# SMART PLATFORM DEVELOPER’S GUIDE

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>THE FEATURES AND DESIGN OBJECTIVES OF SMART PLATFORM</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Dynamically Find and Join</td>
<td>2</td>
</tr>
<tr>
<td>2.2</td>
<td>The combination of Delegated Communication and Peer to Peer Communication</td>
<td>3</td>
</tr>
<tr>
<td>2.3</td>
<td>Automatically Manage and Resolve of the Agent Dependencies</td>
<td>3</td>
</tr>
<tr>
<td>2.4</td>
<td>XML based ICL (Interagent Communication Language)</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>ARCHITECTURE</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>THE DS</td>
<td>4</td>
</tr>
<tr>
<td>4.1</td>
<td>The Startup of DS</td>
<td>4</td>
</tr>
<tr>
<td>4.2</td>
<td>Alter the default listening address of DS (Setup two separated Smart Platforms environments on a LAN)</td>
<td>5</td>
</tr>
<tr>
<td>4.3</td>
<td>Internal Structure of DS</td>
<td>5</td>
</tr>
<tr>
<td>4.4</td>
<td>The Agent Registers to the Directory Service</td>
<td>6</td>
</tr>
<tr>
<td>4.5</td>
<td>Blackboard</td>
<td>6</td>
</tr>
<tr>
<td>4.5.1</td>
<td>The Message Group</td>
<td>6</td>
</tr>
<tr>
<td>4.5.2</td>
<td>The built-in message group----SysEvent(System Event)</td>
<td>7</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Message Format</td>
<td>7</td>
</tr>
<tr>
<td>4.6</td>
<td>The Facilitator</td>
<td>8</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Three Kinds of Status of Agent</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>THE CONTAINER</td>
<td>9</td>
</tr>
<tr>
<td>5.1</td>
<td>The Startup of a Container</td>
<td>9</td>
</tr>
<tr>
<td>5.2</td>
<td>Container Find and Join the DS</td>
<td>10</td>
</tr>
<tr>
<td>5.3</td>
<td>Agent Joins the Smart Platform via Container</td>
<td>10</td>
</tr>
<tr>
<td>5.4</td>
<td>Configure Your Container</td>
<td>10</td>
</tr>
<tr>
<td>5.5</td>
<td>The GUI of Container</td>
<td>12</td>
</tr>
<tr>
<td>5.6</td>
<td>Kill an Agent</td>
<td>12</td>
</tr>
<tr>
<td>5.7</td>
<td>Start a Previously Running Agent</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>MONITOR AGENT</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>INSTALL AND RUN SMART PLATFORM</td>
<td>14</td>
</tr>
<tr>
<td>7.1</td>
<td>Installation of Smart Platform</td>
<td>14</td>
</tr>
<tr>
<td>7.2</td>
<td>How to Run Smart Platform</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>THE ADK (AGENT DEVELOPMENT KIT)</td>
<td>15</td>
</tr>
<tr>
<td>8.1</td>
<td>The VC++ version ADK</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>a) Step 1: Import the ADK Static Library</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>b) Step 2: Include the Header file When Necessary</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>c) Step 3: Initialize the COM</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>d) Step 4: Inherit the CAgent class</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>e) Step 5: Overload some virtual methods of the CAgent class</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>f) Step 6: Call the Methods of CAgent Class: Register and Subscribe.</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>g) Step 7: Publish Message when Necessary</td>
<td>17</td>
</tr>
</tbody>
</table>
1 Introduction

As an important notion in the vision of Ubiquitous Computing, Smart Space or Intelligent Environment is defined as an enhanced a physical space where people can get the services of computer systems without approaching to computers or using the cumbersome keyboard or mouse to interact with them. Typically a Smart Space is a distributed system that involves many sensors, perception devices, software modules and computers. To develop and support such a complicated system, some type of software platform is a must.

Smart Platform is just designed and developed as a solution for this software platform of Smart Space. This Developer’s Guide describes the features and the architecture of Smart Platform, and provides a description of how to develop an agent on this platform.

We just assume a general familiarity with software engineering practices and programming language concepts. Some familiarity with the basics of the XML language is also helpful, as XML serves as the basis communication language throughout the whole platform.

2 The Features and Design Objectives of Smart Platform

The Smart Platform is modeled as a Multi-Agent System, in which the basic unit is agent. Each agent is an autonomous process that either contributes some services to the whole system or uses the service of other agents to achieve a specific goal. As a software platform, Smart Platform implements a runtime environment for the agents and an agent develop kit, which include a programming interface and some debug tools, for the developers.

To accommodate the characteristics of Smart Space system, Smart Platform is designed with the following features.

2.1 Dynamically Find and Join

In the traditional way, if we want to access a service on the network, we must have the prior knowledge of the access address of the service, i.e. the IP address of the server and the port of the service on the server. While typically there will be dozens of agents in a decent Smart Space system, it is so tedious if we have to manually interconnect every agent. Instead, in the “find and join” mechanisms, Smart Platform uses IP multicasting to dynamically discover and assemble the computation environment, eliminating the need of manual configuration.
2.2 The combination of Delegated Communication and Peer to Peer Communication

In Smart Platform, we provide two different communication modes, “Delegated” and “Peer to Peer”. In “Delegated” mode, all the messages between agents are mediated by a special runtime module called DS. Agents post and accept messages without caring who is the other party. DS will forward the messages to the proper agent according to the contents of the message. Thus, the agents are significantly decoupled with each other. However, in this mode, the DS is burdened with a heavy load, so it is not suitable for the case where agents need high volume of data exchange. On the other hand, in “Peer to Peer” mode, the connection is established between agents directly, this is for the case where two agents need high bandwidth and real time communications. Here DS also plays an important role for helping the two agents to setup and maintain the communication channel.

Agents are free to select the proper communication mode according to their specific requirements.

2.3 Automatically Manage and Resolve of the Agent Dependencies

In a multi-agent system, agents must collaborate with each other. One agent may use the services provided by another agent. We called this relationship as agent dependence. The topological structure of the dependencies between agents may be a tree or even a net. The Smart Platform can manage and resolve those dependencies.

When each agent joins the computation environment, it must announce the services it provides and the services it depends. Smart Platform will store this information in a persistent storage. If an agent starts up asking for a service and the agent, which provide this service happens not to be in running, the Smart Platform will use the stored knowledge to locate the agents and automatically launch the agent. This feature is called “Agent Dependency Resolution”.

By this mechanism, it is no longer needed to manually start all the agents in a Smart Space system. Just start some core agents in a system; the whole system can be put into a determinate state.

2.4 XML based ICL (Interagent Communication Language)

ICL is the definition of the syntactic structure, and the semantic to some extent, of the messages between agents. We choose the XML as the basis of the syntax of the ICL in Smart Platform. The inherent advantages of XML benefit the Smart Platform in some aspects. (1) The extensibility and its ability to flexibly describe almost all kinds of data makes the ICL can be easily extended. (2) As the one of standard of technology of Internet, it eases the inter-operation of the Smart Platform with other heterogeneous systems. (3) There are many software libraries for the processing of XML in both industry and academe, which ease the development of the Smart Platform.
3 Architecture

Smart Platform runs upon network-connected computers in a Smart Space. Smart Platform masks the boundary of the involved computers and provides a uniform running environment and highly structured communication model for the software modules run on the platform.

The runtime environment is composed of three kinds of components, which are Agent, Container, and DS.

1) An Agent is the basic encapsulation of the software modules of the systems.
2) Each computer participating in the runtime environment will host a dedicated process called Container, which provides system-level services for and maintains the agents that run on the same computer. In some sense, Container is the mediator between agent and DS. It makes the low-level communicate details transparent to agent developers and provides a simple communication interface for agent.
3) There is one global dedicated process called DS in the environment. The DS mediates the “Delegated” communications between agents and provide services such as agent query, dependency resolution.

Figure 1 shows the architecture of the whole Smart Platform. Later in this document we will talk about the DS, the Container, a specially agent ---- the Monitor and the ADK (Agent Development Kits).

4 The DS

4.1 The Startup of DS

Normally there should be only one DS runs on a Smart Platform; otherwise, two DS on a same platform may bring the Smart Platform into an indeterminate state. Therefore, we build into DS a mechanism to ensure there is only globally one instance of DS running in the system.

1) When a DS is started, it tries to locate a DS on the LAN using UDP multicast, just
like a Container does.

2) If (1) fails, it means no other instance of DS runs on the LAN, then it starts to listen to its multicast address and wait for Container or agent to connect.

3) If (1) succeeds, it means there is already a DS runs on the LAN, then this DS informs the user and terminates itself.

4.2 Alter the default listening address of DS (Setup two separated Smart Platforms environments on a LAN)

Although as mentioned above, Smart Platform prevents running more than one instance of DS on a LAN by default, we do provide a mechanism to do so by manually altering the default listening port and address of the DS. Consider the following case:

Group A and Group B are working on two completely different projects, they are on the same LAN and they both use the Smart Platform. If they have to share a single DS, there will be some intervention between their work, especially in the developing and debugging phase. Therefore, it makes sense if we can setup two or even more Smart Platform on a same LAN, running without intervention with each other. This is achieved by configure the DS to listen on different addresses.

When the DS is started, it loads the configuration file (Config.xml) in its working directory. The Config.xml file describes the listening multicast address of the DS. If there is no such a file or the file format is wrong, the DS will use the default address (234.5.6.7:5678). The following is a sample of the Config.xml file:

```xml
<Config>
  <DSConfig>
    <DS_Multicast_Address>234.5.6.7</DS_Multicast_Address>
    <DS_Multicast_Port>5917</DS_Multicast_Port>
  </DSConfig>
  <ContainerConfig>
    <AutoReTryInterval>30</AutoReTryInterval>
  </ContainerConfig>
</Config>
```

*Fig. 2 Sample of Config.xml*

The `<DS_Multicast_Address>` element describes the *Multicast IP* on which the DS listens, this must be a D class IP, or else the DS may fail to listen to the address. The `<DS_Multicast_Port>` element describes the *Multicast Port* on which the DS listens. The `<ContainerConfig>` element is intended for the Container, so we do not care about it here.

4.3 Internal Structure of DS

The structure of the DS can be divided into three subcomponents, each provides a different set of services, as illustrated in [Figure 1](#).

1) The Directory Services subcomponent provides services such as agent register and query. When an agent is started, it must register its basic information to Directory Service, such as its
name, the services it provides, the services it depends and its development language. The Directory Service provides the directory query service too, that is to say, agents can query it on which host the agent provides the specified service runs.

2) The Blackboard subcomponent implements the “Delegated” mode communication service for the agents in Smart Platform. We will describe this component in detail later in chapter 4.5.

3) The Facilitator subcomponent is responsible for the management and maintenance of the agent dependencies. We will also describe this component in detail later in chapter 4.6.

4.4 The Agent Registers to the Directory Service

When an agent is started, it connects to the local container in the first step. Then the agent should register itself to the Directory Service so it can join the Smart Platform. It sends its registration information through the container to the Directory Service. This information includes its name, development language, the name of the container, the services it provides and the services it depends. When receiving those information, the Directory Service will store the name of the agent, the name of the container and the services this agent provides into the Directory Service database. This database will be used to find out agents to satisfy the dependencies. If the dependency of the agents is not null (i.e. it depends on some other service), the Directory Service notifies the Facilitator to check the dependencies. After that, the Directory Service replies a message to the container saying the agent has successfully registered to the Directory Service and the container will notify the agent in turn.

4.5 Blackboard

Blackboard plays a key role in the “Delegated Communication” mode. Every agent can publish some kind of messages on the Blackboard; it can also subscribe the desired message from the Blackboard. The Blackboard is responsible for forwarding the messages to the appropriate agents by the notion of Message Group as described below. In addition, it should also check the validity of a message when an agent publishes it. If it is invalid, it just ignores the message. (This function hasn’t been implemented in this version) Delegated Communication mode is ideal for most inter-agent communications for its inherent advantage of loose coupling.

4.5.1 The Message Group

The message group is a key notion in the Delegated Communication mode. All the defined inter-agent messages are grouped into message groups, which make a logical point where agents can publish message into and subscribe messages from. The messages in a message group usually have more commonalities than the messages in other groups. For example, all the messages about the control of the cameras may be grouped into a CameraControl Message Group. There may be many message groups on the Blackboard. A message in a group can describe either an event or a command. For example, a camera agent may publish a message to the PeopleInRoom group when it detects a person comes into the room from the door. It is also possible for a TTS (Text To Speech) agent subscribe the ArtificialVoice group. When an agent
wants the TTS agent to read a short sentence out, it may simply publish a message in this group.

If an agent is interested in a certain message group, it may subscribe this message group. When a message of this group arrives, the Blackboard will forward this message to the agent, and typically the agent will response by invoke a message processing method. (We will discuss the details later in ADK in chapter 8)

By the notion of Message Group, the agent can communicate with each other without the prior knowledge who is the other party. All they need to care is the message and the proper message group itself.

4.5.2 The built-in message group----SysEvent(System Event)

SysEvent is a built-in message group on the Blackboard. The messagees in the SysEvent group describe the status change of the Smart Platform. Only the Blackboard can publish SysEvent messages. A system event message may describe one of the following events.

a) A Container joins or leaves the Smart Platform.
b) An Agent joins or leaves the Smart Platform.
c) An Agent subscribes or unsubscribes a message group.
d) The status of the agent changes (e.g. from unsatisfied to satisfied).

4.5.3 Message Format

The word “message” in Smart Platform has different meaning for the agent developers and the Smart Platform itself. We call them user-level messages and low-level messages respectively.

User-level message is what is seen by the agent developers. While a user-level message is published and transferred in the Smart Platform, it will be encapsulated into a low-level message, which is what is seen by the runtime entities in the Smart Platform. Put it another way, the low-level message is transmitted from the Container to the Blackboard then from the Blackboard to the Container where the subscriber is in. Then the Container extract the encapsulated user-level message from the low-level message and passes it to the agent.

A user message in the Smart Platform must be a valid XML document. For example:

```
<SysEvent>
  <QueryType>New-Agent</QueryType>
  <Agent>
    <Name>Monitor alfa 1.0</Name>
    <CtnID>1</CtnID>
    <ID>2</ID>
    <DevLanguage>Visual C++ 6.0</DevLanguage>
    <Services />
    <Dependencies />
    <Status>Satisfied</Status>
  </Agent>
</SysEvent>
```

Fig.3 An example of a user-level message
The syntax of low-level messages is also based on XML. Following is the format of a low-level messages:

```xml
<Msg>
  <Command NeedReply="No">Publish</Command>
  <Sender AgtName="Blackboard" CtnID="0" AgtID="0" CtnName="Directory Services" />
  <Receivers>
    <Groups>
      <Group Name="SysEvent" />
    </Groups>
    <Agents />
  </Receivers>
  <Contents>
    <SysEvent>
      <QueryType>New-Agent</QueryType>
      <Agent>
        <Name>Monitor alfa 1.0</Name>
        <CtnID>1</CtnID>
        <ID>2</ID>
        <DevLanguage>Visual C++ 6.0</DevLanguage>
        <Services />
        <Dependencies />
        <Status>Satisfied</Status>
      </Agent>
    </SysEvent>
  </Contents>
</Msg>
```

Fig. 4 An example or low-level message

Besides the user level message that is contained in the `<Contents>` element, there are also several other fields of the low-level message are visible to the agent developers. They are the the name of the agent that publishes the message as the `AgtName` attribute of the `<Sender>` element indicate, the name of the Container where the publisher is as the `CtnName` attribute of the `<Sender>` element indicate and the `<ReplyTag>` element. `<ReplyTag>` is used to implement the query-and-return type of communication. When an agent expects it’s a query message to be replied, it will specify a ReplyTag while publish the message. When a appropriate agent wants to answer the query, it will set the InReplyTo parameter to the same value as the ReplyTag value of the received message while publish the reply message. The when the original agent gets the reply, it may know which message the current received message replies to.

4.6 The Facilitator

The Facilitator automatically maintains and resolves the dependencies of the running agents.
When a new agent joins in, the Facilitator checks whether the dependencies of this agent can be satisfied by the currently running agent. If the dependencies cannot be satisfied, the Facilitator will try to start some agents to satisfy them according to the information of the previous running agent.

Every time an agent joins or leaves the Smart Platform, Facilitator will check whether the dependencies of other agents will be affected. If that’s the case, the Facilitator will change the affected agents’ status and inform the agent of the status change.

### 4.6.1 Three Kinds of Status of Agent

As mentioned above, there are three kind of status of an agent: *Satisfied*, *Satisfying* and *Unsatisfied*. They are described as following.

- **Satisfied**: All the dependencies of this agent are satisfied by currently running agents.
- **Satisfying**: The Facilitator is now trying to satisfy the agent's dependencies.
- **Unsatisfied**: The dependencies of this agent cannot be satisfied.

## 5 The Container

In Smart Platform, a special daemon process should run on every host that participates in the runtime environment. It behaviors like an agency between agents on the same computer and the DS, we call this process the Container. It communicates with the DS, and provides the Smart Platform system services for the agents that run on the same host. For example, agents can query other agent’s status via the Container, agents can register itself to the DS via the Container, and agents can subscribe or publish message via the Container. All the agents only communicate with the container on its local host, all the low-level communications of the Smart Platform is accomplished and masked by the Container.

### 5.1 The Startup of a Container

There should be no more than one container runs on a host, so when a container is started, it will check if there is another instance of container is now running on the same host. If it is the case, it will notify the user and terminates itself.

Then the container tries to find the address of the DS and join itself to the Smart Platform. The procedure of the container joins the DS will be discussed in chapter 5.2.

If the registration fails, the Container will notify the user that no DS can be found in the LAN, the user may choose to let the container retry immediately, try again later or quit immediately. We will discuss more details on it in later chapter 5.4.

After the Container successfully registers itself to the DS, the Container will listen on the TCP port 1234 of localhost to wait for the agents’ startup.
5.2 Container Find and Join the DS

Whenever a container starts up, it will locate the DS and connect to it. As the DS is ready, it will listen on a port of a pre-configured multicast IP. We call the IP the **DS Multicast IP** and the port the **DS Multicast Port**. When a Container is started, first, it sends a UDP multicast packet to the **DS Multicast IP and Port** to discover the DS. When DS receives this packet, it will multicast its host name (IP) and its TCP listening port, so that after the container receives them, it can establish a reliable TCP connection to the DS. As soon as the connection is established, the container registers its information to the Directory Services on the DS. As we have assumed the Smart Platform runs on a LAN and there should be no more than one Container run on a host, so we just use the host name to identify a Container. The Directory Service also assigns a unique ID to the container, which is used for low-level communication between the DS and the Container.

5.3 Agent Joins the Smart Platform via Container

All the agents communicate with the Container via local TCP connection.

When an agent is started, it tries to connect the TCP port 1234 on the same host, where the container should be listening. Once connected, the Container will assign a unique agent ID to identify the agent. Then the agent try to register itself to the Smart Platform, it sends a message to the DS via the Container. We have discussed the procedure in previous chapter 4.4.

In addition, the agent also sends the full path of its executable file to the Container. The container stores the name of the agent and the path of it to a persistent storage (actually the system registry in the Windows platform), so that when the container wants to start an agent once running on the host, it can use the stored information to find out where the executable file is and launches it when necessary.

5.4 Configure Your Container

As has been discussed in chapter 4.2, user can configure the multicast address of the DS, so that more than one Smart Platform can run on the same LAN. When there are several DSs run on a LAN, the user can also configure the Container and tell it which DS to join.

Just like the DS, the container uses the configuration file (Config.xml) from its working directory to learn the **DS Multicast Port** and **DS Multicast IP**. The format of the Config.xml has been showed in Figure 2. The **<DS_Multicast_Address>** element describes the **DS Multicast IP**. The **<DS_Multicast_Port>** element describes the **DS Multicast Port**. The container will try to locate the DS via this multicast address.

If there is no such a Config.xml file or the file format is wrong, the Container will use the default address (234.5.6.7:5678).

There is also a **<ContainerConfig>** element in the Config.xml. Its child element, **<AutoReTryInterval>**, shows the interval (valued in seconds) that the container wait before automatically retry to connect to the DS, when it loses the connection with the DS or when it can not find the DS while startup and the user let it to retry laterly. If this value is non-zero and the connection between the DS and the Container do not exist, the container will try to find and join
the DS every AutoReTryInterval seconds, until the connection is re-established or the user terminates it.

If there is no such a Config.xml file or the file format is wrong, the interval is set to zero, which means when the connection is lost, the Container will notify the user of this event. It is up to the user to choose to terminate the container or try to connect to the DS immediately.

If the DS is not ready when a container is started, the container will not be able to find and join the Smart Platform. Then the container will notify the user that no DS is not found on LAN, and let the user choose what to do next. The choices are different depending on whether the AutoReTryInterval is zero or not. When the interval is not zero, three choices are available, they are “Yes”, “No” and “Cancel”. When the interval is zero, only “Yes” and “No” are available. The following is the meaning of these three choices:

Fig. 5  The Container can’t find the DS

“Yes”: Try to find and join the DS again immediately.
“No”: Terminate itself.
“Cancel”: Try to find and join the DS after every AutoReTryInterval second.
5.5 The GUI of Container

![GUI of Container](image)

Fig.6 The GUI of Container

The Fig.6 shows the GUI of the container. There are five areas on the interface, they shows the status and information of the container and some information about the DS.

Area 1 gives the container's basic information: the name of the container (that is the name of the host) and the development language of the container (in current, we only have VC++ version).

Area 2 gives the DS's basic information, if the container successfully connect to the DS, the name (IP) and the port of the DS will be showed here.

Area 3 shows all the currently running agents on the same computer as container.

Area 4 shows the information of a particular agent that is selected in area 3. The information includes the name of the agent, the develop language of the agent, the services it provides and the services it depends.

Area 5 shows the status of the container or the lastest event that takes place in this container.

5.6 Kill an Agent

Container allows the user to kill any agent managed by it. First select the agent you want to kill in the agents’ list box (area 3), click the “Kill Agent” button (Button 6 in Fig 6). Then the container will send a message to the agent to inform the agent to quit. As a programming convention, the agent should terminate itself, as soon as it receives this message.
5.7 Start a Previously Running Agent

The Container provides a helpful shortcut to run an agent which has run on the computer managed by the container before. As the container has stored the agent name with its path in a persistent storage, it is not necessary for you to find the executable file of the agent and launch it with the operating system command whenever you want to start this agent again. Just click the “Start Agent” button (7 in Fig. 6), the Container will lookup the storage, and check if the agents’ executable file recorded in the storage is still available. The container will delete the items that are not available now automatically. Then a dialog will pop up. All the agents available are listed in the dialog with their path. User can select one and launch it.

The agent information in the persistent storage is also used in the agent dependency resolution procedure. Whenever the DS asks the Container to do so, the container will look up the system registry, find the path of the agent desired, and launch it.

6 Monitor Agent

Fig. 7 The GUI of the Monitor Agent

The Monitor agent is a special agent that serves as the watcher and the debugger of the Smart Platform. The Figure above shows a picture of the Monitor’s GUI. There are total four areas on the GUI of the Monitor.

Area 1, which is the tree view of the monitor shows the basic information and status of the Smart Platform. The sub-folders of the Smart Platform folder of the tree indicate the containers that participate in the Smart Platform, while the sub-node of the Container shows the agents that run on the same host as the Container. The small icon of before the name of the agent gives the...
status of the agent. Sun(leanor)means Satisfied , cloudy(leanor)means Satisfying while dark clouds(leanor)means Unsatisfied

User can use the Monitor agent to monitor any agents or message groups in the Smart Platform for debugging purpose. The monitored folder shows the currently monitored agents or message groups. All the monitored messages will be list in corresponding sub-folders.

The Area 2 gives the status of the Blackboard. You can find out how many message groups are registered on the Blackboard and all the subscribers of a desired message group.

Area 3 shows the dependencies of the currently selected agent in the left tree view. Form the right to the left, the illustration gives the following information: the agent who use the services that provides by the selected agent, the services that provides by the selected agent, the select agent, the services that the selected agent depends on (white means this dependence is currently satisfied while red means not) and the agents who provide services for the selected agent.

Area 4 gives the detail information of the selected item in the left view. If the selected item is a folder of a monitored message group or agent, the monitored messages will be list in the list box. Double click the message you can see the contents of the message.

The right button context menu is also very useful in the GUI of the Monitor agent, you can achieve many functions list below of the Monitor agent use this menu, all these functions can ease the debugging of the agent.

- Monitor or un-monitor a select agent or group
- Kill a selected agent
- Start a agent on the selected container
- Send a message to the selected message group

7 Install and Run Smart Platform

7.1 Installation of Smart Platform

Fig. 8 A picture during the installation of Smart platform
The above picture shows an image of the installation of Smart Platform. Follow the InstallShield Wizard, you can easily install Smart Platform to your computer. The Smart Platform is a completely free to non-commercial purpose, so any serial number can be accepted by the user information dialog during the installation.

7.2 How to Run Smart Platform.

After the installation of Smart Platform, a “Smart Platform” folder is created in the Programs folder of startup menu of Windows. The shortcut or Smart Platform runtimes, such as the DS, the Container and the Monitor is placed in this folder. You can easily launch the necessary component of Smart Platform from the startup menu. Figure 9 shows this.

Both the DS and the Container will automatically enter the system tray after they are started, as the above Figure 10 has showed.

8 The ADK (Agent Development Kit)

The developers of the Smart Space system use ADK to develop agents that can be run on the Smart Platform. ADK is provided in the form class library, in which several core base classes are implemented. Through using and extending these classes, developers can rapidly build agents without the need to know the low-level details of the Smart Platform. We currently developed both VC++ version and java version of the ADK. The VC++ version is packaged in a static library form.
and the Java version is package in a JAR file form.

There are totally four classes in the ADK. However, in most case, agent developers only have to know two of them, the CAgent class and the TXML class. These two classes expose basic agent services to the developers, while others implement the low-level details of the communication in the Smart Platform. Only those developers have special purpose or permission need to take care of those low-level classes.

Now we will talk about how to develop agent using the VC++ version ADK. Developers can refer to the ADK VC++ Version Reference for more details.

8.1 The VC++ version ADK

The current VC++ version of ADK includes 3 files:

- ADK.h    the C++ header file
- ADKD.lib the debug version of the static link library of ADK
- ADKR.lib the release version of the static link library of ADK

We just assume you have installed the Smart Platform to <SPDIR>. The ADK.h can be located in <SPDIR>\Include directory while ADKD.lib and ADKR.lib can be located in the <SPDIR>\Lib directory. To use the ADK, developer should follow the following instruction.

a) Step 1: Import the ADK Static Library

Import a proper version of ADK static link library to your VC++ project. You can achieve this by configure the settings in the VC++ IDE Menu: Project -> Settings -> Library.

b) Step 2: Include the Header file When Necessary

Include the ADK.h header file, where you want to use the CAgent class or the TXML class.

c) Step 3: Initialize the COM

As the ADK used the COM to parse the XML document, the agent program should initialize the COM, before it use the ADK. It is also necessary to clean the COM when the program terminates, the following code segment shows a example of how to initialize and uninitialize the COM.

```cpp
BOOL CAgentApp::InitInstance()
{
....
CAgentDlg dlg;
    m_pMainWnd = &dlg;
    CoInitialize(NULL); //Initialize the COM
    int nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
        // TODO: Place code here to handle when the dialog is
        // dismissed with OK
    }
    else if (nResponse == IDCANCEL)
    {
        // TODO: Place code here to handle when the dialog is
```
// dismissed with Cancel
}  
CoUninitialize(); // Uninitialize the COM
......

**d) Step 4: Inherit the CAgent class.**

For example, suppose you want to develop a new agent called the TestAgent. You may define a new class CTestAgent who inherits the CAgent class. You should specify the Agent’s name, dependencies, provided services, develop language and other information when you call the constructor of the CAgent class.

**e) Step 5: Overload some virtual methods of the CAgent class.**

There are three methods can be overloaded if necessary, they are:

```cpp
virtual void OnDisconnected();
virtual void OnDependencySatisfied();
virtual void OnDependLost(CString strLeavingAgtName, CString strDepname);
```

You can refer to the ADK VC++ Version Reference for more details of those methods.

**f) Step 6: Call the Methods of CAgent Class: Register and Subscribe.**

An agent should call the Register method at the very beginning of its initialization, in order to register to the DS and participate the Smart Platform environment.

```cpp
BOOL Register();
```

If succeeds, the agent can call the Subscribe method of CAgent class, to subscribe any desired message group.

```cpp
void Subscribe(CString strGrpName, NOTIFY_CALLBACK callback, CString strTemplate="");
```

When an agent subscribes a message group, a callback function should be provided. Each time the Blackboard forward a message of this subscribed group to the agent, the given callback function will be called, giving the agent the chance to process the message. Agent should handle the message in the call back function. The following is the prototype of the call back function:

```cpp
typedef void (CAgent::* NOTIFY_CALLBACK) (CString & strMsg, CString & strCtnName, CString & strAgtName, CString & strReplyTag);
```

**g) Step 7: Publish Message when Necessary**

If the agent wants to publish a message to a message group, it can simply call the Publish method.

```cpp
void Publish(CString strGrpName, CString strContents, CString strReplyTag);
```