System Design Document

JAMES Project
15-413 Software Engineering
Fall 1997
Carnegie Mellon University
Pittsburgh, PA 15213

Revision History:
10/13/97 Bernd Bruegge. Created.
12/01/97 Vincent Mak. Revised.
12/01/97 Mike Samuel. Revised.
12/08/97 Allen Dutoit. Fixed figures and tables.

Preface:
This document addresses the requirements of the JAMES system. The intended audience for this document are the analysts and developers of the project.

Target Audience:
Analysts, Developers

JAMES Members:
Gordon Cheng, Li-Lun Cheng, Christopher Chiappa, Arjun Cholkar, Uhyon Chung, Aveek Datta, John Doe, Phillip Ezolt, Eric Farng, William Ferry, Sang Won Ham, Kevin Hamlen, Pradip Hari, Yenni Kwek, Tze Bin Loh, Alexander Lozupone, Christopher Lumb, Vincent Mak, Darren Mauro,
Table of Content:

1. Goals and Trade-offs (Architecture team)
2. System Decomposition (Architecture team)
3. Concurrency Identification (Travel team)
4. Hardware/Software Mapping (Simulation team)
5. Data Management (Maintenance team)
6. Global Resource Handling (Architecture team)
7. Software Control Implementation (VIP team)
8. Boundary Conditions (Logbook team)
9. Design Rationale (All teams)

List of Diagrams

Figure 2.1.1 - JAMES System Architecture
Figure 2.1.2 - HCI Architecture
Figure 4.2.1 - OSI Vehicle Model
Figure 4.2.2 - Logical Connectivity Diagram
Figure 4.2.3 - Physical Connectivity Diagram
Figure 6.1.1 - User Diagram for Authentication Scheme
Figure 7.1.1 - Maintenance Decomposition
Figure 7.1.2 - Maintenance Subsystem Layered Decomposition

1. Goals and Trade-offs

The goal of the Java Architecture for Mobile Extended Services (JAMES) System is to design a system for the automotive industry, Mercedes Benz in particular, for value-added computerized features to be included with their automobiles. For additional background information on the JAMES system, please look at the Problem Statement Document. Many of the features that will be supported by JAMES are discussed in detail in the Requirements Analysis Document (RAD).

This is one of the first times the industry is trying develop such a system so there are still many unknowns. Industry and consumer tastes are constantly changing so this must be taken into account when designing the JAMES system. JAMES must be made extensible, scalable and platform independent so that it can remain flexible enough for any changes in the future. This is the primary reason why Java has been chosen as the programming language for developing JAMES.

JAMES is modeled as a set of assistants communicating via a common open infrastructure. This will consists of a reusable class library with a set of API's as well as a set of assistants (Travel, Logbook, Maintenance and VIP) to demonstrate an early model of some of its capabilities. The key to the project is the class library and API's because it will speed up the development of other smart card assistants for the car. Once the class libraries have been designed and the API's defined, many different types of assistants can be written to work with the JAMES system. Of course, this will also require that the class libraries and API's to be extensible.

The conceptual prototype of the JAMES system must be delivered by the end of the semester. Much of the hardware technology, specifically the Java Smart Card and the card reader, which we need to make JAMES into a working commercial system, is still in the experimental stages. Due to the time constraint and the limited resources of the students in the course, we will focus on rapid prototyping instead of completeness of functionality. Our hope is to produce class libraries and API's that will allow additional functionality to be added in the future when the resources are available. The assistants that we create now for JAMES are only meant to be a demonstration of things to come.
For the conceptual prototype, portability of the system must take priority over efficiency. JAMES is still an experimental system so it must remain flexible and portable to allow for modifications and new ideas. Part of the requirements is that JAMES be extensible, scalable and platform independent so portability is a must. Once portability has been achieved and the system is ready to become a practical commercial product then efficiency can become a priority.

Since JAMES is meant to be used by regular consumers, we must stress usability of the system over functionality. JAMES must be easy to understand and use. We do not want to overwhelm the user with features to the point where the system becomes unusable or an inconvenience. Reliability is also very important for the system because it is something that the user will come to depend on. Cost is not an important issue for the conceptual prototype but will become one once we try to make this into a commercial product.

To summarize, this document describes the architecture of a conceptual prototype of the JAMES system with simple functionality that can eventually be used as the basis for development of assistants for the automotive industry.

2. System Decomposition

2.1 System Decomposition

The JAMES system consists of eleven subsystems. They are Travel, VIP, Logbook, Maintenance, Vehicle, Notification Service, Authentication Service, Name Service, Database Service, Trip Planning Service and User Interface. Travel, VIP, Logbook and Maintenance represent assistants. The assistants are not dependent on each other and are not required to run at the same time. The Vehicle is an abstract representation of the physical car or of the simulator. The Vehicle will act as a provider of event and state information to the assistants. The Vehicle has information about the state of the car. It also owns resources such as GPS and the Navigation Subsystem. The assistants can only access this information through the Vehicle. The Notification Service is a tool for letting individual assistants know about events in the vehicle. The clients can subscribe to events through the Notification Service. The Name Service will provide information about physical locations of objects to any other subsystem. The Authentication Service will be called for each assistant subsystem to enforce authorized access of all resources. The Database Service represents the static storage of the system. The Trip Planning Service is a service used by the Travel Subsystem to create a trip. User Interface will provide functionality to accommodate user input and output. It will act as an intermediary between the user and the system.

The subsystems communicate with each other through an implementation of a remote method invocation. Subsystems don’t have to be running on the same machine and can communicate with each other remotely.
Assistants

Travel
The Travel Assistant will allow the user to plan a trip on a PC at home or in the office and save it on the SmartCard. The saved trip can then be used for guided driving on the road. The Travel Assistant will display
the current position of the vehicle on a map on a digital display. It will be able to warn the driver if he/she went off the path of the pre planned trip. The Travel Assistant will also accommodate the user in dynamic addition or deletions of steps of the trip.

**VIP**
The VIP subsystem saves preferences for various components in the vehicle (examples include seat positions, climate control, etc.). The user can save preferences to the smartcard, or allow the subsystem to automatically save preferences. The same is true of settings. At any time the user can force his/her preferences to be set, or he/she can set a preference requiring the subsystem to assert his/her preferences as soon as the subsystem initializes.

**Logbook**
The Logbook Assistant allows the driver or a fleet manager for the vehicle to track business and personal use of the vehicle and automates filing tax or reimbursement forms for business use.

**Maintenance**
The Maintenance Assistant allows a history of maintenance events, such as repairs, oil changes, etc., be stored electronically. This history can be viewed and edited by the appropriate parties. In addition, events can be 'recommended', such as a change of seat belts as decided by Mercedes or an oil change, as decided by mileage.

**Bonus System**
Will keep track of bonus points from preferred service providers. It is scheduled to be implemented next design cycle, January-March 1998.

**Services**

**Authentication Service**
The Authentication Service is responsible for checking if the particular user is authorized to perform an operation. Every user request from any assistant subsystem will be checked for access rights by the Authentication Service.

**Notification Service**
The Notification Service will notify assistants when an event has occurred. One example of an event is turning off the ignition in the vehicle. Another might be an update on the current location from the GPS receiver. The Notification Service will only notify the other subsystems of the events they are subscribed to. If the VIP is the only subsystem subscribing to the turning ignition off in the car, then it will be notified of it. The Travel and Logbook might both subscribe to the update location event and they will notified of it. We will be using an implementation of the the Notification Service from the OWL system.

**Event Channel Allocation**
An event channel is a channel onto which a subsystem can push a listener which calls a specified method when another subsystem pushes an event it is interested in, onto that channel. Events contain the event name and an array of arguments.

There will be a total of nine event channels to service the four subsystems currently in development and the HCI subsystem.

Each assistant has two dedicated channels, one for incoming events that it needs to monitor, and one for events it generates that are of interest to multiple subsystems. There is also a single channel for subsystems to register their interfaces with the HCI subsystem.

**Event Channel Naming Convention**
Channel dedicated to particular subsystems have names that start with the subsystem name followed by "In" or "Out".
Table 2.1.2 - Event Channels

<table>
<thead>
<tr>
<th>Channel</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogbookIn</td>
<td>Events generated by user from interface</td>
</tr>
<tr>
<td>LogBookOut</td>
<td></td>
</tr>
<tr>
<td>MaintenanceIn</td>
<td></td>
</tr>
<tr>
<td>MaintenanceOut</td>
<td></td>
</tr>
<tr>
<td>TravelIn</td>
<td>Events generated from interface or by associated hardware (GPS)</td>
</tr>
<tr>
<td>TravelOut</td>
<td>Periodic updates of trip status.</td>
</tr>
<tr>
<td>VIPIn</td>
<td>Events generated by user from interface</td>
</tr>
<tr>
<td>VIPOut</td>
<td>Notification of success/failure of user requests</td>
</tr>
<tr>
<td>Vehicle</td>
<td>Alert subsystems of change in operation mode (drive, park, etc.), and vehicle instrument changes</td>
</tr>
<tr>
<td>HCI</td>
<td>Allows subsystems to register interfaces</td>
</tr>
</tbody>
</table>

Name Service
The Name Service provides information about where a particular object resides physically. Because our objects may not be running on the same computer, the Name Service is a necessary tool for remote methods. When objects on the network become available, they notify the Name Service of their name and location. The Name Service is able to bind the object reference to the location of the object. When clients invoke the object, the Name Service is able to provide them with the physical location of the requested object.

Database Service
The Database Service will provide subsystems with access to static storage of data. It is needed for the Maintenance Subsystem in order to store maintenance history. The Travel Subsystem will need to store planned trips.

Trip Planning Service
The Trip Planning Service will create a trip based on user requirements. It will be used by the Travel Subsystem.

Vehicle
The vehicle subsystem provides access to physical components such as the seat and mirror, and to the state of the vehicle such as speed or position at a level of abstraction that hides the specific implementation of the components from the assistants.

During development, the vehicle subsystem provides an interface to two simulators: AIM/CANalyzer and SA/RT.

- AIM provides access to the vehicle components including: seat, mirror, steering wheel, air conditioning, added accessories, dashboard meters (speedometer, odometer, date, clock), and location and heading information from GPS.
- SA/RT allows the vehicle subsystem to provide access to the seat.

The vehicle subsystem also provides interfaces to

- the card reader
- information on the operation mode of the vehicle
- the vehicle model and serial number
- information on the user's access rights

**User Interface**

Users interact with JAMES assistants through the JAMES User Interface subsystem. The User Interface subsystem is designed to be used with multiple types of hardware, and supporting multi-modal (i.e., visual, speech-based) interaction. In addition, the User Interface subsystem should be capable of interfacing with a wide variety of JAMES assistants.

The User Interface subsystem should:

- Make intelligent use of the "mix" of hardware systems in the vehicle to provide the most sensible multi-modal experience for the user
- Prevent the individual JAMES assistant subsystems from dealing with details of user interfaces (abstraction)
- Provide an architecture that is open to new types of user interfaces (e.g., a cellular phone, new types of speech hardware)

**Subsystem Architecture**

The system consists of four layers: Interface Components Routing Event Handling Assistant Subsystems

---

**Figure 2.1.3 - HCI Architecture**

- **Speech Recognition Interface Component**
- **Speech Synthesis Interface Component**
- **JAVA AWT Visual Interface Component**
- **Cellular Phone Interface Component**

The **Interface Components** layer includes the modules with which a user can interact. These may encompass multiple modalities including Java AWT graphic interfaces, speech interfaces, and other modules (for example, cellular phones).
The **Routing Layer** will be responsible for making sure the proper event managers and interface components communicate. This layer may be implemented through the Java 1.1 event handling mechanism, or by some custom-written event-handler (such as the modified OWL notification system).

The **Event Handler layer** manages communications between the Interface Components and the Assistant Subsystems. At this layer, the various assistants should have the capability to make decisions based on the components that are/are not available. For example, if both a speech synthesis interface component and a Java AWT graphical interface component are both available, the event handler will be able to decide that a particular piece of information should go to one or both interfaces. So, if a user requests information from the maintenance subsystem, the event handler layer may choose to convey the information only on the visual display, and not have it read aloud.

The **Assistant Subsystems** provide a set of methods that may be called by the event handler. These assistants should have no information about what modality (i.e., speech, graphical, text-based) of interface components are available. The Assistant Subsystems simply respond to requests from the event handlers for information by sending messages containing the requested information. The Assistant Subsystems may also send messages to the event handlers to convey information generated within the assistant (as opposed to information delivered in response to a request).

**The Interface Components**

Inside each of the interface components (such as the speech recognition component) resides some assistant-specific data. For example, the speech recognition interface component needs to be aware of various phrases it should recognize for the VIP assistant. Each assistant should provide this information as part of its interface layer.

---

<table>
<thead>
<tr>
<th>Subsystem A</th>
<th>Subsystem B</th>
<th>Relationship (A to B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>All assistants</td>
<td>Vehicle</td>
<td>client/server</td>
</tr>
<tr>
<td>All assistants</td>
<td>Authentication Service</td>
<td>client/server</td>
</tr>
<tr>
<td>All assistants</td>
<td>Notification Service</td>
<td>peer to peer</td>
</tr>
<tr>
<td>All subsystems</td>
<td>Name Service</td>
<td>client/server</td>
</tr>
<tr>
<td>All assistants</td>
<td>Database Service</td>
<td>client/server</td>
</tr>
<tr>
<td>Travel</td>
<td>Trip Planning Service</td>
<td>client/server</td>
</tr>
<tr>
<td>All subsystems</td>
<td>User Interface</td>
<td>peer to peer</td>
</tr>
</tbody>
</table>

- **Assistants** query the **Vehicle** server for information pertaining to the state of the vehicle or a resource that the Vehicle controls. The Vehicle itself doesn't need to call any methods on the assistants. This is a **client/server** relationship.
- The **Authentication Service** will act as a server to the client **assistants**, to allow only authorized access. This is a **client/server** relationship.
- The **assistants** will subscribe to events by sending an request to the **Notification Service**. When the Notification Service will know about the events, it will dispatch their occurrence to the clients. The client needs to know the interface of the Notification Service in order to sent its subscription. The Notification Service needs to know the interface of the client in order to notify it of an event. This is a **peer to peer** relationship.
- The **Name Service** will be a server to any other **subsystem** of JAMES. It will contain information about locations of objects. Objects will call a bind on the Name Service. Clients will call a look-up method on the Name Service. All other subsystem will know the interface of the Name Service, but the service itself doesn't need to know about them. This is a **client/server** relationship.
- The **Database Service** is a server providing access to static information as requested by client **assistants**.
This is a client/server relationship.

- The Trip Planning Service creates a trip when it is called from the Travel Subsystem. This is client/server relationship.
- The User Interface and the rest of the subsystems of JAMES need to know each about others interface. The subsystems will send output to the User Interface. The User Interface needs to call appropriate methods on the rest of the system upon processing user input. This is a peer to peer relationship.

**Software Bus**
The software bus will comply with the Common Object Request Broker Architecture (CORBA) and allow distributed objects to talk to each other even if they are located on different machines. We will be using Visibroker from Visigenic as our Object Request Broker implementation.

### 2.1.1 Layers & Partitions

Each assistant depends on the Vehicle Subsystem to provide it with information about the current state of the vehicle. Assistants also depend on the Notification Service to know about events in the vehicle. The assistants depend on the Authentication Service. The remote method invocation between all subsystems of JAMES depends on the Name Service. Assistants depend on the Database Service for access to static information. The Travel Subsystem depends on the Trip Planning Service.

---

### 3. Concurrency Identification

**Logbook Subsystem.** The Database, Vehicle, SmartCard, and Authorization, defined in Figure 3.1 of the Logbook Subsystem RAD, are the independent objects. While trips are collections of legs, the legs may change without affecting the trip data.

The following control threads have been identified:

- Thread managing the current state of a trip on a car.
- Thread managing the GUI, the graphic user interface, for the Logbook Assistant on the car.
- Thread interfacing with the card.
- Thread managing the GUI on the PC assistant, the part of the Logbook Assistant which runs on a user's home or office PC, with additional threads if smart searches are performed.

The Logbook subsystem does not provide access to multiple users at the same time. It will only perform one action on the database at any particular time. Since there is a single state there are no concurrency issues in the Logbook subsystem. Potential problems for managing the data exist, but should be managed by the Database, which is outside the realm of the Logbook subsystem.

**Maintenance Subsystem.** The data from the MaintenanceHistoryEngine and the RecommendationsEngine, defined in Figure 15 of the Maintenance Subsystem RAD, are independent from each other, though they both relate to the same vehicle. VehicleInfo and UserInfo are independent, though they are both stored on the smartcard. Each user's request initiates a thread of control. (I.e., when I insert my card, this initiates a sequence of operations: pulling down history info, recommendations info, etc. However, only one user will be using any of the assistant at a time, so only one thread is needed for this.

If we're going to be able to respond to notifications given by the Vehicle Subsystem, then journaling those somewhere would be a separate thread of control. It would wait for a notification to arrive and then record the problem somewhere. The user interface in the vehicle or while running the Maintenance Subsystem on the PC only provides access to a single user at a time. The back-end, centralized databases must provide access to multiple simultaneous requests for records.

A single "query" will not consist of multiple queries. Multiple queries, however, can be handled in parallel by different subsystems. The ideal plan would be for two major pieces of data (maintenance history and recommendations) that come from outside services to be requested and fetched in parallel. Depending on the communications link, two requests could be slower than one, especially if a single back-end system is servicing the
Vehicle Subsystem. The Vehicle object is independent. The Vehicle object, defined in figure 3.5.2.1 of the Vehicle Subsystem RAD, consists of many parts, such as the Dashboard, Seat, Mirror, Steering Wheel, Air Conditioning, and Accessory which can run concurrently based on signals received by their implementations. There are 5 threads of logical control. These include the Onboard Computer (the Vehicle's thread), the Smart Card (thread for cardlets), the Seat Simulator (SART), AIM, and the ECUs (Electronic Control Units). The threads run concurrently. There is one user of the vehicle at a time.

The "query" system is handled by the Notification subsystem, which will have a notification server on the onboard computer, and notification clients on the cardlets. The system allows for events to be pushed via channels, and for clients to pull (request) information from a channel. Many channels can exist concurrently, with pushing and pulling operating in parallel.

Travel Subsystem. The independent objects of the Travel Assistant are the Navigation System, the Map Server, and the SmartCard. The control threads all come from the driver or planner of the route when he or she plans a route and saves it onto the card, views a route while on the road, or updates the route on the fly. The system does not provide access to multiple users at the same time. A single "query", however, can consist of multiple queries, to the SmartCard, the Navigation System, and the map service all at the same time (when viewing or updating a route). These queries could be handled in parallel by three subsystems, but the timing of all three would have to correlate with the desired information which is needed at the time. In planning a route, no concurrency is needed: a map is called up, a route is selected, then the route is saved onto the SmartCard.

For now, only one assistant will be on the card at the same time, no concurrency issues need to be resolved between the Travel Assistant and other parts of the system, but when the Notification Subsystem allows more than one subsystem to run on the card, concurrency may be necessary for other parts of the JAMES system to run at the same time. Also, waiting for events from the GPS to determine position will be running on the notification server concurrently. But otherwise, there is no concurrency in the subsystem (one thread at a time).

VIP Subsystem. VIP does not support multiple users, because it interacts with one user's settings at a time so two users should never be hitting the subsystem at once. As to whether VIP can run concurrent execution that depends upon the Vehicle if VIP are always listening for notification events then VIP could submit multiple items for storage/setting before one returns.

The only item that VIP cannot execute concurrently is storing/reading the card. VIP will be required to wait to finish writing/read before it can write/read again.

Essentially VIP Assistant will do the following: VIP will listen on the software bus for event notification by HCI/Vehicle that it needs to do something, i.e. the subsystem needs to store settings, or set settings. At this point VIP will send the request to set values or retrieve values to VIP and wait for notification of completion of the task by the VIP assistant. At this point VIP will go back to listening on the software bus until another notification has been received.

Notification Subsystem. There is concurrency in the notification system. Notification works as follows:
1. There is a notification server. This will most likely be running on the vehicle subsystem.
2. There are notification client beans. These are objects to be used by any object which wants to use the notification system.
3. There are event channels. Notification clients can be subscribed to the event channels as: push producers (they will put events onto the channel), push consumers (they will take pushed events from the channel), pull consumers (they will send requests for information onto the channel), and pull producers (they will take requests from the channel and supply data).
4. Each of these consumers and producers has a listener in the event channel. The listener is an object which contains an interface to a method to be called when something is pushed/pulled onto the event channel.
5. Example, when someone pushes a event onto the channel, the server looks at all the push listeners in the channel. For each one, it calls the a method specified by the listener. In other words, the listener tells the server what method to call when someone pushes something onto the event channel.

User Interface. User interface objects for each subsystem will be independent. Speech and visual interface will also be independent. The system does not provide access to multiple users.
A single "query" can consist of multiple queries for example when the user first inserts the card, the unified interface is queried, but since that unified interface needs information from multiple subsystems, this single query spins off multiple queries.

### 4. Hardware/Software Allocation

**Hardware platform:** As provided in the Problem Statement Document, Part 7, and other information given by the clients, the target environment consists of the following parts:

- **Vehicle** The JAMES system is to be used by drivers driving a Mercedes Benz vehicle. During this project a simulator provided by the client represents the vehicle. Two simulators have been provided, the AIM/CANalyzer system and the SA/RT seat simulator. AIM runs in a LINUX environment, while the CANalyzer runs in a Toshiba Personal Computer MS DOS 5.00 environment. SA/RT is compiled for a machine equipped with a Windows NT 4.0 environment.

- **Cyberflex Card.** The processor on the card is a 8 bit 1 MHz CPU (Motorola 6805), with 4KB EEPROM on chip and a 1.2 Kbyte library. The maximum application size is 2.8 KB bytes (= 4.0 KB - 1.2 KB). The cards that we have available in the lab have a maximum application size of 1896 bytes. The stack size is 16 bytes. JAVA applications which run on the card are called cardlets.

- **Smart Card Readers.** The Litronic Argus 210 reads Cyberflex smart cards. The reader must be attached to an outside computer, such as one that runs Windows NT, via a serial RS 232 cable. To interface with the card reader, the outside machine requires a DLL which is only compiled for the Windows NT environment currently.

- **GPS receiver.** The system is using a GPS receiver unit Eagle Explorer from Lowrance. The GPS has serial port connectivity and can hook into any PC. The GPS outputs location data in strings according to the NMEA 183 standard.

The vehicle interface between the SA/RT seat simulator, AIM/CANalyzer, and the JAMES subsystems (Travel, Logbook, VIP, Maintenance) will run on a machine with the Java 1.1 Virtual Machine in order to provide portability to different machines as well as be able to talk with the JAMES assistants which will also be running Java. However, the vehicle simulator interface will also require links to the Smart Card Reader and SA/RT which can only be done using a Windows NT environment and links to the AIM/CANalyzer system which can be done through the LINUX environment. JAMES assistants such as VIP, Maintenance, Travel, and Logbook will reside as Java applets, which will either run on the Java card as cardlets or run on machines with card readers. As stated previously, only the Windows NT machines currently support the card readers.

Parts of the JAMES assistant, such as a trip planner, will need to run on an external computer system from the vehicle. This external computer system should be a capable of running a full version of Java (or possibly Visual Basic) and have connection to Smart Card Reader. JAMES assistants will read data from the SmardCard, store it on the hard drive, and allow the user to manipulate it as well as print out reports. Any external computer system capable of supporting a JAVA 1.1 virtual machine and capable of running Windows NT should be fine.

Table 4.0.1 is a summary of the mapping between subsystems and machines/environment:
### Table 4.0.1: Hardware/Software/Environment Mapping

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Environment</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA/RT</td>
<td>Windows NT 4.0</td>
<td>Running on machine electra.se.cs.cmu.edu.</td>
</tr>
<tr>
<td>AIM</td>
<td>LINUX</td>
<td>Running on machine james.se.cs.cmu.edu.</td>
</tr>
<tr>
<td>CANalyzer</td>
<td>Toshiba PC MS DOS 5.00</td>
<td>On laptop provided by client.</td>
</tr>
<tr>
<td>Smart Card Reader</td>
<td>Windows NT 4.0</td>
<td>Reader connected to machine java.se.cs.cmu.edu via rs232.</td>
</tr>
<tr>
<td>GPS receiver</td>
<td>Windows NT 4.0</td>
<td>The GPS hooks into the serial port.</td>
</tr>
<tr>
<td>JAMES Assistants (on vehicle)</td>
<td>Cyberflex Card Java</td>
<td>On Cyberflex Cards</td>
</tr>
<tr>
<td>JAMES Assistants (external)</td>
<td>Java 1.1</td>
<td>Needs connection to machine with Card Reader (Needs Windows NT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Needs to be able to have GUI interface</td>
</tr>
<tr>
<td>Vehicle Simulator</td>
<td>Java 1.1</td>
<td>Needs connection to machine with Card Reader (Needs Windows NT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Needs connection to machine with AIM (Needs LINUX)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Needs connection to machine with SA/RT (Needs Windows NT)</td>
</tr>
<tr>
<td>Object Request Broker</td>
<td>Windows NT 4.0</td>
<td>CORBA implementation from Visigenic.</td>
</tr>
</tbody>
</table>

### 4.1 System Performance

As stated in the goals and tradeoffs section, the emphasis is on having the system be flexible and extensible rather than having the system be optimized for speed for any specific platform.

#### 4.1.1 General system performance

The system should react in a timely manner. The system reaction times for the various events depend on the nature of
the task and datapath. A full datapath, such as a method call from the VIP assistant to adjust the seat for example, could originate from the cardlet running on the smart card, travel through the smart card reader to the Windows NT machine via RS232 cable, and then be sent through AIM, and then to the vehicle subsystem, and then to the SA/RT seat simulator which will receive seat configurations from a file on a hard disk. Clearly, the path which the data travels to get from the cardlet to the seat simulator is twisted and low reaction time cannot be expected. Again, the system is designed to demonstrate flexibility and extensibility for the prototype, rather than for optimality of speed.

One question pertinent to this issue is "How long is the user willing to wait for this service?" The following are acceptable response times and transaction rates for different services:

**Logbook.** When the system is doing a complex operation, like downloading a trip object, a 5 seconds should be acceptable. The average transaction rate should be about one leg object /10 minutes. The maximum would be as fast as someone can press the button, 10 times a second.

**Maintenance.** The response time desired for the maintenance subsystem to read information of the card and to get diagnostic information has to be less than a few seconds in the car. In general, there will be one user per MaintenanceTransactionManager object(RAD 3.5.3.2). In this case, there will be at most one request every few second. Each MaintenanceTransactionManager will talk to the database service. The rate of transactions on the database will be a database issue, but can be expected to be more than one transaction per second.

**Travel and VIP.** The response time desired for the onboard real-time services should not exceed more than 0.5 seconds for travel operations, and 5 seconds for vehicle adjustments, such as setting a seat, especially while in the car where the user is less patient and more irritable. The only time the travel assistant will need to request GPS information when the user is changing route info, so the rate of transaction is expected to be low.

**4.1.2 Input/Output Performance**
Software speech systems, utilizing a limited, restricted grammar, can respond within a couple of seconds which is an acceptable performance level. The input necessary for a speech system is mainly a microphone that is sensitive enough so that it can pick up the user more than two feet away. The speech system (not the input device) will have to deal with filtering out noise as well as for recognizing speech. The main output device for the speech system is a speaker, probably separate from the car stereo system.

The other main interface through which the user is going to interact with the travel assistant is a touchscreen. It will provide both the input and output support. Any response time around 0.5 seconds would be acceptable.

**4.1.3 Processor allocation**
Processor allocation is as described above in the table 4.0. However, an important issue is the speech recognition and route calculation. Both are CPU intensive, and may require independent processors with greater computing power than an 80486 chip.

**4.1.4 Memory allocation**
Memory allocation is constrained for storage of data onto the cards, due to their limited space. For the "onboard" computer, 8 MB RAM is sufficient.

**4.2 Network Connectivity / Architecture**
The different parts of the system, as described in section 4.1 communicate to each other on different levels. The OSI (Open Systems Interconnection) model (figure 4.2.1) shows an instance of the seat as part of the 7 layers of the network architecture of JAMES. It depicts data entering the physical layer, traveling up through the corresponding layers to the assistant (logical) layer and proceeding back down to the physical layer, which then transverses to other vehicle subsystems.
The different components of the JAMES system are connected logically on the assistant layer of the OSI model. The logical connectivity diagram (figure 4.2.2) describes the assistant (logical) level. It consists of the class diagrams of this instance (Seat). It describes the protocols which the different subsystems and classes use to communicate. The AIM and SA/RT are currently under research at Daimler Benz. As such, the seat implementation is used as a bridge pattern to connect the AIM and SA/RT and other stubs to the vehicle subsystems. In the future, if AIM is implemented, an actual seat may be connected to the vehicle via AIM instead of directly through an ECU.
The physical connectivity diagram (figure 4.2.3) describes the physical level of the OSI model. It consists of the class diagrams of this instance (Seat). It shows the network topology which connects the different machines which the processes reside on.
5. Data Management

The SE lab computers are interconnected through a local area network.

AIM, Adactive information management, resides on the computer James.SE and is physically connected to the Canalyser.

SART resides on the computer Electra.SE.

The Card Reader is connected to Java.SE via an RS232 wire.
Data for this system is stored and accessed in several forms. The major location, used by all subsystems, is the SmartCard itself. Many of the subsystems also use remote, centralized databases. A few subsystems have other data sources, such as GPS, CD-ROM, a speech library, and more. However, in our domain, only the first two sources are read/write; other sources will be read only.

For all read/write data sources, access will be conceptualized through the use of APIs and stubs. Actual implementation will be recommended in this document, however all subsystems can ignore the actual implementation details and deal with the databases in more conceptual units such as 'WriteData' 'GetQuery'.

First let's consider the SmartCard itself. It will be more than a floppy disk and will have persistent data storage for all the subsystems. The limited storage capacity on the SmartCard is the major concern and hence all subsystems will be compacting their data as much as possible. This will be through using integer codes rather than strings, as for example in the Maintenance System where work done will be stored as a number that represents actual service. Similarly all other subsystems will compact their data to fit on the SmartCard.

For all subsystems it is expected that the SmartCard query rate and amount of data transferred per query will be small. The only way to enumerate a number is through tests, but we expect an average load of no more than 1 query to the card per second. However, there are boundary conditions, such as when VIP initializes the car's settings depending on the preferences stored on the card. Then we expect the limiting factor to be the I/O bandwidth or the processing power of the CPU.

Each subsystem has their own data requirements for the SmartCard. The Maintenance subsystem will store N recent Maintenance events. Since the complete history of events will exist both on the SmartCard and centralized databases, this arises problems with synchronization, which will be discussed below. Currently N is set equal to three. The format of this data will be dealer code, maintenance code, flags, and date.

Logbook and Travel have similar conceptual requirements; they store one trip (the current) one on the card. Logbook stores data in forms of trip legs, of which they expect to be no more than a 100 legs. Each leg is expected to be two-three integers. Travel stores way points, which are points of interest on the trip. They expect a trip to have up to 50 waypoints. Waypoints will consist of integer-encoded data such as type (restaurant, etc.) and <.... Travel also expects waypoints to dynamically change if the driver takes a detour from the pre-planned travel route.

VIP will store user preferences as several objects on the card. These settings will change as the user moves around in the car (i.e., moves his seat, etc.). Their data storage is expected to be very small and depends on how many settings they will store.

For all of the SmartCard data, every subsystem except Maintenance needs read/write access for the driver. Maintenance info should only be updated by a mechanic or authorized garage representative.

Central databases will be used mainly by two subsystems, Maintenance and Logbook. The central database is used to store, in a more permanent fashion, large amounts of data that should be easily accessible through many means. A industry-tested approach is to use a robust RDBMS such as Oracle or mySQL. Other options include object oriented databases such as Poet, or a simple set of files approach. We suggest using Oracle, however due to time or monetary constraints, any of the options can be used, as discussed above.

Any respectable RDBMS or similar system will deal with concurrency issues, usually through the multiple-readers-single-writer model. This document will assume that this implementation detail will be handled in the central DB software itself. The servers will be accessed through TCP/IP over the Inter- or IntraNet.

Maintenance will use central database(s) stored with Daimler-Benz. Logbook on the other hand will use central database(s) stored with the company which requests travel history. Hence the API is crucial to allow external companies to create their own central servers to support the LogBook subsystem.

A major issue with central databases is synchronization; it is possible that data stored on the card may be out of sync with data on central database servers, or be conflicting. For example, a person may have his car serviced at a dealership without a hookup to the central database, and hence have an entry on the SmartCard, but not in the central database. At the same time the central database recommendation engine realizes that the car is due for service. So when the user finally goes to a 'connected' dealer who uploads the data from the SmartCard to the central database, it must realize that the service completed is the service pending from the recommendation service.
Central databases will have three access levels; read, add, and edit. The driver and other authorized peoples have read access. Mechanics and authorized garage representatives have add access. Supervisors and other authorized 'super users' will have access to edit (and delete) entries in the central database.

For the Maintenance sub-system a further data management issue is synchronization. This issue arises due to the duplication of data on both the SmartCard and central database(s). Currently the model includes possibly many (one or more) central databases and one SmartCard. Synchronization will simply make sure the given central database is in sync with the SmartCard. This is important since the SmartCard is very limited and older data will eventually be deleted completely, while central databases are designed to hold more data for a longer time.

Other data sources are read-only, such as GPS, CD-ROMs, speech libraries and more. These data sources are assumed to feed data at a reasonable rate for the subsystem which needs it.

There are also several legacy systems for the Maintenance subsystem. These legacy systems are database servers that use SQL, such as Oracle, and mainframe systems. Some of these systems are rational, some are not. Some legacy servers are also proprietary. One system currently being used is called ASSYST, which provides recommendation functionality and also information about the car. Maintenance will provide access to these legacy servers when needed and possible.

---

### 6. Global Resource Handling

The system requires authentication to prevent users from interfering with each other's data, and to allow the system to distinguish between multiple users using the same card. The Java SmartCard provides an authentication service, and no matter what applets are running on the card, authentication will be required before any access to the onboard vehicle processor is allowed. Users will have to authenticate their identity by means of pin number or password entered on the vehicle touch screen or computer keyboard. The authentication system will have to allow for multiple users using the same Java SmartCard, and provide the user's name to any services requesting it. Specifically, the logbook and VIP subsystems have already determined that they will need to be able to differentiate between users, and protect one users data from another.

Figure 6.1.1 below shows different types of users that are available in the authentication system.
The following is a brief description of each user:
Normal User: Not known by the bonus system of a company: minimal access rights.
Normal Driver: Is known by the company, and has full access rights to all data.
Everybody: A generic user. Can only read radio display data
Maintenance Activator: The generic superclass for garages. Each maintenance activator has a specific role based on his specific type, and has rights depending on this type.
Company Supplier: The user is an employee of a dealer or a garage which offers maintenance support to their customers related to the maintenance assistant.
Fleetmanager: Leader of a carpool.
MBDealerShop: An authorized Mercedes-Benz dealer, he can read and write radio data, car data, history, address data, contract data, dealer file, and MLC file. MLC is similar (but is licensed by Mercedes-Benz, but not owned by it), and has the same rights.
Garage type KDB: A service consultant for Mercedes-Benz. Has the same rights as the MBDealerShop, except that it cannot write contract data.
Garage Type A: A non-Mercedes-Benz sponsored dealer. Can read radio data, car data, or history, but nothing else, and cannot write anything.
Garage Type p: Police, can read all of the data that the MBDealerShop can, but cannot write any data.
Garage Type N: A non-common Mercedes-Benz garage. It can read radio data, car data, and history, and can write the history.
Garage Type T: A Tire dealer with a Mercedes-Benz service contract and 24 hrs service. It can read everything that the MBDealerShop can except the dealer file and the MLC file, and can write the history.
There will be a vehicle-wide name server that can be extended to a network-wide name server once the computing capacity allows. A subsystem will subscribe to events using its system name, and using these names subsystems can send events to other subsystems and receive events from other systems. Subsystems will also be able to query a well known name-server to see if a given system exists.

By the concurrent nature of the system and its event-driven structure, multiple subsystems will need to simultaneously share certain global resources. The SmartCard will need persistent storage to be able to store passwords or pin numbers for authentication. Other persistent card storage will need to be shared by the VIP, Travel, Maintenance, and Logbook subsystems. Different read/write accesses will be granted to different applets to maximize throughput while preventing conflicting unfinished state changes by multiple applets accessing the same service.

The system will never allow direct access to any resource because of security, portability, and extendibility concerns. Direct access might allow a small performance gain in non-concurrent environments, but because of the concurrent nature of the system, and the need for implementation-transparent resources, the disadvantages of direct access far outweigh the possible gain, and will only lead to software that quickly becomes obsolete.

7. Software Control Implementation

7.1 External control flow (between subsystems)

Each system is independent of one another. Therefore, the control flow consists of the assistants receiving events from the vehicle, and also consists of assistants querying the vehicle for information on the vehicle's current state. Furthermore, the Logbook and Travel subsystems use common information, and therefore they also may communicate this information to each other.

Logbook. The logbook assistant essentially has to respond to serialized events: the driver pressing the buttons in the car, the car's ignition turning on or off, or the data being sent to the card. This can be best done through an event loop where the Logbook Subsystem waits for an event, process it, and wait again.

Travel. The control flow between different pieces of the travel system is determined by the user. Depending on what information the user wants, the travel subsystem will query the vehicle to obtain the information that the user requested.

Maintenance. The maintenance subsystem will use a transaction manager to process input from input from the user( see figures 7.1.1 and 7.1.2 ). Most of the procedures will be simple, for example, reading or adding to the maintenance history database. The Maintenance Transaction Manager (MTM) will only process one request at a time. Once a request has been completed, control will be returned to the user.
Figure 7.1.1: Decomposition of the Maintenance System

- Database Manager
  - addEntry
  - editEntry
  - deleteEntry
  - getName
  - setName
  - getCount
  - setCount
  - open
  - close
  - first
  - last
  - next
  - previous
  - find
  - filter
  - fetch

- Input Device
- HCI Event Handler
- Output Device

- Maintenance Transaction Manager
  - addEntry
  - editEntry
  - deleteEntry
  - getHistory
  - synchronize
  - stopCard

- Card Interface
- Maintenance Cardlet
  - MaintApplet

- External Database
- User Interface/Terminal
- Java Smartcar
**Vehicle.** The vehicle control flow is such that external control consists of two cases. The first case is when a system queries the vehicle for certain info about the vehicle's state. The vehicle simply handles the query and returns the requested info to the subsystem that made the query. The second case is when the vehicle wants to notify a subsystem of an event. In this case, the vehicle subsystem simply passes the event to the notification service, who then takes charge of delivering notification of that event to the appropriate subsystem.

**VIP.** The control flow of VIP is simple, when an event occurs, the VIP system will be notified, and the appropriate action will occur. It waits until something of interest occurs, i.e., the card is inserted, the save button is pressed, etc.

### 7.2 Concurrent control

Each JAMES subsystem is independent of one another. They all can be run concurrently. Within the subsystems, the concurrency is described below.

**Maintenance.** Only one user can use a single MTM at any given time, so this should not cause any concurrency. However, multiple MTMs could use the same database concurrently. If there was ever the case where multiple MTMs try to access the same entry within the same database then we will use the rule of multiple readers, single writer.

**Travel.** There is no concurrency within the travel subsystem.
Vehicle. Vehicle will primarily be run on a thread based system so much of their functionality can and will be executed concurrently.

Logbook. No components in the logbook system run in parallel.

VIP. VIP can only set one user's preferences at a time, so there is no concurrency within VIP.

7.3 Internal control (within a single process)

Maintenance. Maintenance is user request driven, so there is a single thread of control.

VIP. VIP is notified when an event occurs via procedure calls. So, when the "save" button is pressed, the VIP subsystem would be notified of that event, and would then execute the appropriate procedure.

Travel. Because the SmartCard does not use threads, the only control flow will consist of procedure calls. However, because of the limited stack space available on the SmartCard, the number of procedure calls must be kept to a minimum.

Logbook. Process control is done by using one thread with an event loop that waits for events, and processes them, and the process repeats again.

Vehicle. Because of the way the threads are coordinated the only internal control is between child and parent threads.

7.4 User Interface

Each subsystem has its own user interface and event loop. The HCI subsystem will coordinate the User Interfaces for each assistant so that they are consistent with one another.

8. Boundary Conditions

8.1 Service Boundary Conditions

There are two types of services:
Generic services are integral to the operation of JAMES, and must always be available. In general they must be in place before any assistants or application specific services can function. They also generally interact with a wide range of assistants and so may require more resources.

Notification Service and Name Service
Unlike other services or assistants, the notification service is running whenever JAMES is active, and must be one of the first subsystems to execute since many assistants will depend on it to sort out their initial state.
The name service must initialize early so that subsystems can register on startup before pushing listeners onto the event channels they need to monitor. It must also run constantly to allow dynamically downloaded applets to register.

Authentication and Database services
The authentication service is needed before applets need to access or change the system state.
The database services is needed after applets have registered and before they need to modify any vehicle state.

Application specific services may be initialized or terminated by the subsystems that they service. Their scope is limited since they are not always available.
8.2 Subsystem Boundary Conditions

Some subsystems must always run. The vehicle and logbook subsystems must always be available. Other assistants will eventually be dynamically downloaded, and can be stopped and started at will.

Logbook Subsystem
On initialization, the Logbook subsystem must gain access to the onboard leg/trip database. It uses information from the database to determine whether a trip is in progress. Since the Logbook subsystem does not provide services to other systems, on initialization, it only needs to register itself with the Name Service.

Under normal circumstances the Logbook subsystem should not terminate except on shutdown of the entire system. If an unrecoverable error occurs, the Logbook subsystem will deregister itself with the Name Service and terminate. The user should be notified when this happens. No other subsystems use Logbook services, therefore none will require notification if the Logbook subsystem shuts down.

The Logbook subsystem should mark the current trip (if one is in progress) in the database on system termination, since this is necessary on startup.

Measurements which cannot be made due to link failure (for example, odometer readings or GPS locations) will be designated in the database records as "Unavailable." This is an exception condition and the user will be notified. The existence and structure of backup links are issues to be resolved by the designers of the JAMES hardware. If links to required subsystems (such as the vehicle) are re-established by some means (outside the scope of the Logbook subsystem), their services can be continued.

Recovery from system failures is not a condition which marks the beginning or end of a leg, so it does not cause re-initialization. Rather, it is performed as needed ("on the fly"). If a recoverable error occurs, the Logbook subsystem does not assume it should start a new leg.

Vehicle Subsystem
On initialization the Vehicle subsystem must set the initial status on parts of the vehicle to create the accessories list, start up the notification server, and interact with the VIP system to set vehicle preferences. If available, the VIP settings will be used. Interaction with the Maintenance subsystem may be needed if information concerning the initialization status should be stored in the Maintenance database.

The vehicle subsystem is vulnerable to two categories of failures, component and system failures. Component failures which are defined as parts of the vehicle that do not acknowledge commands within a given number of attempts. System failures occur when vehicle subsystems go down. Component failures are recoverable. Internal representation of the component will be disabled so that no more commands will be sent to that component, and the system will continue to operate. System failures can be recovered via hard recovery (restart of the vehicle subsystem). Recovery can occur at 'higher levels.' It may be necessary to restart the vehicle to recover from System Failure, or have physical repairs performed. Any failures may be passed to the vehicle to be stored in a log of errors.

VIP Subsystem
The VIP subsystem acts in coordination with the Vehicle subsystem to make any adjustments which the user has stored on the card on system startup.

The VIP subsystem may encounter errors due to a failure in the vehicle, when preferences which are asserted to the vehicle are not properly set. If such a failure occurs, the assertion will be tried again, but only once.

Maintenance Subsystem
The Maintenance subsystem does not need to perform any explicit initialization or shutdown functions, and it does not depend on other subsystems.

Travel Subsystem
The dynamic model of startup for the Travel subsystem is Authenticate, Get Preferences, Get Route & Maps, Get GPS, and Display Route w/ GPS information and directions. The preferences, route, maps, and GPS must be available at startup. The Travel subsystem must register GPS, map, trip planning, and traffic services with the Name service.

The Travel subsystem can be terminated if the user feels that the functionality is not needed. If it is terminated, it does not have to notify other systems, because all subsystems are independent. On termination, stored information and preferences have to be saved to the available data storage.

In case of link failures the user must be notified, the system should revert to the previous action, and reiterate communication efforts. Upon recovery from failure information stored locally and remotely should be reloaded. The steps for recovery will follow initialization, except for retrieving the data that was not lost during failure.
9. Design Rationale

This section describes the rationale behind the most significant design decisions of JAMES. The design rationale is presented here as the discussion of a number of issues: each design decision is listed as:

- the issue to which it is responding,
- a number of proposals that could also respond to the given issue,
- arguments for and against each proposal, and
- the resolution of the issue, that is the documentation of the decision.

9.1 Issues related to system decomposition
9.1.1. Issue: What is the scope of the name service?
9.1.2. Issue: How should the Event Notification Service be realized?

9.2. Issues related to hardware/software mapping
9.2.1. Issue: Should there be a monitor display in the vehicle?
9.2.2. Issue: How can the Vehicle subsystem accommodate several independent simulators?
9.2.3. Issue: Should a specific platform be targeted for the external system assistant?
9.2.4. Issue: How should the Vehicle subsystem interface to SA/RT?
9.2.5. Issue: What is a practical way to map individual components of the vehicle so the subsystems can differentiate between components and utilize their different attributes in an efficient manner?
9.2.6. Issue: Where should the Logbook subsystem reside?

9.3. Issues related to data management
9.3.1. Issue: What is the most optimal solution to synchronize the maintenance items?
9.3.2. Issue (sub issue of above): How much of another dealer's information should the dealer be able to see?
9.3.3. Issue: Should the trips be divided into legs?
9.3.4. Issue: Should everything be kept on the SmartCard?

9.4. Issues related to global resource handling
9.4.1. What should be provided for authentication?
9.4.2. Issue: how to overcome limitations of Java card?
9.4.3. Issue: Should the user be able to disable authentication?

9.5. Issues related to software control
9.5.1. Issue: Which remote method implementation to use?
9.5.2. Issue: Should demo assistants be implemented on the card or the computer?

9.1 Issues related to system decomposition

9.1.1. Issue: What is the scope of the name service?

What kinds of objects should be registered and what role does the name service play in assisting communication between subsystems, services, etc.

This issue is relevant to anybody developing either a subsystem or a commonly used service which needs to communicate with other subsystems or services. This is especially relevant when there is some overlap between subsystems because the name server alone with the event notification service provide a means of communication and data/resource sharing between subsystems. The goal of this project is to produce a system capable of running
multiple dynamically loaded/unloaded applets that must share resources. The name service can either provide a means for the event notification service to locate subscribers, or it can be extended to dramatically increasing resource utilization and overall system efficiency by matching services to subsystems. The issue is also relevant from a safety/security standpoint. If concurrent subsystems are going to be competing for resources, a method must be found to ensure that essential subsystems have access to the resources they need, and protects sensitive information by providing a means for subsystems to authenticate before getting rights to access certain services.

Definitions:

Name Server - A well known service that allows subsystem and/or other services to register, and provides means for communication between registered object.
Subsystem - A java applet or cardlet that provides part of the functionality of the JAMES system by utilizing services in a client server relationship.
Service - Provide a java front end to hardware or software that can be used by subsystems.
<table>
<thead>
<tr>
<th>Proposal 1: Register only subsystems</th>
<th><strong>For:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick implementation time.</td>
<td></td>
</tr>
<tr>
<td>Satisfies name server requirements. Provides means for communication between subsystem.</td>
<td></td>
</tr>
<tr>
<td>Robust. Services always available. Subsystems designed with services built in.</td>
<td></td>
</tr>
<tr>
<td>Extensible. Can be extended to allow service registry.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Against:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inflexible. Must match subsystem to particular hardware</td>
</tr>
<tr>
<td>Inefficient. Does not allow for sharing of services.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposal 2: Register services and subsystems</th>
<th><strong>For:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficient. Eases sharing of commonly used services by multiple subsystems.</td>
<td></td>
</tr>
<tr>
<td>Easy subsystem development. Allows development of libraries of services, and development of front end interfaces for multiple services that provide same functionality using different hardware.</td>
<td></td>
</tr>
<tr>
<td>Flexible. Developer always has option of hard coding little used services instead of using a name service.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Against:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Longer implementation time. Must work out issues of security and priority when subsystems are competing for resources.</td>
</tr>
<tr>
<td>Not robust. Subsystem must fail if it cannot find required service.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposal 3: Treat services as subsystems - Peer to Peer</th>
<th><strong>For:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Peer to peer relationship between subsystems/services more flexible. All the same advantages as proposal II.</td>
<td></td>
</tr>
<tr>
<td>Services can utilize other services in a hierarchical fashion.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Against:</th>
</tr>
</thead>
<tbody>
<tr>
<td>More complex security/priority/authentication issues.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposal 4: Extend proposal I to Proposal III later</th>
<th><strong>For:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick implementation time. Allows subsystem developers to work concurrently with name service developers.</td>
<td></td>
</tr>
<tr>
<td>Extensible.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Against:</th>
</tr>
</thead>
<tbody>
<tr>
<td>May be difficult to port services developed now to run on peer-to-peer network.</td>
</tr>
</tbody>
</table>
Resolution: proposal 4

It is necessary to build a minimal name server interface to work with the event notification system, but since the system must be delivered on 9 December, 1997, and since the card is limited in its size, and cannot run multiple threads it is not feasible to implement a full peer-to-peer name service that can properly address security issues through authentication, and safety issues through prioritization. This makes proposals II and III impractical. A peer-to-peer naming system is necessary for the long term objectives of the course, to allow dynamic loading/unloading of applets and services, and allow them to efficiently share resources. Proposal IV manages to address both the issues of proposal I while allowing a peer-to-peer naming service to be built on top once there are cards, subsystems, and services that can take advantage of this.

9.1.2. Issue: How should the Event Notification Service be realized?

One fundamental issue in the design of the JAMES system is event notification. Subsystems need a common way to communicate events to one another. An event can take on a wide range, such as a turn of the ignition key switching the car from one operation mode to another (off → start), a message from the GPS service saying that the vehicle has reached a certain coordinate, or a message from one subsystem to another - perhaps communication between the Travel and Logbook assistants. An event might also be from the user such as a preference adjustment from the driver of the vehicle. The question is, how should this event notification service be realized?
**Proposal 1: Reuse OWL's notification service**

The first option, reusing the Event Notification Service from the OWL project, has the advantages of being already implemented in JAVA, well documented with an object model and API, and also with source code in Perforce. It has push and pull capabilities, allowing clients to push information to an event channel, and also to request information from an event channel using pull.

Being designed and implemented for the OWL project, this event notification system appears to need OWL's name service, which limits the choice and design of name services that can be used without having to spend time to modify the existing notification service.

**Proposal 2: Design and write new notification service**

The second option is to design and write a new event notification service customized for JAMES, with special considerations for the JAVA card limitations such as the size of the on card portion of the service, and the way it communicates events between the card and the onboard computer.

This option is obviously unfeasible for the December 9, 1997 deadline, though it certainly would be possible for the next iteration.

**Proposal 3: Buy and use a commercial event handling system.**

The third option is to buy and use a commercial event handling system. A commercial event service is available from Visigenic ([http://www.visigenic.com/download](http://www.visigenic.com/download)). With one of these, no development and implementation time for it would be needed to use it. It would only require the time it takes to be implemented. The service from Visigenic is well documented.

However, a drawback could be that our current JAVA card and the communication between the onboard computer may not be able to handle this implementation, which requires CORBA. This approach may be better suited for future iterations, where the card is likely to be larger in storage and more powerful in size, and when there is more sophisticated communication between the card and the onboard processor. Another drawback is that it may not be easy to customize the service for JAMES, since it would perhaps require the actual source code, which the commercial vendor might not want to release.

Consequently, if JAMES were dependent on a commercial event notification service, then future iterations could be at the whim of commercial vendor, who might change implementation to make the service incompatible with JAMES, perhaps causing much chaos in the future.

**Resolution: Reuse OWL's notification service**

The stake holders for this issue are the different subsystems, as well as the name service, upon which the event notification service is dependent.

It is necessary to build a minimal event notification service to work with the name service, but since the system must
be delivered on 9 December, 1997, and since the card is limited in its size and power it is not feasible to implement a customized event notification service for JAMES at present, though it may be feasible to do so in future iterations. The commercial notification system is likely to not fit the card constraints, as well as making it hard to customize to the JAMES system, since source code might not be released. Though these two options are discounted now, they deserve strong consideration in the future.

Thus the only option that fits the current constraints of time and hardware is to reuse the event notification service from OWL.

9.2. Issues related to Hardware/Software mapping

9.2.1. Issue: Is a monitor display in the vehicle needed?

Having a monitor display in the vehicle would require an additional hardware. This additional hardware to be put on would require redesigning parts of the car to make room for the monitor display. However, how much a monitor display is needed? If a monitor display is needed, what kind of a monitor display?
**Proposal 1: No monitor display.**

**For:**
- It is a goal for every industry to minimize cost. Using existing display hardware such as radio display will cut down the cost. To install monitor display inside the car will result to a change on the architecture of the car. Another possibility is having the monitor as a add-on component in the car. However, either options will add at least the monitor display cost to the budget of each car. If the system will function without the monitor display, then then monitor display is not needed.

**Against:**
- Without having monitor display, the system will need to find away to send the information to the driver. This can be done with the existing LED display, such as radio display. Connecting to different display tool, might make the system to be more complex. Moreover since LED display is usually take a small space, it can only show fairly limited text information at a time. The driver cannot afford to stare at the LED in a long period of time while they are driving to avoid the accident.

**Proposal 2: Monitor display, but not necessarily touch screen.**

**For:**
- One major advantage of the flat panel display is that it will allow the system to show icons. Being able to show an pictorial representation of words will be beneficial especially to travel subsystem. Displaying icons will enable drivers to have less time staring at the system.

**Against:**
- To interact with the system, drivers will need separate input device. Choice of input devices such as pointer, keypad, or keyboard will require additional space for the system in the car.

**Proposal 3: Touch screen monitor.**

**For:**
- Touch screen will make the system more user friendly. The drivers would be able to enter their choice directly by touching the screen. They don't have to remember the keyword or number to be entered to a separate input device. In addition, using the proposal 3, the weaknesses of previous two proposals would be addressed. There are five touch technologies: capacitive, resistive, infrared, surface wave and guided wave. Surface wave and guided wave technologies provide good visibility and image clarity. Since wave technology uses glass instead of coating, recalibration is not required.

**Against:**
- The retail price of touch screen ranges from $800-$2,800. Out of the five touch technologies capacitive, resistive, and infrared touch screens may be not suitable for this project. Capacitive technology touch screen can
Resolution: Touch screen

The stake holders for this issue are GUI developers, and the clients. Based on the needs and the requirements, and looking at the strength and weaknesses of each proposal, it is best to use touch screen display. What kind of technologies to be used is still to be discussed in the HCI meeting.

9.2.2. Issue: How can the vehicle subsystem accommodate several independent simulators?

Daimler Benz has provided 2 different and independent simulators, AIM and SA/RT. The vehicle subsystem needs to interface to these 2 simulators as well as to interface to the ECUs of the real components. So that either of the 2 simulators can be used to illustrate james functionality.
### Proposal 1: Use inheritance

By using inheritance, an abstract class `SEAT` would define an interface to 3 concrete seat sub-classes: Aim seat, SA/RT seat, and Real seat. The concrete classes will be responsible for implementing the different versions of the seat.

**For:**
- Straight Forward.

**Against:**
- Inflexible and creates numerous subclasses if extended
- It permanently binds the abstraction to its implementation.

### Proposal 2: Combine inheritance

By following the bridge pattern: 2 super classes classes `Seat_abstraction` and `Seat_implementor` are created. The seat abstraction will define the interface for all seat types. The seat abstraction structure will delegate the actual implementation to the seat implementor structure. i.e. it will have a reference to the implementor super class and the abstractions methods will invoke the implementor methods. The seat implementor will define the interface for the different simulators and the real seat ECU's. The implementor structure will be responsible for implementing the different versions of the structures. The seat abstraction and seat implementors can vary independently.

**For:**
- Extensibility is improved
- Implementation details are hidden from client code.
- The implementor could be selected or changed at runtime, without affecting the seat abstraction hierarchy nor the client (the VIP subsystem)

**Against:**
- Who take the responsibility to choose the implementor?

### Proposal 3: Use multiple inheritance or java interface

Since multiple inheritance is not supported in JAVA it is replaced by using java interfaces combined with single inheritance. By using multiple inheritance, 4 abstractions would be needed: Seat abstraction super class, Aim java interface, SA/RT java interface, ecu java interface. The Seat's subclasses can inherit from the Seat and implements Aim interface or from the seat and implement SA/RT -interface or from the seat and implement real seat interface.

**For:**
- It reduces the number of subclasses

**Against:**
- It produces a complex inheritance hierarchy
- It binds the implementation permanently to the abstraction.
- This solution is not extensible since a new interface must be defined for every every new simulator; the client code must be updated as well.

### Resolution:

Stake holders for this issue, the person designing the client code (a JAMES subsystem or the vehicle class) and the person designing the adapter to the simulator.

Proposal 2 was chosen since it has the following advantages:

1) Both stake holders will proceed with their work independently.
2) Proposal 2 satisfies the project's nonfunctional requirement of being extensible.
3) Proposal 2 allows the use of interchangeable simulators at any given time.

### 9.2.3 Issue: Should a specific platform be targeted for the external system assistant?
<table>
<thead>
<tr>
<th>Proposal 1: Use pure Java</th>
<th>For : Pure java is the way to go</th>
</tr>
</thead>
<tbody>
<tr>
<td>This will mean that the code is written in pure Java and is compatible across all platforms.</td>
<td>This is probably going to be the industrial standard once Sun succeeds in its campaign, and it'll give us the bigger market for this product. There will be no need to provide a separate MAC, Win16, Win32, Unix versions which will reduce the confusion for the users.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Against : We can do better</th>
</tr>
</thead>
<tbody>
<tr>
<td>It's hard to support certain features that allow a better merge with other assistants that run on a certain platform. Pure java is something akin to a toy, and not much useful for real assistant development. It might be great for doing silly animations of SUN mascot, but not much more.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposal 2: Partially dependent code</th>
<th>For : A great way to add features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write certain parts of the code for certain systems. This will give us additional features in certain platforms while still maintaining compatibility for the other platforms.</td>
<td>Detecting for certain features of the platform and using it if it is available is a great way to extend the multi-platform metaphor. This has been used countless number of times where a certain feature on a platform hasn't been available (for example 3D Acceleration). Problems can be worked around by making it available on certain platforms and not others giving us great amount of flexibility in designing the assistant.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Against : System dependent functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>System dependent functionality should be avoided. The main ideal behind JAMES is system independence. There really is nothing that can be offered by using system dependent code that can significantly enhance JAMES, and so this effort is better spent on the assistant development.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposal 3: Restrict to Windows 95/NT platform</th>
<th>For : Windows95/NT is the predominant OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Windows95/NT dependent code in whatever language available that fits the purpose. (For example, Visual Basic and Delphi which have a very low learning curve).</td>
<td>Windows 95/NT is in use by over 90% of computer users and therefore targeting for this OS will target nearly most of the customers. Much of the industry is moving towards this platform, and the other two major platforms (MAC/Unix) is dying out. There is no reason to spend the extra time in writing system independent code in Java. In fact, since compatibility of even &quot;pure&quot; Java is not guaranteed between the platforms, software still needs to be tested across platforms and modified accordingly. In fact, if Java is discarded completely, cumbersome Java classes and the Java interpreter need to be distributed. Furthermore, Java AWT isn't well suited to developing user interfaces, and can take an enormous amount of time to design compared to Visual Basic for example.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Against : Windows 95/NT isn't the only OS around</th>
</tr>
</thead>
<tbody>
<tr>
<td>A lot of the people still use other OSs. To target just a</td>
</tr>
</tbody>
</table>
Resolution:

Use proposal 2: to write partially dependent code.

Method: Detect for a certain system (e.g. win32) and give the user additional functionality or enhancement of the base functionality.

Advantages: There is the option to include enhanced versions, yet be able to run on any system with a Java VM, with the same code. It is also easier to interface with the SmartCard through serializable interfaces once someone writes code for it in Java.

Disadvantages: Everyone needs to learn Java and those who are writing the UI for external system assistants needs to know AWT. AWT can be time consuming to learn.

9.2.4. Issue: How should the Vehicle subsystem interface to SA/RT?

SA/RT is a seat simulation provided by Daimler-Benz. It will be used by the Vehicle subsystem to simulate seat movements when VIP makes a setSeatPosition call, and provide VIP with the seat positions when VIP makes a getSeatPosition call. Currently, the SA/RT provided by Daimler-Benz starts by reading the initial seat position from a file. It allows the seat to be moved only through a Graphic User Interface (GUI). New seat positions can be saved by clicking on the "Store" button of the GUI. This poses a problem to the Vehicle subsystem because it cannot interact with SA/RT through a GUI. New seat positions issued by VIP cannot be transformed into mouse clicks to change the seat position through the GUI.
Proposal 1:
Obtain the libraries from your colleagues in Daimler-Benz to facilitate our own modifications and recompilation of the current SA/RT system. The SA/RT needs to be modified so that AIM can interact with it via identifiers and parameters rather than through the given GUI.

For:
- The graphical display of the current SA/RT need not be rewritten.
- Increased understanding on the API of SA/RT.
- Client would more readily accept testing with the provided SA/RT.

Against:
- The libraries are licensed and would cost too much to be transferred to CMU without breaking the copyright law.
- Modifications to current SA/RT may not be merged successfully with future versions of SA/RT provided by Daimler-Benz.

Proposal 2:
Request the engineers at Daimler-Benz to modify the current SA/RT so that it will read/write the parameters of the seat (e.g. the position) from/to a file in ASCII format.

For:
- The file interface is simple to implement and understand.

Against:
- The SA/RT must be modified by the engineers in Daimler-Benz. The time taken for modification and delivery of the new SA/RT is uncertain. Any lengthy delay will adversely affect the timely delivery of the JAMES system on the 9 Dec 97.

Proposal 3:
Create an emulator for the SA/RT using JAVA AWT. This emulator will have the same functionality as the current SA/RT. It differs from the current SA/RT in terms of accepting inputs. Instead of getting seat position through a GUI, the emulator will accept parameters from AIM.

For:
- Since the emulator is written in JAVA, it can interact with the AIM (AIM is already equipped with a JNI wrapper) directly through RMI without the need for an additional layer of JNI wrapper over the emulator.
- The estimated time for writing the code and making the emulator fully functional is 2-3 days. Since the time for delivery is known, there is no risk of uncertainty.
- Any errors can be fixed instantaneously because the code is accessible.
- Increased understanding of the code of this emulator can expedite the time taken to integrate it to the rest of the Vehicle subsystem.
- This emulator will function like the original SA/RT.

Against:
- Client may not readily accept testing with this emulator because it is only a substitute for the original SA/RT.
- This emulator may not be easily understood and reused by the SA/RT research team of Daimler-Benz.

Resolution:
Once the modified SA/RT simulator arrives the simulation subsystem will determine whether it provides the necessary functionality.
Proposal 1 cannot be chosen because of the budget constraints of this project and the copyright law.
Proposal 2 cannot be considered unless the Simulation team can get a guarantee from the Daimler-Benz engineers on the timely delivery of the modified SA/RT. This is because a working prototype needs to be demonstrated on 9 Dec 97 for client acceptance and this date of presentation is non-negotiable.
Proposal 3 is favored because the Simulation team is confident of producing all the necessary functionality of the SA/RT in the emulator. The emulator can be demonstrated to the engineers in Daimler-Benz (a copy of the emulator can be posted on the bboard) for approval. Even if changes are to be made, this can be done easily and almost instantaneously because all the libraries and source code are readily available. Good documentation in the source code for this emulator can also help the engineers in Daimler-Benz to reuse and integrate the code in their future research. Furthermore, the time taken to get the emulator up and running (2-3 days) is relatively fast and certain, which helps in the scheduling of the rest of the design process.

9.2.5. Issue: What is a practical way to map individual components of the vehicle so the subsystems can differentiate between components and utilize their different attributes in an efficient manner?

A vehicle is made up of many components, some of which do not change from year to year or model to model, but some that constantly do. This issue raises the question as to how to identify components while taking into consideration: The mapping of logical components to physical components or ECUs. The changing of a component in regard to the component description already being on a component list. The replacing of a component with a new component not on the components list. Access speed. Bus ID size (bytes). Portability. Project time constraints. And, the need for a total description encoded in the bus ID verses other options.
<table>
<thead>
<tr>
<th>Proposal 1:</th>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total description in the component ID that maps every specific component type with a specific serial number ID.</td>
<td>The subsystems could retrieve the specific attributes for the specific serial number ID and transfer only the appropriate preferences to the component's ECU.</td>
</tr>
<tr>
<td></td>
<td>Even the non-functional attributes of a component could be identified, i.e. Seat color.</td>
</tr>
<tr>
<td></td>
<td>A list of these component serial number IDs would give the user a complete blueprint of the vehicle.</td>
</tr>
<tr>
<td>Against:</td>
<td>Component list would have to be updated every time a component is changed.</td>
</tr>
<tr>
<td></td>
<td>Non-functional attributes not needed to be known by user to control functional attributes.</td>
</tr>
<tr>
<td></td>
<td>Accessing component list would be slower due to larger size of list.</td>
</tr>
<tr>
<td></td>
<td>Component list would be very large if it's to contain every attribute of the components.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposal 2:</th>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify the vehicle by model and have a general description number as a component ID and then have the ECU accept the most attributes as the physical device can handle.</td>
<td>The subsystems could retrieve functional attributes for the general component and transfer the preferences to the component's ECU.</td>
</tr>
<tr>
<td></td>
<td>Component list would not have to be updated every time a component is changed because the component would fall under a general category.</td>
</tr>
<tr>
<td></td>
<td>Generic GET/SET commands could be used to adjust components.</td>
</tr>
<tr>
<td></td>
<td>Non-functional attributes not needed to be known by user to control functional attributes.</td>
</tr>
<tr>
<td></td>
<td>Accessing component list would be faster due to smaller size of list.</td>
</tr>
<tr>
<td></td>
<td>Component list would be much smaller without the non-functional attributes of each component.</td>
</tr>
<tr>
<td>Against:</td>
<td>Mapping of logical components to physical components would not be precise. Non-functional attributes of a component would not be identified, i.e. Seat color.</td>
</tr>
<tr>
<td></td>
<td>Bus IDs must be known to all subsystems. Therefore, the subsystems cannot use more readable command like SetSimpleSeat.</td>
</tr>
<tr>
<td></td>
<td>Bus IDs could grow past fixed byte size if functional components increase beyond expectation.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposal 3:</th>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple &quot;AdjustXComponent&quot; commands mapped to each component type. Do not have component bus ID.</td>
<td>Mapping of logical components to physical components would be precise. Only data to</td>
</tr>
</tbody>
</table>
numbers but have multiple "AdjustXComponent " commands mapped to each component type that would transfer only preferences that are specific to the attributes the physical component can handle, (i.e. AdjustSimpleSeat, AdjustComplexSeat.)

known attributes would be sent to components ECU.

- More readable commands. The component bus IDs would not have to be learned by the subsystems.

Against:

- Multiple commands would have to be used for each group of component.
- Multiple commands would have to be designed present to the complexity of the component.
- Component's complexity type would have to be given to subsystems before subsystems could invoke desired SET commands.
- Non-functional attributes are not given to subsystems. Only functional attributes would be given to subsystems.
- New and more complex components would require new coding of commands to accept new attributes.

Resolution:

Proposal 2: Identify the vehicle by model and have a general description number as the component ID.

The main objective being discussed in this issue is the ability to set or adjust a component to its fullest capacity. The stake holders, the other subsystems, the client, and project management, all want to see total functionality of the vehicle components. Total component identification as depicted in proposal 1 achieves this but at the cost of slowing the system down due to trying to access large detailed component files and at a cost of a great deal of coding to serialize every component. Proposal 3 also achieves total functionality of the components but again the coding time and the ability to add or replace components could be burdensome. Proposal 2 however, transfers the necessary data to the component's ECU and doesn't get involved with non-functional data that may indeed be attributes to the component but are not vital to the ECU setting the component.

Therefore, due to the time constraints of the project and the amount of memory on the smart card, the most practical way to map components is to identify the vehicle by model and then have a general description number as a component ID. Then have the ECU accept the most attributes as the physical device can handle.

9.2.6. Issue: Where should the Logbook subsystem reside?

A fundamental question facing the Logbook team is the hardware allocation of the Logbook subsystem. Currently, the Logbook subsystem must run for the most part in the vehicle because of the client's requirement that all trips be recorded (even when the SmartCard is not in the reader) and the SmartCard acts as a transfer medium between the vehicle and some external system. However, it has been suggested that there are many advantages to putting subsystems on the SmartCard instead of in the vehicle. Because this will make the recording of every trip impossible, maybe some compromise can be reached.
<table>
<thead>
<tr>
<th>Proposal 1: Implement on the SmartCard</th>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Easy to upgrade/fix: Fixing or upgrading the Logbook subsystem software would simply involve issuing a new SmartCard.</td>
</tr>
<tr>
<td></td>
<td>• Better security: Security would be enhanced because only the Java VM processor on the SmartCard itself would have access to the data on that SmartCard.</td>
</tr>
<tr>
<td></td>
<td>Against:</td>
</tr>
<tr>
<td></td>
<td>• Doesn't satisfy client's requirements. Trips may be lost if the SmartCard is not in reader.</td>
</tr>
<tr>
<td></td>
<td>• More limitations: Implementing exclusively on the SmartCard introduces memory constraints which limit the # of trips which may be recorded before a transfer of data to the external system must occur and makes the implementation more difficult in terms of the amount of memory the assistant can use.</td>
</tr>
<tr>
<td></td>
<td>• No time: Moving the implementation to the SmartCard exclusively would require the change of all the documents and a complete redesign of the system which there is no time for before the end of the semester.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposal 2: Implement in the vehicle</th>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Speed: The vehicle CPU and storage device will undoubtedly be faster than the SmartCard thus increasing the speed of the Logbook subsystem.</td>
</tr>
<tr>
<td></td>
<td>• Less limited: Implementing in the vehicle lessens the limitations of the # of trips that can be recorded, memory used by the assistant, etc.</td>
</tr>
<tr>
<td></td>
<td>• Already planned: Because the Logbook team been planning to do it this way, they would not have to rewrite all of the documents and redesign the whole subsystem.</td>
</tr>
<tr>
<td></td>
<td>• Satisfies client's requirements: Implementing in the vehicle allows for the satisfaction of the client's requirement that all trips be recorded. If not, a trip might be lost if the SmartCard is not in the reader.</td>
</tr>
<tr>
<td></td>
<td>Against:</td>
</tr>
<tr>
<td></td>
<td>• Too hard to upgrade/fix: Fixing or upgrading the Logbook system software in the vehicle would be difficult requiring a professional to open up the car and get access to the onboard computer.</td>
</tr>
<tr>
<td></td>
<td>• Lessened security: Because the data on the card has to be read by the computer in the vehicle, it would not be hard for someone to write an assistant to read this data and use it for their own purposes.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposal 3: Implement on both the SmartCard and in the vehicle</th>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Doing both increases scalability: Implementing</td>
</tr>
</tbody>
</table>
on both increases scalability because the information in the database can be used by other "cardlets" in the future that have their own GUIs, etc.

- Satisfies client's requirements: Implementing partly in the vehicle allows for the satisfaction of the client's requirement that all trips be recorded. If not, a trip might be lost if the SmartCard is not in the reader.

Against:

- Too late: It is too late to change now. It would involve rewriting all of the documents and redesigning the whole system.

Resolution:

Implementing exclusively on the SmartCard is not practical because it would be impossible to satisfy the client's requirement that all trips be recorded. Therefore, the only two options left are implement exclusively in the vehicle or come to a compromise. Although a compromise would solve many of the problems of implementing exclusively in the vehicle, the simple fact that there is no time to change everything so late in the semester makes this option unrealistic. Therefore, the Logbook team must continue with what it's been doing and implement exclusively in the vehicle.

---

9.3. Issues related to data management

9.3.1. Issue: What is the most optimal solution to synchronize the maintenance items?

Maintenance items can be fragmented between the card and the dealer database. This could happen if the car is serviced at a dealer who has no database of his own (in emergency situations). When these items exist on the card, the database of the dealer where the driver regularly goes to service his car, is out of sync. as it does not know of these new maintenance items.

Also as dealers do not share information between each other; due to sensitivity of their data; even though two dealers could have a database and a driver goes to these two dealers to get his car serviced; a synchronization problem could still exist, due to lack of information sharing.
<table>
<thead>
<tr>
<th>Proposal: Use the card to store the maintenance history</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storing the maintenance history on the card is a logical way to store data about a vehicle.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- One step Synchronization: Storing the history on the card makes the synchronization process easy and is performed in one single step, when the dealer inserts the card into his card reader at the shop. The maintenance history on the card is compared with the maintenance history on the dealers database and any missing items are updated. Thus the synchronization process is completed in a single step.</td>
</tr>
<tr>
<td>- Data is not fragmented.: By using the card to store the maintenance history, all data will remain on the card and in ONE physical location. Thus all maintenance data pertaining to the car will not be fragmented between dealer databases and the card.</td>
</tr>
<tr>
<td>- Users can keep their local databases or maintenance log in sync: Since the whole maintenance history will be present on the card the users can view the maintenance history without having to download it from the Internet. This enables synchronization of their local databases or maintenance history logs.</td>
</tr>
<tr>
<td>- Recommendations can be generated on the fly: The added advantage of using the card is that recommendations derived from the current vehicle condition and the past maintenance history, can be generated on the fly and can advise the driver to take certain actions depending on the priority of the recommendations.</td>
</tr>
<tr>
<td>- No communication infrastructure required: By storing the maintenance history on the card, the communication infrastructure required for complete synchronization of maintenance items between the card and the dealers is eliminated.</td>
</tr>
<tr>
<td>- No dealer dependent synchronization issues: Since the history is always on the card, this arrangement would work with all types of dealers, namely -</td>
</tr>
<tr>
<td>1. MB dealers with local databases.</td>
</tr>
<tr>
<td>2. Garage mechanics with no local databases.</td>
</tr>
<tr>
<td>3. MLCs.</td>
</tr>
<tr>
<td>4. Dealers with communication infrastructure.</td>
</tr>
<tr>
<td>5. Dealers without communication infrastructure.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Against:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Card might run out of space: Since the card has very little space, it is highly likely that the card will run out of space very fast.</td>
</tr>
<tr>
<td>- Not scalable: This proposal is not scalable as if the card runs out of space nothing can be done except to get a new card with larger storage space.</td>
</tr>
</tbody>
</table>
Proposal 2: Dealer has a local database and reads/writes new items from/to card

Whenever the dealer receives the card he queries for any new items and updates his database if any. Then he performs a set of tasks and updates the card so that the items are available for other dealers to update their own databases. It is assumed here that no communication infrastructure exists between the dealers and the MLCs.

For:

- No space constraints: Since the driver would normally go to a dealer that he regularly goes to, it is very rare that gets his car tuned up from another dealer. Thus only in these rare situations would the card have any maintenance items on it, which the dealer erases off the card when the driver takes his car back to him for service. Hence the space requirements for this proposal is very low.
- Low synchronization times: Since the number of maintenance items is low the I/O produced while synchronizing the dealer database is typically low and hence the synchronization times are low.
- No communication infrastructure is needed: Since the card is used to transfer information between dealers, no communication links are needed between dealers (via MLCs possibly).
- Low security concerns: As the card has only a little amount of data the security risk is low if the card is stolen or lost. It may of concern to only the dealer who has done the last service and hence does not affect all the dealers who have performed some maintenance job on the car.

Against:

- A user cannot do two "outside" jobs successively: If the user goes to two dealers and both of them have their respective databases, when one person performs an update, the old data on the card is over written by the new data. But this poses no problem as long as the user goes back to his original dealer and updates his history. But in the mean time if the user goes back to the second dealer or goes to a third, the old history is lost and the user’s original dealer has no way of tracing it.
- Data is too fragmented: Except for the user’s regular dealer, the data present with the other dealers will be too fragmented. They will just have bits and pieces of the maintenance history.
- Waste of space: As the data is fragmented, the space on a dealers database is essentially wasted, since he cannot reconstruct the maintenance history of the car. Thus all that he has is rendered useless.
Proposal 3: History is stored in an MLC database and dealer has access to the MLC database

The dealer first downloads the maintenance history from an MLC database. The maintenance app. then reads the maintenance data off the card compares it with the maintenance history downloaded from the MLC database. If new items are found then those are added to the MLC database. It is assumed here that all the Mercedes Benz dealers have a Internet / Intranet connection to the MLC database in their region.

For:

- Synchronization problems are reduced to a low degree: Since this architecture proposes a central database for all the dealers, within an area, and as it is unlikely that a user goes to dealers outside a particular radius, the database is always in sync.
- No fragmentation: As all the dealers in the area update the same MLC database there is no data fragmentation.
- Space requirements for the card are minimal: Since the maintenance data is never stored on the card, except when the user is outside his regular area, storage requirements of the card are minimal.
- Security problems are reduced to a minimum: No security problems exist for such an architecture as no data is present on the card. Security issues can exist if the car is serviced at a dealer outside the user's normal area. However as this would be rare, security risks are low.
- Low synchronization times: Since the synchronization process mostly consists of updates to the MLC database via an Intranet / Extranet link very high bandwidths are available as opposed to synchronization bandwidths available on RS232 interfaces. However once in a while synchronization needs to be done between the maintenance items on the card and the MLC database (when the user services his car outside his area), but since these incidents are relatively low in number the average sync. times are low.
- Scalable: Since MLC databases can be easily scaled in size and bandwidth, this option is highly scalable.

Against:

- A large communication infrastructure is needed: As all the Mercedes Benz dealers have to be connected via an Intranet/ Extranet a large communication infrastructure is needed and it may not be in the best of interests of Mercedes Benz to spend on such a network.
- User has no way of updating his personal maintenance log: The user however cannot maintain a log of his own as most of the times no maintenance items are stored on the card. Hence unless he gets an explicit account on the MLC databases, he has no way of viewing the maintenance history of the car.
Proposal 4: Use mail slots for dealers!

A pre selected number of dealers can be issued access to the car. This can be done by the user initially when he buys the car or activates the card in a way similar to what he does while selecting a Primary Care Physician for his medical care. After having done that specific slots of maintenance items are created for these dealers along with a slot for a wild card dealer for emergencies. When a dealer performs a maintenance task he updates the slots of the other dealers. Thus each dealer looks for items in his slot. If items exist he updates his local database. Apart from the ability to provide for a number of dealers this proposal resembles the "Use card for maintenance history" proposal.

NOTE: The wild card slot is not updated. The wild card slot is used only to write to in case of "non regular" dealers. Also since these dealers are "non regular" and enough number of different dealers are already chosen by the user "non regular" dealers do not store any maintenance items about the car on their local database.

For:

- Support for multiple "regular" dealers: Since dealers have their own slot, a number of dealers can be supported. This makes it more comfortable for a user especially if he regularly goes to two dealers, probably one when he wants to service the car on a working day and the dealer is on his way and the other who is near his house.
- Data will not be fragmented, hence no wasted space: Data will not be fragmented as all the dealers will have all the information about the maintenance history of the car. Hence space will never be used up on a dealer's machine by useless information and this will result in no wastage of space. Of course this is however restricted by the size of the storage space on the card.
- Data synchronization between dealers is high: Provision of slots increases the chances of synchronization of a number of dealers.
- No communication infrastructure required: Since the data exists on the card, no communication infrastructure is required.
- Users can keep their local maintenance log in sync: Since all the data is going to be passed via the card, the users have a chance to maintain their local maintenance logs in sync. They also can afford to do this even after a couple of services as the data is always stored unless he goes to each dealer in a round robin fashion and each dealer updates his database and frees up the space in the mail slot, resulting in the original information being lost (from the user's perspective).
- One step synchronization: Since the dealer just queries his own mail slot and uploads all data that he does not find in his local database, the synchronization is achieved in one step automatically.

Against:

- Synchronization times are high: Since the dealer has to check his slot for any new updates, synchronization times can get relatively high due to the limited bandwidth of an RS232 interface.
- Space constraints: If a user goes to a dealer regularly and does not distribute his visits regularly amongst the other dealers, the other dealer's slots could get filled up and they could run out of space.
- Possible security risk: Since data will always exist on the card, unless the user selects only one dealer, the possibility of a security risk exists if the card is lost or stolen.
- Redundant Data: Since each dealers' mail slot has to be updated, there is redundant data sitting...
Resolution:

Proposal 1 is a very good option, but currently the card is not going to be able to hold the entire history due to space constraints as argued by Argument 7. This could be an option to look into, in the future. Proposal 2 is a down to earth option as far as the current architecture of the card is concerned. However it poses a significant number of synchronization problems, if the user is not diligent enough in carrying out his maintenance tasks in the appropriate order as argued in Argument 5. It also results in data fragmentation and wasted space at a "non regular" dealer's site as argued in Argument 6 and Argument 7 respectively. Therefore Proposal 2 is dropped. Proposal 3 is by far the best proposal to keep all the maintenance data in sync., however Daimler Benz would incur a huge cost, just to set up the infrastructure. This from the client's perspective it is unimplementable and so this proposal is dropped too. Proposal 4 is sort of a compromise between the space constraints as argued by Argument 7 of Proposal 1; since although the space requirements are higher than Proposal 2 they are certainly lower than Proposal 1; and relieves the user from having to be very careful. It also results in no wasted space for the "non regular" dealers. Thus it takes care of the problems proposed by Arguments 5 and Argument 6 of Proposal 2. Also normally a user would register no more than two dealers that he uses regularly, and so Argument 10 of Proposal 4 does not hold for most cases. Thus it can be seen that Proposal 4 is by far the best choice for synchronization and is implementable at a low cost.

9.3.2. Issue (sub issue of above): How much of another dealer's information should the dealer be able to see?

The amount of information that a dealer can see depends on the sensitivity of the data. Thus how much information in the maintenance items is sensitive is needed to be determined.
Proposal 1: See all the information

For:
- Easy troubleshooting: Since the whole information is available all the details of the past services are available and this aids the mechanic in his troubleshooting job.

Against:
- Security Violation: Some sensitive data might be in the maintenance record and hence could violate security.

Proposal 2: See all the information except details of other dealers

For:
- Sensitive information is protected: Sensitive information which gives rise to security risks is protected.

Against:
- Dealers might not give enough information: Due to sensitive issues arising from competition dealers might not provide enough information needed by another dealer.

Proposal 3: Not able to see any maintenance tasks not performed by him!

The dealer has no access to maintenance items that he has not performed. This is to protect dealer sensitive information.

For:
- Maximum Security: Since no information about services done by another dealer are available to him, this produces maximum security of dealer specific information which is sensitive.

Against:
- Useless: The databases of dealers will always be out of sync. with each other.

Proposal 4: Provide a template to all dealers which will be used to fill in shared information!

A template will be provided which will take care of hiding sensitive information. At the same time however this template will force the dealers to enter information about all the fields, without which the maintenance app. will not go to the next screen or task.

For:
- High Security: Since all the dealer specific information will be concealed, this option presents high security.
- Easy troubleshooting: The template forces the dealers to enter the necessary information and hence this aids the other dealers / mechanics in troubleshooting, as it should.

Resolution: Provide a template to all dealers which will be used to fill in shared information.

Since Proposal 1 disregards the need for security of sensitive dealer specific information, this is obviously not the right choice. Proposal 3 on the other hand provides no information about past services done to the car, to the dealer if he has not performed them. This is indeed inadvisable as it makes the job of the mechanic very difficult, especially while troubleshooting. Proposal 2 however strikes in between and provides only necessary information, thereby protecting dealer specific sensitive information. The Argument 2, which is against Proposal 2 is dealt with by using Proposal 4. Therefore Proposal 4 is best choice.

9.3.3. Issue: Should the trips be divided into legs?
Trips can be divided into legs, which would show the information about when the user started up and shutdown the car. Legs are not directly in the requirement, but are a good feature to provide. Dividing the trip into legs can give more information to the user, but may increase time to market and if implemented wrong, may add to confusion for the user.

<table>
<thead>
<tr>
<th>Proposal 1: Divide the trip into legs</th>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td>This would mean that each trip is divided into legs. These legs are started when the user starts up a new trip or turns on the car, and are ended when the trip is ended or turns off the car.</td>
<td>• Dividing the trip into legs gives fuller information about the trip. Legs provide a very good way to organize the trip and to display the information. • Even if the information isn't necessary, it helps modularize. The information doesn't necessarily have to be directly presented to the user. It can be stored away and presented in a more efficient way. The user might not ever see an object called &quot;leg&quot;.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Against:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• This extra information might not be necessary. This information is not directly related to the requirements. They do not mandate legs. So why should they be implemented? • Not enough space. Space on the current SmartCard is quite restricted. This might not be a consideration because JAMES is designed for the future and more efficient SmartCards will be available.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposal 2: Don't divide trips into legs</th>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td>For:</td>
<td></td>
</tr>
<tr>
<td>• Are legs really necessary? • Legs would add to implementation time. Legs can potentially add to the implementation time, and as time appears to be our most significant constraint. So it might be in our interest not to implement them.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Against:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Legs would marginally add to implementation time. Theoretically the time spent implementing legs should be minimal; as it is essentially functionality that would be passed on to the trip object. A trip without legs could be thought of as a trip with a single leg.</td>
</tr>
</tbody>
</table>

Resolution:
It has been decided in favor of the first proposal; legs will be used to implement the project.

9.3.4. Issue: Should everything be on the SmartCard?
It is a simple and convenient choice to have all the information on the SmartCard. Of course, there are arguments for and against this choice.
<table>
<thead>
<tr>
<th>Proposal 1: Keep everything on the SmartCard</th>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Convenience: This choice minimizes the load on the user (owner of the car or a dealer). They do not have to worry about any other places where relevant information can be stored and whether they can access it.</td>
</tr>
<tr>
<td></td>
<td>• Security considerations: If all the information resides in one place (the SmartCard) it is easier to protect it (or certain part of it) from unauthorized (properly) access.</td>
</tr>
<tr>
<td></td>
<td>• Consistency: If all the information is stored in one place (the SmartCard) there are no problems of consistency.</td>
</tr>
<tr>
<td></td>
<td>Against:</td>
</tr>
<tr>
<td></td>
<td>• Avoiding an unnecessary overload: Clearly, there exists relevant information that need not be on the card. Consider the following example. Assume a CD-ROM with maps of the area in the CD-ROM reader in the car. The SmartCard need not copy maps from the CD-ROM to itself. Also, among the different items stored on the card, some may have much higher priority than others. Example: the date of the last oil change is much more important than the dates of oil changes before that.</td>
</tr>
<tr>
<td></td>
<td>• Backups. The possibility of the SmartCard being lost or broken needs to be taken into account. Any really important information has to be backed up. That means not everything is on the SmartCard.</td>
</tr>
<tr>
<td></td>
<td>• Non practical (at least for now): The storage space on the SmartCard is very small (of the order of KiloBytes) at the moment. The amount of relevant information is much bigger.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposal 2: Keep everything in a database</th>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Realistic: Storage on the card is limited. Moving all the data to a database solves this problem.</td>
</tr>
<tr>
<td></td>
<td>• Safety: A stationary database is much less likely to crash as compared with a SmartCard (and information can be duplicated on separate computers). Not to mention that it is much more difficult to loose a database than a SmartCard.</td>
</tr>
<tr>
<td></td>
<td>Against:</td>
</tr>
<tr>
<td></td>
<td>• Consistency: Redundant information raises redundancy issues.</td>
</tr>
<tr>
<td></td>
<td>• Convenience: The problem of consistency mentioned in the previous argument leads to a much worse one: who is going to do the upgrades in the databases ??</td>
</tr>
<tr>
<td></td>
<td>• The Card??? What for? For some assistants like Maintenance if everything is in a database it may be simple to work directly with the database. If the card does not help us to interact with the database why use the card at all ??</td>
</tr>
</tbody>
</table>
Resolution: To have InformationManager on the Card

It is possible to resolve the issue in a way which incorporates all the good things from both proposals and does not seem to have their drawbacks. I suggest to put an agent InformationManager (IM) on the card and let it handle the information. IM will assign priority to different items and move them in and out of the Card according to the priority. It will ask the user to insert the card into the card reader of his PC or to go on the Net when the Card is inserted, but it will never ask for anything more complicated than that. It will "ask" by sending some text on the cars display or speak out using speakers in the car before the card is taken out. From the point of view of the user all the information is on the Card.

9.4. Issues related to Global Resource Handling

9.4.1. Issue: What should be provided for authentication?
Proposal 1: Simple Login and Password

This, the simplest method, would only provide a password service. Each subsystem must then decide and enforce any access rights for the different kinds of users that use that subsystem.

For:
- Simplicity: This would likely require the least amount of time, which is important because of the limited time remaining, and would give the highest probability of having something that could be demonstrated to the client.

Against:
- Lack of Security. Obviously, extreme security problems, that would have to be solved very soon, because the subsystems would be responsible for handling their own access rights. It would also be nice to be able to demonstrate an authenticated user's limited access to the data in the system, if time allows. And, time could very well allow for at least a simple set of read/write protections as well as the login/password.

Proposal 2: Limited Read/Write Accesses as well as Login/Password

This scheme would allow for at least a small, fixed, set of different types of users (e.g. dealer, vehicle owner, etc.), and have a set of "default" access rights for each type of user. Upon login, the service would find the type of the user, and return the access rights for that user to the subsystem making the authentication request.

For:
- More Demonstrable. This would allow us to also demonstrate the ability to restrict an authenticated user's access to data. That is, the user would not be allowed to read/modify data for which that user's access to the data could result in security problems, (such as, falsifying the vehicle history so that the vehicle looks more promising to a prospective buyer...).
- More Secure than proposal 1. No longer has to trust (or depend on) the subsystems with access rights.

Against:
- Not Extensible. If rights are predetermined, then trying to add new types of users to the system would be a major headache, as well as modifying the users that exist. As a result, security holes will be tougher to fix.
- A security concern. The security would be too coarse, in allowing only read-write access to the subsystem as a whole, rather than to each piece of data that the subsystem would use. Hence, there might be cases where users could never be given a "correct" set of access rights.

Proposal 3: Like Proposal 2, but more complex Read/Write Access rights.

Each subsystem would be allowed to define the types of users that will want to use that subsystem, and the access rights that each type could have to the data in that subsystem. Upon startup of JAMES, each subsystem would give its type of users and the access rights for each user to the authentication service. This scheme would also allow for more fine-grained access rights as well, so that for each "file" on the card, each user

For:
- Best security of the three proposals. This would allow for a unix style access rights scheme, where users could have access by file, thus making a good access rights fit easier to obtain.
- Easiest to fix holes. Since the access rights are read in at startup from each subsystem, the access rights could be more easily changed, allowing for an easier time patching of security holes.
would have certain access rights, as opposed as to each subsystem.

<table>
<thead>
<tr>
<th>Against:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Feasibility for December 9. This would probably be the most difficult service to provide, and perhaps is not possible to implement for this semester. It's important to get something working and demonstrable, and if this proposal may become a time sink, resulting in nothing to demonstrate. Proposal is something that is &quot;best&quot; for down the road, but may not be doable by early December.</td>
</tr>
</tbody>
</table>

Resolution:

Each subsystem needs to work with this authentication service, so they all have a stake in this. Clearly, both client and project management will also have a stake in this issue. The current plan is that the VIP team will attempt to get services required by each subsystem, and by Thursday at 5:30 these services will be collated into one document. The primary goal is to get a login/password system going, and that each team, unless otherwise specified, is responsible for handling its own access rights for each user once authenticated. If time allows (and if found appropriate to do so), the access rights may be handled by our authentication service, with each team giving the types of users/rights wanted for each user. Ideally, Proposal 3 is best of the three proposals in terms of the long run, but for short-term concerns, proposal 1 or 2 may be more realistic.

### 9.4.2. Issue: How to overcome limitations of Java card?

Part of James project is to write subsystems in Java and have them running on the Java card. The following is the main limitations, on which this issue will be focused.

Limited resource.
- Application size is limited to 2.8KB.
- Stack size is limited to 16 Bytes.

Limited data types available.
- Boolean, byte, short are the only data types available.

Limited communication.
- No network.
- Communication buffers in smart card are byte type.

No threads
- no multithreading or garbage collection
Proposal 1: Have the subsystems running on the onboard computer.

The smart card is used only as a data storage media

For:
- Implementation of subsystems will be easier and faster: The onboard computer, can have all the features of Java and none of the limitations of the Card.
- Communication among subsystems will be easier: Having the subsystems running on the onboard computer, it will be a lot easier to communicate among subsystems. Threads, RMI, or CORBA can be used. In addition, by using temporary storage in the onboard computer to read from and write to the smart card only at the beginning and end of driving session, the needs for communicating with the smart card can be lessened even more. As a result, not only the implementation of communication part of James (such as notification) is easier, but also the overall speed of James will be faster.

Against:
- There is no longer merits of smart card. A floppy diskette can be used instead.
- After all, this is not likely what the client wants

Proposal 2: Have subsystems running on the smart card and use the onboard computer to run an assistant system

For:
- Some of the merits of smart card can be kept and while some load can be taken off the smart card by using the assistant system on the onboard computer for the purposes like communicating with the vehicle, tokenizing message streams, storing data, etc.

Resolution:

Have subsystems running on the smart card and use the onboard computer to run an assistant system

9.4.3. Issue: Should the user be able to disable authentication?

The man issue with this is whether the user should be able to disable the Authentication procedure that is run when the card is first implemented. This would give anyone with the card access to all assistants on the card.
### Proposal 1: Permit the user to disable Authentication

The user should be able to select an option on the VIP HCI screen to allow the Authentication to be disabled the next time the user inserts the card.

**For:**
- If the authentication is allowed to be disabled then any user will be able to access the system allowing multiple users to use the system without having to know the password. This would be useful in the case where the user may only have one card among many people and there is still a need to use the assistants on the card.
- The user should be allowed to disable authentication should the choose to because the user may find it irritating to constantly be logging in if they feel that they are the only one that will use that card.
- The main advantage of this is that it is simple and easy and allows multiple users to share one card and assistants and will be quick to implement which is a major issue at this time.

**Against**
- The major problem with disabling the Authentication involves security and allowing users who have not purchased certain assistants to use them. Also one must consider that disabling authentication would allow anyone who uses that card to alter the users settings at will.

### Proposal 2: Permit guest login password

The system should support a guest login option so that a generic user can be permitted to use some of the assistants but not all.

**For:**
- A guest login would be very useful for a card that is expected be used by many users. The guest login allows a generic user to use some assistants that are relevant to any user but it prevents the guest from having the ability to change the main users settings.

**Against:**
- Allowing the guest login only for generic users may limit the number of useable assistants by generic users too much. Suppose that only the main user has access to the VIP assistant but while a generic user is driving the car they wish to change to fuel efficiency settings, if they don't have access to that assistant they cannot change that setting.
- While the guest login idea is a good idea, it would be forced to have fixed user profiles and add complexity to the system. While it would be nice to have a user customizable guest profile, due to the time constraints it does not appear to be feasible.

### Proposal 3: Permit a default user profile in the event of a login failure.

The system should permit a generic login after three unsuccessful login attempts. This would permit a non

**For:**
- This would allow for a generic login without having to know any passwords. The login would only allow the use of a few assistants but
authorized user to use some of the assistants.

| would enable multiple users to use a single card. |

Against:

- The problem with having a default profile for a user with login failure is that anyone can use the card even if the main user does not desire them to. Essentially it allows a stolen card to always be useful to someone which is not a desirable option.
- The problem with this idea is that there is very little time left to complete items of complexity with this could easily developed into. It is something that could be added in an extension, but there is very little time now.

Resolution:

The main problem with this issue is time. Ideally one would want to have a guest login that can be customizable by the primary user, but due to time constraints that may not be able to be implemented in time. With that in mind I would recommend allowing the user to disable authentication if they choose to. This allows the user not to be annoyed by being required to login every time they insert their card. This also allows for multiple users to be able to use the same assistants if desired with little change to the system required. This option is only really valid while only one user can be stored on the card. When the card is advanced enough to store multiple users on the card, this will no longer be necessary except as a comfort feature for the user. At the point where the card is sufficiently advanced the recommendation with be for a user specified guest profile for non registered users. This provides the best combination of functionality and security.


9.5. Issues related to software control

9.5.1. Issue: Which remote method implementation to use?

If the assistants run on the card and the services and the vehicle object on the onboard computer, a remote method mechanism is needed. Currently the two major options available commercially are CORBA and RMI.
<table>
<thead>
<tr>
<th>Proposal 1: Use CORBA</th>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• CORBA will allow multiple language implementation. This is important because currently the legacy systems is written in C++ and our code is going to be in Java. Therefore CORBA is needed to communicate between Java and C++ objects. Even if everything was written in Java, we should allow for future change of languages.</td>
</tr>
<tr>
<td></td>
<td>Against:</td>
</tr>
<tr>
<td></td>
<td>• CORBA does not fit on the current card. CORBA classes size up to roughly 400K, this 100 times the size of the card.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposal 2: Use RMI</th>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• RMI is cleaner, easier to implement No need to worry about type marshaling in RMI. There is also less overhead in both performance and programming complexity.</td>
</tr>
<tr>
<td></td>
<td>• RMI is much cheaper RMI is part of the JDK, which relatively inexpensive. Commercial licenses for CORBA run around $20K.</td>
</tr>
<tr>
<td></td>
<td>Against:</td>
</tr>
<tr>
<td></td>
<td>• RMI is not supported on the Java Card Java RMI is not supported on the current version of SmartCard.</td>
</tr>
<tr>
<td></td>
<td>• Does not allow multiple languages implementation. This is bad because if the language were to change in the future a new software bus would have be implemented.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposal 3: Write our own implementation of remote method calling</th>
<th>For:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• If we make a &quot;custom-made for JAMES&quot; protocol, all the features of CORBA need not be included. But it is not guaranteed that it will fit on the card.</td>
</tr>
<tr>
<td></td>
<td>Against:</td>
</tr>
<tr>
<td></td>
<td>• We don't have the expertise in this area to be able to develop such protocol quickly. We have a limited resource such as time and expertise. This is out of our scope.</td>
</tr>
<tr>
<td></td>
<td>• If new subsystems are added on in the future, it will take extra work to make them talk to our special protocol. This would make our system less extensible and hurt the buzzword compliance.</td>
</tr>
</tbody>
</table>

9.5.2. Issue: Should we implement and demo assistants on the card, or
### Proposal 1: Use the Card as a "Smart" storage device with the assistants themselves running on the onboard computer

**For:**
- This is a reasonable way to deal with size limitations of the card. Time is running out and we need a way to deal with the size limitation.
- Once larger capacity cards become available, the assistants could be moved to the card. Because CORBA and RMI make the location of the objects transparent to the clients, we can later on (when the size of Java Cards increases) move the corresponding objects onto the card without having to change the software bus or topology of the system.

**Against:**
- Might be against professor's expectations of having assistant run on the card.

### Proposal 2: Simulate a larger capacity card on a computer and run one of the remote method brokers from there.

**For:**
- Could easily be moved onto the card when large capacity cards become available. There are larger capacity cards predicted to come out in the future. The top level topology won't have to change.

**Against:**
- Might be against professor's expectations of having assistant run on the card.

### Resolution:

The final resolution lies with the approval of the professor, client and the Architecture team. I would propose to combine proposal A (use CORBA) and sub_A (Use the Card as a "Smart" storage device with the assistants themselves running on the onboard computer).

CORBA is a standard implementation of remote method invocation. It will allow to utilize different language implementation within the system. Since CORBA doesn't fit on the card, I would propose to move the assistant objects on the onboard computer and use the card mainly for storage of trips, user preferences, etc. When larger cards capable of supporting CORBA become available, those objects can be moved on the card. Because of CORBA's location transparency moving the object to a different location, shouldn't pose a problem for the overall topology of our system.