OTOMATISASI SISTEM TRANSPORTASI *MONORAIL* BERBASIS TEKNOLOGI MULTI-AGEN

SKRIPSI

oleh:
JOHAN
04 04 03 0555

DEPARTEMEN TEKNIK ELEKTRO

FAKULTAS TEKNIK UNIVERSITAS INDONESIA

DEPOK

GENAP 2007/2008

OTOMATISASI SISTEM TRANSPORTASI *MONORAIL* BERBASIS TEKNOLOGI MULTI-AGEN

SKRIPSI

oleh:
JOHAN
04 04 03 0555

SKRIPSI INI DIAJUKAN UNTUK MELENGKAPI SEBAGIAN PERSYARATAN MENJADI SARJANA TEKNIK

DEPARTEMEN TEKNIK ELEKTRO

FAKULTAS TEKNIK UNIVERSITAS INDONESIA

DEPOK

GENAP 2007/2008

PERNYATAAN KEASLIAN SKRIPSI

Saya menyatakan dengan sesungguhnya bahwa skripsi dengan judul:

OTOMATISASI SISTEM TRANSPORTASI *MONORAIL* BERBASIS TEKNOLOGI MULTI-AGEN

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, 25 Juni 2008

(Johan)
04 04 03 0555

PENGESAHAN

Skripsi dengan judul:

OTOMATISASI SISTEM TRANSPORTASI *MONORAIL* BERBASIS TEKNOLOGI MULTI-AGEN

dibuat untuk melengkapi sebagian persyaratan menjadi Sarjana Teknik pada Program Studi Teknik Elektro Departemen Teknik Elektro Fakultas Teknik Universitas Indonesia dan disetujui untuk diajukan dalam sidang ujian skripsi.

Depok, 25 Juni 2008

Dosen Pembimbing,

F. Astha Ekadiyanto ST, M.Sc.

NIP. 132.166.489

UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada:

F. Astha Ekadiyanto ST, 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.

Johan
NPM 04 04 03 0555
Departemen Teknik Elektro

Dosen Pembimbing
F. Astha Ekadiyanto ST, M.Sc.

OTOMATISASI SISTEM TRANSPORTASI *MONORAIL* BERBASIS TEKNOLOGI MULTI-AGEN

ABSTRAK

Pemrograman berorientasi agen merupakan suatu paradigma baru dalam rekayasa perangkat lunak. Pendekatan ini memodelkan aplikasi sebagai kumpulan agen, yang diantaranya bersifat otonom, proaktif, dan mampu untuk berkomunikasi sehingga mampu membawa konsep dari teori kecerdasan buatan ke dalam bidang sistem terdistribusi. Agen bergerak yang menerapkan konsep pergerakan kode, data dan keadaan antar perangkat jaringan merupakan hasil dari kedua disiplin ilmu tersebut.

Dalam skripsi ini dilakukan pengembangan sebuah aplikasi untuk otomatisasi sistem transportasi *monorail* berbasis teknologi multi-agen. Aplikasi ini dikembangkan dengan menggunakan Java Agent DEvelopment Framework (JADE) dan MySQL. JADE merupakan salah satu *platform middleware* terbuka yang berorientasi pada agen dan dibangun dengan bahasa pemrograman Java. Sedangkan MySQL adalah sebuah sistem manajemen *database* relasional terbuka. Layanan diwujudkan oleh agen-agen yang saling berkomunikasi dan agen-agen yang bergerak. Agen-agen tersebut masing-masing merepresentasikan pusat akses data (agen *server*), tempat keluar masuk penumpang (agen stasiun), dan kendaraan (agen kereta). Suatu agen *server* dan beberapa agen stasiun dapat didistribusikan pada beberapa komputer dalam suatu jaringan lokal. Sementara agen-agen kereta bergerak antar *host* tersebut. Semua agen ini tetap terintegrasi dalam suatu sistem intra-*platform*.

Evaluasi kinerja sistem dilakukan dengan memperhatikan *delay* pewaktuan, lalu lintas data, penggunaan sistem memori dan waktu CPU. Hasil evaluasi menunjukkan bahwa *delay* tidak menjadi suatu isu yang penting, data sebesar 2400 – 22000 byte untuk sebuah komunikasi, konsumsi memori sebesar ±40 MB, dan penggunaan waktu CPU sampai dengan <40% saat sistem *idle.*

Kata Kunci: Agen, Kecerdasan Buatan, Sistem Terdistribusi, Agen Bergerak, JADE, MySQL, *Monorail*

Johan
NPM 04 04 03 0555
Electrical Engineering Department

Supervisor
F. Astha Ekadiyanto ST, M.Sc.

AUTOMATION OF MONORAIL TRANSPORTATION SYSTEM BASED ON MULTI-AGENT TECHNOLOGY

ABSTRACT

Agent oriented programming is a relatively new software paradigm. This approach models an application as a collection of agents that are characterized by, among other thing, autonomy, proactivity and an ability to communicate, with result that it is able to bring concepts from the theories of artificial intelligence into the mainstream realm of distributed systems. Mobile agents which apply the concept of code, state, and data mobility between networked machines, are results of these two disciplines.

In this work, an application for automation of monorail transportation system is built based on multi-agent technology. This application is developed using Java Agent DEvelopment Framework (JADE) and MySQL. JADE is one of many open source middleware platform that oriented on agent and built with Java language. MySQL is an open source relational database management system. The service is formed by communicating agents and mobile agents. Each agent is representing a main data access center (server agent), the place where passengers enter and exit (station agent) and the vehicle (train agent). A server agent and some station agents can be distributed on several computers in a local network. While the train agents move between these hosts. All the agent are still integrated in a intra-platform system.

Performance evaluation focused on the delay, data traffic, CPU time and system memory usage. Evaluation results show that delay is not an important issue, 2400 – 22000 bytes of data on a communication, memory usage is ±40 MB, and CPU time usage is up to <40% when the system is idle.

Keywords: Agent, Artificial Intelligence, Distributed System, Mobile Agent, JADE, MySQL, Monorail

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

DAFTAR SINGKATAN

ACL	Agent Communication Language
AID	Agent Identifier
AMS	Agent Management System
AOP	Agent Oriented Programming
CPU	Central Processing Unit
DF	Directory Facilitator
FIPA	Foundation for Intelligent Physical Agents
GUI	Graphical User Interface
HTTP	Hyper Text Transfer Protocol
IMTP	Internal Message Transport Protocol
J2ME	Java 2 Micro Edition
JADE	Java Agent DEvelopment Framework
JDBC	Java Database Connnectivity
JSP	Java Server Pages
MTP	Message Transport Protocol
MTS	Message Transport Service
RDBMS	Relational Database Management System
RMI	Remote Method Invocation
SQL	Structured Query Language
TCP	Transmission Control Protocol
UML	Unified Modeling Language

BAB 1

PENDAHULUAN

1.1 LATAR BELAKANG

Agent Oriented Programming (AOP) merupakan suatu paradigma yang relatif baru tentang perangkat lunak yang membawa konsep dari teori kecerdasan buatan ke dalam bidang sistem terdistribusi. AOP dapat dikatakan sebagai evolusi dari pemrograman berorientasi objek. AOP memodelkan sebuah aplikasi sebagai kumpulan komponen yang disebut agen yang memiliki sifat, diantaranya, otonom, proaktif, dan mampu untuk berkomunikasi. Bersifat otonom, agen secara bebas dapat mengerjakan tugas yang rumit dan kadang memakan waktu lama. Bersifat proaktif, agen dapat mengambil inisiatif untuk mengerjakan tugas yang diberikan tanpa stimulus eksplisit dari pengguna. Bersifat komunikatif, agen dapat berinteraksi dengan entitas lain untuk membantu mencapai tujuan. Model arsitektural dari agen secara intrinsik merupakan *peer to peer*, agen dapat memulai komunikasi dengan agen lain atau menjadi subjek dari komunikasi yang datang setiap saat.

JADE *(Java Agent DEvelopment Framework)* merupakan salah satu sistem *middleware* terdistribusi berbasis agen dengan infrastruktur yang secara fleksibel memperbolehkan perluasan menggunakan modul tambahan. Karena JADE ditulis dalam Java, terdapat keuntungan dari fungsi-fungsi set bahasanya yang lengkap dan pustaka dari pihak ketiga, sehingga menghasilkan set abstraksi pemrograman yang kaya dan memperkenankan pengembang untuk membangun sistem multi-agen JADE dengan relatif sedikit keahlian dalam teori agen.

MySQL merupakan sebuah *Relational Database Management System* (RDBMS) yang dimiliki dan disponsori oleh perusahaan MySQL AB yang merupakan bagian dari Sun Microsystems. Sebagai sebuah *Database Management System* (DBMS), MySQL mendukung fungsi-fungsi antara lain manajemen penyimpanan data, memelihara keamanan data, memelihara *metadata*,

menangani transaksi, mendukung bermacam-macam konektivitas, optimasi kerja, memelihara mekanisme *back-up* dan *recovery*, serta menangani pengambilan data dan modifikasi.

Sistem transportasi *monorail* merupakan salah satu alternatif sarana angkutan masal yang cukup populer di negara-negara tertentu. Sistem yang digunakan ada yang bersifat otomatis maupun semiotomatis. Dengan pendekatan teknologi multi-agen, otomatisasi sistem transportasi ini dapat dilakukan. Dengan adanya suatu aplikasi yang mampu mengakomodasi transaksi pembelian tiket oleh calon penumpang, mendeteksi jumlah penumpang secara *real time*, mendeteksi penumpang yang masuk dan keluar, serta mengendalikan mobilitas dari kereta sebagai sarana angkutan; efisiensi dan optimasi sumber daya serta peningkatan layanan dapat dicapai. Dengan pendekatan teknologi multi-agen dan penggunaan *database* secara tepat aplikasi tersebut dapat diwujudkan.

1.2 PERUMUSAN MASALAH

Masalah yang akan diselesaikan mencakup pembangunan aplikasi untuk otomatisasi sistem transportasi *monorail* yang dibuat dengan pendekatan teknologi multi-agen. Aplikasi dibuat sedemikian rupa sehingga dapat mengakomodasi transaksi pembelian tiket oleh calon penumpang, mendeteksi jumlah penumpang secara *real time*, mendeteksi penumpang yang masuk dan keluar, serta mengendalikan mobilitas dari kereta sebagai sarana angkutan. Setiap aktivitas penumpang tercatat dalam suatu *database* terintegrasi.

1.3 TUJUAN PENELITIAN

Tujuan skripsi ini adalah:

1. melakukan studi literatur dan studi kasus lebih lanjut mengenai teknologi multi agen dengan mengimplementasikan teknologi multi-agen pada suatu aplikasi,

2. membangun aplikasi untuk otomatisasi sistem transportasi *monorail,*

3. melakukan analisis dan evaluasi terhadap aplikasi yang telah dibangun.

1.4 BATASAN MASALAH

Pekerjaan yang dilakukan dalam skripsi ini dibatasi pada simulasi dari aplikasi yang telah dibangun dalam suatu jaringan komputer lokal. Aplikasi yang telah dibangun terbatas pada intra-*platform* (JADE). Sistem transportasi *monorail* yang disimulasikan bersifat sederhana, dalam arti hanya terdiri atas satu jalur tanpa *looping* dari agen kereta. Simulasi transaksi pembelian tiket oleh calon penumpang yang diakomodasi oleh agen stasiun mengasumsikan bahwa alat pembayaran adalah suatu *smart card*.

1.5 METODOLOGI PENELITIAN

Metodologi penelitian yang digunakan mengacu pada sebuah metodologi untuk analisis dan perancangan sistem multi-agen menggunakan JADE [3] yang disediakan dalam paket distribusi JADE.

1.6 SISTEMATIKA PENELITIAN

Penelitian yang dilakukan mulai dari studi literatur, studi kasus yaitu pembangunan program aplikasi, dan berpuncak pada evaluasi hasil kerja.

1.7 SISTEMATIKA PENULISAN

Skripsi ini dibagi dalam beberapa bab, masing-masing membahas pokok bahasan tertentu dalam integral tujuan akhir skripsi ini.

Bab pertama, Pendahuluan, berisi latar belakang, perumusan masalah, tujuan penelitian, batasan masalah, metodologi penelitian, sistematika penelitian,

dan sistematika penulisan. Bab kedua menggambarkan landasan pengetahuan tentang istilah-istilah yang digunakan dalam skripsi ini. Bab ketiga menggambarkan langkah pengerjaan skripsi atau perancangan sistem. Bab keempat berisi implementasi dan pengujian serta pembahasan mengenai evaluasi kerja dari berbagai aspek yang menjadi fokus pengukuran. Bab kelima berisi kesimpulan dari skripsi ini.

BAB 2

TEKNOLOGI MULTI-AGEN, JADE, DAN MYSQL

2.1 TEKNOLOGI MULTI-AGEN

2.1.1 Konsep Sistem Agen dan Multi-agen

Istilah "agen" dapat memiliki banyak arti karena istilah ini telah digunakan pada berbagai bidang, misalnya kecerdasan buatan, *database*, sistem operasi dan literatur jaringan. Meskipun demikian, pengamatan lebih dalam mengidentifikasi dua penggunaan istilah ini: gagasan lemah dan gagasan kuat tentang agen. Gagasan lemah tentang agen disetujui oleh kebanyakan peneliti, sedangkan gagasan kuat tentang agen lebih kontroversial dan masih menjadi subjek penelitian.

Gagasan lemah tentang agen menyatakan sebuah sistem komputer berbasis perangkat lunak dengan sifat-sifat sebagai berikut:

- *Autonomy*: agen beroperasi tanpa intervensi langsung dari manusia atau yang lain, dan memiliki semacam kendali terhadap keadaan dan tindakannya;
- *Social Ability*: agen berinteraksi dengan agen lain (mungkin juga manusia) melalui semacam bahasa komunikasi agen;
- *Reactivity*: agen merasakan lingkungannya dan menanggapi seiring waktu perubahan yang terjadi pada lingkungan tersebut;
- *Pro-activeness*: selain bertindak menanggapi lingkungannya, agen mampu mengambil inisiatif untuk bertindak guna mencapai tujuannya.

Gagasan kuat tentang agen merupakan perluasan dari gagasan lemah, selain itu memiliki sifat-sifat yang menyerupai sifat-sifat manusia, seperti kepercayaan, keinginan, dan tujuan.

Walaupun tidak ada definisi tunggal terhadap agen, semua definisi yang ada setuju bahwa agen pada dasarnya merupakan sebuah komponen perangkat

lunak khusus yang memiliki otonomi menyediakan antarmuka yang dapat beroperasi terhadap sistem yang berubah-ubah dan/atau bertindak seperti agen manusia, bekerja untuk beberapa klien dengan mencari agendanya sendiri. Walaupun sebuah sistem agen dapat merupakan satuan agen yang bekerja dalam suatu lingkungan, namun pada umumnya terdiri dari beberapa agen. Sistem multi-agen ini dapat memodelkan sistem yang kompleks dan memungkinkan agen memiliki kesamaan maupun konflik tujuan. Agen-agen ini dapat saling berinteraksi secara langsung (bertindak terhadap lingkungan) maupun tidak langsung (melalui komunikasi dan negosiasi). Agen dapat memutuskan untuk bekerja sama mencapai hasil yang saling menguntungkan atau berkompetisi untuk mencapai tujuannya sendiri.

Konsisten dengan gagasan lemah, dalam tulisan ini diasumsikan definisi agen adalah:

agen tinggal dalam suatu platform, yang menyediakan suatu mekanisme tertentu untuk agen tersebut berkomunikasi dengan menggunakan nama, seberapa kompleks maupun mendasarnya lingkungannya (seperti sistem operasi, jaringan, dan lain-lain).

Jadi setiap agen memiliki nama yang unik untuk identifikasi. Sistem multi-agen adalah sistem yang terdiri dari bermacam-macam agen yang saling berinteraksi.

2.1.2 Agen dan Objek

Para pemrogram yang sudah terbiasa dengan bahasa berorientasi objek seperti Java, C++, atau Smalltalk kadang gagal melihat sesuatu yang baru dalam gagasan agen. Saat seseorang tidak mampu menyadari properti relatif dari agen dan objek, hal ini tidaklah mengherankan. Objek didefinisikan sebagai entitas komputasional yang mengenkapsulasi beberapa keadaan, dan mampu melakukan aksi, atau *methods* pada kondisi ini, dan berkomunikasi dengan mengirimkan pesan. Berikut ini, perbedaan antara agen dan objek akan dirangkum.

Walaupun terdapat persamaan-persamaan, terdapat pula perbedaan yang signifikan antara agen dan objek. Pertama adalah derajat di mana agen dan objek bersifat otonom. Pendefinisian karakteristik pada pemrogramanan berorientasi objek merupakan prinsip dari enkapsulasi-ide bahwa objek dapat mengendalikan keadaan internalnya sendiri. Dalam bahasa pemrograman seperti Java, kita dapat mendeklarasikan *variable* (dan *method*) menjadi *private,* yang berarti mereka hanya dapat diakses dari dalam objek. Kita dapat pula mendeklarasikan mereka *public,* yang berarti mereka dapat diakses dari manapun, dan demikianlah kita harus melakukan ini untuk *method* agar mereka dapat digunakan oleh objek lain. Namun penggunaan *public instance variable* umumnya dianggap sebagai gaya pemrograman yang buruk. Dengan cara ini, satu objek dapat dianggap menunjukkan otonomitas terhadap keadaannya: objek dapat mengendalikan keadaannya. Namun objek tidak menunjukkan kendali terhadap *behaviour*-nya. Jika *method* m dibuat tersedia bagi objek lain untuk di panggil, maka dapat dilakukan kapanpun; objek tidak memiliki kendali apakah *method* harus dieksekusi atau tidak. Tentunya, objek harus membuat *method* tersedia bagi objek lain, jika tidak kita tidak akan bisa membangun sistem. Ini normalnya bukanlah isu, karena jika kita membangun sebuah sistem, lalu merancang objek untuk itu, mereka diasumsikan memiliki "tujuan umum". Namun, dalam sistem multi agen, tidak ada tujuan umum yang dapat diasumsikan. Tidak dapat disetujui bahwa agen i akan mengeksekusi aksi a hanya karena agen lain j menginginkan a, yang mungkin bukan yang terbaik bagi i. Kita tidak berpikir bahwa agen saling meng-*invoke method* satu sama lain, melainkan me-*request* aksi untuk dilakukan. Jika j me-*request* i untuk melakukan a, i dapat melakukan aksi tersebut ataupun tidak. Lokasi kendali dalam hal keputusan untuk mengeksekusi aksi atau tidak berbeda dalam sistem agen dan objek. Pada kasus berorientasi objek, keputusan terletak pada objek yang meng-*invoke method*. Pada kasus agen, keputusan terletak pada agen yang menerima *request.*

Perbedaan penting kedua antara sistem objek dan agen adalah dalam hal gagasan *flexible (reactive, pro-active, social) autonomous behaviour*. Model objek standar tidak memiliki apapun tentang bagaimana membangun sistem yang mengintegrasikan tipe-tipe *behaviour* ini. Seseorang dapat yakin bahwa kita dapat

membangun program berorientasi objek yang mengintegrasikan tipe-tipe *behaviour* ini. Namun argumen ini melewatkan intinya, bahwa model pemrograman berorientasi objek tidak memiliki apapun yang harus dilakukan terhadap tipe-tipe *behaviour* ini.

Perbedaan penting ketiga antara model objek standar dan cara pandang sistem agen adalah bahwa setiap agen dianggap memiiki *thread of control*-nya sendiri - pada model objek standar, terdapat *thread control* tunggal dalam sistem. Tentunya, banyak usaha yang dilakukan untuk *concurrency* dalam pemrograman berorientasi objek. Sebagai contoh, bahasa Java menyediakan *built-in constructs* untuk pemrograman *multi-thread*. Terdapat banyak bahasa pemrograman tersedia yang secara spesifik dirancang untuk memperbolehkan pemrograman berorientasi objek yang *concurent.* Namun bahasa-bahasa tersebut tidak menangkap ide agen sebagai entitas *autonomous*. Bagaimanapun, perlu dicatat bahwa *active objects* mendekati konsep *autonomous agents* – walaupun mereka bukanlah agen yang mampu melakukan *flexible autonomous behaviour*.

Sebagai tambahan, terdapat dua hal yang secara kualitatif membedakan interaksi agen dari interaksi yang muncul dalam paradigma teknik perangkat lunak seperti paradigma berorientasi objek. Pertama, interaksi berorientsi agen secara umum muncul melalui suatu bahasa komunikasi agen tingkat tinggi. Konsekuensinya, interaksi umumnya dihubungkan pada tingkat pengetahuan; dalam hal tujuan mana yang harus diikuti, pada waktu kapan, dan oleh siapa. Kedua, karena agen adalah pemecah masalah yang fleksibel, beroperasi dalam lingkungan di mana mereka hanya memiliki kendali dan kemampuan observasi parsial, interaksi-interaksi perlu ditangani dengan cara fleksibel yang serupa. Kemudian, agen memerlukan perangkat komputasional untuk membuat keputusan yang *context-dependent* tentang sifat dasar dan lingkup dari interaksi mereka dan untuk menginisiasi (dan merespon kepada) interaksi yang tidak perlu diramalkan pada waktu merancang.

Sebagai rangkuman, pandangan tradisional terhadap objek dan pandangan kita terhadap agen paling tidak memiliki tiga perbedaan, yaitu:

- agen memiliki gagasan yang lebih kuat terhadap otonomitas dari pada objek, dan secara khusus, mereka memutuskan untuk mereka sendiri apakah harus melakukan aksi atau tidak terhadap *request* dari agen lain;
- agen mampu melakukan *flexible (reactive, pro-active, social) behavior* dan model objek standar tidak memiliki apapun tentang tipe-tipe *behavior* tersebut; dan
- sebuah sistem multi agen pada dasarnya *multi-threaded,* bahwa setiap agen diasumsikan paling tidak memiliki satu *thread* kendali.

2.1.3 Agen Bergerak

Agen bergerak adalah sebuah paradigma yang diperoleh dari dua disiplin ilmu yang berbeda. Yang pertama adalah kecerdasan buatan, menghasilkan konsep dari agen, dan yang kedua adalah sistem terdistribusi, menetapkan konsep pergerakan kode. Berdasarkan definisi umum, agen bergerak adalah semua agen tak bergerak (seperti *autonomous, reactive, proactive* dan *social*), ditambah kemampuan untuk berpindah; agen tersebut dapat berpindah antar-*platform* untuk melaksanakan tugasnya.

Dari sudut pandang sistem terdistribusi, agen bergerak adalah sebuah program dengan identitas unik yang dapat memindahkan kode, data dan keadaannya antar perangkat jaringan. Untuk mencapai hal ini, agen bergerak dapat menghentikan aktivitasnya setiap saat dan melanjutkannya kembali saat berada di lokasi lain.

Agen bergerak terdiri dari tiga bagian: kode, keadaan, dan data. Kode adalah aspek dari agen yang dieksekusi saat berpindah ke sebuah *platform*. Keadaan adalah lingkungan eksekusi data dari agen, termasuk *counter* program dan *stack* eksekusi. Data terdiri dari variabel-variabel yang digunakan oleh agen, seperti pengetahuan, *file identifier*, dan lain-lain.

Beberapa keuntungan dari agen bergerak adalah sebagai berikut.

1. *Asynchronous and Independent Processing*

   Sekali berpindah ke *platform* baru, agen tidak memiliki hubungan dengan pemiliknya untuk melakukan tugasnya. Agen hanya perlu mengirimkan kembali hasilnya.
2. *Fault Tolerance*

   Agen dapat mengalamati dan berguna saat kondisi buruk dengan bergerak ke *platform* alternatif saat masalah terdeteksi. Sama halnya saat tempat tujuan bermasalah, perantara dapat dipilih sebagai *host* sementara.
3. *Sea of Data Applications*

   Agen bergerak sangat cocok untuk aplikasi yang perlu memproses data dari jauh dalam jumlah yang besar. Agen bergerak dapat berpindah ke data, dibandingkan sebaliknya, yang dalam beberapa kasus merupakan pilihan yang lebih efisien.

Agen bergerak juga memiliki beberapa kekurangan, diantaranya adalah sebagai berikut.

1. *Scalability and Performance*

   Walaupun agen bergerak mengurangi beban jaringan, agen bergerak juga dapat menambah beban proses. Hal ini dikarenakan agen tersebut umumnya diprogram dengan bahasa interpretasi dan juga sering perlu untuk mengamati standar *interoperability* yang teliti sehingga menimbulkan tambahan proses data.
2. *Portability and Standardization*

   Agen tidak dapat saling beroperasi jika tidak mengikuti standar komunikasi yang umum. Penggunaan standar-standar ini, seperti OMG MASIF *(Mobile Agent System Interoperability Facility)* atau FIPA *(Foundation for Intelligent Physical Agents)* diperlukan, khususnya untuk pergerakan inter-*platform*.

3. *Security*

   Penggunaan agen bergerak dapat menimbulkan masalah keamanan. Kode bergerak memiliki potensi ancaman dan harus diautentifikasi dengan hati-hati sebelum dibaca.

Dalam sistem agen bergerak terdapat dua jenis utama perpindahan: perpindahan kuat dan perpindahan lemah. Perpindahan kuat lebih rumit dan dalam kasus dimana eksekusi dari agen dihentikan, terjadi perpindahan, dan eksekusi dimulai kembali dari instruksi selanjutnya. Teknik ini memerlukan penyimpanan dan perlindungan terhadap keadaan agen saat proses perpindahan. Implemenasi dari teknik ini dapat menjadi rumit karena memerlukan akses ke parameter internal dari eksekusi agen, umumnya hanya tersedia bagi sistem operasi, dan dapat sangat tergantung pada arsitektur. Sebaliknya, perpindahan lemah tidak mengirim keadaan agen dan lebih sederhana; eksekusi agen selalu dimulai kembali dari awal kode. Perpindahan jenis ini memerlukan agen diimplementasikan sebagai mesin keadaan terbatas jika keadaan harus dijaga.

Sebuah *itinerary* menentukan lokasi yang harus disinggahi agen bergerak untuk menyelesaikan sekumpulan tugasnya. Terdapat dua jenis dasar *itinerary*:

- *Static Itineraries,* ditentukan pada waktu pembuatan agen tanpa kemungkinan modifikasi saat eksekusi agen;
- *Dynamic Itineraries,* ditentukan saat eksekusi agen berdasarkan tujuan dan keinginan.

Metode hibrid merupakan jenis tambahan dapat menggabungkan kedua jenis ini.

2.1.4 *Agent Platform*

Teknologi multi-agen telah banyak dikembangkan untuk berbagai domain berbeda dalam sektor bisnis dan riset. Pada umumnya, agen dikembangkan diimplementasikan dalam bahasa pemrograman tingkat tinggi (C++ atau Java). Namun, terdapat banyak alasan untuk memilih bahasa Java. salah satunya adalah

portabilitas dari program Java di mana pengguna dapat menggunakan pada *hardware platforms* atau sistem operasi secara bebas. Aplikasi yang sama dapat berjalan baik pada PC dengan Microsoft Windows atau Unix/Linux atau pada perangkat kecil seperti *personal digital assistant* (PDA) atau telepon genggam dengan Windows CE, Symbian atau sistem operasi lain yang mendukung Java.

Pada dasarnya, perangkat pengembangan agen atau umum disebut dengan *agent platform,* menyediakan kumpulan pustaka Java untuk spesifikasi kelas-kelas agen penggunanya dengan atribut dan *behavior* spesifik. Sejenis lingkungan *runtime* yang disediakan oleh *agent platform* lalu digunakan untuk menjalankan aplikasi agen. Lingkungan *runtime* ini diaplikasikan pada Java juga, secara khusus menjamin transportasi pesan di antara agen, registrasi dan deregistrasi dari agen dalam komunitas *(white pages services)* juga registrasi dan pencarian layanan yang disediakan agen-agen itu sendiri *(yellow pages services)*. Beberapa perangkat pilihan lain dapat juga menjadi bagian dari *agent platform runtime*, sebagai contoh *graphical viewer* untuk pesan yang dikirimkan diantara agen, dan lain-lain.

Tabel 2.1 memberikan gambaran sebagian besar perangkat pengembangan agen yang saat ini tersedia [6]. Atribut *security* telah ditambahkan sebagai suatu properti dari *agent platform* untuk memastikan komunikasi yang aman (umumnya melalui SSL), otorisasi, otentifikasi, dan lain-lain. Tanda √ memunjukkan bahwa agent platform memiliki properti tertentu sedangkan tanda × memunjukkan sebaliknya. Tanda ? digunakan jika tidak terdapat referensi terhadap properti tersebut pada sumber yang tersedia dan dapat diasumsikan, bahwa *platform* tersebut tidak memilikinya.

Tabel 2.1 Berbagai macam *agent platform* [6]

JAVA-based Agent Development Toolkits/Platforms - Overview							
Agent platform	Developer	FIPA compatibility			Open-source	J2ME version (lightweight)	Security (authent, SSL, ...)
		Agent Communication (ACL, protocols, ...)	Agent Management (AMS, Df, AOC)	Inter-platform Messaging (MTP)			
JADE (Java Agent DEvelopment Framework)	CSELT http://jade.cselt.it/	√	√	√	√	√	√
FIPA OS	Emorphia http://fipa-os.sourceforge.net/	√	√	√	√	√	√
ZEUS	British Telecom http://labs.bt.com/projects/agents/zeus/	√	√	√	√	×	√
JACK (Jack Intelligent Agents)	Agent Oriented Software http://www.agent-software.com/	√	√	√	×	×	√
GRASSHOPPER 2	IKV++ Technologies AG http://www.grasshopper.de/	√	√	√	×	√	√
ADK (Agent Development Kit)	Tryllian http://www.tryllian.com/	√	×	×	×	?	√
JAS (Java Agent Services API)	Fujitsu, HP, IBM, SUN, ... http://java-agent.org/	√	√	√	√	×	?
AgentBuilder	IntelliOne Technologies http://www.agentbuilder.com/	×	×	×	×	×	?
MacKit (Multi-Agent Development Kit)	MacKit Team http://www.madkit.org/	×	×	×	√	×	?
Comtec Agent Platform	Communication Tecnologies http://www.agentland.com/Download/Intelligent_Agent/Comtec%20Agent%20Platform-500.html	√	√	√	√	×	×
Bee-gent	Toshiba http://www2.toshiba.co.jp/rdc/beegent/index.htm	×	×	×	×	×	√
Aglets	IBM Japan http://www.trl.ibm.com/aglets/	×	×	×	√	×	√

2.1.5 Aplikasi Sistem Agen dan Multi-agen

Teknologi agen adalah perluasan natural dari pendekatan berbasis komponen, dan memiliki potensi memberikan dampak yang besar terhadap hidup dan aktivitas kita, ini merupakan salah satu bidang yang sangat dinamis dan menarik dalam ilmu komputer saat ini. Beberapa domain aplikasi di mana teknologi agen akan memainkan peran yang penting mencakup: *Ambient Intelligence*, penyampaian *seamless* dari komputasi *ubiquitous*, komunikasi berkelanjutan, dan antarmuka pengguna yang cerdas pada perangkat industri dan konsumen; *Grid Computing*, dimana pendekatan sistem multi-agen akan mengefisiensikan penggunaan sumber daya dari infrastruktur komputasi berperforma tinggi pada aplikasi keilmuan, teknik, medikal dan komersial; *Electronic Business*, di mana pendekatan berbasis agen telah mendukung automasi dan semi-automasi dari aktivitas pengumpulan informasi dan transaksi pembayaran melalui Internet; *the Semantic Web*, di mana agen diperlukan untuk menyediakan layanan, dan mengoptimalkan penggunaan sumber daya yang tersedia, kadang bekerja sama dengan agen lain; *Bioinformatics and Computational Biology,* di mana agen cerdas dapat mendukung eksplorasi koheren dari revolusi data yang muncul dalam biologi; dan bidang lain yang mencakup pengawasan dan kendali, manajemen sumber daya, dan ruang, aplikasi militer dan manufaktur, untuk contohnya.

2.2 JADE

JADE adalah sebuah *platform* perangkat lunak yang menyediakan fungsi-fungsi *middleware-layer* dasar yang tidak terikat pada aplikasi tertentu dan menyederhanakan realisasi dari aplikasi terdistribusi yang memanfaatkan abstraksi agen perangkat lunak (Wooldridge and Jennings, 1995). Keuntungan utama dari JADE adalah mengimplementasikan abstraksi ini pada bahasa pemrograman berorientasi objek, Java, yang menyediakan API yang sederhana dan mudah dimengerti. Abstraksi agen mempengaruhi beberapa pilihan rancangan sederhana sebagai berikut:

- sebuah agen adalah anonim dan proaktif,
- agen dapat mengatakan „Tidak' dan tidak terikat,
- sistemnya adalah *Peer-to-Peer*.

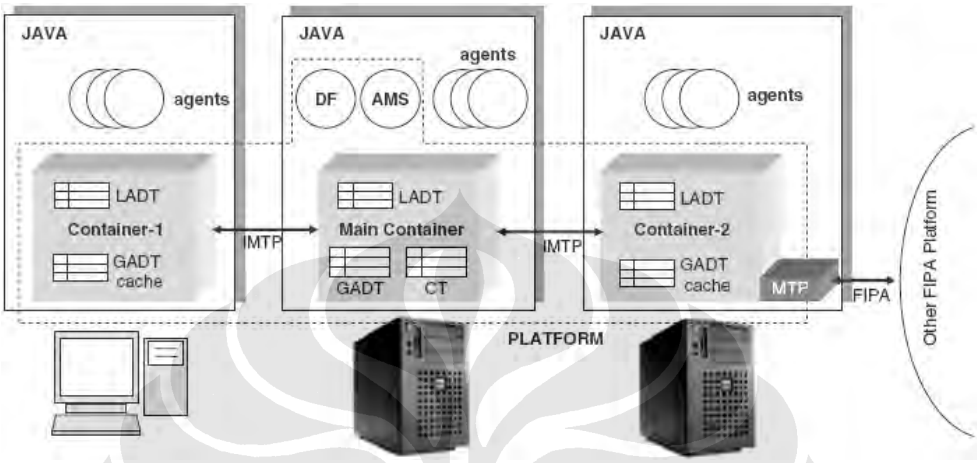
Berdasarkan ketiga pilihan rancangan tersebut, JADE diimplementasikan untuk menyediakan kepada pemrogram fungsi-fungsi utama yang siap digunakan dan mudah dikostumasi sebagai berikut:

- sepenuhnya terdistribusi,
- sesuai dengan spesifikasi FIPA,
- penyampaian pesan asinkron efisien,
- implementasi *white pages* dan *yellow pages,*
- manajemen siklus hidup agen yang sederhana namun efektif,
- mendukung pergerakan agen,
- mendukung mekanisme pendaftaran,
- perangkat grafik,
- mendukung bahasa ontologi dan konten,
- pustaka protokol interaksi,
- integrasi bermacam teknologi berbasis *web,*
- mendukung *platform* J2ME dan lingkungan nirkabel,
- antarmuka *in-process* untuk menjalankan atau mengendalikan *platform* dan komponen terdistribusinya dari aplikasi eksternal,
- kernel yang dapat diperluas dirancang agar pemrogram dapat memperluas fungsi-fungsi *platform* melalui penambahan layanan terdistribusi pada tingkat kernel.

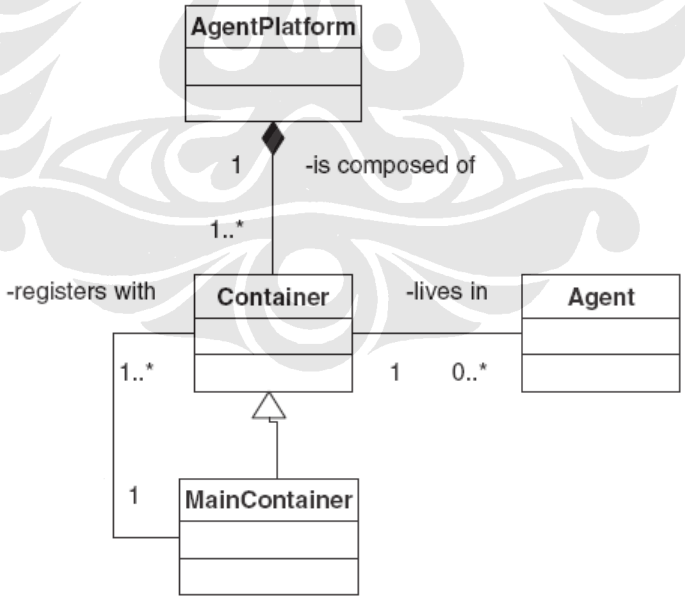
2.2.1 Arsitektur JADE

Gambar 2.1 menunjukkan elemen arsitektural utama dari *platform* JADE. JADE terdiri dari kontainer-kontainer agen yang dapat didistribusikan dalam jaringan [2]. Agen hidup dalam kontainer yang merupakan proses Java yang menyediakan JADE *run-time* dan seluruh layanan yang diperlukan untuk *hosting* dan eksekusi agen. Terdapat kontainer khusus, disebut *main container*, yang merepresentasikan titik utama dari *platform*, merupakan kontainer pertama yang

dijalankan dan kontainer-kontainer lain harus bergabung ke *main container* dengan mendaftar. Diagram UML pada Gambar 2.2 menunjukkan hubungan antarelemen arsitektural utama dari JADE [2].



Gambar 2.1 Hubungan antarelemen arsitektural utama dari JADE [2]



Gambar 2.2 Diagram UML hubungan antarelemen arsitektural utama dari JADE [2]

Sebagai titik utama, *main container* memiliki beberapa tanggung jawab khusus, yaitu:

- mengatur *Container Table* (CT), yaitu registri dari referensi objek dan alamat *transport* dari semua titik kontainer yang menyusun *platform*;
- mengatur *Global Agent Descriptor Table* (GADT), yaitu registri dari semua agen yang terdapat dalam *platform*, termasuk kondisi dan lokasinya sekarang;
- memuat AMS dan DF, dua agen khusus yang masing-masing menyediakan manajemen agen dan layanan *white pages*, dan layanan *yellow page* awal dari *platform.*

Yang menjadi pertanyaan umum adalah bagaimana jika *main container* mengalami *system bottleneck*. Pada kenyataannya hal ini tidak menjadi masalah karena JADE menyediakan *cache* GADT yang diatur secara lokal oleh tiap kontainer. Operasi *platform* umumnya tidak melibatkan *main container*, namun hanya *cache* lokal dan dua kontainer yang memuat agen yang merupakan subjek dan objek dari operasi. Ketika sebuah kontainer harus mencari dimana penerima pesan berada, pertama-tama ia mencari pada LADT (*Local Agent Descriptor Table*) kemudian, jika pencarian gagal, *main container* dihubungi untuk memperoleh referensi yang sesuai, yang ditempatkan pada *cache* lokal untuk penggunaan di masa depan. Karena sistem bersifat dinamik (agen dapat bermigrasi, mati, atau agen lain dapat muncul), kadang-kadang sistem menggunakan nilai *cache* lama sehingga alamatnya tidak valid. Untuk itu, kontainer menerima sebuah *exception* dan dipaksa untuk me-*refresh cache*-nya terhadap *main container*. Aturan penggantian *cache* adalah LRU (*Least Recently Used*), yang dirancang untuk mengoptimalkan percakapan yang panjang dibandingkan sporadis, percakapan tukar pesan tunggal yang tidak cukup umum dalam aplikasi multi-agen.

Identitas agen dimuat dalam *Agen Identifier* (AID), terdiri dari kumpulan slot yang mengikuti struktur dan semantik yang didefinisikan oleh FIPA. Elemen paling dasar dari AID adalah nama agen dan alamatnya. Nama dari sebuah agen adalah pengidentifikasi unik global yang dibuat oleh JADE dengan merangkaikan

nama lokal terhadap nama *platform*. Alamat agen adalah alamat *transport* yang diberikan oleh *platform*, dimana setiap alamat *platform* sesuai dengan MTP *(Message Transport Protocol)*. Pemrogram agen juga diperbolehkan menambah alamat transportnya sendiri ke AID ketika, untuk alasan spesifik aplikasi, ia ingin menggunakan MTP privat agennya sendiri.

Saat *main container* dijalankan, dua agen khusus secara otomatis dijalankan oleh JADE, yang fungsinya didefinisikan oleh standar FIPA *Agent Management,* yaitu:

1. *Agent Management System* (AMS) adalah agen yang mengawasi seluruh *platform*. AMS merupakan titik hubung semua agen untuk berinteraksi guna mengakses *white pages* dari *platform* juga mengatur siklus hidupnya. Setiap agen perlu mendaftar ke AMS (otomatis oleh JADE saat mulainya agen) untuk memperoleh AID valid.
2. *Directory Facilitator* (DF) adalah agen yang mengimplementasikan layanan *yellow pages*, yang digunakan setiap agen yang ingin mendaftarkan layanannya atau mencari layanan lain yang tersedia. JADE DF juga menerima pendaftaran dari agen yang ingin diberitahukan saat sebuah pendaftaran atau modifikasi layanan dibuat sesuai beberapa kriteria spesifik. Beberapa DF dapat dijalankan bersamaan untuk mendistribusikan *yellow pages* pada beberapa domain. DF ini dapat difederasikan, jika perlu, dengan membangun *cross-registrations* satu sama lain yang memperbolehkan propagasi permintaan agen dalam suatu federasi.

2.2.2 Paket-Paket JADE

Sumber/komponen dari *platform* JADE diorganisir ke dalam hierarki paket dan sub-paket Java, dimana setiap paket, mengandung kumpulan kelas dan antarmuka yang mengimplementasikan fungsi khusus. Paket-paket JADE yang utama adalah sebagai berikut.

1. `jade.core` mengimplementasikan kernel dari JADE, lingkungan *run-time* terdistribusi yang mendukung seluruh *platform* dan perangkatnya. Ini

mencakup kelas `jade.core.Agent` dan kelas-kelas *run-time* dasar yang diperlukan untuk mengimplementasikan kontainer agen, yaitu:

- `jade.core.event` mengimplementasikan layanan notifikasi kejadian terdistribusi;
- `jade.core.management` mengimplementasikan layanan manajemen siklus hidup agen terdistribusi;
- `jade.core.messaging` mengimplementasikan layanan distribusi pesan;
- `jade.core.mobility` mengimplementasikan mobilitas agen dan layanan kloning, termasuk transfer keadaan dan kode dari agen;
- `jade.core.nodeMonitoring` memungkinkan kontainer untuk saling mengawasi dan menemukan container yang tak tercapai atau mati;
- `jade.core.replication` memungkinkan replikasi *main container* sebagai pilihan *failover* saat terjadi kesalahan serius pada *main container* yang asli.

2. `jade.content` dan sub-paketnya berisi kumpulan kelas yang mendukung pemrogram untuk membuat dan memanipulasi ekspresi konten kompleks berdasarkan ontologi dan bahasa konten yang diberikan.
3. `jade.domain` berisi implementasi agen AMS dan DF, sesuai spesifikasi standar FIPA, ditambah perluasan spesifik JADE. Setiap sub-paketnya mengandung kelas yang merepresentasikan entitas yang berbeda dari ontologi JADE. Ontologi-ontologi ini ditunjukkan oleh Tabel 2.2.
4. `jade.gui` berisi beberapa ikon dan komponen Java kegunaan umum yang berguna untuk membangun GUI berbasis Swing untuk agen JADE.
5. `jade.imtp` berisi imlementasi JADE IMTP *(Internal Message Transport Protocol).*
6. `jade.lang.acl` berisi dukungan untuk FIPA *Agent Communication Language* (ACL) termasuk kelas ACLMessage, *parser, encoder*, dan kelas pembantu unuk merepresentasikan pesan ACL.
7. `jade.mtp` berisi kumpulan antarmuka Java yang harus diimplementasikan oleh JADE MTP.

8. `jade.proto` berisi implementasi sejumlah protokol kegunaan umum, termasuk yang didefinsikan oleh FIPA.
9. `jade.tools` berisi implementasi semua perangkat grafik JADE.
10. `jade.util` berisi beberapa kelas utilitas tambahan.
11. `jade.wrapper` bersama dengan kelas `jade.core.Profile` dan `jade.core Runtime` menyediakan dukungan untuk antarmuka JADE *in-process* yang memperbolehkan aplikasi Java eksternal menggunakan JADE sebagai pustaka.
12. FIPA adalah sebuah paket yang berisi modul IDL *(Interface Definition Language)* yang dispesifikasikan FIPA untuk MTP berbasis IIOP.

Tabel 2.2 Ontologi JADE yang telah didefinisikan [2]

Ontology	Package	Description
`FIPA-Agent-Management`	`jade.domain.FIPAAgent Management`	Entities, exceptions and actions needed to interact with the AMS and the DF, according to the FIPA specs
`JADE-Agent-Management`	`jade.domain.JADEAgent Management`	JADE extensions to the FIPA-agent-management ontology
`JADE-Introspection`	`jade.domain. introspection`	JADE extensions related to the monitoring of platform events
`JADE-Mobility`	`jade.domain.mobility`	JADE extensions related to agent mobility
`JADE-Persistence`	`jade.domain.persistence`	JADE extensions related to agent persistence
`DFApplet-Management`	`jade.domain.DFGUI Management`	Ontology used by the DF GUI to interact with the DF. It allows multiple GUIs of the same DF, including GUIs implemented as applets

2.2.3 *Message Transport Service*

Berdasarkan spesifikasi FIPA, sebuah *Message Transport Service* (MTS) merupakan salah satu dari tiga layanan yang paling penting yang dibutuhkan *agent platform*. Sebuah MTS mengatur semua pertukaran pesan di dalam dan antar-*platform*.

2.2.3.1 Message Transport Protocol (MTP)

Untuk dapat saling berkomunikasi antar-*platform*, JADE menerapkan semua standar protokol-protokol pengiriman pesan (*Message Transport Protocol* atau MTP) yang didefenisikan oleh FIPA, dimana setiap MTP memasukan definisi dari *transport protocol* dan standar *encoding* dari amplop pesan. Saat ini beberapa MTP yang ditunjukkan oleh Tabel 2.3 tersedia pada domain publik, dan masing-masing dikembangkan sebagai suatu *file* jar terpisah.

Tabel 2.3 JADE MTP yang terdapat di domain publik [2]

Transport Protokol	*Encoding* Pesan	Pengembang
HTTP dan HTTPS	XML	Universitat Autonoma o Barcelona (UAB), Spanyol
IIOP (Sun Implementation)	COBRA IDL	Tim JADE
IIOP (ORBacus Implementation)	COBRA IDL	Geovanni Rimassa, Universita di Parma, Italy
JMS	Java data structure	Edward Curry, University of Galway
Jabber XMPP	Java data structure	Universitat Politecnica of Valencia, Spanyol

2.2.3.2 Internal Message Transport Protocol (IMTP)

JADE IMTP (*Internal Message Transport Protocol*), digunakan secara eksklusif untuk pertukaran pesan antara agen-agen yang hidup di dalam kontainer yang berbeda namun dalam *platform* yang sama. IMTP dianggap berbeda dengan *inter-platform* MTPS, seperti HTTP. IMTP hanya digunakan untuk komunikasi internal *platform*, sehingga tidak membutuhkan kompatibilitas dengan FIPA.

JADE dirancang untuk mengizinkan pemilihan IMTP pada waktu peluncuran. Untuk saat ini, terdapat dua implementasi IMTP yang tersedia. Yang pertama didasarkan pada Java RMI, dan merupakan pilihan *default*. Yang kedua didasarkan pada protokol yang menggunakan TCP *socket* yang menghindari ketidaktersediaan Java RMI pada lingkungan J2ME.

2.2.4 Pembuatan Agen

Membuat agen pada JADE dilakukan dengan mendefinisikan sebuah kelas yang meng-*extend* kelas `jade.core.Agent`, dan mengimplementasikan `setup()` *method*, seperti yang ditunjukkan pada kode di bawah ini:

```
import jade.core.Agent;

public class HelloWorldAgent extends Agent {
      protected void setup() {
            System.out.println("Hello World");
      }
}
```

Lebih tepatnya, sebuah kelas seperti HelloWorldAgent *class* di atas, menyatakan suatu jenis agen sama seperti Java *class* pada umumnya menyatakan suatu jenis objek. Tetapi tidak seperti objek Java pada umumnya, yang diatur oleh referensi mereka, sebuah agen selalu diadakan oleh JADE *run-time* dan referensinya tidak pernah berada di luar dari agen itu sendiri (kecuali agen tersebut melakukannya secara eksplisit). Agen tidak pernah berinteraksi melalui pemanggilan metode, tetapi dengan bertukar pesan (*asynchronous message*).

Bagian `setup()` *method* ditujukan untuk menyertakan inisialisasi agen. Pekerjaan sebenarnya dari agen dinyatakan dalam *behaviours*. Contoh operasi yang dinyatakan dalam `setup()` *method* antara lain: menampilkan GUI, membuka koneksi dengan *database*, mendaftarkan layanan ke *yellow pages*, dan menjalankan *behaviour*.

2.2.4.1 Tanda Pengenal Agen

Konsisten dengan spesifikasi FIPA, setiap agen diidentifikasi dengan menggunakan *agent identifier*. Pada JADE, *agent identifier* dinyatakan sebagai *instance* dari `jade.core.AID` class. Sebuah objek AID menyertakan sebuah GUID *(Globally Unique ID)* dan sejumlah alamat. Penamaan dalam JADE memiliki format *<local-name>@<platform-name>*. Misalkan sebuah agen

bernama “johan” dan berada di *platform* yang bernama “Elektro”, maka agen tersebut memiliki nama johan@Elektro sebagai GUID. Alamat-alamat yang disertakan dalam AID adalah alamat dari *platform* yang ditempati oleh agen. Alamat-alamat ini hanya digunakan jika agen ingin berkomunikasi dengan agen yang berada di *platform* yang berbeda.

Nama lokal dari sebuah agen ditentukan saat *start-up* oleh pembuat agen, dan harus unik di dalam *platform* yang ditempatinya. Jika sebuah agen dengan nama lokal yang sama telah ada di *platform* tersebut, maka JADE *run-time* akan mencegah pembentukan agen.

2.2.4.2 Inisialisasi Agen

Inisialisasi agen dapat dilakukan setelah proses *compile* terhadap *file* sumber dilakukan. Proses *compile* untuk *agent class* dilakukan sama seperti *java class* pada umumnya, tetapi dengan menentukan *classpath* pada *environment variable* terlebih dahulu.

```
Javac <agent-class>.java
```

Setelah proses *compile* berhasil agen dapat dijalankan bersamaan dengan *main container* melalui perintah:

```
java jade.Boot <agent-name>:<agent-classes>
```

Jika *main container* sudah dibentuk sebelumnya, maka agen harus diletakkan dalam *container* baru dengan menambahkan pilihan `-container` pada baris perintah:

```
java jade.Boot -container <agent-name>:<agent-classes>
```

Cara yang disebutkan di atas merupakan cara menjalankan agen melalui baris perintah (*command line*). Selain cara tersebut, masih banyak cara menjalankan agen, seperti melalui GUI RMA, dan melalui kode yang mengirim permintaan ke AMS.

2.2.4.3 Mematikan Agen

Meskipun sebuah agen sudah selesai melakukan tugasnya, dan tidak memiliki pekerjaan lain, agent tersebut akan tetap hidup. Untuk mematikan agen tersebut, maka `doDelete()` *method* harus dipanggil.

2.2.4.4 Mengirim Argumen ke Agen

Agen dapat menerima argumen saat *start-up*, yang diterima sebagai *array* dari objek. Pengambilan argumen dilakukan dengan menggunakan `getArguments()` *method* dari Agent *class*. Ketika menjalankan agen pada baris perintah, argumen diletakkan dalam tanda kurung, dan dipisahkan oleh spasi.

```
Java jade.Boot -container <agent-name>:<agent-classes>(arg1 arg2
arg3);
```

2.2.5 Penugasan Agen

Pekerjaan sebenarnya dari sebuah agen dinyatakan dalam *behaviour*. *Behaviour* menunjukkan tugas yang dibawa sebuah agen, dan diimplementasikan sebagai sebuah objek dari kelas yang meng-*extends* `jade.core.behaviours.Behaviour`. Untuk membuat agen mengeksekusi suatu *behaviour*, maka *behaviour* tersebut harus ditambahkan pada agen dengan menggunakan `addBehaviour()` *method*. *Behaviour* dapat ditambahkan setiap saat di dalam `setup()` *method* atau di dalam *behaviour* lain.

2.2.5.1 Penjadwalan dan Eksekusi Agen

Sebuah agent dapat mengeksekusi beberapa *behaviour* secara bersamaan. Tetapi, *behaviour* tidak berjalan dengan serempak seperti *Java thread,* melainkan sebuah *behaviour* akan terus dijalankan sampai sampai `action()` *method* selesai atau diberhentikan sementara.

Pendekatan ini memiliki beberapa keuntungan, yaitu:

- memungkinkan *java thread* tunggal untuk setiap agen, yang sangat menguntungkan, terutama pada lingkungan dengan sumber daya terbatas, sepeti telepon seluler;
- kinerja yang lebih baik, karena perpindahan *behaviour* lebih cepat daripada perpindahan *thread*;
- menghilangkan permasalahan sinkronisasi antar-*behaviour* yang mengakses suatu sumber daya secara bersamaan;
- ketika terjadi perpindahan *behaviour* aktif, keadaan dari agen dapat disimpan untuk dilanjutkan di lain waktu.

Alur eksekusi dari *agent thread* digambarkan pada Gambar 2.3 [2]. Ketika tidak ada *behaviour* yang tersisa untuk dieksekusi, maka *agent thread* akan memasuki mode *sleep* agar tidak mengkonsumsi CPU. *Thread* ini akan bangun kembali saat terdapat *behaviour* yang harus dieksekusi.

2.2.5.2 One-Shot Behaviours, Cyclic Behaviours, dan Generic Behaviours

Ketiga jenis *behaviour* utama yang disediakan JADE adalah sebagai berikut.

1. *One-shot behaviours*

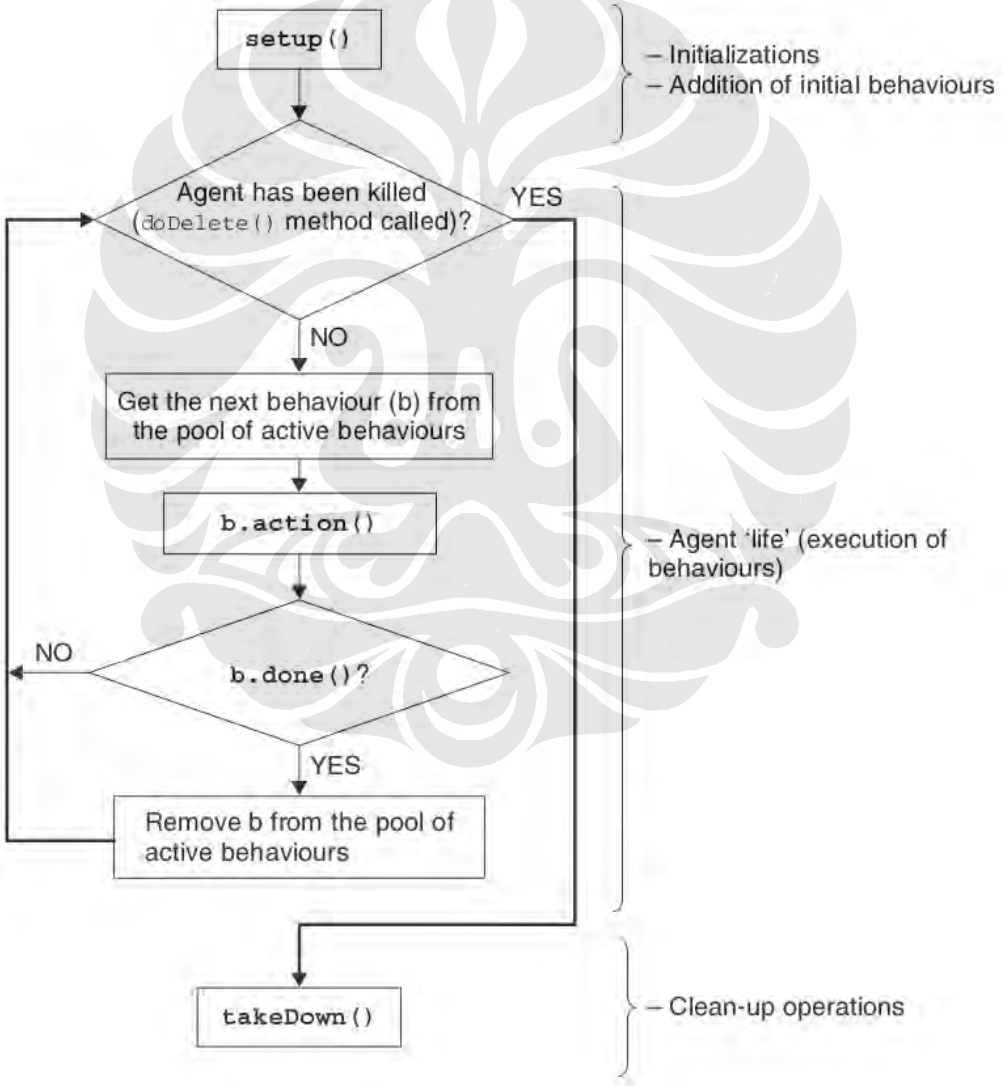
   Dirancang untuk selesai dalam satu fase eksekusi (`action()` *method* hanya dieksekusi satu kali).

2. *Cyclic behaviour*

   Digunakan untuk pekerjaan yang dilakukan secara berulang-ulang dan tidak pernah berakhir.

3. *Generic behaviour*

   Mengintegrasikan sebuah pemicu keadaan dan mengeksekusi operasi berbeda tergantung dari nilai keadaan. *Behaviour* selesai ketika kondisi yang diinginkan sudah tercapai.



Gambar 2.3 Diagram alir urutan eksekusi *agent thread* [2]

2.2.5.3 Operasi Penjadwalan

JADE telah menyediakan dua buah *class* siap pakai dalam paket `jade.core.behaviours` yang dapat diimplementasikan untuk menghasilkan *behaviour* yang dieksekusi pada waktu yang ditentukan, yaitu sebagai berikut.

1. *Waker behaviour*

   Memiliki `action()` dan `done()` *method* yang diimplementasikan untuk mengeksekusi `onWake()` *abstract method* setelah waktu yang ditentukan terlewati. Setelah eksekusi `onWake()` *method*, maka *behaviour* selesai.

2. *Ticker behaviour*

   Memiliki `action()` dan `done()` *method* yang dieksekusi setiap selang waktu tertentu. *Behaviour* ini tidak pernah selesai, kecuali jika `stop()` *method* dipanggil secara eksplisit.

2.2.6 Komunikasi Agen

Komunikasi agen pada JADE dilakukan bedasarkan *asynchronous message passing*. Setiap agen memiliki sebuah *'mailbox' (message queue)* dimana JADE *run-time* meletakkan pesan yang dikirimkan oleh agen lain. Setiap ada pesan yang diterima oleh *message queue*, maka agen yang dituju akan diinformasikan.

Format pesan pada JADE sesuai dengan format yang ditentukan oleh FIPA-ACL. Setiap pesan memiliki *field* sebagai berikut:

- pengirim pesan;
- daftar penerima pesan;
- tindakan komunikatif (*performative*) yang menunjukkan apa tujuan dari pengirim pesan (Misalkan jika *performative* pesan berupa REQUEST, maka pengirim mengharapkan penerima pesan melakukan sesuatu);
- isi pesan, berisi informasi sebenarnya yang dikirimkan;
- bahasa pesan, menunjukkan *syntax* yang digunakn pada pesan tersebut;
- *Ontology*, menunjukkan tata bahasa yang digunakan pada isi pesan;

- beberapa *field* tambahan untuk pengaturan tambahan, seperti: *conversation-id, reply-to reply-with, in-reply-to* dan *reply-by*.

2.2.6.1 Mengirim Pesan

Mengirim pesan kepada agen lain dilakukan dengan mengisi *field* dari objek ACLMessage, dan memanggil `end()` *method*. Contoh kode di bawah ini mengirimkan pesan berisi "Hello agent" kepada agen bernama Agent1:

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);

msg.addReceiver(new AID("Agent1", AID.ISLOCALNAME));

msg.setContent("Hello agent");

send(msg);
```

2.2.6.2 Menerima Pesan

Sebuah agen dapat mengambil pesan dari *message queue* dengan menggunakan `receive()` *method*. Metode ini mengembalikan pesan pertama yang terdapat dalam *message queue*, atau null jika *message queue* kosong.

```
ACLMEssage msg = receive()
If (msg != null) {
// Pemrosesan pesan yang diterima
}
```

2.2.6.3 Menahan Behaviour yang Menunggu Pesan

Sebuah *behaviour* yang berfungsi untuk menunggu pesan yang masuk biasnya berupa *cyclic behaviour* yang terus dieksekusi secara berulang-ulang. Eksekusi terhadap *behaviour* seprti ini dapat menyebabkan kerja yang berat pada CPU. Untuk menghindari hal ini, dapat digunakan `block()` *method* yang menandai *behaviour* tesebut sebagai *'blocked'*, sehingga agen tidak menjadwalkan

behaviour tersebut untuk dieksekusi. Jika terdapat pesan baru di dalam *message queue*, maka *behaviour* yang sebelumnya ditahan, akan menjadi aktif kembali.

2.2.7 Keadaan Agen

Pada JADE dikenal enam jenis keadaan agen. Keadaan suatu agen dapat diketahui dengan melakukan *debug* terhadap agen menggunakan *Introspector Agent*, yaitu salah satu dari *debugging tools* yang didukung oleh JADE. Jenis-jenis keadaan agen tersebut adalah sebagai berikut.

1. *Active*

   Keadaan ini menunjukkan bahwa suatu agen aktif mengeksekusi *behaviour*-nya. Untuk membuat keadaan suatu agen menjadi *active* dapat dilakukan dengan menggunakan *syntax* `doActivate()`.

2. *Suspended*

   Keadaan ini menunjukkan bahwa suatu agen sedang tidak aktif dan tidak mengeksekusi atau menunda seluruh *behaviour*-nya sampai agen tersebut *active* kembali. Untuk membuat keadaan suatu agen menjadi *suspended* dapat dilakukan dengan *syntax* `doSuspend()`.

3. *Idle*

   Keadaan ini menunjukkan bahwa suatu agen *active* namun tidak terdapat *behaviour* untuk dieksekusi sehingga agen tersebut tidak melakukan apa-apa.

4. *Waiting*

   Keadaan ini menunjukkan bahwa suatu agen sedang menunggu dan menghentikan eksekusi *behaviour*-nya selama periode waktu tertentu atau sampai agen tersebut *active* kembali. Saat agen ini *active* kembali maka ia akan melanjutkan kembali *behaviour* tersebut. Untuk membuat keadaan

suatu agen menjadi *waiting* dapat dilakukan dengan *syntax* `doWait()` atau `doWait(millis)`.

5. *Moving*

   Keadaan ini menunjukkan bahwa suatu agen sedang berpindah dari suatu `Location` menuju `Location` lain. Keadaan ini dikenal pada agen bergerak. Untuk membuat keadaan suatu agen menjadi *moving* dapat dilakukan dengan *syntax* `doMove(destination)`.

6. *Dead*

   Keadaan ini menunjukkan bahwa suatu agen telah mati. Untuk mematikan suatu agen dapat dilakukan dengan menggunakan *syntax* `doDelete()`.

Jika kita melakukan *debug* terhadap suatu agen menggunakan *Introspector Agent,* maka terdapat empat jenis proses transisi keadaan yang dikenali yakni *Suspend, Wait, Wake Up,* dan *Kill. Suspend* merupakan proses transisi keadaan agen menjadi *suspended*. *Wait* merupakan proses transisi keadaan agen menjadi *waiting*. *Wake Up* merupakan proses transisi keadaan agen menjadi *active*. *Kill* merupakan proses transisi keadaan agen menjadi *dead*.

2.3 MySQL

MySQL merupakan sebuah *Relational Database Management System* (RDBMS) yang mendukung *multi-thread*, *multi-user*, dan SQL (*Structured Query Language*) *database server*. MySQL dimiliki dan disponsori oleh perusahaan MySQL AB yang merupakan bagian dari Sun Microsystems. Kode sumber dari proyek MySQL tersedia di bawah lisensi GPL *(GNU General Public License),* dan berbagai perjanjian *proprietary* lainnya.

2.3.1 Konsep *Relational Model*

Konsep *relational model* pertama kali diusulkan oleh Dr. Edgar F. Codd pada tahun 1970. Model ini kemudian dijelaskan dalam 12 peraturan yang dikenal dengan nama *Codd's Twelve Rules*.

Sebuah *Database Management System* (DBMS) yang ideal akan memenuhi semua peraturan tersebut. Tetapi, saat perancangan *database*, peraturan yang dianggap paling penting adalah peraturan pertama dan kedua saja. Berikut ini adalah ringkasan dari kedua peraturan tersebut.

1. Peraturan 1

   Peraturan ini menyatakan bahwa data terletak di dalam tabel-tabel. Sebuah tabel mengelompokan informasi mengenai subyek tertentu. Baris mendeskripsikan informasi mengenai sebuah hal, sedangkan kolom mendeskripsikan karakteristik dari setiap hal. Perpotongan antara baris dan kolom berisi sebuah nilai dari atribut spesifik sebuah hal tunggal.

2. Peraturan 2

   Data tidak diambil atau direferensikan oleh lokasi fisik. Data harus diperoleh dengan mereferensikan tabel, *key* yang unik, dan satu atau beberapa nama kolom.

MySQL merupakan sebuah RDBMS, yang artinya tidak hanya mendukung penyimpanan data pada *table,* tetapi juga menangani *relationship* antar-*table* tersebut.

2.3.2 *File* Data MySQL

MySQL memiliki *file* dengan beberapa ekstensi, antara lain:

- FRM, merupakan *file* yang digunakan untuk mendefinisikan format *table*;
- MYD, *file* yang menyimpan data;

- MYI, digunakan sebagai *file* index;
- MRG, tipe *file* yang digunakan sebagai daftar nama *table* yang digabungkan *(merged).*

Pembuatan masing-masing *file* diatas tergantung pada tipe *table*. Pada MySQL, terdapat beberapa tipe *table* sebagai berikut:

- BDB, merupakan *tipe* table yang ditangani oleh *Berkeley DB handler*;
- MEMORY, merupakan *table* yang tersimpan pada *memory*, sehingga data hanya akan tersedia selama *server* MySQL aktif;
- InnoDB, tipe *table* yang ditangani *InnoDB handler*;
- ISAM, merupakan tipe *table default* MYSQL versi terdahulu, saat ini tipe *table default* MySQL adalah MyISAM (Sementara ISAM masih tetap didukung untuk kompabilitas);
- MERGE, sebuah *table* virtual yang tercipta dari tipe *table* MyISAM;
- MyISAM, merupakan tipe *table default* MySQL, mendukung index dan telah dioptimasi dari sisi kecepatan dan kompresi.

2.3.3 *Executable file* pada MySQL

MySQL menyediakan *file-file* pendukung yang dapat dieksekusi secara *Command Line*. *File-file* tersebut antara lain:

- `myisamchk`, berfungsi untuk melakukan *chek* dan *repair* pada *table* MyISAM;
- `myisampack`, berfungsi untuk melakukan kompresi pada *table* MyISAM ke dalam *table read-only;*
- `mysql`, berfungsi untuk melakukan akses pada data MySQL, dapat digunakan dalam mode interaktif maupun mode *batch* (Mode interaktif

digunakan jika ingin langsung mengeksekusi *query* pada *database.* Mode *batch* digunakan untuk mengeksekusi *query* yang tersimpan pada *file script*, dan menyimpan hasilnya pada sebuah *file);*

- `mysqladmin`, menyediakan *interface* untuk tugas-tugas administratif, seperti mengambil informasi mengenai konfigurasi MySQL, melakukan *setting password*, menghentikan *server*, menciptakan dan menghapus *database*, dan lain-lain;

- `mysqldump`, meng-*copy table* ke dalam *file* untuk data *back-up*. Juga dapat digunakan untuk memindahkan *database* ke *server* lain;

- `mysqlhostcopy`, membuat *back-up* dari sebuah *database;*

- `mysqlshow`, menampilkan daftar *database* yang berada pada MySQL. Juga ditampilkan daftar *table* dan informasi lainnya.

2.3.4 Instalasi MySQL *Community Server*

Untuk menjalankan MySQL *Community Server* sebagai *service* pada *Windows* dilakukan dengan membuat sebuah *file* konfigurasi dengan nama my.ini, atau menggunakan *file* konfigurasi awal yang sudah disediakan oleh MySQL. Untuk memudahkan proses instalasi, masukkan lokasi *folder* bin pada direktori mysql ke dalam PATH pada *environment variable Windows*.

Setelah *file* konfigurasi ditentukan, masukkan perintah:

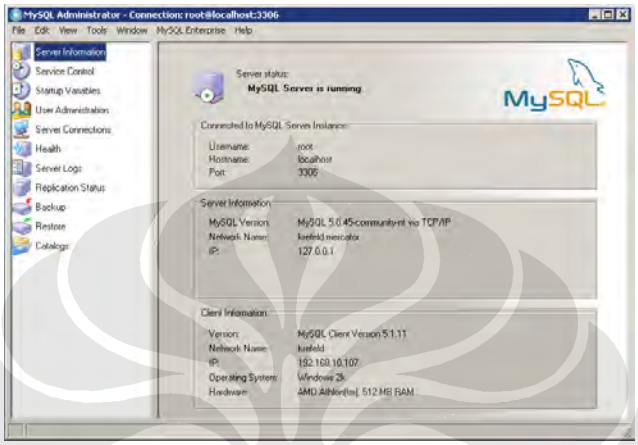
```
mysqld --install <nama-service> --defaults-file=<file-konfigurasi>
```

Setelah *server* MySQL ter-*install* sebagai *service*, maka *server* MySQL akan dijalankan secara otomatis setiap kali sistem operasi dijalankan. *Server* MySQL sebagai *service* dapat dihentikan dengan menggunakan perintah `mysqld --remove`.

Untuk mengakses *server* MySQL dengan menggunakan *command prompt*, dapat digunakan perintah `mysql -u <user> -p <password>`.

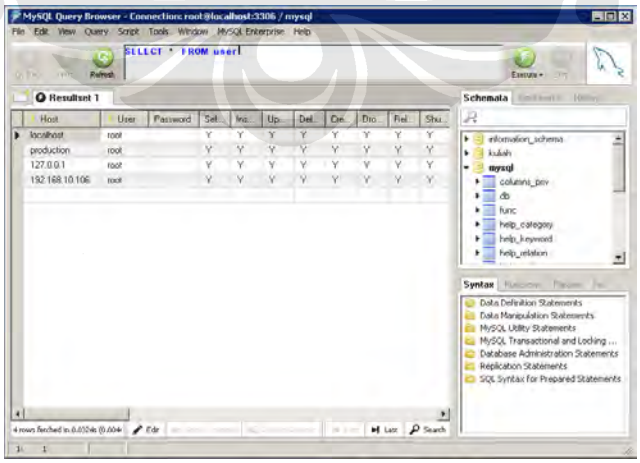
2.3.5 MySQL *Administrator*

MySQL *Administrator* merupakan sebuah program tambahan yang disediakan oleh MySQL AB untuk mempermudah pengawasan dan pengaturan lingkungan MySQL dengan menampilkan data secara grafis. MySQL *Administrator* ditunjukkan oleh Gambar 2.4.



Gambar 2.4 MySQL *Administrator*

MySQL Administrator juga menyertakan MySQL *Query Browser* yang berguna untuk memasukkan, membaca, dan mengubah hasil *query* MySQL secara grafis. MySQL *Query Browser* ditunjukkan oleh Gambar 2.5.



Gambar 2.5 MySQL *Query Browser*

BAB 3

PERANCANGAN SISTEM *MONORAIL* AGEN

3.1 SKENARIO SISTEM *MONORAIL* AGEN

Program aplikasi yang dibangun dalam skripsi ini mengasumsikan sistem *monorail* dengan skenario-skenario berikut.

1. Skenario Penumpang

   Pertama-tama calon penumpang memiliki suatu *smart card,* baik bersifat *contact* ataupun *contactless*. Dalam *smart card* ini terdapat tiga atribut yang harus ada, yakni nama, id yang bersifat unik, dan jumlah nominal uang yang dimiliki. Calon penumpang melakukan transaksi pembelian dengan menggunakan *smart card*-nya pada *smart card reader* yang tersedia di stasiun dengan memasukkan input stasiun yang akan dituju pada mesin. Stasiun yang dapat dituju hanya stasiun setelah stasiun tempat calon penumpang berada sesuai rute kereta. Calon penumpang yang kemudian menunggu kedatangan kereta berada dalam suatu area tunggu khusus. Seandainya calon penumpang membatalkan transaksi atau membatalkan perjalanannya, ia harus kembali meletakkan *smart card*-nya pada suatu *smart card reader* untuk keluarnya penumpang. Dengan demikian sistem dapat mengetahui bahwa transaksi dibatalkan. Jumlah nominal pada *smart-card* dapat berubah seandainya terdapat kebijakan bahwa pembatalan transaksi akan dikenakan biaya. Hal ini tergantung kebijakan bisnis pemilik sistem. Demikian pula penumpang yang turun dari keretapun harus meletakkan *smart card*-nya pada *smart card reader* untuk keluarnya penumpang.

2. Skenario Stasiun

   Saat program stasiun dijalankan, ia akan menunggu pendeteksian oleh agen *server*, baru kemudian dapat aktif. Stasiun melalui *smart card*

reader mengetahui jumlah penumpang yang terdapat disana setiap saat. Setiap saat ada input terhadap *smart card reader*, agen stasiun mengirim pesan ke agen *server* untuk meng-*update database* penumpang. Seandainya terdapat calon penumpang dengan jumlah tertentu atau setelah durasi waktu tertentu, agen stasiun dapat memanggil kereta dengan mengirimkan pesan pada agen *server*.

3. Skenario *Server*

Saat program *server* dijalankan ia akan mendeteksi stasiun-stasiun sesuai dengan konfigurasinya. Setelah semua stasiun terdeteksi, ia menghidupkan agen kereta. Agen *server* bertugas meng-*update* database setiap ada pesan permintaan oleh agen stasiun. Jika terdapat calon penumpang stasiun manapun, maka agen kereta mulai dijalankan/dimobilisasikan. Agen *server* juga bertugas menjalankan agen kereta yang *idle* setiap ada permintaan oleh agen stasiun. Agen *server* juga merespon setiap pesan yang datang dari agen kereta untuk mengetahui status penumpang-penumpang di stasiun yang disinggahinya.

4. Skenario Kereta

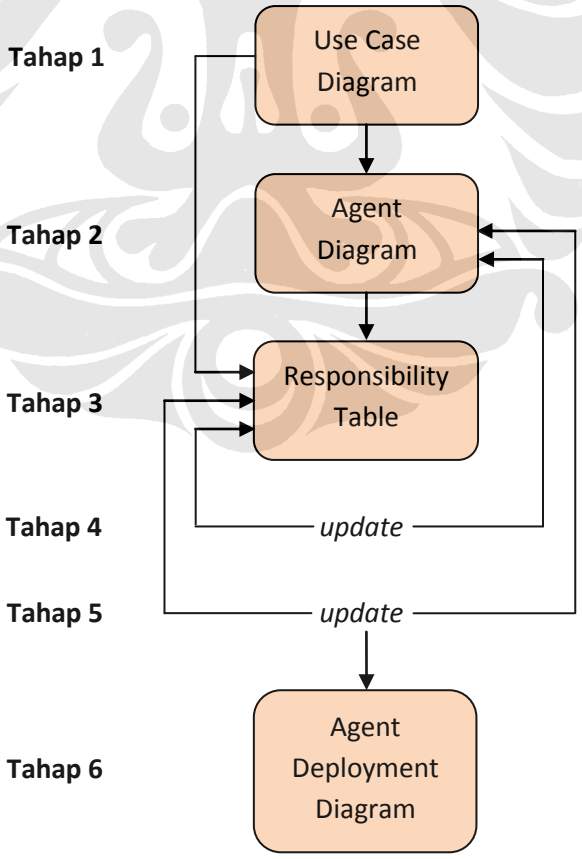
Saat agen kereta hidup ia segera memperoleh identitas dan lokasi agen-agen stasiun. Agen kereta berjalan untuk pertama kali saat ada pesan permintaan oleh agen *server*. Untuk mobilitas selanjutnya akan ditentukan sendiri oleh agen kereta. Ia akan berhenti pada setiap stasiun, mengirim pesan permintaan kepada agen *server* untuk mengetahui status penumpang pada stasiun tersebut. Selanjutnya ia menentukan sendiri aksi apa yang harus dilakukan. Pada setiap stasiun yang disinggahinya ia juga mengirim pesan kepada agen stasiun bahwa ia telah sampai pada lokasi tersebut, sehingga akses penumpang yang naik dan turun diizinkan. Saat telah melewati stasiun akhir dan tidak terdapat penumpang, ia kembali ke agen *server* dan tidak aktif, dengan mengasumsikan kereta memerlukan perawatan ataupun istirahat. Agen kereta ini dapat *suspend* sampai ada panggilan selanjutnya, mati ataupun di-*migrate* ke *server* yang lain

(misalnya *server* dengan rute stasiun sebaliknya) tergantung topologi rute stasiun. Pada simulasi ini agen kereta akan mati saat kembali ke agen *server.*

3.2 ANALISIS SISTEM *MONORAIL* AGEN

Tahap ini akan menganalisis pemecahan terhadap skenario yang telah dipaparkan di pada bagian sebelumnya menggunakan metodologi secara ilmiah.

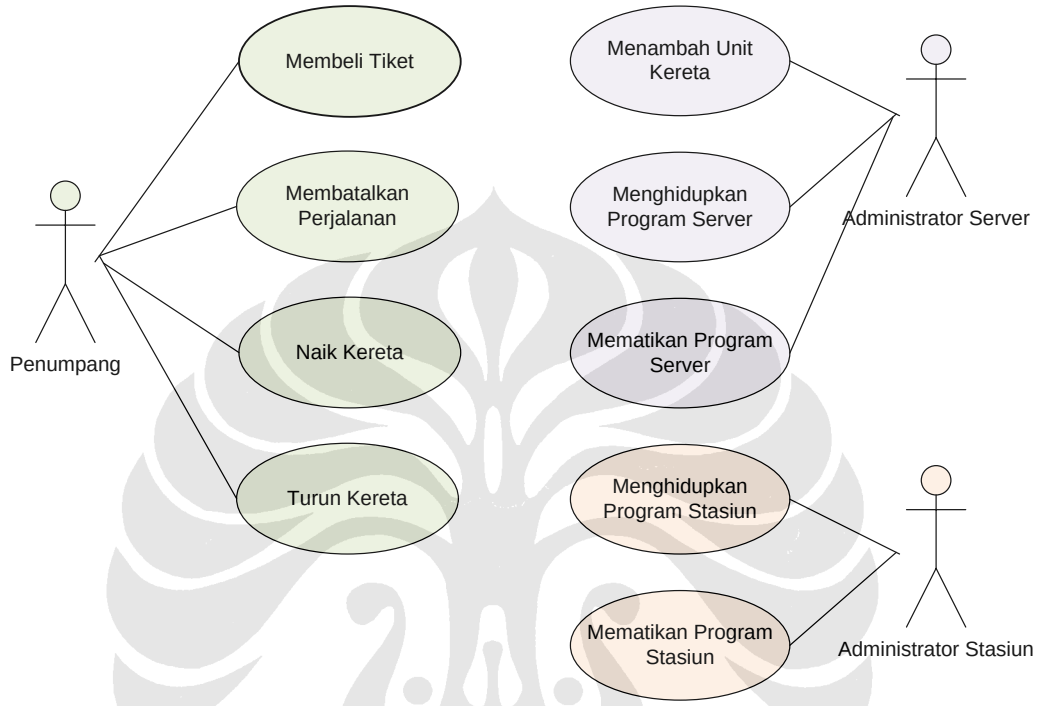
Gambar 3.1 menunjukkan tahapan-tahapan pada fase analisis [3]. Fase analisis bertujuan untuk memperjelas masalah tanpa atau dengan sedikit membahas solusinya. Dari fase ini akan dihasilkan empat buah artefak, yaitu *use case diagram, agent diagram, responsibiliy table,* dan *agent deployment diagram.* Pada gambar terlihat bahwa pada tiap tahapnya terjadi *update* terhadap *agent diagram* dan *responsibility table*.



Gambar 3.1 Tahapan-tahapan pada fase analisis [3]

3.2.1 Tahap 1: Diagram *Use Case*

Berdasarkan deskripsi skenario pada sub bab 3.1, diagram *use case* dapat dibuat seperti pada Gambar 3.2 berikut. Aktor-aktor pada diagram ini adalah pengguna aplikasi.



Gambar 3.2 Diagram *use case* sistem *monorail* agen

3.2.2 Tahap 2: Identifikasi Awal Tipe Agen

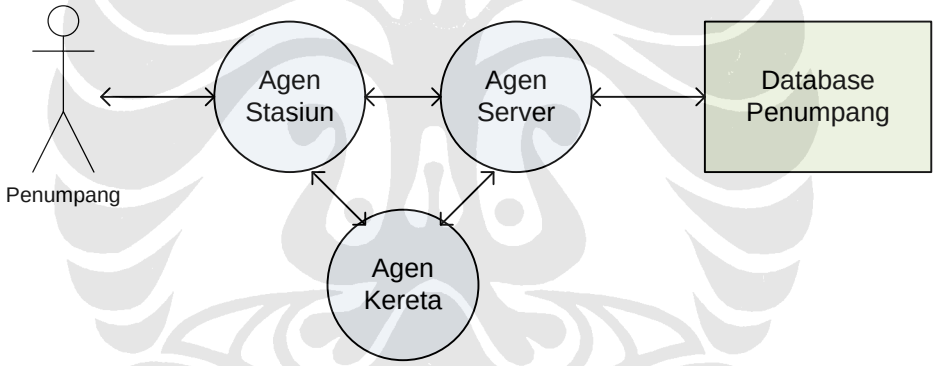
Tahap ini mengidentifikasi jenis-jenis agen yang utama pada skenario dan menghasilkan artefak utama yang penting dalam fase analisis yaitu diagram agen. Panduan yang digunakan dalam identifikasi ini adalah:

- menambahkan satu jenis agen untuk setiap *user*/alat,
- menambahkan satu jenis agen untuk setiap sumber daya (termasuk *legacy software*).

Pada sistem *monorail* agen dapat kita identifikasi tiga buah agen yang utama, yaitu agen *server*, agen stasiun, dan agen kereta.

Diagram agen yang ditunjukkan pada Gambar 3.3 melibatkan simbol-simbol yang dapat diartikan sebagai berikut:

- tipe-tipe agen: merupakan tipe-tipe agen yang sebenarnya, dan dilambangkan dengan lingkaran,
- manusia: orang-orang yang harus bertinteraksi dengan sistem, dilambangkan dengan simbol aktor pada UML,
- sumber daya: sistem luar yang harus berinteraksi dengan sistem, dilambangkan dengan kota,
- hubungan perkenalan: dilambangkan dengan panah yang menghubungkan objek-ebjek dari elemen-elemen yang dijelaskan sebelumnya, menjelaskan bahwa elemen-elemen yang dihubungkan tersebut akan saling berinteraksi dalam beberapa cara, selama sistem beroperasi.



Gambar 3.3 Diagram agen sistem *monorail* agen setelah tahap identifikasi awal tipe agen

Dalam hubungannya dengan sistem-sistem eksternal/*legacy*, sistem berbasis agen yang akan dikembangkan akan menggunakan agen *transducer*. Agen *transducer* adalah agen yang akan berperan sebagai antar muka (*interface*) antara sistem yang diwariskan (*legacy system)* dan agen-agen yang lain yang terdapat di dalam sistem. Cara kerja agen *transducer* adalah dengan menerima pesan yang datang dari pengirim pesan, menterjemahkan pesan tersebut dari bahasa sumber ke bahasa yang akan dituju, kemudian mengirimkannya ke tujuan. Sebagai contoh, sebuah agen akan mengirim pesan ke *legacy system*, maka ia akan mengirimkan pesan ke agen *transducer*. Agen *transducer* akan menerima

pesan tersebut kemudian menterjemahkannya ke bahasa yang dimengerti oleh *legacy system*. Setelah pesan tersebut diterjemahkan, barulah kemudian pesan tersebut diteruskan ke tujuan.

Jika nantinya dalam pengembangan sistem ini, ditemukan komunikasi dengan sistem-sitem eksternal/*legacy* yang tidak dapat diselesaikan dengan agen *transducer*, maka akan digunakan salah satu dari metode berikut.

1. Menyisipkan *wrapper*. Caranya adalah dengan memasukan kode-kode ke dalam *legacy system* yang dilengkapi dengan kode-kode sumber daya *legacy*. Kode yang dimasukan ini akan memungkinkan sumber daya untuk berkomunikasi dengan menggunakan bahasa agen.
2. Menulis ulang kode-kode. Metode ini dilakukan dengan menulis ulang kode-kode yang terdapat di *legacy system*, dan menambahkan kemampuan berkomunikasi dalam bahasa agen.

3.2.3 Tahap 3: Identifikasi Tanggung Jawab

Pada tahap ini, untuk setiap jenis agen, tugasnya masing-masing dinyatakan secara informal dan intuisi berdasarkan diagram *use case*. Hasilnya adalah artefak yang disebut *responsibility table* seperti yang ditunjukkan pada Tabel 3.1.

3.2.4 Tahap 4: Identifikasi Hubungan

Pada tahap ini dilakukan analisis interaksi antaragen. Hal yang menjadi fokus perhatian adalah agen mana yang ingin berinteraksi, dan dengan agen mana pula interaksi tersebut akan dilakukan. Memperbaharui interaksi berarti memperbaharui hubungan antar agen, yang berakibat pada penambahan dan spesifikasi tanggung jawab yang harus dikerjakan oleh agen yang terlibat dalam interaksi tersebut. Sebagai hasilnya dapat terjadi *update* terhadap *agent diagram* dan *responsibility table*.

Tabel 3.1 *Responsibility table* sistem *monorail* agen setelah tahap identifikasi tanggung jawab

Jenis Agen	Tugas
Agen Stasiun	1. Menghitung jumlah penumpang yang menunggu di stasiun setiap saat
	2. Mendeteksi penumpang yang melakukan transaksi pembelian tiket
	3. Mendeteksi jika terdapat kesalahan dalam transaksi penumpang dan menginformasikannya kepada calon penumpang
	4. Mendeteksi penumpang yang melakukan pembatalan perjalanan
	5. Mendeteksi penumpang yang naik ke kereta
	6. Mendeteksi penumpang yang turun dari kereta
	7. Mendeteksi jika terdapat kereta yang singgah
	8. Mengirim setiap perubahan atribut penumpang berupa pesan ke agen *server*
Agen *Server*	9. Meng-*update* database setiap ada perubahan atribut penumpang
	10. Menjalankan agen kereta jika terdapat penumpang di manapun
Agen Kereta	11. Berpindah antara lokasi agen-agen stasiun
	12. Menentukan stasiun mana yang harus disinggahi dan yang tidak berdasarkan data penumpang
	13. Memberitahu agen stasiun jika telah sampai pada lokasinya

3.2.5 Tahap 5: Perbaikan Agen

Bagian ini berisi apa yang diperlukan agen untuk melaksanakan dapat tugasnya, dan bagaimana, kapan, serta dimana informasi ini diperoleh/disimpan.

3.2.5.1 Dukungan

Agen stasiun pada program sistem *monorail* agen ini memerlukan dukungan informasi berupa daftar stasiun yang dilalui kereta dan identitas dari agen stasiun itu sendiri untuk dapat menyediakan pilihan stasiun tujuan kepada calon penumpang. Untuk mengakomodasi hal ini diterapkan dua buah cara. Pertama, saat program stasiun dijalankan, ia perlu membaca suatu *file* konfigurasi yang memuat konfigurasi identitasnya sendiri. Kedua, saat agen *server* mendeteksi agen stasiun, agen *server* mengirimkan daftar agen stasiun seluruhnya sekaligus sebagai rute yang akan dilalui kereta kepada agen-agen stasiun,

kemudian masing-masing agen stasiun akan menangani data tersebut. Cara yang digunakan pada penelitian ini adalah cara kedua.

Agen stasiun juga pada implementasinya seharusnya memerlukan dukungan ketersambungan dengan *smart card reader*, yang mampu membaca *smart card* dan menyediakan atribut nama, id, dan jumlah nominal uang yang diperlukan untuk transaksi oleh calon penumpang. Dalam program sistem *monorail* agen ini, kertersambungan dengan *smart card reader* belum diimplementasikan. Hal yang diterapkan adalah menggunakan user *interface* untuk menentukan aksi penumpang guna melakukan simulasi.

Agen kereta kereta juga memerlukan informasi rute stasiun yang harus dilaluinya. Untuk itu saat agen kereta hidup ia segera menerima daftar lokasi yang menjadi rutenya dari agen *server*.

Agen *server* memerlukan informasi pendukung berupa daftar agen stasiun yang harus ditanganinya sekaligus merupakan rute dari agen kereta serta daftar agen kereta yang tersedia. Agen *server* memerlukan input dari pemilik sistem atau membaca dari suatu *file* konfigurasi yang memuat informasi tersebut. Pembuatan dan pembacaan *file* konfigurasi seharusnya didukung oleh program agar tidak harus selalu meminta input pemilik sistem setiap kali program dijalankan. Daftar agen yang dimuat dalam *file* konfigurasi harus sesuai dengan nama global agen yang bersifat unik.

3.2.5.2 Pencarian

Bagian ini berhubungan dengan bagaimana agen terhubung dan dapat saling dikenali. Mekanisme yang paling sederhana dan efektif adalah konvensi penamaan, dimana nama agen agen secara global adalah unik. Hal ini diterapkan pada program sistem transportasi *monorail* agen ini. Selain itu juga diterapkan metode pengiriman identitas agen oleh agen *server* seperti yang telah diungkapkan pada bagian di atas.

3.2.5.3 Manajemen dan Pengawasan

Jenis agen lain yang dapat ditambahkan adalah terkait dengan isu seperti pengawasan kesalahan agen dan mengembalikannya ke keadaan semula. Hal ini terkait dengan penanganan kesalahan. Pada program sistem *monorail* agen Java, sampai tahap ini, tidak ada agen baru yang perlu ditambahkan untuk alasan manajemen dan *monitoring*.

Setelah tahap perbaikan agen, terjadi *update* terhadap *responsibility table* seperti yang ditunjukkan oleh Tabel 3.2. Meskipun agen-agen mendapat tambahan dan spesifikasi tanggung jawab, tidak terlihat perubahan atau tambahan interaksi pada diagram agen. Dengan demikian, diagram agen tetap sama seperti pada Gambar 3.3.

Tabel 3.2 *Responsibility table* sistem *monorail* agen setelah tahap identifikasi hubungan dan perbaikan agen

Jenis Agen	Tugas
Agen Stasiun	1. Menghitung jumlah penumpang yang menunggu di stasiun setiap saat
	2. Memperoleh daftar stasiun dari *server* agen untuk dapat menyediakan layanan
	3. Mendeteksi penumpang yang melakukan transaksi pembelian tiket dan menginformasikannya ke agen *server*
	4. Mendeteksi penumpang yang melakukan pembatalan perjalanan dan menginformasikannya ke agen *server*
	5. Mendeteksi penumpang yang naik ke kereta dan menginformasikannya ke agen *server*
	6. Mendeteksi penumpang yang turun dari kereta dan menginformasikannya ke agen *server*
	7. Mendeteksi jika terdapat kereta yang singgah
	8. Mendeteksi jika terdapat kesalahan dalam transaksi penumpang dan menampilkan informasi kesalahan
	9. Memanggil kereta jika penumpang melebihi kapasitas tertentu atau setelah durasi waktu tertentu

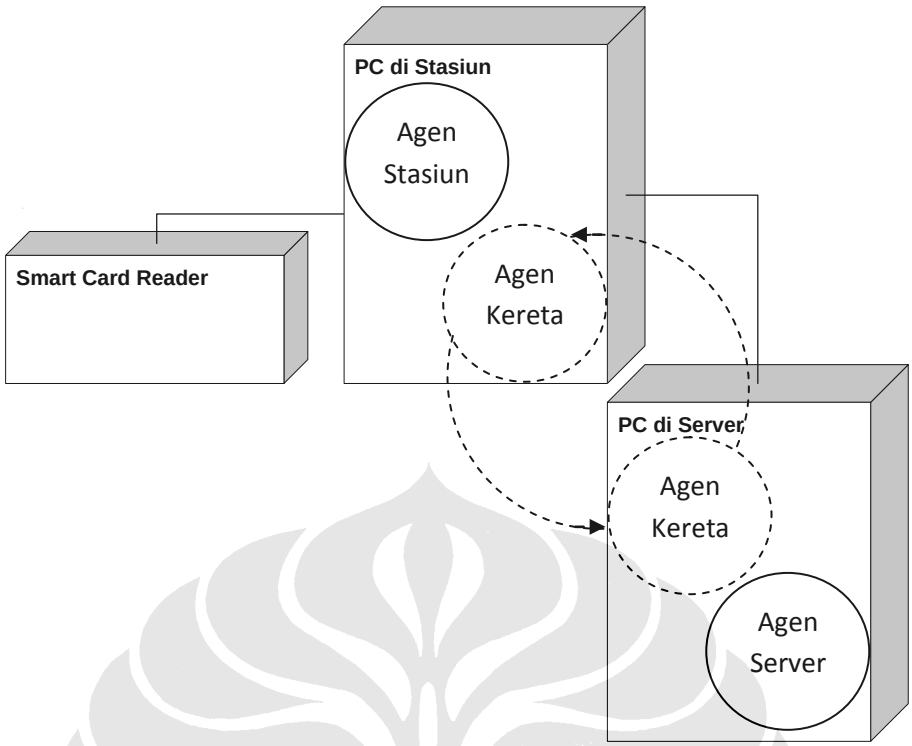
Tabel 3.2 *Responsibility table* sistem *monorail* agen setelah tahap identifikasi hubungan dan perbaikan agen (lanjutan)

Jenis Agen	Tugas
Agen *Server*	10. Meng-*update* database setiap ada perubahan atribut penumpang
	11. Menjalankan agen kereta jika terdapat penumpang di manapun sebagai respon terhadap panggilan oleh stasiun
	12. Mengirim daftar rute kereta/daftar stasiun kepada agen stasiun dan agen kereta
	13. Merespon permintaan status atau daftar penumpang di stasiun yang diminta oleh agen kereta
	14. Mematikan agen kereta jika telah menyelesaikan rutenya
Agen Kereta	15. Berpindah antara lokasi agen-agen stasiun
	16. Melakukan permintaan status/daftar penumpang pada stasiun yang sedang disinggahinya kepada agen *server* untuk menentukan waktu singgah
	17. Menerima daftar stasiun dari *server* agen untuk dapat mengetahui rute yang harus dilaluinya
	18. Memberitahu agen stasiun jika telah sampai pada lokasinya

3.2.6 Tahap 6: Informasi Penempatan Agen

Tahap terakhir pada fase analisis ini menghasilkan artefak berupa *agent deployment diagram*, yang menggambarkan dimana perangkat tempat agen-agen tinggal. Pada dasarnya program sistem *monorail* agen dibangun dengan menerapkan komunikasi *intra-platform* antar agen-agennya. Dengan demikian sistem ini dapat didistribusikan pada beberapa komputer tergantung kebutuhan jumlah stasiun namun tetap berada dalam suatu jaringan lokal.

Agent deployment diagram untuk program sistem *monorail* agen ini ditunjukkan pada Gambar 3.4.



Gambar 3.4 *Agent deployment diagram* sistem *monorail* agen setelah tahap informasi penempatan agen

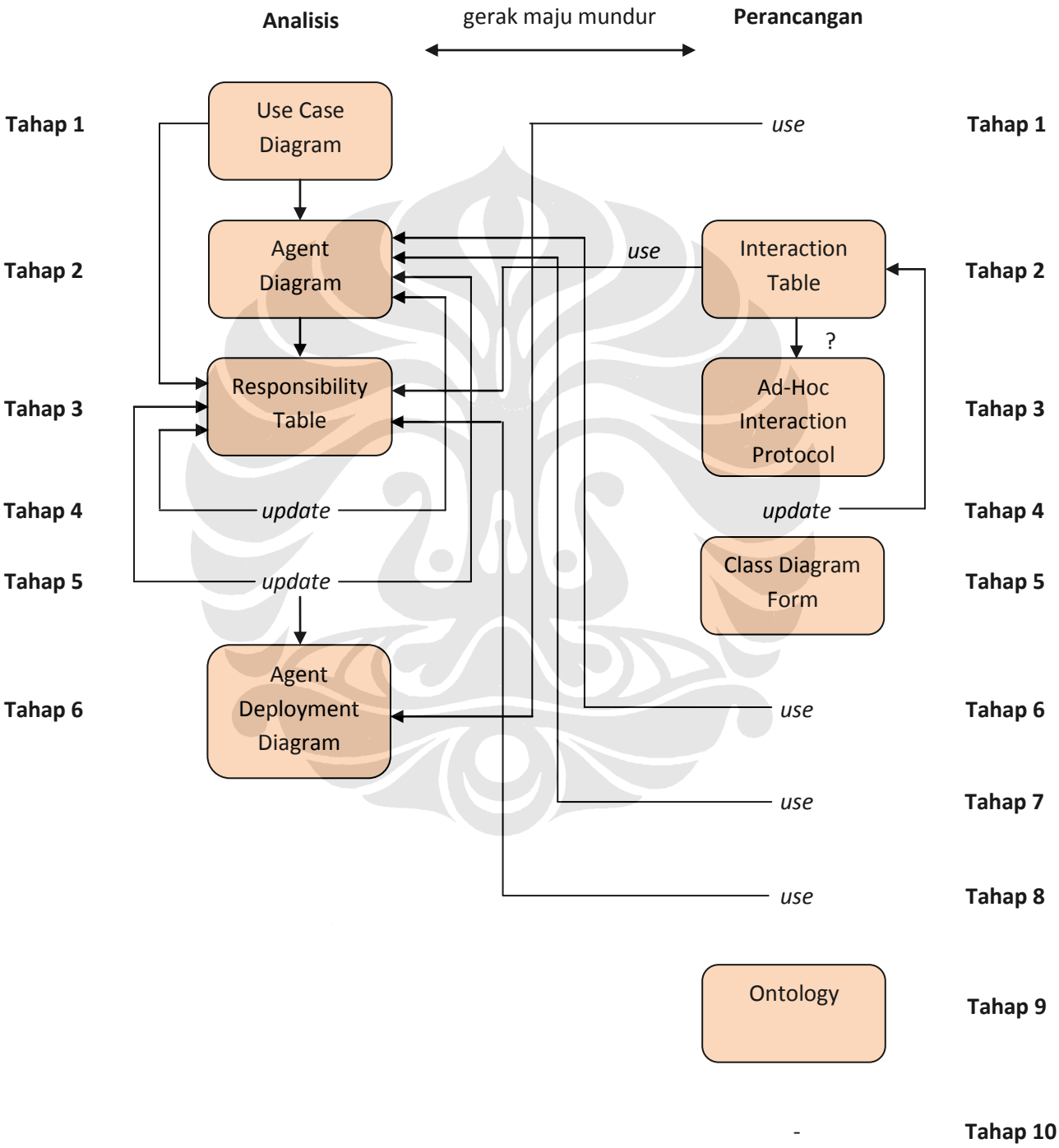
3.3 PERANCANGAN SISTEM *MONORAIL* AGEN

Setelah menyelesaikan tahap analisis, kita memasuki tahap perancangan yang bertujuan untuk menspesifikasi solusi-solusi dari permasalahan. Gambar 3.5 menunjukkan seluruh sub-tahapan dalam fase perancangan berikut artefak-artefak yang dihasilkan dan hubungannya dengan fase analisis [3]. Perlu dicatat bahwa tidak semua sub-tahapan akan dilalui atau dispesifikasi dalam perancangan sistem *monorail* agen karena memang tidak diperlukan pada kasus ini.

3.3.1 Tahap 1: Penamaan Ulang/Penggabungan/Pemecahan Agen

Tahap ini melibatkan pengamatan terhadap artefak yang telah dihasilkan pada tahap analisis perancangan dan menentukan apakah terdapat agen yang harus dipecah atau digabungkan atau dibiarkan apa adanya. Hal ini penting

mengingat jumlah agen akan mempengaruhi kompleksitas dan efisiensi sistem secara keseluruhan. Beberapa aturan untuk menentukan apa yang harus dilakukan pada tahap ini adalah sebagai berikut.



Gambar 3.5 Tahapan-tahapan pada fase perancangan [3]

1. Duplikasi data harus dihindari. Jika terdapat dua atau lebih agen yang berbagi informasi dalam jumlah banyak untuk menjalankan tugasnya, agen-agen ini memungkinkan untuk digabung.

2. Duplikasi kode untuk mengakses sumber daya harus dihindari. Jika terdapat dua atau lebih agen yang memerlukan akses ke sumber daya yang sama, agen-agen ini memungkinkan untuk digabung.

3. Hindari pemecahan agen jika tidak ada alasan yang cukup karena banyaknya agen akan menambah kompleksitas sistem dan mengurangi efesiensi sistem dengan banyaknya komunikasi yang tidak perlu.

4. Setiap agen disituasikan untuk satu mesin. Faktor utama untuk memecah agen adalah isu *deployment*. Jika dua bagian agen harus disediakan pada mesin yang berbeda, bagian-bagian fungsionalitas ini harus disediakan oleh agen yang berbeda.

5. Hindari adanya agen yang terlalu besar dan kompleks. Hal ini membuat sulitnya untuk merancang dan mengatur agen tersebut.

6. Pada beberapa kasus dimana pendekatan *wrapper* digunakan, agen diasumsikan membungkus dan memiliki ukuran sebesar kode Java yang dibungkusnya. Pada kasus seperti ini, akan sulit untuk menggabungkan dan memecah agen.

Pada sistem *monorail* agen, terdapat jumlah tipe agen yang relatif kecil. Oleh karena itu, penggabungan atau pemecahan tipe agen tidaklah diperlukan.

3.3.2 Tahap 2: Spesifikasi Interaksi

Pada tahap ini semua kewajiban dalam *responsibility table* yang memiliki hubungan dengan agen lain menjadi fokus perhatian dan dispesifikasikan ke dalam suatu *interaction table*. Tabel 3.3 menunjukkan *interaction table* untuk sistem transportasi *monorail* agen.

Tabel 3.3 *Interaction table* sistem *monorail* agen setelah tahap spesifikasi interaksi

Interaksi	Tanggung Jawab	Interaction Protocol (IP)	Peranan	Dengan	Kapan
Stasiun menerima daftar stasiun	2	FIPA Request	R	Agen *server*	Setelah seluruh agen stasiun dideteksi oleh agen *server*
Update penumpang baru	3	Contract Net	I	Agen *server*	Ada transaksi baru
Update pembatalan transaksi	4	FIPA Request	I	Agen *server*	Ada pembatalan perjalanan
Update penumpang naik kereta	5	FIPA Request	I	Agen *server*	Ada penumpang turun kereta
Update penumpang turun kereta	6	FIPA Request	I	Agen *server*	Ada penumpang naik kereta
Deteksi kereta datang	7	FIPA Request	R	Agen kereta	Ada kereta datang
Memanggil kereta	9	FIPA Request	I	Agen *server*	Saat ada sejumlah penumpang dan tidak ada kereta dalam periode tertentu
Meminta status penumpang di stasiun	16	FIPA Request	I	Agen *server*	Saat kereta sampai di suatu stasiun
Kereta menerima daftar stasiun	17	FIPA Request	R	Agen *server*	Setelah agen kereta hidup

3.3.3 Tahap 3: Definisi Protokol Interaksi *Ad-Hoc*

Pada kasus saat protokol interaksi yang ada tidak dapat digunakan untuk sebuah interaksi, sebuah protokol interaksi *ad-hoc* didefinisikan menggunakan formalitas yang sesuai.

Pada sistem *monorail* agen interaksi protokol yang digunakan bersifat *default*, karena protokol interaksi yang ada sudah mampu mengakomodasi interaksi antaragen.

3.3.4 Tahap 4: *Message Templates*

Semua peranan protokol interaksi yang diidentifikasi pada tahap sebelumya diimplementasikan sebagai JADE *behaviours*. Pada tahap ini, `MessageTemplate` yang sesuai dispesifikasikan untuk digunakan pada *behaviour* saat menerima pesan masuk. Jenis-jenis `MessageTemplate` yang digunakan ditambahkan pada setiap baris dalam *interaction table*. `MessageTemplate` digunakan untuk memilah jenis pesan yang diterima oleh suatu agen. Cara yang paling sederhana adalah membedakan `ConversationId` untuk setiap `MessageTemplate`.

Berikut ini adalah contoh `MessageTemplate` yang digunakan untuk menerima permintaan menambah penumpang baru pada *database* oleh agen *server*.

```
MessageTemplate createPassenger = MessageTemplate.and(
      MessageTemplate.MatchConversationId("request-for-creating-
      passenger"),
      MessageTemplate.MatchPerformative(ACLMessage.REQUEST));
```

Pada behaviour agen *server* digunakan cara berikut untuk memilah pesan dengan `MessageTemplate` tersebut dan merespon permintaan penambahan penmpang pada *database*.

```
ACLMessage msg = receive(createPassenger);
if ( msg != null ) {

// Respon terhadap pesan msg

}
```

Setelah tahap Tahap 4: *Message Templates* terjadi *update* terhadap *interaction table* berupa penambahan kolom *Template,* seperti yang ditunjukkan oleh Tabel 3.4.

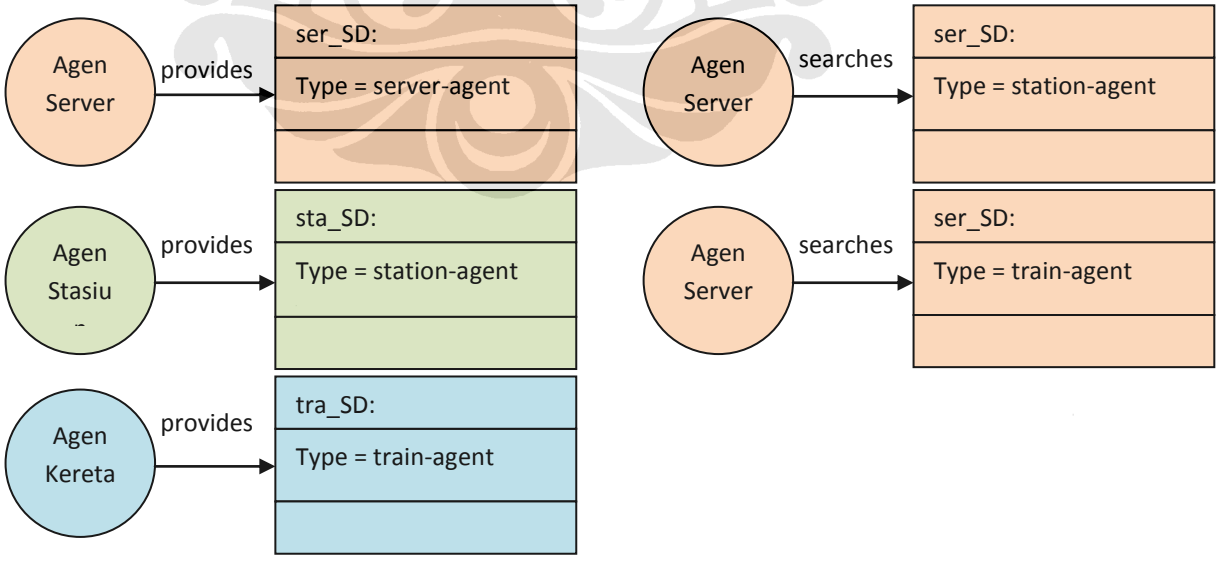
Tabel 3.4 Penambahan kolom *Template* dalam *interaction table* setelah tahap *Message Templates*

Interaksi	Template
Stasiun menerima daftar stasiun	Perf, Conv-id
Update penumpang baru	Perf, Conv-id
Update pembatalan transaksi	Perf, Conv-id
Update penumpang naik kereta	Perf, Conv-id
Update penumpang turun kereta	Perf, Conv-id
Deteksi kereta datang	Perf, Conv-id
Memanggil kereta	Perf, Conv-id
Meminta status penumpang di stasiun	Perf, Conv-id
Kereta menerima daftar stasiun	Perf, Conv-id

3.3.5 Tahap 5: Deskripsi yang Didaftarkan/Dicari

Pada tahap ini, konvensi penamaan dan layanan yang diregistrasi/dicari oleh agen dalam katalog *yellow pages* diorganisir oleh JADE *Directory Facilitator* memiliki bentuk atau aturan tertentu.

Konvensi penamaan bersifat tergantung pada domain dan dianjurkan untuk mengunakan bahasa natural untuk menspesifikasinya. *Class diagram form* untuk sistem transportasi *monorail* agen pada Gambar 3.6 mendeskripsikan layanan registrasi/pencarian yang terjadi pada sistem.



Gambar 3.6 Registrasi dan pencarian layanan pada sistem *monorail* agen

3.3.6 Tahap 6: Interaksi Agen-Sumber Daya

Agen yang berinteraksi dengan sumber daya telah dianalisis pada tahap identifikasi awal tipe agen pada fase analisis. Sumber daya dapat diketegorikan menjadi dua jenis dan dijelaskan sebagai berikut.

3.3.6.1 Sumber Daya Pasif

Contoh sumber daya pasif adalah sebuah *database* yang secara penuh dikendalikan oleh agen yang berinteraksi, sebuah file data dalam *local file system* atau sebuah *library* C yang menyediakan fungsi komputasional. Metodologi ini tidak mencakup interaksi dengan sumber daya pasif.

3.3.6.2 Sumber Daya Aktif

Contoh dari sumber daya aktif adalah sebuah *database* di mana seorang operator (atau program eksternal) dapat memasukkan atau memodifikasi data, sebuah *file log* yang secara kontinu di-*update* oleh program eksternal, sebuah alat yang memunculkan *alarm* dan perangkat lunak pengendali sensor yang yang mendeteksi perubahan lingkungan lokal.

Sistem transportasi *monorail* agen dimaksudkan agar dapat menggunakan sumber daya aktif seperti MySQL *database, smart card reader*, sensor, serta perangkat keras lainnya pada implementasi nyatanya. Namun pada skripsi ini masalah dibatasi hanya pada tahap pembangunan aplikasi untuk simulasi. Sumber daya aktif yang digunakan adalah *database* MySQL untuk menyimpan data penumpang. Java menyediakan JDBC API untuk koneksi dan manipulasi *database* tersebut, yang dimplementasikan pada program ini.

3.3.7 Tahap 7: Interaksi Agen-Pengguna

Terdapat beberapa cara untuk agen berinteraksi dengan seorang pengguna. Interaksi yang paling memudahkan pengguna dan paling umum dipakai adalah

menggunakan *grapical user interface* (GUI). Interaksi dengan GUI ini dibedakan menjadi dua kategori, yakni:

- GUI lokal, umumnya mengimplementasikan Swing, *Abstract Window Toolkit* (AWT) atau beberapa *toolkit* grafis lainnya;
- GUI *web* yang mengimplementasikan teknologi *Java Server Pages* (JSP).

Program sistem transportasi *monorail* agen dibangun dengan mengimplementasikan GUI lokal untuk interaksi dengan pengguna dan menampilkan simulasi.

3.3.8 Tahap 8: *Internal Agent Behaviours*

Pekerjaan sesungguhnya dari suatu agen dilakukan dalam *behaviour* agen. Agen-agen dalam sistem *monorail* agen Java menjalankan pekerjaannya dengan menerapkan berbagai jenis *behaviour* sesuai dengan interaksi yang diperlukan. Beberapa diantaranya akan disebutkan dibawah ini sesuai kondisinya.

Untuk menerima pesan yang dapat datang kapan saja umumnya digunakan `CyclicBehaviour`. `SequentialBehaviour` dan `Parallelbehavour` diimplementasikan pada inisiasi program dan saat urutan atau pewaktuan menjadi isu utama. `TickerBehaviour` digunakan pada pendeteksian *service* agen stasiun dan agen kereta. `WakerBehaviour` diimplementasikan pada agen kereta untuk menunggu penumpang. `OneShotBehaviour` menyusun pekerjaan yang kompleks, ditambahkan untuk pekerjaan yang yang hanya berjalan dalam satu siklus. `ArchieveREInitiator` digunakan di mana interaksi dengan protokol `ContractNet` dilakukan.

3.3.9 Tahap 9: Penentuan *Ontology*

Saat agen-agen dalam suatu sistem berinteraksi, mereka bertukar informasi yang merujuk pada entitas, abstrak atau konkret, yang terdapat dalam lingkungan dimana agen berada. Entitas ini dapat beruapa tipe data *primitive*, seperti String

atau Integer, atau dapat berupa suatu struktur yang kompleks. *Concept* merupakan suatu entitas dengan struktur yang kompleks dan dapat didefinisikan dalam bentuk slot-slot.

Entitas umumnya memiliki suatu hubungan dengan ekspresi yang dapat bersifat `true` atau `false`. Serupa dengan entitas kompleks, ekspresi-ekspresi ini juga memiliki struktur yang didefinisikan oleh *template* dan dapat didefinisikan dalam bentuk slot-slot. *Template-template* ini disebut dengan *Predicate.*

Terdapat pula suatu entitas kompleks khusus yang direpresentasikan oleh *descriptors of actions* yang dapat dilakukan oleh agen. *Template* dari *action descriptors* ini disebut sebagai *AgentAction.*

Suatu *ontology* merupakan kumpulan dari *concept, predicates*, dan *agent action* yang berada dalam suatu domain. Suatu *ontology* baru umumnya dibuat dalam komunikasi antar agen dimana isi dari pesan yang disampaikan merupakan suatu entitas yang kompleks.

Pada program sistem *monorail* agen, digunakan suatu *ontology* baru, **PassengerOntology**, untuk mengirim entitas penumpang antar agen. Untuk komunikasi dengan pesan berupa objek Java lainnya digunakan Java *serialization*. *Ontology* diterapkan pada objek penumpang karena entitas ini memiliki sejumlah atribut yang menggambarkan identitas dan aktivitas penumpang tersebut. Atribut-atribut inilah yang menjadi isi dari *database*. **Passenger** merupakan sebuah *Concept* penumpang dalam **PassengerOntology** yang digunakan pada program ini. Pengiriman objek penumpang ini diterapkan oleh *AgentActionSchema* dengan kelas **Create**.

3.3.10 Tahap 10: Pemilihan *Content Language*

JADE menyediakan dua buah *content language*: SL *languag*e dan LEAP *language*. SL *language* merupakan bahasa konten berupa String yang dapat dibaca oleh manusia, sedangkan LEAP adalah bahasa konten *byte-encoded* yang tidak dapat dibaca manusia. SL cocok untuk aplikasi berbasis agen yang bersifat

terbuka (misalnya agen dari beberapa pengembang yang berjalan pada *platform* yang berbeda harus berkomunikasi). LEAPCodec *class* lebih ringan dibandingkan SLCodec *class* dalam hal pembebanan sistem sehingga cocok untuk sistem dengan keterbatasan *memory.* Sebaliknya jika komputer memiliki kapasitas yang tinggi sebaiknya digunakan SL *language*.

Agen-agen dalam sistem transportasi *monorail* agen Java menggunakan SL *language* sebagai *content language* dalam komunikasinya. Hal ini dilakukan karena selain *ontology* yang digunakan memerlukan SL *language,* sifatnya yang dapat dibaca oleh manusia sangat menolong pada proses *debu*g dan pengujian aplikasi.

BAB 4

IMPLEMENTASI, PENGUJIAN, DAN EVALUASI KINERJA

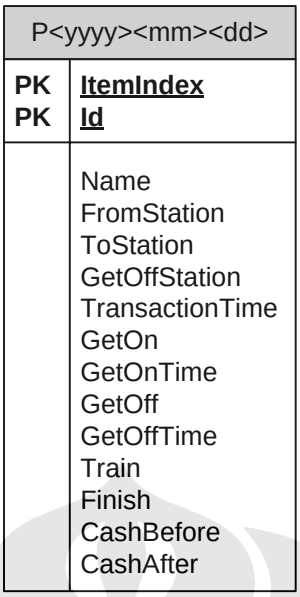
4.1 IMPLEMENTASI

4.1.1 Implementasi *Database*

Implementasi *database* pada sistem *monorail* agen menggunakan MySQL *Community Server* 5.0.45. *Database* pada sistem *monorail* agen digunakan untuk menyimpan data penumpang kereta. Operasi baca dan tulis terhadap *database* dilakukan oleh agen *server*. Jika *database* tidak tersedia maka agen *server* akan membuat *database* baru. Data-data dalam *database* diatur ke dalam *table-table* menurut tanggal saat program dijalankan. Format penamaan *table* yang digunakan adalah P<yyyy><mm><dd>, misalnya P20080403. Hal ini dilakukan untuk menyederhanakan pengelompokkan data penumpang. *Table* yang digunakan adalah tipe *table* MyISAM.

Struktur *database* untuk sistem *monorail* agen ditunjukkan pada Gambar 4.1. Masing-masing kolom pada tabel penumpang memuat informasi yang di akan dijelaskan sebagai berikut:

- **ItemIndex**: urutan penumpang pada *table*, merupakan *primary key,*
- **Name**: nama penumpang sesuai yang terdapat pada *smart card,*
- **Id**: Id penumpang sesuai dengan yang terdapat pada *database*, merupakan *unique key,*
- **FromStation**: nama stasiun tempat penumpang berasal,
- **ToStation**: nama stasiun tujuan penumpang,
- **GetOffStation**: nama stasiun tempat penumpang turun,

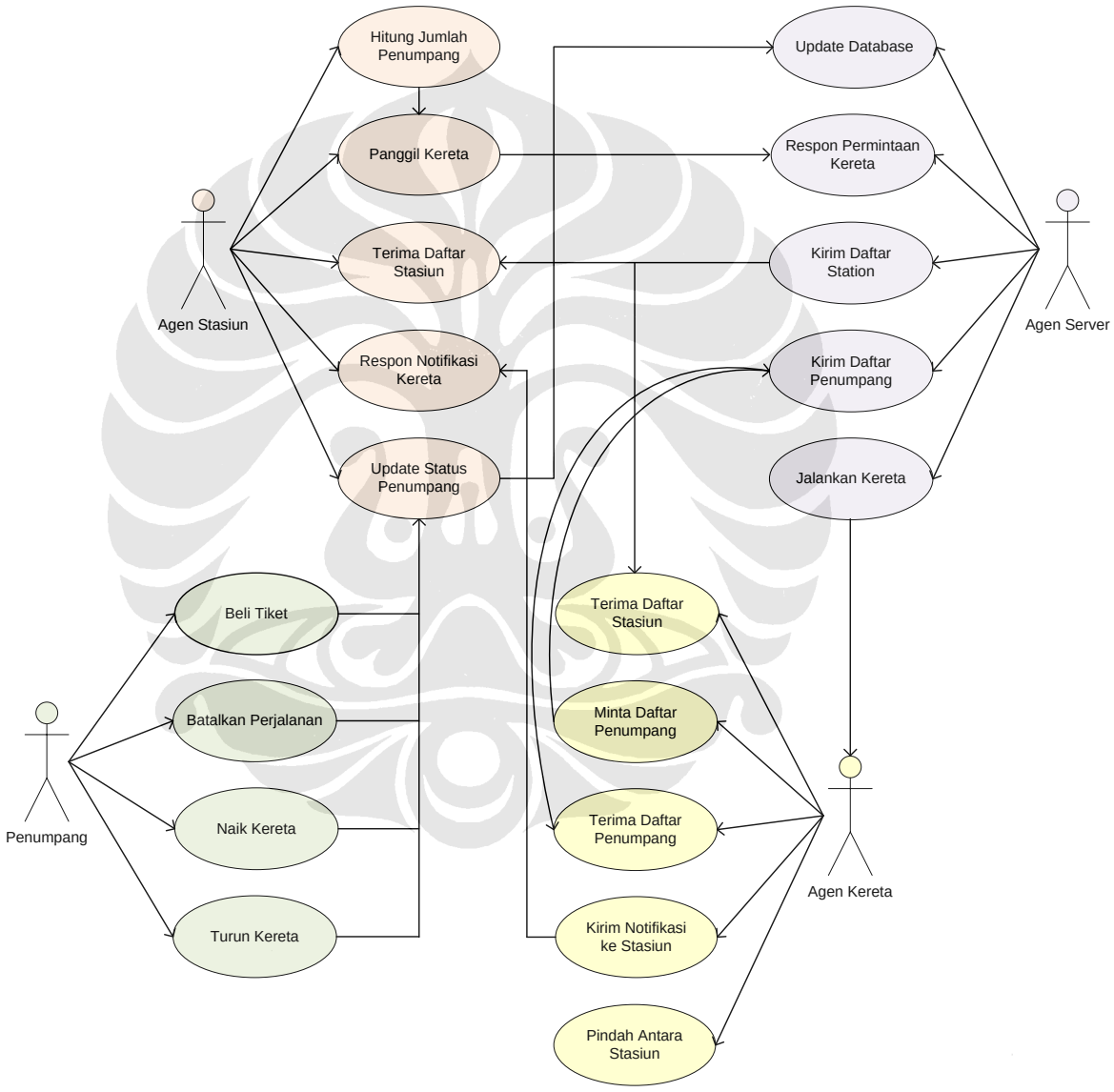


Gambar 4.1 Struktur *database* pada sistem monorail agen

- **TransactionTime**: waktu transaksi pembelian tiket oleh penumpang,
- **GetOn**: status penumpang apakah sudah naik ke kereta atau tidak, bernilai `true` atau `false`,
- **GetOnTime**: waktu kapan penumpang naik ke atas kereta,
- **GetOff**: status penumpang apakah sudah turun dari kereta atau tidak, bernilai `true` atau `false`,
- **GetOffTime**: waktu kapan penumpang turun dari kereta,
- **Train**: nomor kereta yang dinaiki penumpang,
- **Finish**: status penumpang apakah telah keluar dari menyelesaikan perjalanan atau tidak, bernilai `true` atau `false`,
- **CashBefore**: jumlah nominal uang yang terdapat pada *smart card* penumpang sebelum transaksi,
- **CashAfter**: jumlah nominal uang yang terdapat pada *smart card* penumpang setelah transaksi.

4.1.2 Implementasi Agen *Server*

Fungsi-fungsi yang di dukung oleh agen-agen pada sistem *monorail* agen ditunjukkan pada Gambar 4.2 dibawah ini. Gambar ini merupakan perluasan dari diagram *use case* pada Gambar 3.2 dengan menjadikan agen-agen yang terlibat sebagai aktor dengan tujuan memperjelas tugas atau fungsi masing-masing aktor.



Gambar 4.2 Diagram *use case* yang diperluas untuk agen-agen sistem *monorail* agen

Setelah koneksi ke *database* selesai dilakukan, program *server* akan melalukan inisiasi awal agen *server*, yakni registrasi layanan pada *yellow pages* yang difasilitasi oleh *Directory Facilitator* (DF), dan menampilkan GUI dari program. Pada dasarnya GUI program *server* hanya menampilkan isi dari *table* penumpang untuk direpresentasikan secara *real time* dan menampilkan notifikasi *event* yang sedang terjadi kepada administrator *server*. Melalui GUI, administrator *server* dapat menambahkan unit kereta.

Selanjutnya agen *server* akan mendeteksi identitas stasiun-stasiun sesuai dengan daftar yang terdapat pada *file* konfigurasi. Pendeteksian dilakukan secara sekuensial mulai dari stasiun pertama, kedua, dan seterusnya sampai stasiun terakhir. Pendeteksian setiap stasiun diimplementasikan dengan suatu `TickerBehaviour` yang mengeksekusi tugasnya setiap lima detik. Pewaktuan yang terlalu cepat mengakibatkan pembebanan sistem yang tinggi dan kurang efesien, sedangkan pewaktuan yang terlalu lama mengakibatkan sistem tidak optimal. Program *server* menyimpan identitas (*Agent Identifier* atau AID) agen stasiun dalam bentuk sebuah *array*. Setelah seluruh stasiun sukses terdeteksi, agen *server* selanjutnya mengirimkan *array* yang berisi identitas (AID) agen stasiun tersebut kepada setiap agen stasiun. Hal ini untuk menjamin bahwa sistem baru dapat berjalan jika seluruh agen telah mengetahui identitas agen-agen lainnya.

Tugas selanjutnya dari agen *server* adalah menghidupkan satu agen kereta pada kontainer yang sama dan mendeteksi identitasnya menggunakan `TickerBehaviour` pada *yellow pages*. Setelah terdeteksi selanjutnya agen *server* mengirimkan pesan notifikasi kepada agen kereta untuk memberitahukan AID-nya. Kemudian fase mobilitas agen kereta dimulai. Agen *server* akan mendeteksi jika terdapat penumpang di stasiun manapun, jika terdapat status **Finish** bernilai `false`, agen *server* akan mengirimkan pesan untuk menjalankan agen kereta. Selanjutnya tugas agen *server* adalah menerima/merespon pesan dan berinteraksi dengan agen-agen yang lain.

4.1.3 Implementasi Agen Stasiun

Saat program stasiun dijalankan semua fungsi transaksi penumpang belum dapat diakses (tombol di-*disable*) dan daftar stasiun tujuan belum ada. Setelah inisiasi awal agen stasiun, yaitu registrasi layanan ke *yellow pages* yang difasilitasi oleh *Directory Facilitator* dan menampilkan GUI, pekerjaan selanjutnya dari agen stasiun adalah menunggu. Setelah layanannya dideteksi oleh agen *server*, selanjutnya agen stasiun akan menerima pesan dari agen *server* yang memuat *array* AID agen-agen stasiun lainnya. Saat ini pula agen stasiun menyimpan AID dari agen *server*. Hal ini dilakukan untuk menjamin bahwa agen stasiun mengetahui dengan pasti identitas agen-agen lainnya. Setelah itu *update* terhadap GUI dilakukan, fungsi-fungsi transaksi diaktifkan (tombol di-*enable*) dan daftar stasiun tujuan ditampilkan. Selanjutnya program stasiun menunggu masukan dari pengguna stasiun. Mulai dari tahap ini pekerjaan dari agen stasiun adalah berinteraksi dengan agen-agen lainnya dan dengan pengguna.

4.1.4 Implementasi Agen Kereta

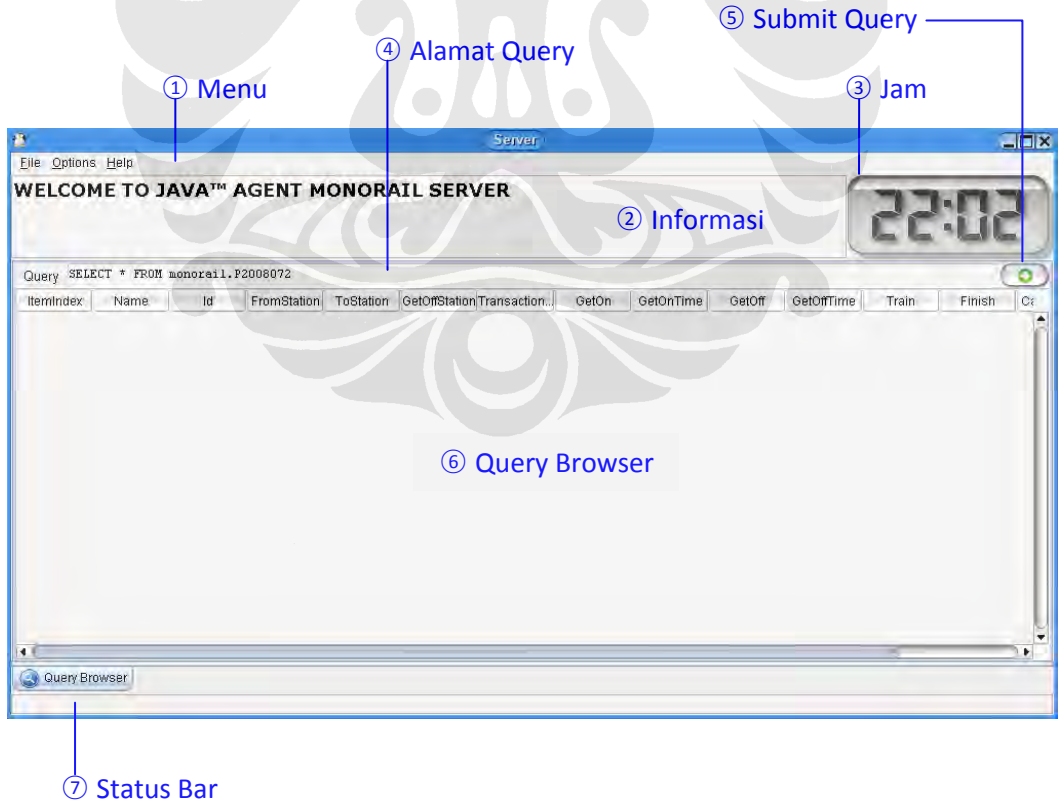
Agen kereta dihidupkan oleh agen *server*. Saat agen kereta hidup ia segera mendaftarkan layanannya ke *yellow pages* yang difasilitasi oleh *Directory Facilitator*. Agen kereta dijalankan oleh agen *server* untuk pertama kalinya saat ada penumpang di stasiun. Agen kereta bergerak antara kontainer yang satu ke kontainer yang lain sambil berinteraksi dengan agen stasiun dan agen *server*. Saat sampai pada suatu stasiun, ia akan me-*request* pesan ke agen *server* untuk mengetahui status penumpang di stasiun tersebut. Jika ada penumpang yang akan naik ke ataupun penumpang yang akan turun dari kereta pada stasiun tersebut, ia akan menjalankan `WakerBehaviour` sebagai penentu waktu singgah stasiun dan mengirim notifikasi kepada agen stasiun untuk memperbolehkan akses naik dan turun kereta, dan jika tidak maka ia akan bergerak ke stasiun selanjutnya. Sesuai skenario yang telah dipaparkan, pada stasiun akhir, agen kereta mewajibkan seluruh penumpang yang ada untuk turun, kemudian ia akan kembali ke agen *server* dan mati.

Pada simulasi, datangnya kereta ditunjukkan dengan munculnya GUI kereta jika terdapat penumpang yang akan naik atau turun pada stasiun tersebut. GUI kereta memuat daftar penumpang (ditampilkan dalam bentuk tombol-tombol berisi nama dan id penumpang) yang terdapat pada kereta serta menyediakan input untuk menurunkan dan menaikkan penumpang pada stasiun yang disinggahi.

4.1.5 Implementasi GUI

Implementasi GUI pada sistem *monorail* agen dilakukan pada program *server* dan program stasiun serta kereta untuk mensimulasikan keadaan sesungguhnya.

GUI dari program *server* pada sistem *monorail* agen ditunjukkan oleh Gambar 4.3. Komponen-komponen dari GUI program *server* berikut fungsinya ditunjukkan oleh Tabel 4.1.



Gambar 4.3 GUI program *server* dari sistem *monorail* agen

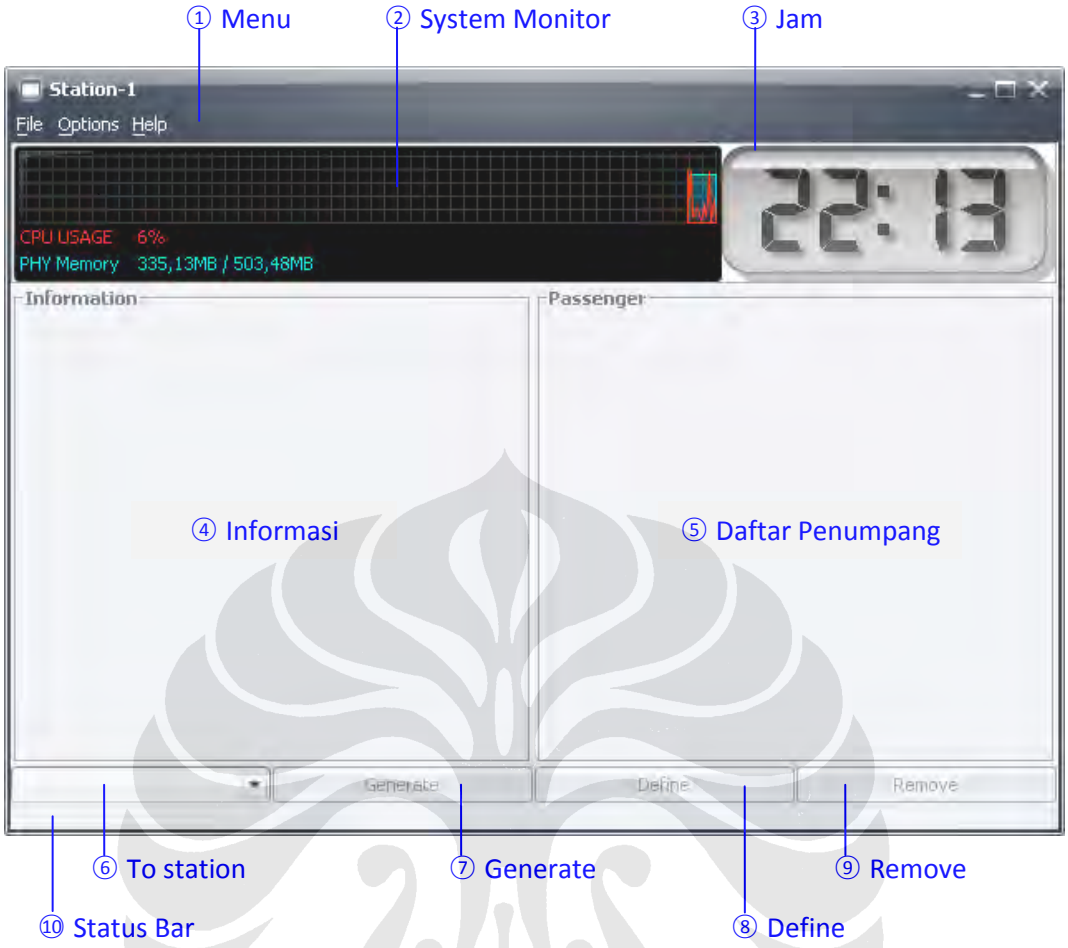
Tabel 4.1 Komponen-komponen utama GUI program *server* dari sistem *monorail* agen

Komponen	Class	Fungsi
1. Menu	`JMenu`	Menampilkan menu dari program, termasuk menu untuk menambahkan unit kereta, keluar dari program, mengubah tema, dan menampilkan tentang program
2. Informasi	`JTextArea`	Menampilkan *event* yang sedang terjadi
3. Jam	`LcdClockPanel`	Menampilkan waktu
4. Alamat Query	`JTextArea`	Menuliskan *syntax* MySQL untuk menampilkan data penumpang
5. Submit Query	`JButton`	Menyetujui alamat *query* yang telah dituliskan
6. Query Browser	`JTable`	Menampilkan data penumpang yang terdapat pada *database*
7. Status Bar	`JTextField`	Menampilkan pesan fungsi komponen-komponen GUI

GUI dari program stasiun pada sistem *monorail* agen ditunjukkan oleh Gambar 4.4. Komponen-komponen dari GUI program stasiun berikut fungsinya ditunjukkan oleh Tabel 4.2.

Tabel 4.2 Komponen-komponen utama GUI program stasiun dari sistem *monorail* agen

Komponen	Class	Fungsi
1. Menu	`JMenu`	Menampilkan menu dari program termasuk menu untuk mematikan program, mengubah tema, dan menampilkan tentang program
2. System Monitor	`SystemMonitor`	Memonitor penggunaan sumber daya komputer
3. Jam	`LcdClockPanel`	Menampilkan waktu
4. Informasi	`JTextArea`	Menampilkan *log event* yang terjadi
5. Daftar Penumpang	`JTextArea`	Menampilkan daftar penumpang di stasiun
6. To Station	`JComboBox`	Menampilkan pilihan stasiun tujuan
7. Generate	`JButton`	Membuat penumpang secara acak
8. Define	`JButton`	Membuat penumpang secara spesifik (Name, Id, dan Cash)
9. Remove	`JButton`	Membatalkan perjalanan penumpang terakhir
10. Status Bar	`JTextField`	Menampilkan pesan fungsi komponen-komponen GUI



Gambar 4.4 GUI program stasiun dari sistem *monorail* agen

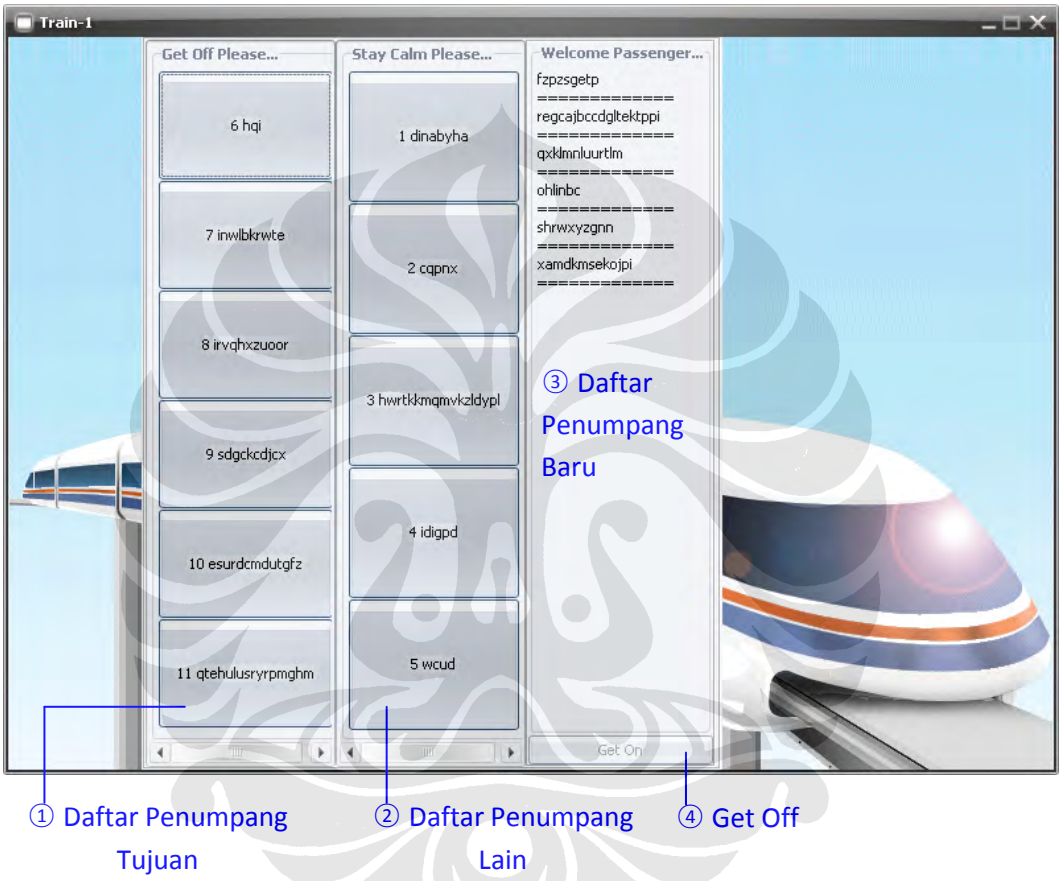
GUI dari kereta pada sistem *monorail* agen ditunjukkan oleh Gambar 4.5. Komponen-komponen dari GUI kereta berikut fungsinya ditunjukkan oleh Tabel 4.3.

Tabel 4.3 Komponen-komponen utama GUI kereta dari sistem *monorail* agen

Komponen	Class	Fungsi
1. Daftar Penumpang Tujuan	`JPanel`	Menampilkan tombol-tombol untuk menurunkan penumpang yang stasiun tujuannya sama dengan stasiun singgah
2. Daftar Penumpang Lain	`JPanel`	Menampilkan tombol-tombol untuk menurunkan penumpang yang stasiun tujuannya tidak sama dengan stasiun singgah

Tabel 4.3 Komponen-komponen utama GUI kereta dari sistem *monorail* agen (lanjutan)

Komponen	Class	Fungsi
3. Daftar Penumpang Baru	`JTextArea`	Menampilkan daftar penumpang yang baru menaiki kereta
4. Get Off	`JButton`	Menaikkan penumpang



Gambar 4.5 GUI kereta dari sistem *monorail* agen

4.2 PENGUJIAN

Pengujian terhadap sistem *monorail* agen dilakukan dalam bentuk pengujian fungsionalitas modul-modul dan komponen-komponen pada GUI. Hasil dari pengujian tersebut disajikan dalam Tabel 4.4.

Tabel 4.4 Pengujian fungsionalitas modul-modul dan komponen-komponen GUI sistem *monorail* agen

Kelompok	Fungsi	Hasil
Program *Server*	Mampu melakukan pembacaan terhadap *file* konfigurasi	Ya
	Mampu mendeteksi seluruh stasiun	Ya
	Mampu menghidupkan dan menjalankan agen kereta	Ya
	Mampu membaca dan menulis *database* penumpang	Ya
	Komponen Jam berfungsi	Ya
	Komponen Alamat Query dan Submit Query berfungsi	Ya
	Mampu menampilkan isi *database* pada GUI secara dinamis	Ya
	Mampu menampilkan informasi setiap *event* yang terjadi	Ya
	Menu untuk menambah unit kereta berfungsi	Ya
	Menu terminasi program berfungsi	Ya
	Menu pengubahan tema berfungsi	Ya
	Menu untuk menampilkan tentang program berfungsi	Ya
Program Stasiun	Mampu menambah penumpang secara acak	Ya
	Mampu menambah penumpang secara spesifik	Ya
	Mampu membatalkan perjalanan	Ya
	Pilihan stasiun tujuan ditampilkan dengan benar	Ya
	Mampu menampilkan *log event* yang terjadi	Ya
	Menampilkan daftar penumpang dengan benar	Ya
	Menu terminasi program berfungsi	Ya
	Menu pengubahan tema berfungsi	Ya
	Menu untuk menampilkan tentang program berfungsi	Ya
	Komponen System Monitor berfungsi	Ya
	Komponen Jam berfungsi	Ya
Program Kereta	Agen mampu hidup pada waktu seharusnya	Ya
	Agen kereta mampu bergerak antara stasiun sesuai rute	Ya
	Mampu menampilkan GUI pada stasiun yang benar	Ya
	Mampu melewati stasiun yang seharusnya tidak disinggahi	Ya
	Pengelompokkan daftar penumpang stasiun tujuan berfungsi	Ya
	Pengelompokkan daftar penumpang stasiun lain berfungsi	Ya
	Menampilkan penumpang yang baru menaiki kereta	Ya
	Mampu menaikkan penumpang	Ya
	Mampu menurunkan penumpang	Ya
	Menampilkan daftar penumpang yang baru menaiki kereta	Ya
	Menunggu seluruh penumpang turun pada stasiun akhir	Ya
	Agen kereta mati setelah menyelesaikan rutenya	Ya

4.3 EVALUASI KINERJA

Evaluasi kinerja sistem *monorail* agen dilakukan dengan analisis *delay*, pengukuran lalu lintas data dan analisis penggunaan sumber daya jaringan, pengukuran konsumsi sistem memori dan waktu CPU saat *idle*.

4.3.1 Analisis *Delay*

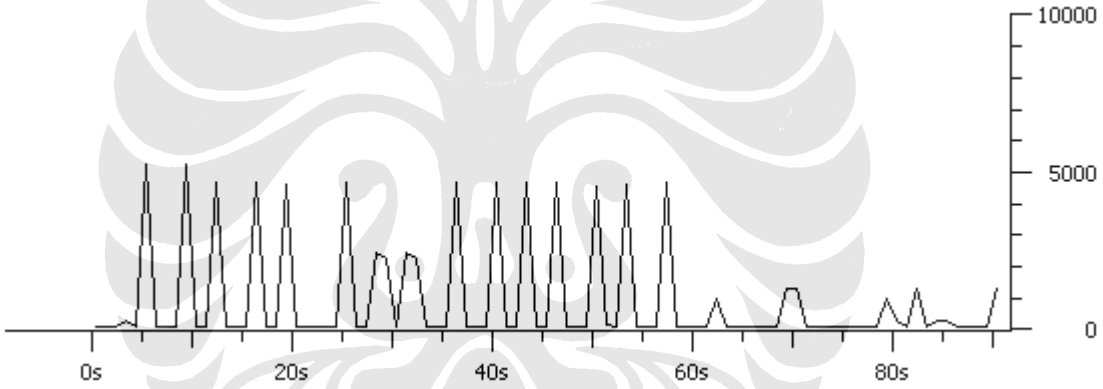
Analisis *delay* dilakukan dengan melihat jedah waktu yang diperlukan saat tombol pada program stasiun ditekan sampai ada notifikasi baik berupa tampilan informasi maupun perubahan atribut penumpang pada *database* di program *server*. Dari pengujian yang dilakukan dalam suatu jaringan lokal dengan menggunakan tiga buah komputer, input pada program stasiun dilakukan setiap detik selama satu menit, menunjukkan tidak terdapat *delay* yang berarti. Pengukuran *delay* secara kuantitatif cukup sulit dilakukan mengingat komunikasi antar-agen (pengiriman pesan) kebanyakan bersifat satu arah. *Delay* terutama diakibatkan pewaktuan pada program misalnya karena penggunaan `TickerBehaviour`. Dalam suatu jaringan lokal, penerapan sistem berbasis agen perlu memperhitungkan faktor pewaktuan yang tepat. Dengan pengaturan pewaktuan secara tepat, *delay* tidaklah menjadi sebuah isu.

4.3.2 Lalu Lintas Data

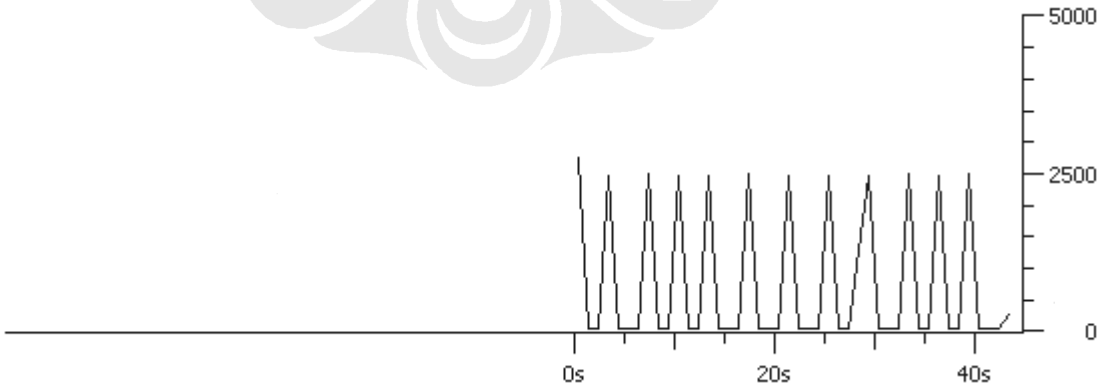
Perhitungan lalu lintas data antara program *server* dan program stasiun dilakukan dengan konfigurasi jaringan yang sama seperti pada analisis *delay*. Pengukuran besar aliran data yang dikirimkan antara program *server* dan program stasiun dilakukan dengan menangkap paket-paket data yang dikirimkan oleh ketiga agen menggunakan aplikasi Wireshark 0.99.8 pada *host* program *server*.

Gambar 4.6 menunjukkan grafik besarnya data yang dikirimkan saat terdapat penumpang baru. Besarnya data yang dikirimkan untuk memulai transaksi adalah ±4500 byte. Gambar 4.7 menunjukkan grafik besarnya data yang

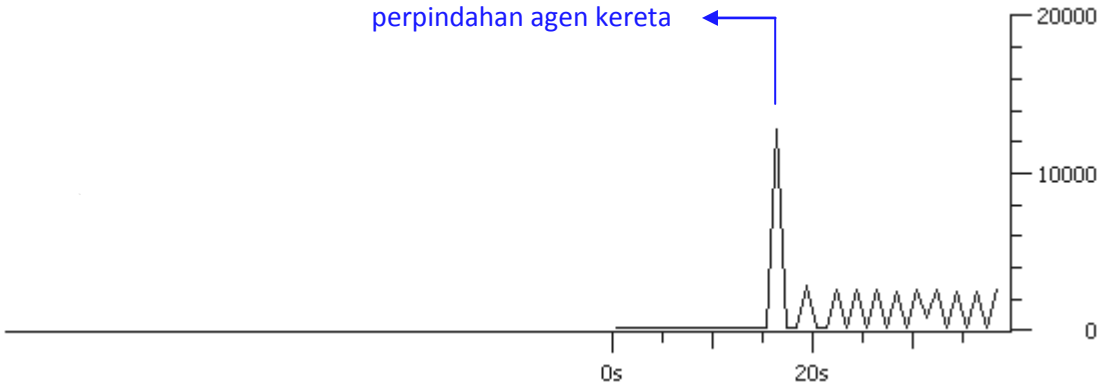
dikirimkan saat terdapat pembatalan perjalanan oleh penumpang, yakni sebesar ±2500 byte. Gambar 4.8 menunjukkan grafik besarnya data yang dikirimkan saat terdapat penumpang yang naik ke kereta, yakni sebesar ±2400 byte. Gambar 4.9 menunjukkan grafik besarnya data yang dikirimkan saat terdapat penumpang yang turun dari kereta, yakni sebesar ±2500 byte. Lonjakan besar data pada awal grafik di Gambar 4.8 dan 4.9 sebesar ±12000 byte merupakan besar data perpindahan agen kereta yang mencakup perpindahan kode, data, dan keadaan. Pada simulasi terdapat tambahan paket data yang berisi daftar penumpang yang terdapat pada kereta. Paket data ini dikirim oleh agen *server* kepada agen kereta pada setiap stasiun yang disinggahinya untuk menampilkan daftar penumpang dalam kereta pada GUI kereta. Besar paket data ini adalah ±22000 byte, seperti yang ditunjukkan pada Gambar 4.9.



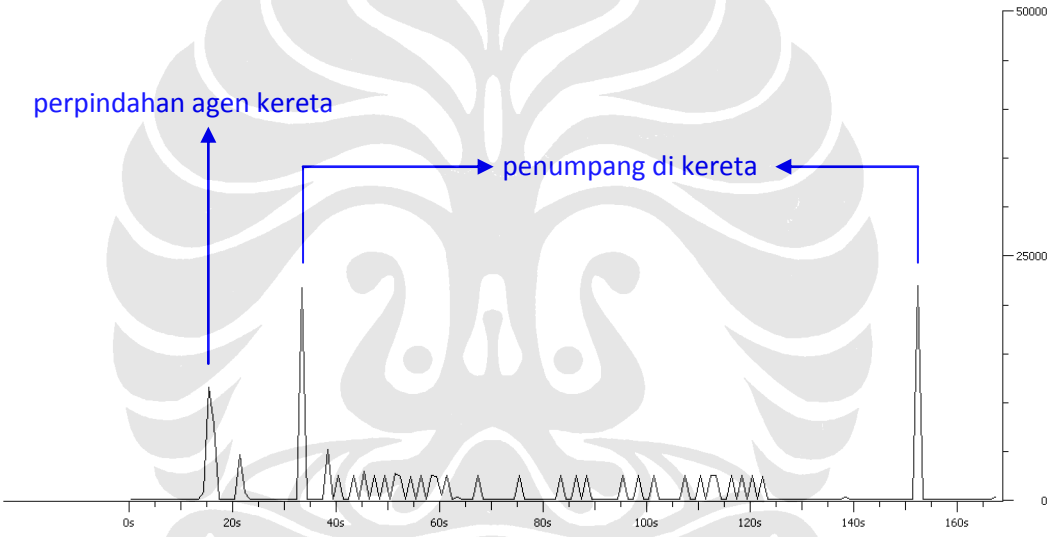
Gambar 4.6 Grafik besarnya data yang dikirimkan saat terdapat penumpang baru



Gambar 4.7 Grafik besarnya data yang dikirimkan saat terdapat pembatalan perjalanan oleh penumpang



Gambar 4.8 Grafik besarnya data yang dikirimkan saat terdapat penumpang yang naik ke kereta



Gambar 4.9 Grafik besarnya data yang dikirimkan saat terdapat penumpang yang turun dari kereta

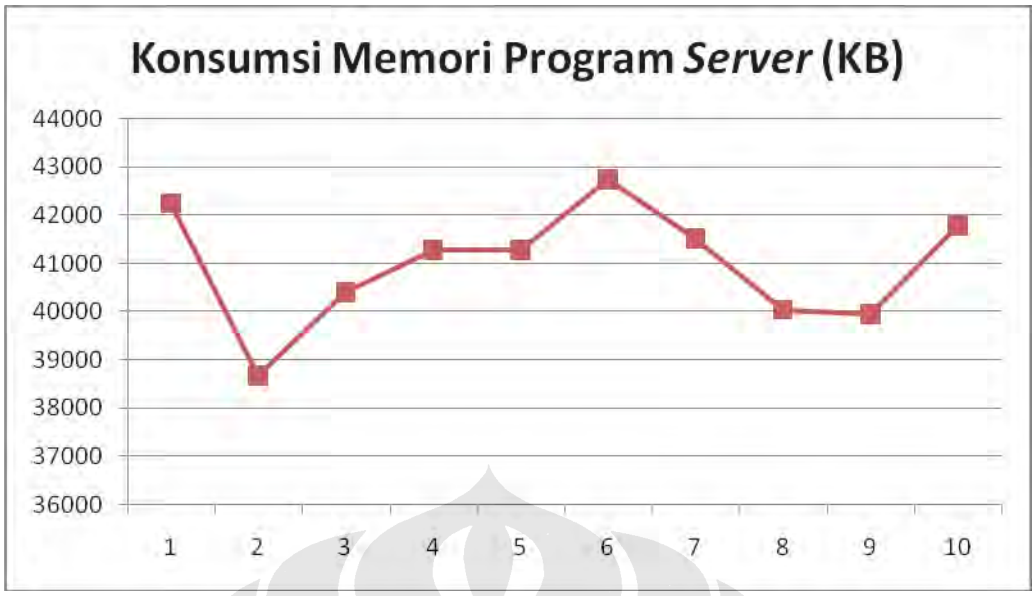
Nilai-nilai yang diperoleh dari hasil pengukuran lalu lintas data pada sistem *monorail* agen (2400 – 22000 byte) terbilang sangat kecil dibandingkan kapasitas jaringan lokal yang umumnya memiliki kapasitas minimum 100 Mbps. Dari hasil pengukuran ini dapat dikatakan bahwa sistem *monorail* agen menggunakan sumber daya jaringan yang relatif kecil.

4.3.3 Konsumsi Sistem Memori dan Waktu CPU

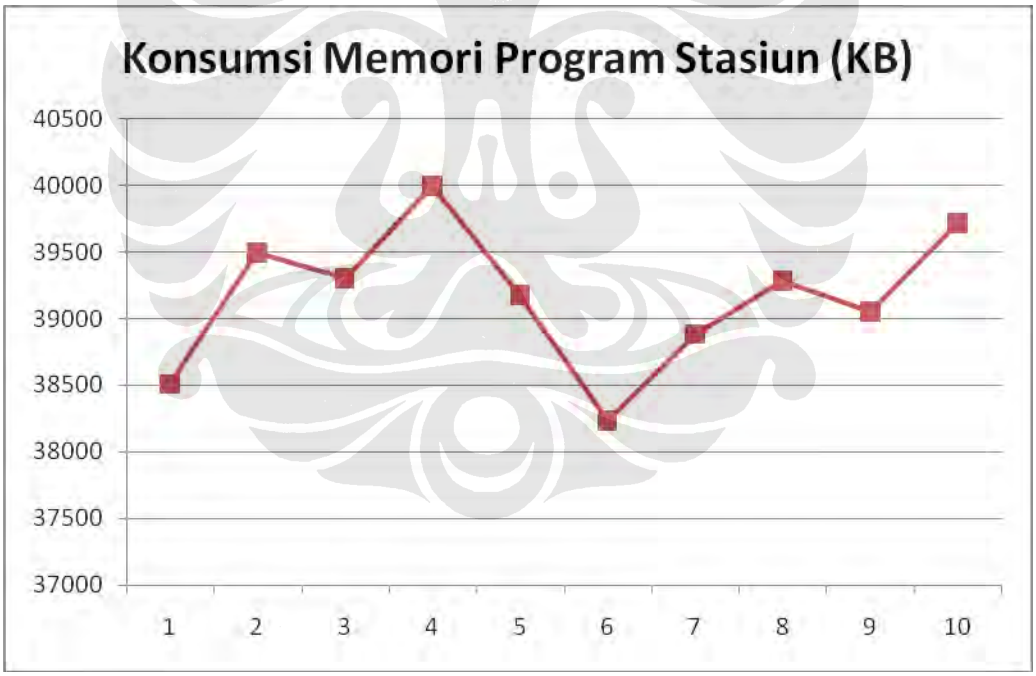
Konsumsi sistem memori diperoleh dengan mengamati nilainya sebelum dan sesudah program dijalankan. Dari pengamatan sebanyak sepuluh kali baik untuk program *server* maupun program stasiun diperoleh rata-rata konsumsi sebesar 40.999,6 kilobyte untuk program *server* dan 39.199,4 kilobyte untuk program stasiun. Tabel 4.5, Gambar 4.10, dan Gambar 4.11 menunjukkan konsumsi memori program *server* dan program stasiun secara detail. Pengukuran dilakukan pada komputer dengan sistem memori sebesar 512 megabyte dan *clock* sebesar 2,4 GHz.

Tabel 4.5 Konsumsi memori oleh program *server* dan program stasiun dari sistem *monorail* agen

Program	Konsumsi Memori (KB)			
	Sebelum	Sesudah	Selisih	Rata-rata
Server	262916	305164	42248	
	263020	301700	38680	
	261468	301884	40416	
	261456	302748	41292	
	261424	302712	41288	
	260952	303700	42748	
	260912	302432	41520	
	261020	301068	40048	
	261052	301016	39964	
	260972	302764	41792	40999,6
Stasiun	260400	298912	38512	
	260504	300000	39496	
	260360	299664	39304	
	260384	300380	39996	
	261800	300980	39180	
	261908	300144	38236	
	261676	300560	38884	
	261532	300816	39284	
	261372	300428	39056	
	261272	300988	39716	39166,4



Gambar 4.10 Konsumsi memori oleh program *server* dari sistem *monorail* agen



Gambar 4.11 Konsumsi memori oleh program stasiun dari sistem *monorail* agen

Pengukuran penggunaan waktu CPU baik oleh program *server* maupun program stasiun dilakukan pada lingkungan pengukuran yang sama seperti pada

pengukuran besar konsumsi sistem memori. Pengukuran penggunaan waktu CPU dilakukan saat aplikasi sedang *idle* atau tidak terdapat input apapun dari pengguna. Hasil pengamatan menunjukkan penggunaan waktu CPU oleh program *server* adalah <20% dan oleh program stasiun adalah <40%. Adanya penggunaan waktu saat *idle* oleh kedua program disebabkan oleh penggunaan `CyclicBehaviour` dan `TickerBehaviour` pada agen.

4.4 PEKERJAAN MENDATANG

Beberapa kekurangan pada sistem *monorail* agen mencakup otomatisasi hidupnya agen kereta baru saat ada panggilan oleh agen stasiun serta isu mobilitas untuk banyak agen kereta. Untuk itu pada pengembangan selanjutnya kekurangan-kekurangan tersebut perlu ditelaah ulang dan diperbaiki. Antarmuka program dapat dikembangkan dengan fitur penulisan *file*, mengakomodasi pilihan, menampilkan bantuan dan dokumentasi dari program.

Sistem yang telah diselesaikan pada penelitian ini memanfaatkan sumber daya terdistribusi dalam lingkungan komunikasi antar-agen intra-*platform* JADE. Pengembangan sistem dapat dilakukan pada lingkungan komunikasi antar-agen inter-*platform*. Selain itu, dalam penelitian ini rute kereta yang diskenariokan bersifat sederhana dan tanpa *looping* dari agen kereta. Pengembangan lain yang dapat dilakukan adalah implementasi sistem pada skenario yang lebih kompleks.

BAB 5

KESIMPULAN

Setelah melakukan studi literatur, analisis, perancangan, implementasi, pengujian, serta evaluasi kinerja sistem *monorail* berbasis teknologi multi-agen, dapat ditarik beberapa kesimpulan sebagai berikut.

1. Pada sistem *monorail* agen, *delay* tidak menjadi sebuah isu yang penting, penggunaan sumber daya jaringan relatif kecil (2400 – 22000 byte), konsumsi memori sebesar ±40 MB, dan adanya penggunaan waktu CPU saat *idle* (sampai dengan <40%).

2. Pengaturan pewaktuan terhadap pekerjaan agen menjadi isu yang penting dalam membangun aplikasi berbasis agen. Aspek mobilitas dan reliabilitas agen menjadi isu yang penting pada sistem *monorail* agen.

3. *Ontology* sebaiknya digunakan untuk komunikasi antaragen dengan dengan entitas yang kompleks sebagai isi pesan. Jika isi pesan relatif sederhana strukturnya, penggunaan Java *serialization* akan memberikan kemudahan. Penggunaan SL *language* akan memudahkan proses *debugging* dan *testing* aplikasi.

4. Pengerjaan proyek ini belum menggali lebih dalam dan lengkap kemampuan yang ditawarkan oleh teknologi multi-agen. Meskipun demikian penelitian ini telah mampu mengaplikasikan teknologi multi-agen.

5. Penggunaan JADE sebagai *platform* untuk membangun sistem multi-agen memberikan kemudahan dengan dukungan dokumentasi dan kumpulan pustaka Java yang lengkap. Sistem multi-agen cocok menjadi model pengembangan aplikasi cerdas yang memanfaatkan sumber daya terdistribusi.

DAFTAR ACUAN

[1] MySQL AB (2008). *MySQL 5.0 Reference Manual.* Diakses 6 Juni 2008. http://dev.mysql.com/doc/refman/5.0/en/.

[2] Fabio Bellifemine, Giovanni Caire, Dominic Greenwood, *developing multi-agent system with JADE.* (England: John Wiley & Sons Ltd, 2007)

[3] Magid Nikraz, Giovanni Caire, Parisa A. Bahri, "A Methodology for the Analysis and Design of Multi-Agent System Using JADE".

[4] Joko Nurjadi, "Mengenal Lebih Jauh MySQL". *PCMedia,* Maret 2007: hal. 74 - 76.

[5] Elhadi Shakshuki, Yang Jun, "Multi-agent Development Toolkits: An Evaluation". (© Springer-Verlag Berlin Heidelberg 2004)

[6] Pavel Vrba, "JAVA-Based Agent Platform Evaluation". (© Springer-Verlag Berlin Heidelberg 2003)

[7] Joan Ametller, Sergi Robles, Joan Borrell, "Agent Migration over FIPA ACL Messages". (© Springer-Verlag Berlin Heidelberg 2003)

[8] Rainer Unland, Matthias Klusch, Monique Calisti (ed.), *Software Agent-Based Applications, Platforms and Development Kits*. (Germany: Birkhäuser Verlag, 2005)

[9] Federico Bergenti, Marie-Pierre Gleizes, Franco Zambonelli (ed.), *METHODOLOGIES AND SOFTWARE ENGINEERING FOR AGENT SYSTEMS The Agent-Oriented Software Engineering Handbook.* (United States of America: Kluwer Academic Publishers, 2004)

DAFTAR PUSTAKA

Bellifemine, Fabio, Agostino Poggi, Giovanni Rimassa, "Developing Multi-Agent System with JADE". (© Springer-Verlag Berlin Heidelberg 2003)

Bellifemine, Fabio, et al. "JADE Administrator's Guide". (TILAB, 2007)

Bellifemine, Fabio, et al., "The JADE Platform and Experiences with Mobile MAS Applications".

Bellifemine, Fabio. "Java Agent Development Framework". (TILAB, 2001)

Caire, Giovanni. "JADE Programming for Beginners". (TILAB, 2003)

Henderson-Sellers, Brian, Paolo Giorgini, *Agent-Oriented Methodologies*. (Idea Group Publishing, 2005)

Sun Microsystems Inc. (1998). *JDBC 2.0 Standard Extension API.*

Tamma, Valentina, et al. (ed), *Ontologies for Agents: Theory and Experiences.* (Germany: Birkhäuser Verlag, 2005)