArrayOutputStream hoho = new ByteArrayOutputStream();

    byte[] apdu = new byte[] { CLA, INS_READ_ADDITIONAL_DATA, 0, 0, 2, 0,
            (byte) ADDITIONAL_DATA_LENGTH };

    ResponseAPDU res = sendAPDU(new CommandAPDU(apdu));

    if (!res.isSuccess()) {
        if (res.isEqual(ERR_GET_ADD_DATA_FAILED))
            throw new ResponseErrorException(res,
                    "Get additional data failed");
        else {
            System.out.println("error: " + res.getSW1SW2());
            throw new ResponseErrorException(res, "Unable to get data");
        }
    }

    byte[] hasil = res.getData();
    byte[] chunk2 = new byte[hasil.length - 3];
    System.arraycopy(hasil, 3, chunk2, 0, chunk2.length);

    // ambil Warga Negara
    byte[] buffer7 = new byte[LENGTH_NEGARA];
    System.arraycopy(chunk2, 0, buffer7, 0, LENGTH_NEGARA);
    short x7 = (short) (buffer7.length - 1);
    while (buffer7[x7] == 0 && x7 > 0) {
        x7--;
    }
    byte[] wargaNegara = new byte[x7 + 1];
    System.arraycopy(buffer7, 0, wargaNegara, 0, x7 + 1);

    hoho.write(wargaNegara);
    hoho.write(DELIMITER.getBytes());

    // Status Kawin
    byte[] statusKawin = new byte[LENGTH_STATUS];
    System.arraycopy(chunk2, 15, statusKawin, 0, LENGTH_STATUS);
```


Implementasi aplikasi..., Donny, FT UI, 2008

```
    hoho.write(statusKawin);
    return hoho.toByteArray();
}

public void setAddData(int sequence, byte[] dataTambahan)
        throws VerifyingRequiredException, ResponseErrorException,
        TerminalException {

    int maxLength = -1;

    switch (sequence) {
    case 0:
        maxLength = LENGTH_NEGARA;
        break;
    case 1:
        maxLength = LENGTH_STATUS;
        break;
    default:
        return;
    }

    byte[] dataBuffer2 = new byte[maxLength];
    System.arraycopy(dataTambahan, 0, dataBuffer2, 0,
            maxLength > dataTambahan.length ? dataTambahan.length
                    : maxLength);

    byte[] apdu = new byte[7 + dataBuffer2.length];
    apdu[0] = CLA;
    apdu[1] = INS_WRITE_ADDITIONAL_DATA;
    apdu[2] = 0;
    apdu[3] = (sequence != 1) ? 0 : (byte) 0x80;
    apdu[4] = (byte) (dataBuffer2.length + 2);
    apdu[5] = (byte) (sequence);
    apdu[6] = (byte) dataBuffer2.length;

    System.arraycopy(dataBuffer2, 0, apdu, 7, dataBuffer2.length);
    ResponseAPDU res = sendAPDU(new CommandAPDU(apdu));

    if (!res.isSuccess()) {
        if (res.isEqual(ERR_VERIFICATION_REQUIRED))
            throw new VerifyingRequiredException();
        else if (res.isEqual(ERR_STATE_CORRUPTED))
            throw new ResponseErrorException(res,
                    "State medical record corrupted");
        else if (res.isEqual(ERR_SET_ADD_DATA_FAILED))
            throw new ResponseErrorException(res,
                    "Set additional data failed");
        else {
            System.out.println("Error: " + res.getSW1SW2());
            throw new ResponseErrorException(res, "Unable to write data");
        }
    }
}

public void closeApplet() throws ResponseErrorException, TerminalException {
    // no implementation
}

@Override
```


Implementasi aplikasi..., Donny, FT UI, 2008

```
    public byte[] getAID() {
        // TODO Auto-generated method stub
        return APPLET_AID;
    }

    private static final byte[] APPLET_AID = new byte[] { (byte) 0xF3, 0x60, 0,
            0, 0x1, 0x07 };

    private boolean authenticated = false;

    private static final int LENGTH_INSTITUSI = 20;
    private static final int LENGTH_ID_DOKTER = 12;
    private static final int LENGTH_TGL = 8;
    private static final int LENGTH_TEXT_AREA = 100;
    private static final int LENGTH_NEGARA = 15;
    private static final int LENGTH_STATUS = 1;

    // Additional Data length
    private static final int ADDITIONAL_DATA_LENGTH = 16;

    /* APDUs */

    /* CLA */
    private static final byte CLA = (byte) 0x90;

    /* INS */
    private static final byte INS_READ_DATA = (byte) 0x30;
    private static final byte INS_WRITE_DATA = (byte) 0x31;
    private static final byte INS_VALIDATE_PIN = (byte) 0x32;
    private static final byte INS_RESET_PIN = (byte) 0x33;
    private static final byte INS_GET_RECORD_LENGTH = (byte) 0x34;
    private static final byte INS_READ_ADDITIONAL_DATA = (byte) 0x35;
    private static final byte INS_WRITE_ADDITIONAL_DATA = (byte) 0x36;

    /* P1 */
    // none so far
    /* P2 */
    // none so far
    /* End of APDU */
    private static final byte END_OF_APDU = (byte) 0x00;

    /* APDU Errors */
    private static final byte[] ERR_VERIFICATION_FAILED = new byte[] { (byte) 0x51, 0 };
    private static final byte[] ERR_VERIFICATION_REQUIRED = new byte[] { (byte) 0x52, 0 };
    private static final byte[] ERR_DATA_CORRUPTED = new byte[] { (byte) 0x53, 0 };
    private static final byte[] ERR_STATE_CORRUPTED = new byte[] { (byte) 0x54, 0 };
    private static final byte[] ERR_SET_DATA_FAILED = new byte[] { (byte) 0x55, 0 };
    private static final byte[] ERR_GET_ADD_DATA_FAILED = new byte[] { (byte) 0x56, 0 };
    private static final byte[] ERR_SET_ADD_DATA_FAILED = new byte[] { (byte) 0x57, 0 };

    public static final String DELIMITER = "++++";
}
```

Lampiran 3 *Listing* Program Modul Terminal

3.1 Modul InsertMedicalRecord

```
package edu.ui.smartcard.terminal.module.pkm;

import edu.ui.smartcard.connector.CardEvent;
import edu.ui.smartcard.connector.ResponseErrorException;
import edu.ui.smartcard.connector.TerminalException;
import edu.ui.smartcard.data.InvalidPINException;
import edu.ui.smartcard.data.VerifyingRequiredException;
import edu.ui.smartcard.data.mahasiswa.MahasiswaUIApplet;
import edu.ui.smartcard.data.pkm.PKMAppletConnector;
import edu.ui.smartcard.terminal.ModulePanel;
import edu.ui.webservice.smartcard.isidata.IsiDataLocator;
import edu.ui.webservice.smartcard.isidata.IsiDataPortType;
import java.rmi.RemoteException;
import javax.swing.JOptionPane;
import javax.xml.rpc.ServiceException;


public class InsertMedicalRecord extends ModulePanel {

    /** Creates new form InsertMedicalRecord */
    public InsertMedicalRecord() {
        initComponents();
        setNullVariables();
    }


    private void initComponents() {
        //Inisialisasi komponen dan desain GUI
    }

    private void submitBtnActionPerformed(java.awt.event.ActionEvent evt) {
      //GEN-FIRST:event_submitBtnActionPerformed
      // TODO add your handling code here:
        try {
            //Mendapatkan PIN
            IsiDataLocator isiDataLocator = new IsiDataLocator();
            IsiDataPortType isiData = isiDataLocator.getIsiDataPort();

            String isiPIN = isiData.getPin();

            //Inisiasi PKMConnector
            PKMAppletConnector pkmConnector = new PKMAppletConnector(terminal);
            pkmConnector.selectApplet();
            pkmConnector.openWriteSession(isiPIN);

            int counter = 0;
            pkmConnector.setMedicalRecord(counter++, institusi.getText().getBytes());
            pkmConnector.setMedicalRecord(counter++, idDokter.getText().getBytes());
            pkmConnector.setMedicalRecord(counter++, tglPeriksa.getText().getBytes());
            pkmConnector.setMedicalRecord(counter++, anamnesa.getText().getBytes());
            pkmConnector.setMedicalRecord(counter++, diagnosa.getText().getBytes());
            pkmConnector.setMedicalRecord(counter++, terapi.getText().getBytes());

            pkmConnector.closeWriteSession();
```

```
            pkmConnector.closeApplet();

            JOptionPane.showMessageDialog(this, "Data medis telah dimasukkan.",
                    "Sukses Memasukkan Data", JOptionPane.INFORMATION_MESSAGE);

            setNullVariables();
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (ServiceException e) {
            e.printStackTrace();
        } catch (InvalidPINException e) {
            e.printStackTrace();
        } catch (VerifyingRequiredException e) {
            e.printStackTrace();
        } catch (ResponseErrorException e) {
            System.out.println(e.getResponse().getSW1SW2());
            e.printStackTrace();
        } catch (TerminalException e) {
            e.printStackTrace();
        }
    }//GEN-LAST:event_submitBtnActionPerformed

    public void cardInserted(CardEvent event) {
        //Jika modul ini tidak sedang di-view, maka tidak melakukan apa2
        if (!this.isActive()) return;

        super.cardInserted(event);

        try {
            ambilData();
        } catch (ResponseErrorException e) {
            e.printStackTrace();
        } catch (TerminalException e) {
            e.printStackTrace();
        }
    }

    public void cardRemoved(CardEvent event) {
        super.cardRemoved(event);

        setNullVariables();
    }

    /** Mengeset semua field menjadi tidak dapat diisi sampai identitas mahasiswa
     *  telah terlihat di GUI.
     */


    private void setNullVariables() {
        lblNama.setText(NULL_TEXT);
        lblKodeOrganisasi.setText(NULL_TEXT);
        lblNPM.setText(NULL_TEXT);
        institusi.setEditable(false);
        institusi.setText(NULL_TEXT);
        idDokter.setEditable(false);
        idDokter.setText(NULL_TEXT);
        tglPeriksa.setEditable(false);
        tglPeriksa.setText(NULL_TEXT);
        anamnesa.setEditable(false);
        anamnesa.setText(NULL_TEXT);
```

```
    diagnosa.setEditable(false);
    diagnosa.setText(NULL_TEXT);
    terapi.setEditable(false);
    terapi.setText(NULL_TEXT);
    submitBtn.setEnabled(false);
}

private void enableField() {
    institusi.setEditable(true);
    idDokter.setEditable(true);
    tglPeriksa.setEditable(true);
    anamnesa.setEditable(true);
    diagnosa.setEditable(true);
    terapi.setEditable(true);
    submitBtn.setEnabled(true);
    institusi.setText("");
    idDokter.setText("");
    tglPeriksa.setText("");
    anamnesa.setText("");
    diagnosa.setText("");
    terapi.setText("");
}

private void ambilData() throws ResponseErrorException, TerminalException {
    MahasiswaUIApplet mhsApplet = MahasiswaUIApplet.
            getInstanceApplet(terminal);
    mhsApplet.selectApplet();
    lblNama.setText(mhsApplet.getNama());
    lblNPM.setText(mhsApplet.getNPM());
    lblKodeOrganisasi.setText(mhsApplet.getKodeOrganisasi());
    mhsApplet.closeApplet();

    enableField();
}

/**
 * Return the name of this module. This name will be shown in module tab.
 *
 * @return module name
 */
public String getName() {
    return "Insert Medical Record";
}

// Variables declaration - do not modify//GEN-BEGIN:variables
private javax.swing.JTextArea anamnesa;
private javax.swing.JTextArea diagnosa;
private javax.swing.JTextField idDokter;
private javax.swing.JTextField institusi;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel2;
private javax.swing.JLabel jLabel3;
private javax.swing.JLabel jLabel4;
private javax.swing.JLabel jLabel5;
private javax.swing.JLabel jLabel6;
private javax.swing.JLabel jLabel7;
private javax.swing.JLabel jLabel8;
private javax.swing.JLabel jLabel9;
private javax.swing.JScrollPane jScrollPane1;
private javax.swing.JScrollPane jScrollPane2;
```

```
    private javax.swing.JScrollPane jScrollPane3;
    private javax.swing.JLabel lblKodeOrganisasi;
    private javax.swing.JLabel lblNPM;
    private javax.swing.JLabel lblNama;
    private javax.swing.JButton submitBtn;
    private javax.swing.JTextArea terapi;
    private javax.swing.JTextField tglPeriksa;
    // End of variables declaration//GEN-END:variables

    public static final String NULL_TEXT = "-";
}
```

3.2 Modul ViewMedicalRecord

```
package edu.ui.smartcard.terminal.module.pkm;

import edu.ui.smartcard.connector.CardEvent;
import edu.ui.smartcard.connector.ResponseErrorException;
import edu.ui.smartcard.connector.TerminalException;
import edu.ui.smartcard.data.mahasiswa.MahasiswaUIApplet;
import edu.ui.smartcard.data.pkm.PKMAppletConnector;
import edu.ui.smartcard.terminal.ModulePanel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.text.ParseException;
import java.util.Date;

public class ViewMedicalRecord extends ModulePanel {

    /** Creates new form ViewMedicalRecord */
    public ViewMedicalRecord() {
        initComponents();
        records.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                javax.swing.JComboBox cb = (javax.swing.JComboBox) event.getSource();
                int index = cb.getSelectedIndex();
                System.out.println("selectedIndex: "+index);
                updateTampilanRecord(index);
            }
        });
        setNullVariables();
    }

    private void initComponents() {
        //Inisialisasi komponen dan desain GUI
    }

    public void cardInserted(CardEvent event) {
        if (!this.isActive()) return;

        super.cardInserted(event);

        try {
            ambilDataMahasiswa();
            ambilDataTambahan();
            ambilMedicalRecord();
            inisialisasiRecord();
            updateTampilanRecord(0);
```

```
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ResponseErrorException e) {
        System.out.println("errornya: " + e.getResponse().getSW1SW2());
        e.printStackTrace();
    } catch (ParseException e) {
        e.printStackTrace();
    } catch (TerminalException e) {
        e.printStackTrace();
    }
}

public void cardRemoved(CardEvent event) {
    super.cardRemoved(event);

    setNullVariables();
}

private void ambilMedicalRecord() throws IOException, ResponseErrorException,
        TerminalException {
    PKMAppletConnector pkmConnector = new PKMAppletConnector(terminal);
    pkmConnector.selectApplet();

    int jumlahRecord = pkmConnector.getRecordLength();
    medicalRecord = new String[jumlahRecord][6];

    for (int i = 0; i < medicalRecord.length; i++) {
        String chunkString = new String(pkmConnector.getMedicalRecord(i));
        medicalRecord[i] = chunkString.split(PKMAppletConnector.DELIMITER_REGEX);
    }

    pkmConnector.closeApplet();
}

private void setNullVariables() {
    lblNama.setText(NULL_TEXT);
    lblNPM.setText(NULL_TEXT);
    lblKodeOrg.setText(NULL_TEXT);
    lblTglLahir.setText(NULL_TEXT);
    lblJenisKelamin.setText(NULL_TEXT);
    lblKewarganegaraan.setText(NULL_TEXT);
    lblStatus.setText(NULL_TEXT);
    lblGolonganDarah.setText(NULL_TEXT);
    records.removeAllItems();
    institusi.setText(NULL_TEXT);
    idDokter.setText(NULL_TEXT);
    tglPeriksa.setText(NULL_TEXT);
    anamnesa.setText(NULL_TEXT);
    diagnosa.setText(NULL_TEXT);
    terapi.setText(NULL_TEXT);
}

private void ambilDataMahasiswa() throws ParseException,
        ResponseErrorException, TerminalException {
    MahasiswaUIApplet mhsApplet = MahasiswaUIApplet.
            getInstanceApplet(terminal);
    mhsApplet.selectApplet();

    String npm = mhsApplet.getNPM();
    lblNPM.setText(npm);
```


Implementasi aplikasi..., Donny, FT UI, 2008

```
        lblNama.setText(mhsApplet.getNama());
        lblKodeOrg.setText(mhsApplet.getKodeOrganisasi());

        Date tglLahir = mhsApplet.getTanggalLahir();
        lblTglLahir.setText(tglLahir.toString());

        lblJenisKelamin.setText(mhsApplet.getJenisKelamin());
        lblGolonganDarah.setText(mhsApplet.getGolonganDarah());

        mhsApplet.closeApplet();
    }

    private void ambilDataTambahan() throws IOException,
            ResponseErrorException, TerminalException {
        PKMAppletConnector pkmConnector = new PKMAppletConnector(terminal);
        pkmConnector.selectApplet();

        dataTambahan = new String[2][6];
        for (int i = 0; i < dataTambahan.length; i++) {
            String chunkString = new String(pkmConnector.getAddData(i));
            dataTambahan[i] = chunkString.split(PKMAppletConnector.DELIMITER_REGEX);
        }

        int counter = 0;
        lblKewarganegaraan.setText(medicalRecord[1][counter++]);
        lblStatus.setText(medicalRecord[2][counter++]);

        pkmConnector.closeApplet();
    }

    private void inisialisasiRecord() {

        int jumlahRecord = 20;
        String[] recordString = new String[jumlahRecord];
        for (int i = 0; i < jumlahRecord; i++) {
            recordString[i] = "Record ke-"+(i + 1);
            records.addItem(recordString[i]);
        }

        records.setSelectedIndex(0);
    }

    private void updateTampilanRecord(int index) {
        if (index < 0) return;

        int counter = 0;
        institusi.setText(medicalRecord[index][counter++]);
        idDokter.setText(medicalRecord[index][counter++]);
        tglPeriksa.setText(medicalRecord[index][counter++]);
        anamnesa.setText(medicalRecord[index][counter++]);
        diagnosa.setText(medicalRecord[index][counter++]);
        terapi.setText(medicalRecord[index][counter++]);
    }

    /**
     * Return the name of this module. This name will be shown in module tab.
     *
     * @return module name
     */
```

```
    public String getName() {
        return "View Medical Record";
    }

    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JTextArea anamnesa;
    private javax.swing.JTextArea diagnosa;
    private javax.swing.JTextField idDokter;
    private javax.swing.JTextField institusi;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel10;
    private javax.swing.JLabel jLabel11;
    private javax.swing.JLabel jLabel12;
    private javax.swing.JLabel jLabel13;
    private javax.swing.JLabel jLabel14;
    private javax.swing.JLabel jLabel15;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JLabel jLabel4;
    private javax.swing.JLabel jLabel5;
    private javax.swing.JLabel jLabel6;
    private javax.swing.JLabel jLabel7;
    private javax.swing.JLabel jLabel8;
    private javax.swing.JLabel jLabel9;
    private javax.swing.JScrollPane jScrollPane1;
    private javax.swing.JScrollPane jScrollPane2;
    private javax.swing.JScrollPane jScrollPane3;
    private javax.swing.JLabel lblGolonganDarah;
    private javax.swing.JLabel lblJenisKelamin;
    private javax.swing.JLabel lblKewarganegaraan;
    private javax.swing.JLabel lblKodeOrg;
    private javax.swing.JLabel lblNPM;
    private javax.swing.JLabel lblNama;
    private javax.swing.JLabel lblStatus;
    private javax.swing.JLabel lblTglLahir;
    private javax.swing.JComboBox records;
    private javax.swing.JTextArea terapi;
    private javax.swing.JTextField tglPeriksa;
    // End of variables declaration//GEN-END:variables

    public static final String NULL_TEXT = "-";
    private String[][] medicalRecord;
}
```


Implementasi aplikasi..., Donny, FT UI, 2008