ESPRIT Project N. 25 338

Work package I

Pilot Application 1

Requirements & Architectural Design

ID:	WP_I_Req&ArcDesign V. 1.2	Date:	15.05.1998
Author(s):	Hans-Guenter Stein, FAST e.V.	Status:	deliverable
	Lioba Gebauer, FAST e.V.		
	Abhaya Rajakarunanayake, FAST e.V.		
Reviewer(s):		Distribution:	Project internal & EC reviewers

Change History

Document Code	Change Description	Author	Date
WP_I_Req&ArcDesign	Version 0.5. No changes.	Stein, Rajakarun-anayake	25.02.98
WP_I_Req&ArcDesign	Version 0.8. Input from Gebauer	Stein, Gebauer, Rajakarun-anayake	06.03.98
WP_I_Req&ArcDesign	Version 1.0. Changes by Stein	Stein, Gebauer, Rajakarun-anayake	16.03.98
WP_I_Req&ArcDesign	Version 1.1 Changes by Stein	Stein, Gebauer, Rajakarun-anayake	07.04.98
WP_I_Req&ArcDesign	Version 1.2 Changes by Stein	Stein, Gebauer, Rajakarun-anayake	15.05.98

Overview

In workpackage I a set of services is established, based on a framework that provides a generic method for constructing domain specific, user customizable services for context related information retrieval on the internet. The approach taken is to logically separate information sources (raw data) from services (data processing applications). The aim is to give end users high level access to customized information by providing them with access to services rather than letting them deal with locating and filtering raw data from potentially highly distributed information sources on the internet.

The framework is validated by two pilot applications implementing services in the domains of stock portfolio management and regional event notification. The applications are implemented in the Bavaria Online Citizens network which provides a large base of potential users. System components will be initially deployed at FAST and five different Bavaria Online Citizens network nodes.

The requirements for workpackage I are specified by identifying use cases both for the generic scenario of locating and using services and for the domain specific functionality required by the two pilots. The architectural design maps these use cases on the agent-based system components provided by the technical workpackages D, E, F and H and provides sequence charts to illustrate the system internal control flow for each of the use cases. The architectural design serves as input for the design work of the workpackages providing the required technologies (namely WP D, E, F and H).

1 Agent-based Application Framework

1.1 Motivation

Efficient information management and retrieval is becoming the key success factor to almost any human activity – be it in business or private life. The well known problem of 'information overload' is omnipresent in the Internet. Thus the challenge we face is to structure the contents of the web and to develop easy to use tools that assist information consumers in locating the information sources that best fit their needs.

Thus the major objective of the FollowMe pilot applications in WP I is to exploit the mobile agent technology to develop a support infrastructure for information consumers. In this document section we outline an agent interaction framework that facilitates an information brokering and information retrieval infrastructure build on top of the architectural concepts developed in WP A, B, C, D, E, F, G and H. Users can customize agents that serve as interfaces to information services. These agents are potentially mobile so they may move whenever appropriate depending on the specific tasks they are designed to service. The systems supports location transparent agent and data addressing giving both users and agent developers maximum flexibility with respect to mobility. Users may interact with their personalized agents through a variety of different device types as outlined in WP H: User Access.

The pilot applications will be deployed at a number of internet nodes that are part of the Bavaria Online user organization. The Bavarian Online user organization was chosen as a partner since they provide a large user base and a distributed network of nodes where agents can run.

NOTE: Section 3 (Use Case Models) and section 4 (System Specification) provide a design related analysis of the system requirements for the intended pilot applications. Since the purpose of these document sections is to provide a guideline to detailed system design and implementation, annotations to use cases and sequence charts have been added wherever we saw the need for further investigations that are beyond the scope of this document but will be an issue for the system design and implementation phase.

1.2 Architectural Framework for WP I Applications

The architectural framework for WP I applications is designed to enhance information retrieval and information filtering in large scale, heterogeneous networks like the Internet. It is build on top of the architectural concepts developed by the other project partners as illustrated in

Figure1.

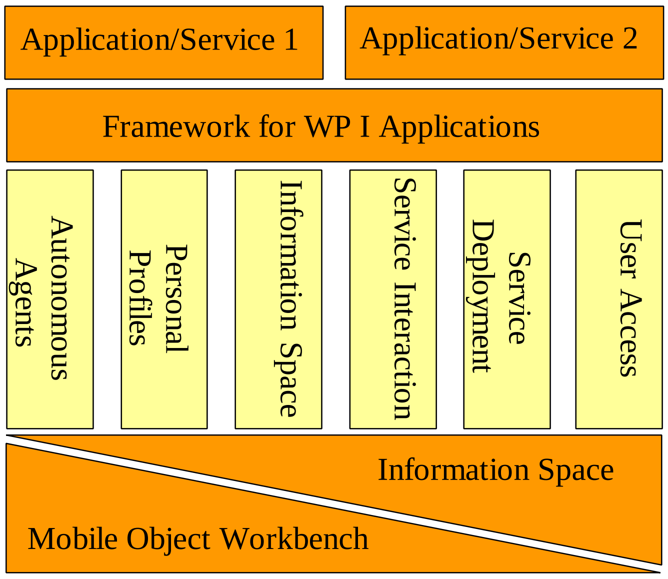

Figure1: Architecture Framework for WP I Applications

Upon designing this framework, we identified the following core issues related to any system of distributed data sources:

- The use of any database is not only defined by the sheer amount of collected data, but by the applications or services that operate on the contents of the database to provide information in form of customized results to the users of such systems.
- In order to enable the development of useful database applications, any database needs to provide a meta-model describing the structure of the offered data.
- To gain the most from largely distributed databases, the development of database applications should be de-coupled from the maintenance of the databases themselves. That way, the contents of data sources can be re-used and re-combined when developing new applications according to the needs of information consumers.

A first analysis of above issues leads to the most important design decision for the WP I framework: *to de-couple the roles of service providers and content providers*. *Service providers* implement applications that make use of raw data offered by *content providers*. They define meta-models describing the data structures their application are capable of dealing with. In order to enable the *service providers*' application to make use the data offered by a *content provider*, the data needs to be structured according to meta-models that form supersets of the meta- models of the *service providers*. This decision leads straightforward to the following core axioms:

- Users of our system will no longer (as is with the Web today) address *content providers* to obtain information in raw data format. Instead they will address services that provide them with already refined information according to their individual needs. The services therefore need to be customizable by the individual user.

- Services do not operate on a predefined or hardcoded set of data sources, but on specific data structures. They may use any data source available at runtime that offers relevant data as long as it offers an interface the service knows how to use.

These axioms impose the introduction of components *that glue things together*. On the one hand, users need to have an effective way of locating services that fit their needs. On the other hand, there need to exist mechanisms that enable services to locate relevant information sources at runtime. The appropriate concepts to fulfil these requirements are the ideas of brokers, matchmakers or – more simply – directory services.

Thus the core components of the FollowMe system have been identified as (see Figure 2):

- *content/information providers* (offering access to data-objects),
- *service providers* (providing services operating on data available from the *content providers*)
- *information consumers* (users of available services) and
- *directory services* (mediating between the other components).

The architectural pattern behind this component model is the pattern of information funnel, which is described in the FollowMe architecture (WP A).

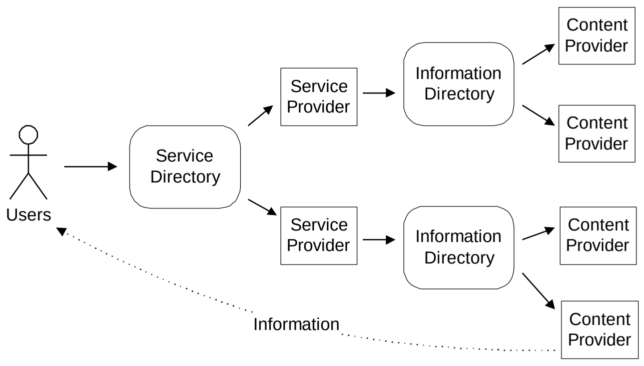

Figure 2: Core components of the FollowMe system

In WP I applications a service is composed of a component related to the information consumer (referred to as *task agent*) and a component implementing an interface to content providers (referred to as *service interaction interface*). A special user related agent (referred to as *personal assistant*) assists the user in organizing the usage of services and handling personalized information.

All agents acting on behalf of a user in WP I applications are instantiated on request by downloading the respective Java classes from so called *agent factories* to the user trusted environment (referred to as *FollowMe places*). The environment is located on a host with permanent online connection (i.e. a local ISP).

1.2.1 Scenario description

To illustrate the basic concepts of the WP I architectural framework for applications, we describe a fairly generic scenario. We assume, that the user in the described scenario already owns a *personal assistant* and is now on his way to select one of the services offered by FollowMe's *service providers*. As an example we introduce a service that delivers information on regional events (i.e. concerts, cinemas, markets, exhibitions).

Step 1:

A user connects to the system by contacting his personal assistant (PA). The user may then make changes to his personal diary, i.e. by stating, that he will be reachable by e-mail during working hours and by fax otherwise (see Figure 3).

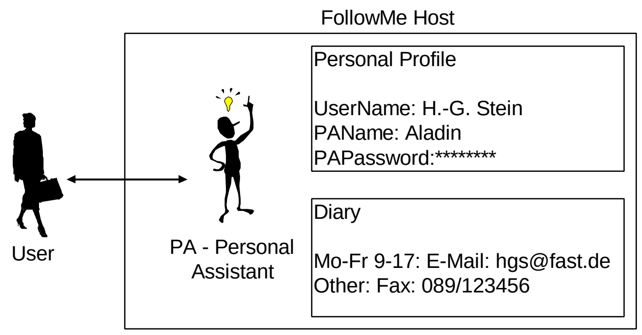

Figure 3: Contacting the personal assistant

Step 2:

The user wants to use one (or more) of the services offered by the system. The PA connects the user to a service directory that allows the user to select a specific service the user is interested in. Note that the PA does not require any knowledge about specific services. This de-couples the user specific components (such as the PA) from the rest of the system (available information sources and services). After selecting a service to subscribe to, the directory service links to the appropriate service provider (agent factory) and an instance of a task agent representing the respective service is created on the FollowMe host to service the individual request of its owner (see Figure 4).

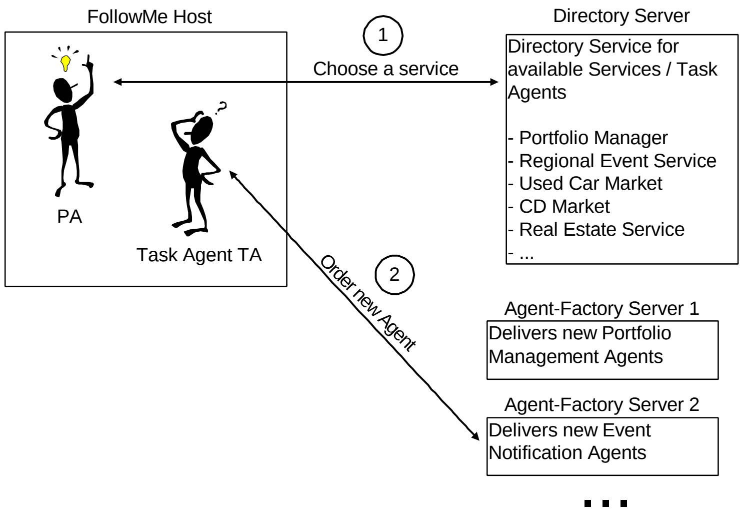

Figure 4: Instantiation of a new task agent

Step 3:

The user may now provide the new task agent with personalized parameters. In case of an event notification service, parameters might include specific event types of interest to the user and location and date of events. Moreover the user specifies a time schedule defining when he wants the results of the service to be delivered (see Figure 5).

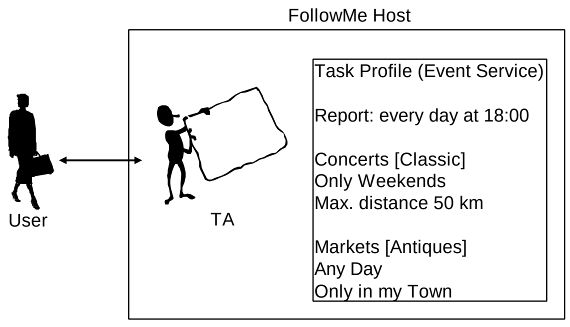

Figure 5: Defining a task profile

Step 4:

After specifying these input parameters the user may disconnect from the system. The specified service will execute automatically and in regular intervals according to the user defined schedule. The agent in charge of executing the respective task will contact another directory service that links it to service interaction interfaces at content provider sites relevant to the application domain of the task agent (in our example these are providers of regional event information). All the agent needs to know is the type of interfaces it is capable of connecting to. There is no need to hardcode the addresses of content providers within the agent. This de-couples service providers (agent providers) from content providers. That way, new content providers could join the system by registering at the directory service without the need for changes to existing services. The same holds for the integration of new services operating on data offered by existing content providers. In our example the agent connects to servers providing information on regional events and queries these servers according to the user specified parameters.

As described in the architectural framework, all objects and thus agents are potentially mobile. Service interaction interfaces are capable of providing an agent runtime environment (a FollowMe place – see WP B: Mobile Object Workbench). Whether an agents makes use of these mobility features depends on the type of service the agent is representing. In applications where communication between a task agent and a service interface is very intensive (i.e. sophisticated negotiation processes) the agent could be designed to move to the interface instead of remotely connecting to it (see Figure 6):

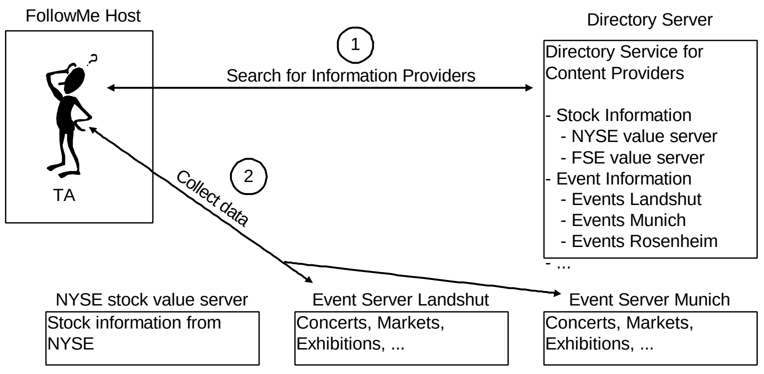

Figure 6: Task execution

Step 5:

After the task agent finished its task of information retrieval and refining, it stores the information in the user's information space for future reference either by the user or by the agent itself (see architecture framework). In addition the agents might be instructed to deliver reports on new information to their user. In our example the user instructed the agent to send a report every day at 18:00. Reporting might as well be triggered by specific changes to data values or other events (i.e. a stock portfolio management agent might be instructed to report to the user immediately when a stock exceeded a specified limit).

To send reports to its user, the task Agent uses the user access components, which provide gateways to a variety of devices like mail boxes, phones, pagers or fax machines (see architecture framework). The kind of device to be used for report delivery is stated in the diary section of the user's personal profile. The task agent consults the personal assistant to obtain this information. In our example the appropriate device for delivering reports at 18:00 is a fax gateway (see Figure 7).

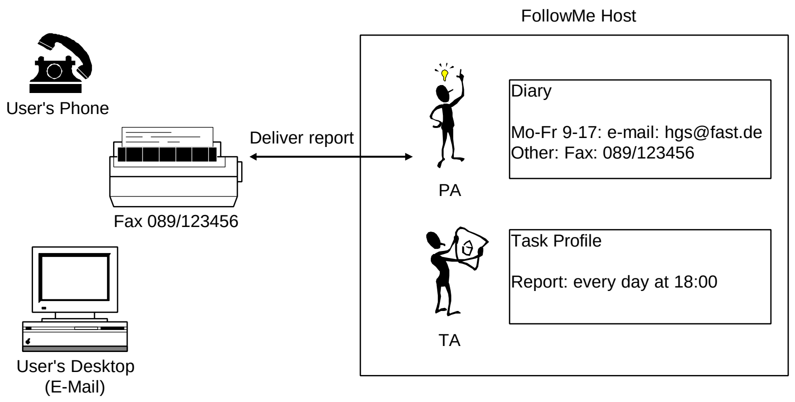

Figure 7: Delivering a report

1.2.2 Pilot Applications

To validate the above described framework, WP I implements two applications and deploys them at a number of Bavaria Online nodes. That way, our pilots will be evaluated by a large number of real users.

The first pilot application is an event notification system and offers basically the features outlined in the above scenario description: it provides Bavaria Online users with information on regional events. Content providers in this scenario are local institutions like cinemas, concert organizers, schools, etc.. Content in this pilot is widely distributed among a variety of different providers, that have close relations to the providers of the service (the Bavaria Online nodes). Thus this pilot serves as a demonstrator of the concepts of automated information retrieval and filtering among a widely distributed set of data sources.

The second pilot is a stock portfolio management application that provides the user with up to date information on share values retrieved from well known stock information providers. This application serves primarily as a demonstrator of the concepts of automated, event generated user notification. The user may define price limits for share values. Whenever such a limit is exceeded at any of the stock exchanges monitored by the user's agent, the user will be instantly notified by using the features offered by the User Access modules (see WP H: User Access).

2 Requirements

In this document section we provide a requirement specification of the framework for WP I applications and the two pilot applications by describing the functionality offered by the system to its external actors using Jacobson’s use case models. The use cases detail the scenario description provided in section 1.2.1.

Figure 8 shows the use case diagram of the whole system for the generic framework of locating and using services. It consists of seven use cases for the actor *user* (= *information consumer*), one use case for the actor *content provider* (= *information provider*) and one use case for the actor *service provider*.

Another actor not explicitly modeled in the use cases is the s*ystem administrator*. The user is offered to contact the administrator in various situations during interaction with the system. The system contacts the administrator when specific system events occur (triggers).These issues need to be detailed in the system design documents.

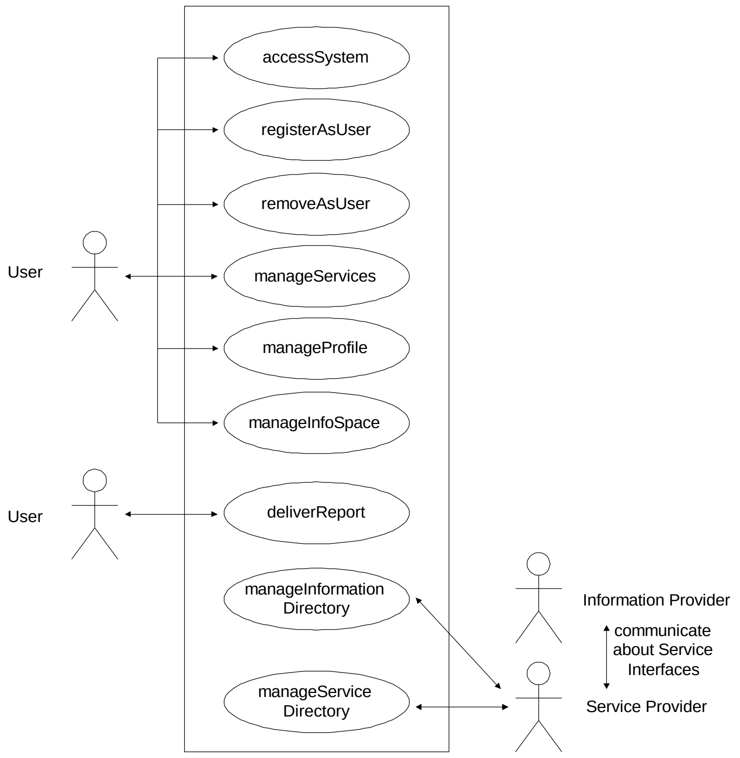

Figure 8: Use Case Diagram

2.1 Use Cases for the Actor User

In this section we identify the use cases for the actor user.

<u>*AccessSystem:*</u>

The use case *accessSystem* is always entered by the user at the beginning of a session and describes how the user connects to the system. The remaining five use cases related to the user's interaction with the system extend this use case as shown in figure 9.

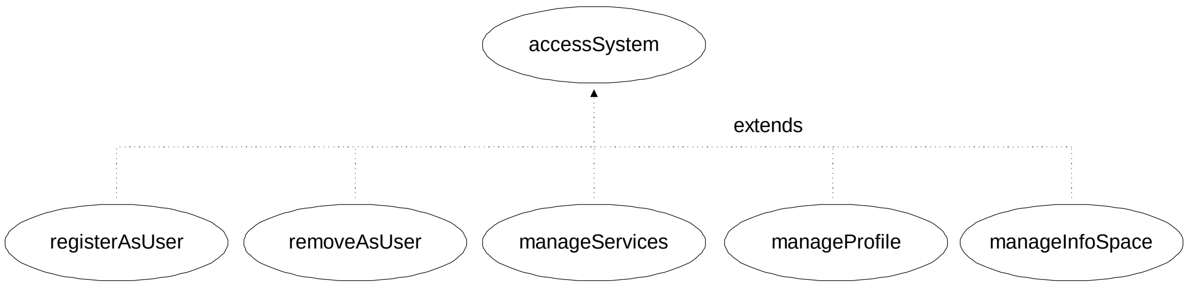

Figure 9: Extensions of the use case *accessSystem*

The use case *accessSystem* is started every time a user connects to the system either by accessing a web-site through a standard web browser via a modem (Netscape or Microsoft browser version 3) or by running a FollowMe specific software package.

He is then prompted to choose one of the following options (*access menu*):

- Authenticate and access personalized data. When successfully authenticated, the user may select one of the following options (*main menu*): *manageServices*, *manageProfile*, *manageInfoSpace*, *removeAsUser* and *exitSystem*. If one of the first four options is selected, the corresponding use case is started, otherwise the user exits from the system.
- Register to the system as a new user. This invokes the use case *registerAsUser*. After being registered successfully he may select one of the above mentioned options. Otherwise the access menu is shown again.
- Contact the system administrator (e.g. via e-mail).
- Exit the system.

<u>*RegisterAsUser:*</u>

The use case *registerAsUser* describes process of a new user registering to the system. During the registration process the user must provide user specific data. The use case *registerAsUser* may be started by yet unregistered users. When the user enters this use case, the system creates a new account from a default template and prompts the user for the following data: real name, password, email address and a system username.

After the user submitted his registration, the system checks if the submitted data is valid (username is a non-empty string and not in use by another user; format of e-mail address is valid; no password mismatch and minimum password length) and then saves the now personalized account information. After this the use case ends.

RemoveAsUser:

The use case *removeAsUser* describes the process of a user unsubscribing from using the system. This use case may be entered by a user any time after the use case *accessSystem* has been started and the user has successfully authenticated. After entering this use case, the user is prompted whether he is really sure that he wants to delete his system account.

When the remove command is submitted, the system removes all components associated with the user and deletes the users system account. After this, the user is notified, that his account was successfully removed, the use case ends and the access menu of the use case *accessSystem* is displayed.

ManageProfile:

The use case *manageProfile* describes how the user interacts with the system to maintain his personal data stored with the user's system account. This use case may be entered by a user any time after the use case *accessSystem* has been started, the user has successfully authenticated and the main menu of *accessSystem* is displayed.

Within this use case the user may choose from one of the following options (*profile menu*): *changePassword*, *manageDiary* and *exitManageProfile*.

To change the password the user needs to enter the old password and the new password and to submit these changes. After submission, the system checks if the new password is valid and saves these changes to the personal data. After successfully changing the password, the *profile menu* is displayed again.

When the user chooses *manageDiary*, he may change information specifying when he is reachable through which device, e.g. 'on weekends the device to contact the user is a fax with fax-number 111222'. The precise data structure and thus the kind of data to be provided by the user is to be specified by *WP-E: Personal Profiles*. After successfully changing these settings, the *profile menu* is displayed again.

The option *exitManageProfile* ends the use case and displays the *main menu* of the use case *accessSystem*.

ManageInfoSpace:

The system offers the user a facility to persistently store data with location transparent access features. This storage facility is used to store information delivered to the user by the system's information retrieval services. The use case *manageInfoSpace* describes how the user may directly access the contents of this *personal information space* (see WP A: 'Architecture' for general information on the concepts of information spaces). This use case may be entered by a user any time after the use case *accessSystem* has been started, the user has successfully authenticated and the *main menu* of *accessSystem* is displayed.

Within this use case the user may change the contents of his *information space* or *exitManageInfoSpace* (which ends this use case leading back to the *main menu* in the use case *accessSystem*). The kind of objects that may be changed depends on the characteristics of the specific applications/services the information relates to. A list of objects that can be changed or deleted is displayed. The user selects the objects to change/delete and submits his request. After an additional confirmation (including a cancel option that leads back to the object list) the system makes the requested changes and the list of objects is displayed again.

ManageServices:

The use case *manageServices* deals with general issues about accessing services in the WP I application framework. It is extended as shown in figure 10.

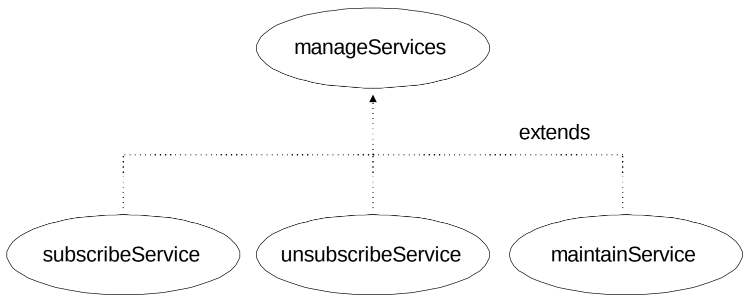

Figure 10: Extension of use case *manageServices*

The use case *manageServices* may be entered by a user any time after the use case *accessSystem* has been started, the user has successfully authenticated and the *main menu* of *accessSystem* is displayed.

Within this use case the system displays the list of services the user is currently subscribed to. The user may choose from the following options (*services menu*): *subscribeService*, *unsubscribeService*, *maintainService* and *exitManageServices*, each of them leading into a new use case as shown in figure 10.

SubscribeService:

One of the processes dealing with the management of services is to subscribe to a service. This process is described in the use case *subscribeService*. This use case may be entered by a user any time after the use case *manageServices* has been entered.

Within this use case the system displays a list of currently available services which the user did not already subscribe to and an *exitSubscribeServices* option that leads back to the services menu of the use case *manageServices*. One entry in the list consists of a service name and a textual description of the features of the service. Initially the list will consist of two entries: stock portfolio manager and regional event notification (the two intended pilots). The user may select one service at a time. The system then displays a more detailed service description and prompts for a confirmation of the subscription process (here cancel leads back to the list of services). After confirmation by the user, the system will register the user as subscribed to that specific service. Application/domain specific details on the subscription process will be discussed in further sections. After successful subscription, the new service is added to the list of services in use (see below), the user gets notified about the successful operation and is linked back to the list of available services (in which the just selected service is now no longer displayed since the user already subscribed to it). If subscription failed, the user gets informed about this and is linked back to the list of available services. The new service can be accessed from the services menu in use case *manageServices* by choosing *maintainService*.

UnsubscribeService:

The use case *unsubscribeService* describes the process of unsubscribing from a specific service. This use case may be entered by a user any time after the use case *manageServices* has been entered

Within this use case the system displays the list of services in use and an exit-*UnsubscribeServices* option that leads back to the *services menu* in the use case *manageServices*. The user may select one service at a time. The system then prompts for a confirmation to unsubscribe from the respective service (cancel leads back to the list of services in use). The user gets notified about the success of this process and is linked back to the list of services in use.

MaintainService:

The use case *maintainService* outlines the general options the user is offered to maintain one of the services he did subscribe to. However, most of the options are domain specific and will be described in more detail for both pilots separately. This use case may be entered from use case *manageServices* by selecting a specific service from a list of services already in use. This leads to another option menu (*maintainService menu*) offering the following: *setTaskParameters/useService*, *scheduleReporting*, *immediateReporting*, *maintainServiceInfoSpace*, and *exitMaintainService*.

The option *setTaskParameters/useService* is very domain specific. Details for both pilots will be described in later sections. Basically the user gets displayed a form that allows him to change parameters that customize the respective service to the user's individual needs. The parameters might define specific data queries or set triggers for actions to be executed under certain circumstances. An *exitSetTaskParameters* option leads back to the *maintainService menu*.

The option *scheduleReporting* enables the user to specify when the respective service should deliver reports to the user. The data structure and thus the kind of scheduling information to be provided by the user is to be decided by *WP-E: Profiles*. An *exitScheduleReporting* option leads back to the *maintainService menu*.

The option *immediateReporting* instructs the system to deliver an immediate, unscheduled report to the user (addressed to the device, the user is currently using). The system checks through which device the user is connected and sends a report to this device - containing the most actual data available. After requesting the immediate report, the user is linked back to the *maintainService menu*.

The option *maintainServiceInfoSpace* is again domain specific. Details for both pilots will be described in later sections. Basically the user may access domain specific data stored in the persistent storage facility (*information space*) and manipulate its contents (e.g. delete outdated data). An *exitMaintainServiceInfoSpace* option leads back to the *maintainService menu*.

DeliverReport:

The use case *deliverReport* describes how services deliver reports to the user. Report contain data composed of information the services have gathered and filtered on behalf of the user. Reports in general are sent to specific devices as specified in the user's *diary* (see use case *manageProfile*). There are two different kinds of reports: regular, scheduled reports and event triggered reports. The system behavior in this use case is domain specific and will be described for both pilots separately.

2.2 Domain specific Use Cases for the Actor User

Here we describe the domain specific aspects of above use cases.

2.2.1 Stock Portfolio Management System

The stock portfolio management system provides the user with an service aiding in managing a stock portfolio. The service persistently stores information on the user's shares and cash account as well as a history of all transactions between stock depot and cash account. Share values are constantly updated by querying stock information providers. The user may define upper and lower limits for the values of his shares. Whenever such a limit is exceeded, the system will generate an immediate report and send it to the user.

In the following section we describe the domain specific aspects of the use case *maintainService* within the use case *manageServices*. The option *setTaskParameters/useService* in the use case *maintainService* is extended as shown in figure 11.

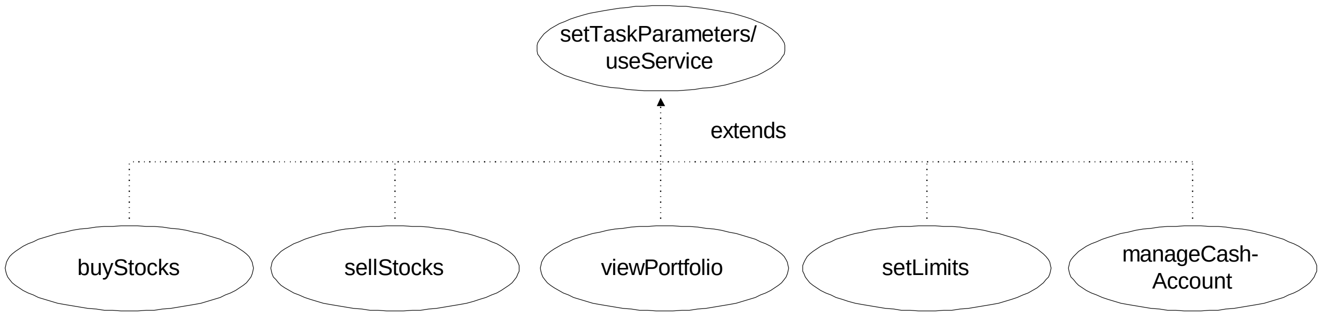

Figure 11: Extension of *setTaskParameters/useService* in the stock domain

<u>BuyStocks:</u>

The use case *buyStocks* outlines the process of buying new shares. Within this use case the user is prompted to enter a company name. The system then computes a list of ticker symbols that might match the user specified company name. This list is presented to the user. Each entry in the list consists of a full name for the found stock entry, a ticker symbol, the name of the service provider the information was retrieved from and the stock exchange at which the stock identified by the ticker symbol is traded. The user selects one or more entries from the list to specify from which information source he wants his agent to retrieve stock quotes. The user must select one of the sources as *mastersource* (see use case *viewPortfolio* for details). Next, the user is prompted to enter more details on the buying transaction: price per share, date of purchase, amount of shares bought. Then the use case *setLimits* is entered. In addition, the user may enter costs for transaction fees. The system then computes the overall costs of the purchase and subtracts the money from the cash account. If there is not enough money in the cash account, the transaction is canceled and the user gets informed about the failure including an option to update the cash account by adding some money (see *manageCashAccount* for details). Note that buying shares from the same stock at different days or at different prices results in keeping two different entries in the portfolio. After completion of the transaction the user is prompted to either specify another transaction by entering a new company name or to go back to *setTaskParameters/useService* options.

<u>SellStocks:</u>

The use case *sellStocks* outlines the process of selling shares. Within this use case the user is displayed the lists of positions currently held in his portfolio. Each position is displayed only once (not multiple times due to multiple data sources for the same position). The user selects the position to be sold and is prompted for details about the transaction: date of selling, price at which the shares where sold, amount of shares sold and transaction fees. The system then computes the overall costs of the transaction and adds the result to the cash account. If an entire position was sold, the entire position is deleted from the portfolio. If only a fraction was sold, the position is kept in the portfolio, but the amount of shares bought/held is reduced to the remaining fraction of the position. After completion of the transaction the user is prompted to either specify another transaction by selecting a new position to be sold or to go back to *setTaskParameters/useService-Functions*.

The use case *viewPortfolio* outlines the process of a user viewing the contents of his portfolio. This use case is almost identical to what the user gets delivered, when he scheduled reporting according to the general description of the processes *scheduleReporting* or *immediateReporting*. When entering *viewPortfolio* the user gets displayed the actual content of his portfolio, an option to set limits to specific portfolio entries (see *setLimits*) and an *exitViewPortfolio* option, which leads back to the *maintainService menu*. The user may choose from different views: view all entries including multiple listings for one position according to multiple data sources; view only one entry per portfolio position using the most up-to-date value; view only one entry per portfolio position using the *mastersource* as defined during the purchase phase. The portfolio will be displayed in a table. Each entry will consist of the following information: company name, ticker symbol, source of value (*information provider*), source of value (stock exchange), number of shares, purchase price and date, actual value with date and timestamp, up and down limits, gain/loss per position (absolute and percentage). In addition the overall values of the portfolio are displayed (value of all stocks at purchase time, actual value of all stocks, overall gain/loss (absolute and percentage). The initial amount of money in the cash account (see *cashstart* in the process *manageCashAccount*), the actual overall value of stocks and cash account and the gain/loss (absolute and percentage) computed from the latter are displayed.

SetLimits:

The use case *setLimits* outlines the process of defining limits for share values. This use case is always automatically invoked, when the user purchases a new stock (see above) or when a previously set limit was exceeded (see use case *deliverReport* below), and can be manually invoked by selecting a specific entry while viewing the portfolio. The user may then set high and low limits to a selected position. When finished, the user is linked back to the previous menu depending on the context in which *setLimits* was invoked (new purchase or viewing the portfolio).

ManageCashAccount:

The use case *manageCashAccount* outlines how a user may maintain his cash account. This use case offers the user the following options: *viewHistory*, *add/withdrawMoney* and *exitManageCashAccount* (leading up one level in the options hierarchy). The option : *viewHistory* lets the user scroll through the listing of the *cash account history file*. An entry in the history describes a single transaction and consists of the values of *cashstart* (the amount of money the user did put into the cash account), *cashnow* (that is *cashstart* minus the money invested by purchasing stocks, paying transaction fees and fees for maintaining the cash account plus revenues from selling stocks), a transaction type (which might be: add/withdraw money, cash account maintenance costs, fee for stock purchase, fee for stock selling, purchase of stocks or selling of stocks), a description for the transaction (i.e. 100 IBM sold at 52.80) and the overall value of the transaction. The user may define different views on the history (i.e. show only transactions of a specific type). Quitting the *viewHistory* process leads back to *manageCashAccount*.

The *add/withdrawMoney* option allows the user to add or withdraw money from the cash account (which changes the value of *cashstart*) or pay fees for maintenance (which changes the value of *cashnow*).

DeliverReport:

The use case *deliverReport* for the domain Stock Portfolio Management describes how the system delivers reports to the user. Reports in general are sent to specific devices as specified in the user's *diary*. There are two different kinds of reports: regular, scheduled reports and event triggered reports.

Scheduled reports are sent to the user at regular intervals according to the parameters the user specified for the stock portfolio service (see use case *manageServices* – option *maintainService* – option *scheduleReporting*). These reports provide the user with a standard overview of his portfolio. This standard view is described in the *viewPortfolio* option in the use case *manageServices*.

Event triggered reports in the stock domain are created whenever a stock limit (see *setLimits* in use case *manageServices*) is exceeded. This kind of report consists of a short notification about the respective event.

Since some devices do not provide direct interaction with the user, there is no way of determining whether a report was received by the user. Thus the system will keep copies of all reports sent to the user in the user's persistent storage facility (*information space*) until the user manually deletes them (see *maitainServiceInfoSpace* in use case *manageServices*).

2.2.2 Regional Event Notification System

The Regional Event Notification System (RENS) will provide users of the Bavaria Online Network with information about upcoming events. A large variety of events will be supported by the RENS: e.g. festivals, movies shown in cinema, or language courses. A detailed definition of events is given below. Content providers will put information about these events into their data bases. The user can specify a query to get notified about events. The query is matched against the events in the data bases of several information providers. A query may specify the type of the event, the time period and a region in which the event must take place. Queries may be executed once or may be repeated on a certain schedule. The service looks for all content providers that provide events for the specified region, queries them, collects the results and forwards them to the user, either offline (e.g. by fax) or on-line in the session of the user.

Definitions

An *Event* is an organized meeting, limited in time and content. It means in the original sense festivities, festivals, balls, concerts, markets, exhibitions, readings, performances, assemblies, sittings and meetings, but also in the broader sense teaching courses of e.g. adult colleges or the red cross, programs of cinemas and theatres, holidays- and recreational programs, special exhibitions of museums and art galleries, days of open doors, advertisement of shop openings, special dinners and public elections.

An Event is described by a hierarchy of types, a title, an organizer, and a description (what?), its time (when?), and its location (where?).

A *query* (in the context of this service) is a characterisation of the type of the event, the time interval, the region and the reporting schedule.

In the following section we describe the domain specific aspects of the option *maintainService* within the use case *manageServices*. The use case *maintainService* is separated into a set of sub-use cases (see figure 12). Please note that this structure deviates slightly from the generic case, because the scheduled reporting is handled slightly different.

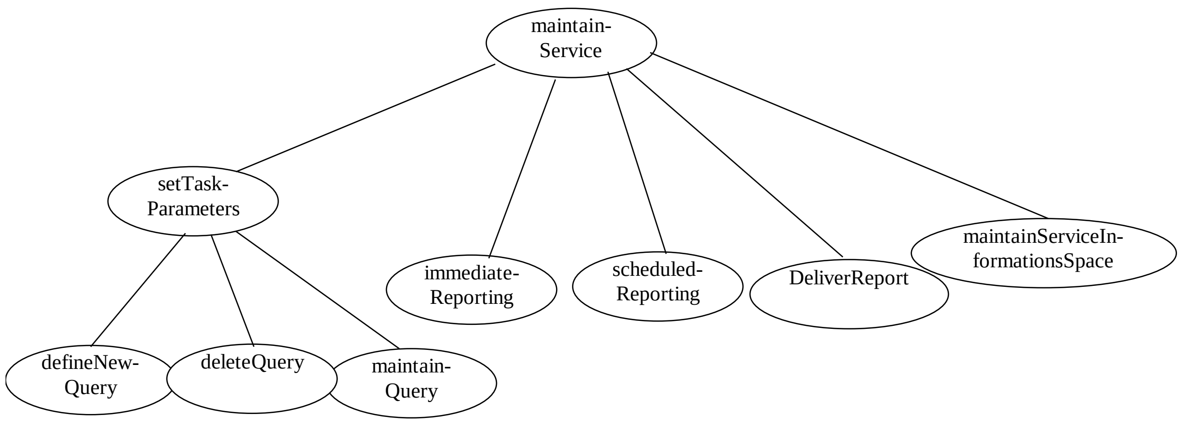

Figure 12: Extensions of the use case *maintainService* for the event notification domain

MaintainService:

Within the use case *maintainService* (consists of *setTaskParameters, immediateReporting, scheduledReporting*, *deliverReport,* and *maintainServiceInformationspace*) for the domain of RENS, the system presents a list of already defined queries to the user. Each query has assigned a name that can be freely chosen. The service allows the user to choose either to define a new query (*defineNewQuery*), to exit or to choose from the following options for each of the defined queries*: manageQuery*, *maintainServiceInformationSpace,* or to select several queries and to delete them (*deleteQuery*).

DefineNewQuery:

When selecting *defineNewQuery*, the user is prompted for a description of a query. The user gives the name of the query, and its details (what, when, where) and optionally the reporting schedule. The data is checked for consistency and the query is added to the list of defined queries. After finalizing the definition the user will return to *maintainService*.

MaintainQuery:

The use case *maintainQuery* coincides mainly with the use case *defineNewQuery*. But all the fields are already preset according to the selected query. The user may choose to change the details of the query, including its name. If the user changes the name of the query, it will be stored as a copy and handled according to *defineNewQuery*. If the name is unchanged the query will be updated by the new data.

ImmediateReporting:

When the user triggers the use case *immediateReporting*, the system will immediately start to look for events and will create a report to be deposited in the user's persistent storage facility.

ScheduledReporting:

If the use case *scheduledReporting* is performed for a specific query, the system schedules the execution of the query. Each time the query is executed a report is generated and sent to the end user.

DeleteQuery:

During the use case *deleteQuery*, all marked queries will be deleted from the list of queries after an confirmation.

<u>*DeliverReport:*</u>

The use case *deliverReport* is entered every time a scheduled query was executed and a report was generated. Within this use case, the respective report is delivered to the user. The way of delivery depends on the entries in the user's *diary*.

<u>*MaintainServiceInfomationSpace:*</u>

Within the use case *maintainServiceInfomationSpace* the user can select a number of available reports related to a selected query. The events contained in the reports are presented. The user can request different sortings of this list (chronological (which is the default), by type, by municipality). The user can also request the deletion of a set of reports.

2.3 Use Cases for the Actors Information and Service Providers

In this section we describe the roles of the actors *service providers* and *information providers*. Within the project, we see two different levels of issues dealing with these actors:

1. Within the project we implement two example services: a *Stock Portfolio Management* and an *Event Notification System*. The process of implementing and advertising these services is part of the system development and not part of the system as an application. From this point of view, the development and deployment of new services is not a feature of the system as an application. Moreover, the participants in these pilot applications are predefined and fixed. Any changes to system functionality (i.e. service upgrades, changes to interface specifications, deployment of interfaces at new *information provider* sites) can be viewed as new system releases. Change management in that sense is not a feature of the system itself. The system functionality required for participation of *service providers* and *information providers* is then reduced to the following features:
   - *Information providers*:
     - *IPs* need an easy to use interface to maintain their database contents. This is domain specific and needs to be addressed for each pilot separately.
   - *Service providers*:
     - Both pilot services will be implemented and maintained by FAST. Nevertheless the system should provide facilities to automatically update all instances of services whenever we decide to expand data structures or service functionality.
2. Besides just implementing these two example services, the aim of WP-I (and the project in general) must be to provide a system architecture that is truly scalable. With respect to *service providers* and *information providers* scalability includes providing certain change management features, that allow third parties to develop their own services and allow *information providers* to enhance their interfaces to provide extended data structures. The system functionality required for these features is as follows:
   - *Information providers* (*IPs*):

    - To enable new *IPs* to join the system, the system needs to provide a facility to advertise a new *IP* site in the *information provider directory*. This includes:
        - *Information provider* name
        - Interface definition
        - A reference to the location of the *IP* site
    - *IPs* need a system facility to interact with *SPs* about requested changes to *service interface definitions*.
    - *IPs* need a system facility to maintain the data contents they provide.
- *Service providers* (*SPs*):
    - To enable third parties to develop and maintain services that conform to our system architecture, the system needs to provide a *service development kit*.
    - *SPs* need a system facility to advertise their services in a *service provider directory* including:
        - Service name and human readable service description
        - A reference to the location of the service
    - *SPs* need a system facility to define service interfaces and make them available to potential *information providers* or assist them in deploying the interfaces.
    - *SPs* need a system facility to interact with *IPs* about requested changes to *service interface definitions*.
    - *SPs* need a system facility to automatically update the (distributed) components of their services.

Details on requirements for the second and more general view on the system are outside the scope of the current project phase (which is to implement a first version of a working system and deploy it at Bavaria Online nodes). Nevertheless the above stated requirements should be kept in mind by developers of *autonomous agents* and *service interaction*. Integrating these change management features into the system should be approached after the more static view described above has been successfully implemented.

3 Architectural Design: Use Case Analysis

In this document section we start with the architectural design by mapping the use case models developed in the previous section onto the agent-based system components developed in the underlying technical workpackages of FollowMe. The architectural design serves as input for the design work of the workpackages providing the technologies required to built the pilot applications (namely WP D, E, F and H). The component interaction models described in the use cases and the event flow models described in the sequence charts in the next document section are subject to changes due to the design decisions that will be made in the technical workpackages. Detailed design of the pilot applications will start after all technical workpackages delivered their design documents and interface specifications.

3.1 System Component Model

Figure 13 provides a basic view on the system components and their relationships. As outlined in the previous document sections, workpackage I provides user customizable services for information retrieval. The user locates these services via his *personal assistant* (*PA*), which holds information specific to the individual user in a *personal profile* (e.g. user name, e-mail address, fax and phone number and the user's diary). The *PA* locates available services via a *service provider directory service*. The services themselves are represented by *task agents* (*TAs*). The user provides these agents with parameters specifying when to deliver which kind of information. These parameters are stored in *task agent profiles*. Each *TA* is specialized in retrieving domain specific, well structured information from *information provider* sites. The interfaces between the legacy database systems of the information providers and the *TAs* are represented by *service interfaces*. Every time a *TA* executes a query for information based on a user defined schedule, it locates available *information provider service interfaces* by contacting an *information provider directory service*. Profile data as well as information retrieved by the *TAs* is persistently stored in the user's *information space* (see workpackage C). The user interacts with his *PA* and *TAs* via *device gateways* provided by the *user access* (see workpackage H).

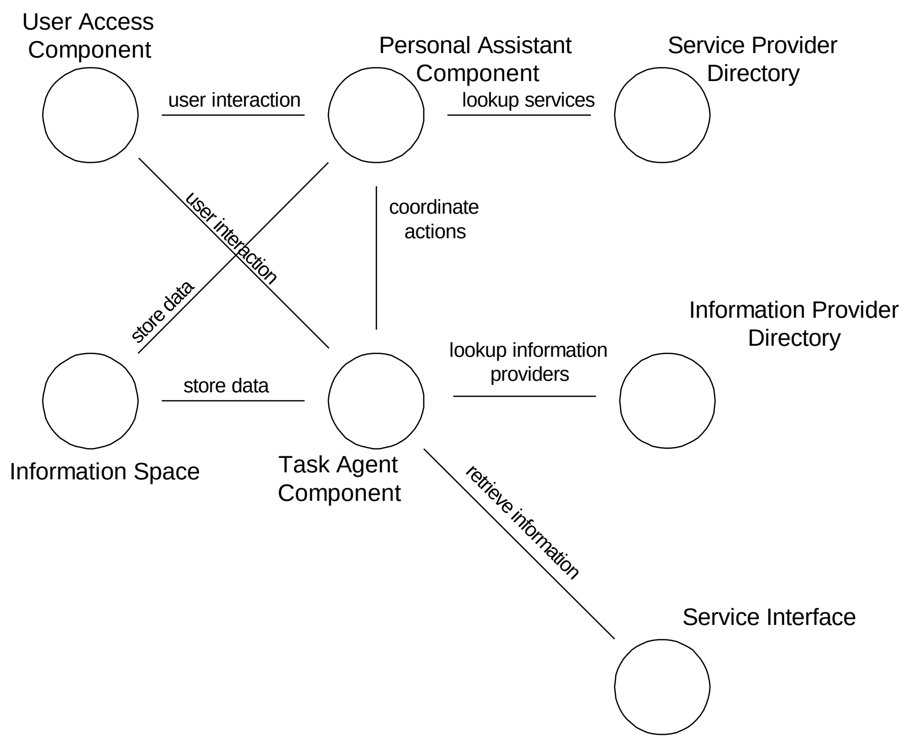

Figure 13: System Components

User Access Component, Personal Assistant Component and Task Agent Component are decomposed in according to Figure 14, Figure 15 and Figure 16.

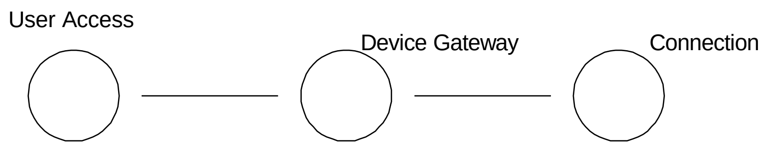

Figure 14: User Access Components

A more detailed design of the *User Access* module and how it connects to the *PA* and *TA* components can be reviewed in WP H: Design (deliverable DH3).

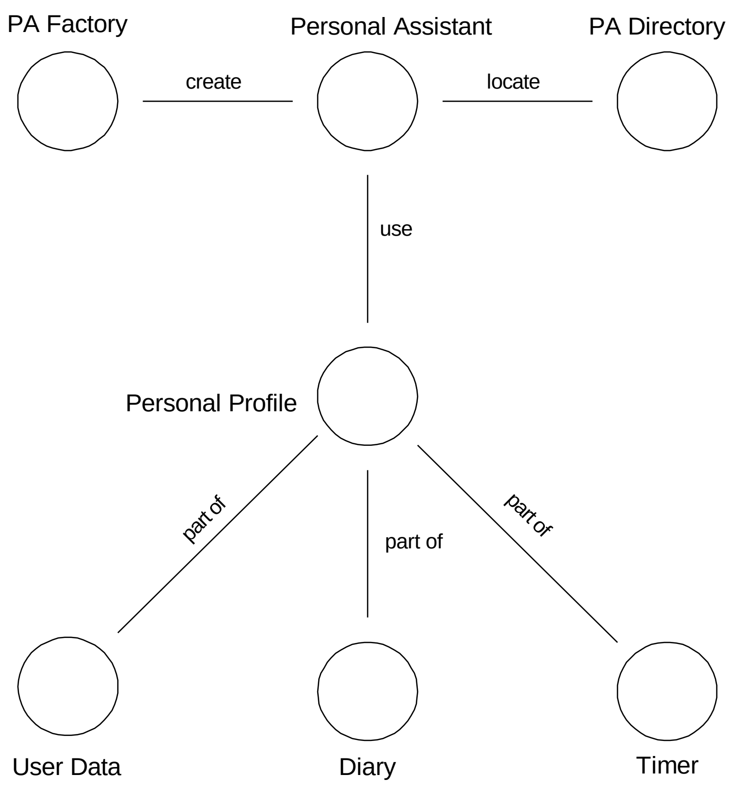

Figure 15: Personal Assistant Components

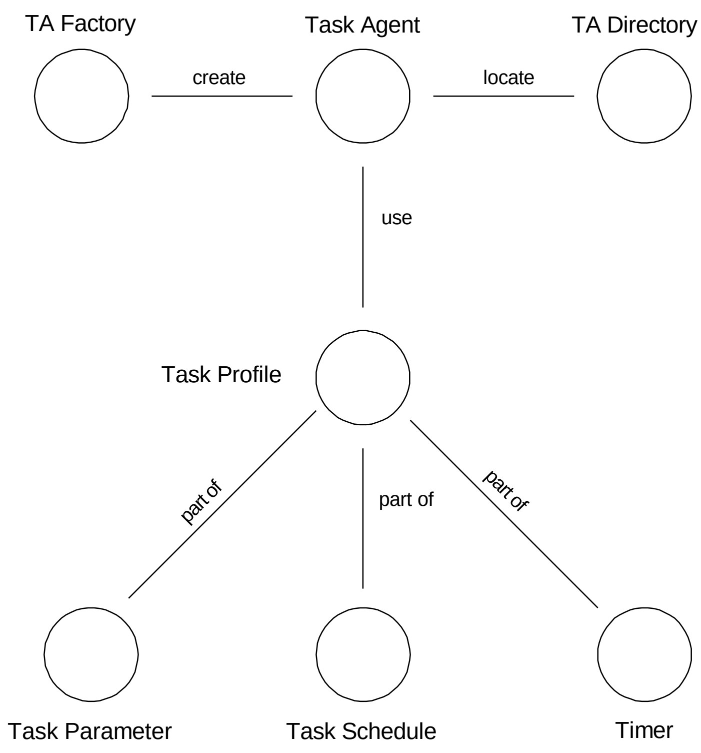

Figure 16: Task Agent Components

The pilot applications provided by workpackage I are distributed systems. Hosting, maintenance and administration of components is distributed among various parties as follows:

1. *Service providers*:
   - advertise their services in a global *service provider directory* (by providing a service description and a link to a *TA factory*). This directory will be implemented using the directory and name trading capabilities of the *Mobile Object Workbench* (WP B. As more and more services are added, the directory could be hierarchically decomposed.
   - host TA factories, that allow creation of TAs (representing a software capable of using their services).
   - host an Information Provider Directory (a listing of links to Service Interfaces to Information Providers to be used by TAs)
   - provide Service Interfaces to Information Providers (might be located either at Service Provider sites or at Information Provider sites)
   - offer Information Providers to deploy FollowMe Service Interfaces to enable agents to use their legacy systems
2. *Information providers:*
   - host information sources (in form of legacy systems)
   - may host *service interfaces* deployed by *service providers* (see above)
3. Parties hosting agents on behalf of users:
   - host places (the runtime environment for agents), *PA*s, *TA*s, agent profiles and *information spaces* on systems that provide permanent online connection and can be trusted by the user with respect to privacy

- host User Access components

4. FollowMe system administrators:
   - host *PA factories*
   - host and maintain *PA* and *TA* directories

3.2 Use Cases for the Actor User

In the following we map the use cases identified in section 2 onto above described system components.

Use Case *accessSystem*

The use case *accessSystem* is started every time a user enters the system by either loading the place software locally or connecting to a place remotely.

He is then prompted to choose one of the following options (*access menu*):

- Enter the name of his *personal assistant* (*PA*). In this case, the systems links the user to his *PA*. If linking to the *PA* is successful, the *PA* prompts the user for his password (echoed on the screen as asterisks) which is checked in turn by the *PA*. If the password is correct, the user may select one of the following options (*PA main menu*): *manageServices*, *manageProfile*, *manageInfoSpace*, *removeAsUser* and *exitSystem*. If one of the first four options is selected, the corresponding use case is started, otherwise the user exits from the system.
- Register to the system as a new user. This invokes the use case *registerAsUser*. After being registered successfully he may select one of the above mentioned options. Otherwise the access menu is shown again.
- Contact the system administrator via e-mail
- Exit the system.

Annotations:

- Whenever the user is online connected to the system, the current online-connection (all parameters required to contact the user – i.e. IP-address,...) need to be stored in the *User Profile* for reference by any other component, that may receive a command to contact the user.

The use case *registerAsUser* outlines how a new user registers to the system by triggering the instantiation of a new *personal assistant*. Is described as follows:

Use Case *registerAsUser*

The use case *registerAsUser* may be started by yet unregistered users. When the user enters this use case, the system creates a new *personal assistant* (*PA*). The *PA* prompts the user for the following data: name, password, email address and a name for the *PA*. The password must be entered twice to ensure its correctness. It is not echoed on the screen. Instead, an asterisk is echoed for every character entered.

After entering all data the user may submit his registration or cancel it. The option cancel ends the use case and displays the access menu of use case *accessSystem*.

If the user submits his registration, the system checks if the submitted data is valid (username is a non-empty string; format of e-mail address is valid; *PA* name is not already in use; no password mismatch and minimum password length) and then creates an initial *information space* and an initial *user profile* for the new user. After this the use case ends.

Annotations:

- A new *personal assistant*, *information space* and *personal profile* is by default created at the FollowMe place from which the request originated.

The use case *removeAsUser* outlines how a user may unsubscribe from using FollowMe services. It is described as follows:

Use Case *removeAsUser*

The use case *removeAsUser* may be entered by a user any time after the use case *accessSystem* has been started and the user has successfully authenticated. After entering this use case, the user is prompted whether he is really sure that he wants to delete his system account.

If the user cancels the operation, the use case is ended and the *PA main menu* (see use case *accessSystem*) is displayed again.

If the remove command is submitted, the system removes all components associated with the user (includes deleting *information space*, *personal assistant*, all *profiles* and *task agents*). After this, the user is notified, that his account was successfully removed, the use case ends and the access menu of the use case *accessSystem* is displayed.

The use case *manageProfile* outlines how the user interacts with the system to maintain his *personal profile*. It is described as follows:

Use Case *manageProfile*

The use case *manageProfile* may be entered by a user any time after the use case *accessSystem* has been started, the user has successfully authenticated and the main menu of *accessSystem* is displayed.

Within this use case the user may choose from one of the following options (*profile menu*): *changePassword*, *manageDiary* and *exitManageProfile*.

To change the password the user needs to enter the old password and the new password (twice) and to submit these changes. After submission, the system checks if the new password is valid and saves these changes to the profile. After successfully changing the password, the *profile menu* is displayed again.

When the user chooses *manageDiary*, he may change information specifying when he is reachable through which device. The data structure and thus the kind of data to be provided by the user is to be decided by *WP-E: Profiles*. After successfully changing the diary settings, the *profile menu* is displayed again.

The option *exitManageProfile* ends the use case and displays the *PA main menu* of the use case *accessSystem*.

Annotations:

- Diary data consists of device type, device address (dependent on device type) and a time interval.
- The time interval may be of various types: daily hours, specific days within a week, weekends,... WP-E specifies how these types can be combined.
- The way the user interacts with the system when managing the diary will be designed in close cooperation between WP-E and WP-I. Profile contents will be displayed using XSL stylesheets. User interaction will be implemented using XML/XSL or HTML forms, JavaScript or Java applets.

The *information space* serves as a storage facility for persistent data with location transparent access features. The use case *manageInfoSpace* outlines how the user may directly access the contents of his *information space*. It is described as follows:

Use Case *manageInfoSpace*

The use case *manageInfoSpace* may be entered by a user any time after the use case *accessSystem* has been started, the user has successfully authenticated and the *PA main menu* of *accessSystem* is displayed.

Within this use case the user may change the contents of his *information space* or *exitManageInfoSpace* (which ends this use case leading back to the *PA main menu* in the use case *accessSystem*). The kind of objects that may be changed depends on characteristics of specific applications/services. A list of objects that can be changed or deleted is displayed. The user selects the objects to change/delete and submits his request. After an additional confirmation (including a cancel option that leads back to the object list) the system makes the requested changes and the list of objects is displayed again.

Annotations:

- As far as the intended pilot application (stock portfolio and event notification) are concerned, we do not see any need to allow the user to directly access the *information space*. However this might be required in the context of future applications.

The use case *manageServices* deals with general issues about accessing services and is described as follows:

Use Case *manageServices*

The use case *manageServices* may be entered by a user any time after the use case *accessSystem* has been started, the user has successfully authenticated and the *main menu* of *accessSystem* is displayed.

Within this use case the system displays the list of services the user is currently subscribed to. The user may choose from the following options (*services menu*): *subscribeService, unsubscribeService, maintainService* and *exitManageServices*.

The use case *subscribeService* outlines how a user subscribes to a new service by instantiating the appropriate *task agent*. This use case is described as follows:

Use Case *subscribeService*

The use case *subscribeService* may be entered by a user any time after the use case *manageServices* has been entered

Within this use case the system displays a list of currently available services which the user did not already subscribe to and an *exitSubscribeServices* option that leads back to the services menu of the use case *manageServices*. One entry in the list consists of a service name and a textual description of the features of the service. Initially the list will consist of two entries: stock portfolio manager and regional event notification (the two intended pilots). The user may select one service at a time. The system then displays a more detailed service description and prompts for a confirmation of the subscription process (here cancel leads back to the list of services). After confirmation by the user, the system will create a new instance of a *task agent* (*TA*) representing the new service. The *TA* creation is accompanied by some initialization process that includes the creation of a default *TA profile* and some changes to the user's *information space*. Details on this are application/domain specific and will be discussed in further sections. After successful creation/initialization of the new *TA*, the new service is added to the list of services in use (see below), the user gets notified about the successful operation and is linked back to the list of available services (in which the just selected service is now no longer displayed since the user already subscribed to it). If *TA* creation/initialization fails, the user gets informed about it and is linked back to the list of available services. The new *TA* can be accessed from the services menu in use case *manageServices* by choosing *maintainService*.

The use case *unsubscribeService* outlines how a user unsubscribes from a specific service. It is described as follows:

Use Case *unsubscribeService*

The use case *unsubscribeService* may be entered by a user any time after the use case *manageServices* has been entered

Within this use case the system displays the list of services in use and an *exitUnsubscribeServices* option that leads back to the *services menu* in the use case *manageServices*. The user may select one service at a time. The system then prompts for a confirmation to unsubscribe from the respective service (cancel leads back to the list of services in use). After confirmation the system removes/kills the respective *TA*, the *TA profile* and deletes all related entries in the *personal profile* and all related data in the *information space*. The user gets notified about the success of this process and is linked back to the list of services in use.

The use case *maintainService* outlines the general options the user is offered to maintain one of the services he did subscribe to. However, most of the options are domain specific and will be described in more detail for both pilots separately. The use case is described as follows:

Use Case *maintainService*

The use case *maintainService* may be entered from use case *manageServices* by selecting a specific service from a list of services already in use. leads to another option menu (*maintainService menu*) offering the following: *setTaskParameters/useService*, *scheduleReporting*, *immediateReporting*, *maintainServiceInfoSpace*, and *exitMaintainService*.

The option *setTaskParameters/useService* is very domain specific. Details for both pilots will be described in later sections. Basically the user gets displayed a form that allows him to change parameters that define the mission of the respective *TA*. The parameters are stored in the *TA profile*. The parameters might define specific data queries or set triggers for actions to be executed under certain circumstances. In addition, the service might offer additional functions not directly dealing with agent instructions. An *exitSetTaskParameters* option leads back to the *maintainService menu*.

The option *scheduleReporting* enables the user to specify when the respective service should deliver reports to the user. This information is stored in the *TA profile*. The data structure and thus the kind of scheduling data to be provided by the user is to be decided by *WP-E: Profiles*. An *exitScheduleReporting* option leads back to the *maintainService menu*.

The option *immediateReporting* instructs the system (the respective *TA*) to deliver an immediate, unscheduled report to the user (addressed to the device, the user is currently using). The system checks through which device the user is connected and sends a report to this device - containing the most actual data available. After requesting the immediate report, the user is linked back to the *maintainService menu*.

The option *maintainServiceInfoSpace* is again domain specific. Details for both pilots will be described in later sections. Basically the user may access domain specific parts of his *information space* and manipulate its contents (i.e. delete outdated data and reports). The amount of data stored in the *IS* might be limited. When such a limit reached, the respective *service/task agent* might prompt the user to delete outdated data prior to any other changes to the *IS* (see use case *deliverReport* for the stock domain). An *exitMaintainServiceInfoSpace* option leads back to the *maintainService menu*.

Annotations:

- The *list of services in use* needs to be stored in the users *personal profile*. Immediate report delivery is asynchronous in the sense that the system won't wait for the user getting the report, reading the report and quitting this action. After requesting the delivery by choosing the *immediateReport* option, the user may continue interaction with the system as usual. The system delivers the report independently of other system activities. The report will be displayed on the users end device in a separate window.

The use case *deliverReport* outlines how reports are delivered to the user. It is described as follows:

Use Case *deliverReport*

This use case describes how the system delivers reports to the user. Reports in general are sent to specific devices as specified in the *PA Profile*. There are two different kinds of reports: regular, scheduled reports and event triggered reports. The system behavior in this use case is domain specific and will be described for both pilots separately.

3.3 Domain specific Use Cases for the Actor User

The domain specific aspects of above use cases describe the domain related system functionality rather then the general way in which system components interact. For that reason the following use case descriptions are only slightly different from the descriptions provided in document section 2.

3.3.1 Stock Portfolio Management System

The use case *buyStocks* outlines the process of buying new shares and is described as follows:

Use Case *buyStocks* (domain *Stocks*)

Within the use case *buyStocks* the user is prompted to enter a company name. The system then computes a list of ticker symbols that might match the user specified company name. This list is presented to the user. Each entry in the list consists of a full name for the found stock entry, a ticker symbol, the name of the service provider the information was retrieved from and the stock exchange at which the stock identified by the ticker symbol is traded. The user selects one or more entries from the list to specify from which information source he wants his agent to retrieve stock quotes. The user must select one of the sources as *mastersource* (see use case *viewPortfolio* for details). Next, the user is prompted to enter more details on the buying transaction: price per share, date of purchase, amount of shares bought. Then the use case *setLimits* is entered. In addition, the user may enter costs for transaction fees. The system then computes the overall costs of the purchase and subtracts the money from the cash account. If there is not enough money in the cash account, the transaction is canceled and the user gets informed about the failure including an option to update the cash account by adding some money (see *manageCashAccount* for details). After all transaction parameters are validated, the system updates the *TA profile* and informs the users about the successful transaction. Note that buying shares from the same stock at different days or at different prices results in keeping two different entries in the portfolio. After completion of the transaction the user is prompted to either specify another transaction by entering a new company name or to go back to *setTaskParameters/useService* options.

The use case *sellStocks* outlines the process of selling shares and is described as follows:

Use Case *sellStocks* (domain *Stocks*)

Within the use case *sellStocks* the user is displayed the lists of positions currently held in his portfolio. Each position is displayed only once (not multiple times due to multiple data sources for the same position). The user selects the position to be sold and is prompted for details about the transaction: date of selling, price at which the shares where sold, amount of shares sold and transaction fees. The system then computes the overall costs of the transaction and adds the result to the cash account. After that, the system updates the contents of the portfolio by changing the *TA profile*. If an entire position was sold, the entire position is deleted from the portfolio. If only a fraction was sold, the position is kept in the portfolio, but the amount of shares bought/held is reduced to the remaining fraction of the position. After completion of the transaction the user is prompted to either specify another transaction by selecting a new position to be sold or to go back to *setTaskParameters/useService-Functions*.

The use case *viewPortfolio* outlines the process of a user viewing the contents of his portfolio. It is described as follows:

Use Case *viewPortfolio* (domain *Stocks*)

The use case *viewPortfolio* is almost identical to what the user gets delivered, when he scheduled reporting according to the general description of the processes *scheduleReporting* or *immediateReporting*. When entering *viewPortfolio* the user gets displayed the actual content of his portfolio, an option to set limits to specific portfolio entries (see *setLimits*) and an *exitViewPortfolio* option, which leads back to the *maintainService menu*. The user may choose from different views: view all entries including multiple listings for one position according to multiple data sources; view only one entry per portfolio position using the most up-to-date value; view only one entry per portfolio position using the *mastersource* as defined during the purchase phase. The portfolio will be displayed in a table. Each entry will consist of the following information: company name, ticker symbol, source of value (*information provider*), source of value (stock exchange), number of shares, purchase price and date, actual value with date and timestamp, up and down limits, gain/loss per position (absolute and percentage). In addition the overall values of the portfolio are displayed (value of all stocks at purchase time, actual value of all stocks, overall gain/loss (absolute and percentage). The initial amount of money in the cash account (see *cashstart* in the process *manageCashAccount*), the actual overall value of stocks and cash account and the gain/loss (absolute and percentage) computed from the latter are displayed.

The use case *setLimits* outlines the process of defining limits for share values and is described as follows:

Use Case *setLimits* (domain *Stocks*)

The use case *setLimits* is always automatically invoked, when the user purchases a new stock (see above) or when a previously set limit was exceeded (see use case *deliverReport* below), and can be manually invoked by selecting a specific entry while viewing the portfolio. The user may then set high and low limits to a selected position. After a request for confirmation to the new limits, the system writes these changes to the TA

Profile for further reference (see use case *deliverReport*. The user is then linked back to the previous menu depending on the context in which *setLimits* was invoked (new purchase or viewing the portfolio).

The use case *manageCashAccount* outlines how a user may maintain his cash account. It is described as follows:

Use Case *manageCashAccount* (domain Stocks)

The use case *manageCashAccount* offers the user the following options: *viewHistory*, *add/withdrawMoney* and *exitManageCashAccount* (leading up one level in the options hierarchy). The option : *viewHistory* lets the user scroll through the listing of the *cash account history file*. An entry in the history describes a single transaction and consists of the values of *cashstart* (the amount of money the user did put into the cash account), *cashnow* (that is *cashstart* minus the money invested by purchasing stocks, paying transaction fees and fees for maintaining the cash account plus revenues from selling stocks), a transaction type (which might be: add/withdraw money, cash account maintenance costs, fee for stock purchase, fee for stock selling, purchase of stocks or selling of stocks), a description for the transaction (i.e. 100 IBM sold at 52.80) and the overall value of the transaction. The user may define different views on the history (i.e. show only transactions of a specific type). Details on this will be described in later document versions. Quitting the *viewHistory* process leads back to *manageCashAccount*.

The *add/withdrawMoney* option allows the user to add or withdraw money from the cash account (which changes the value of *cashstart*) or pay fees for maintenance (which changes the value of *cashnow*).

The use case *deliverReport* for the domain Stock Portfolio Management is described as follows:

Use Case *deliverReport* (domain Stocks)

This use case describes how the system delivers reports to the user. Reports in general are sent to specific devices as specified in the *PA Profile*. There are two different kinds of reports: regular, scheduled reports and event triggered reports.

Scheduled reports are sent to the user at regular intervals according to what the user specified in the *TA Profile* (see use case *manageServices* – option *maintainService* – option *scheduleReporting*). These reports provide the user with a standard overview of his portfolio. This standard view is described in the *viewPortfolio* option in the use case *manageServices*.

Event triggered reports in the stock domain are created whenever a stock limit (see *setLimits* in use case *manageServices*) is exceeded. This kind of report consists of a short notification about the respective event. After such a report was created, the limit rule is flagged to has_already_fired. The rule will then fire no longer until the limit has been changed by the user. No more report will be sent. Next time the user logs on to the system and enters the *PA main menu* (see use case *accessSystem*), the *setLimit* process (see use case *manageServices* for the stock domain) will automatically be invoked and the user is asked to change the respective limit. After the limit was changed, the user is linked back to the *PA main menu*.

Since some devices do not provide direct interaction with the user, there is no way of determining whether a report was received by the user. Thus the system will keep copies of all reports sent to the user in the user's *information space* until the user manually deletes them (see *maitainServiceInfoSpace* in use case *manageServices*).

The amount of reports to be stored in the *IS* is limited. When the limit is reached the user is sent a message that prompts him to clean up the *IS* by deleting outdated reports. The message is sent a limited number of times along or instead of the usual scheduled reports. The next time the user logs on to the system for direct interaction, he is directly prompted to delete old reports. This happens after he authenticated to his *PA* (see use case *accessSystem*). A list of previously sent reports is displayed and the user may select which one to delete. After the user deleted enough old reports, he is linked back to the *PA main menu*.

Annotations:

- Since all transactions are kept in a history file, a number of additional statistics could be offered to the user (i.e. gain/loss in a specific period of time,...)
- Selling of fractions of a portfolio position results in:
  - changing the purchase information of the position in the portfolio so that it appears as if only the remaining fraction has been purchased (i.e. before the transaction: bought 100 Siemens at 01/02/97; after selling 30 of these, the portfolio looks like: bought 70 Siemens at 01/02/97)
  - keeping all value retrieval information
- Selling of a complete position results in:
  - deleting the entire position from the portfolio
  - removing all value retrieval information and instructions
- Experts in the domain of stock management told us that integrating features that provide values for investment funds and option-calls as well as industry related news will increase system acceptance by potential users. Integration of these kind of information is planed for later software releases.

3.3.2 Regional Event Notification System

The use case *maintainService* in the domain of regional event notification is defined as follows:

Use Case *maintainService* (consists of *setTaskParameters, immediateReporting, scheduledReporting, deliverReport,* and *maintainServiceInformationspace*)

In this use case *maintainService*, the user gets the actual list of already defined queries from the TA profile. Each query has assigned a name that can be freely chosen. This service allows the user to choose either to define a new query (*defineNewQuery*), to exit, for each of the defined queries to choose from the following options*: manageQuery, maintainServiceInformationSpace,* or to select several queries and to delete them (*deleteQuery*).

The use case *defineNewQuery* is defined as follows:

> Use Case *defineNewQuery* (domain events)
>
> When selecting *defineNewQuery*, the user is prompted for a description of a query.
>
> The user gives the name of the query, and its details (what, when, where) and optionally the reporting schedule.
>
> The data is checked for consistency and the query is added to the list of defined queries.
>
> After finalizing the definition the user will return to *maintainService*.

Annotations:

- A note on navigational design: After the definition a NewQuery, the user can select the options *immediateReporting* or *scheduledReporting*. In both cases the schedule is saved as defined by the user, although the schedule might either be empty, or there is a schedule, but immediate reporting is selected.

The use case *maintainQuery* is defined as follows:

> Use Case *maintainQuery* (domain events)
>
> *maintainQuery* coincides mainly with *DefineNewQuery*. But all the fields are already preset according to the selected query. The user may choose to change the details of the query, including its name.
>
> If the user changes the name of the query, it will be stored as a copy and handled according to *DefineNewQuery*. If the name is unchanged the query will be updated by the new data.

The use case *immediateReporting* is defined as follows:

> Use Case *immediateReporting* (domain events)
>
> If the user selects immediate reporting for a query, the task agent will immediately start to look for events and will create a report to be deposited in the information space.

The use case *scheduledReporting* is defined as follows:

> Use Case *scheduledReporting* (domain events)
>
> If this use case is performed for a specific query, the TA schedules the execution of the query. Each time the query is executed a report is generated and sent to the end user.

The use case *deleteQuery* is defined as follows:

Use Case *deleteQuery* (domain events)

When choosing *deleteQuery*, all marked queries will be deleted from the list of queries after an confirmation.

The use case *deliverReport* is defined as follows:

Use Case *deliverReport* (domain events)

Each time a report is generated, due to the execution of a scheduled query, the report is delivered to the user. The way of delivery depends on the entries in the personal profile.

The use case *maintainServiceInfomationSpace* is defined as follows:

Use Case *maintainServiceInfomationSpace* (domain events)

The user can select from the available reports from a selected query. The events contained in the reports are be presented. The user can request different sortings of this list (chronological (which is the default), by type, by municipality).

The user can also request the deletion of a set of reports.

3.4 Use Cases for the Actors Information and Service Providers

As outlined in document section 2.3, the implementation of management features that allow automated integration of new services and information sources is outside the scope of the current project phase. Therefore we do not provide an architectural design for these features.

4 Architectural Design: Event Flow Models

4.1 Domain-independent event flow

In this section we describe the system internal event flows according to the use cases identified in section 2. Threads or grouping of events into processes and related object locking need to be addressed in the system design phase.

Figure 17 describes the event flow for the process *connectToPA* of the use case *accessSystem*.

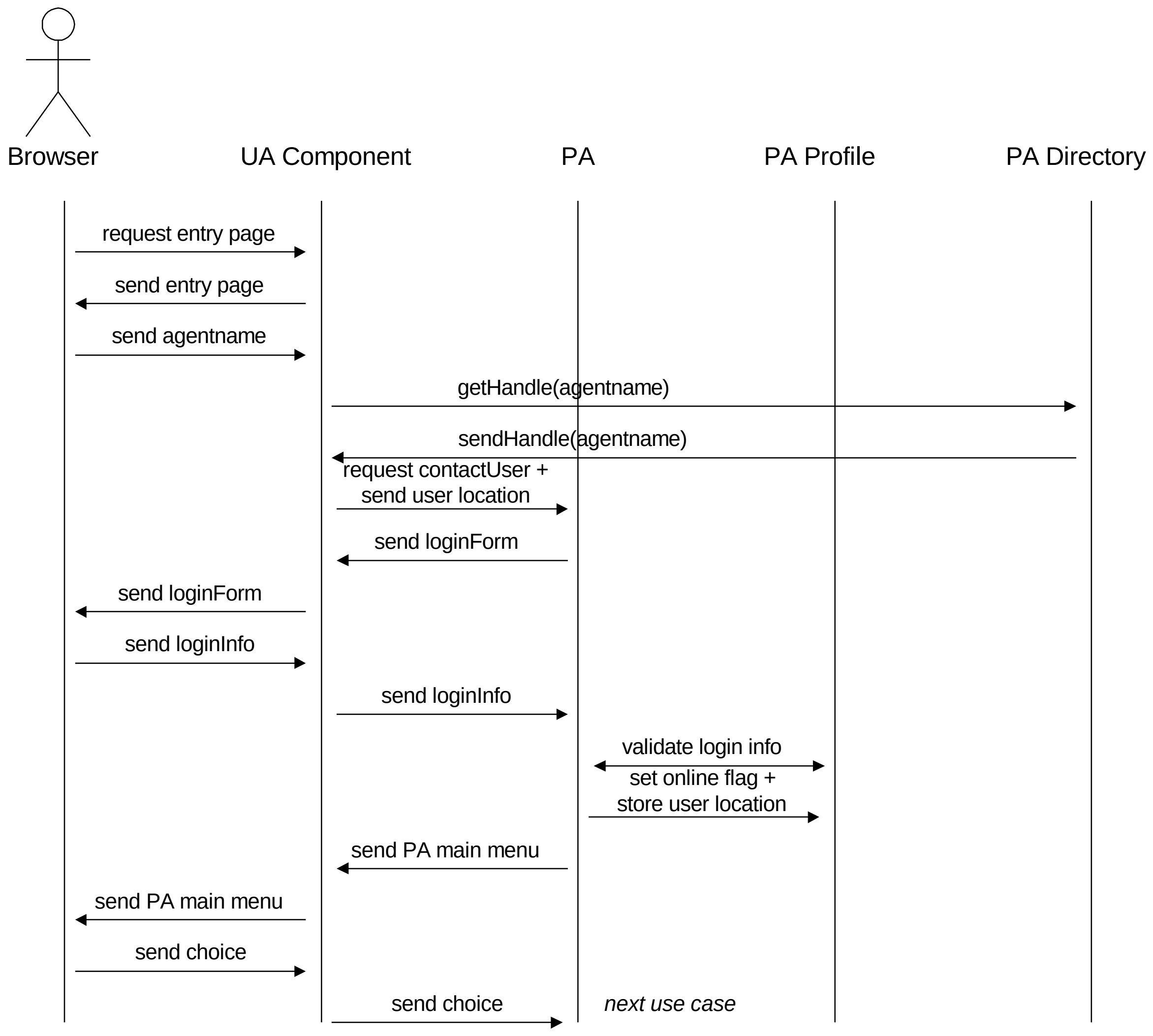

Figure 17: Process *connectToPA* of use case *accessSystem*

Annotations:

- Details on how the user obtains a handle to the *PA*:
  - The user connects to a *place/user access* (*UA*) by loading a HTML form or applet.
  - The user inputs the name of his *PA* and sends it to the *UA*. NOTE: the name is NOT a reference to the *PA* object!
  - The *UA* needs to contact the *PA directory* to obtain a reference/object address to the *PA*.
  - With this reference, the *UA* sends a message to the *PA*, that invokes a method *contactUser* with parameters, that specify the context (*authenticationDialog*) (NOTE: context information is only required, if communication is fully asynchronous!!) and information on where to contact the user (location[addressOfUA]; devicetype[browser]). The procedures of agents contacting a user via the *UA* module is described in greater detail in WP-H: Design (deliverable DH3).
- The *PA* might not in suspended mode stored in the *information space*. This should be transparent to the above process and handled by the respective *place* or *information space*.
- There might occur several exceptions during the process described above:
  - The *UA* might be unable to contact the *PA directory*.

  - The *PA directory* might not be able to locate the *PA*.
  - The *PA* might not respond to the request to contact the user.
  - The *PA* might be unable to send messages to the *UA*. This might occur any time during the process.
  - The *UA* might be unable to forward messages to the user. This might occur any time during the process.
  - The authentication might fail due to the fact, that communication between *PA* and *personal profile* fails.
- Note that some of the user's *TA*s might have reports or tasks stored, that could not be completed offline because they require user interaction (see stock domain for an example: when a stock limit was exceeded, the user must change (or may just have the option to do so) the limit during the next interactive FollowMe session). This requires that prior to displaying the *PA* main menu, the *PA* must check for tasks stored or scheduled by some *TA*. Eventually the *TA*s are invoked automatically to prompt the user for the required interaction (or just notify the user about these pending tasks). This might be implemented by adding a tag in the *PA profile* that points to a specific method of the respective *TA*. Another possibility is to instruct the *PA* to ask all *TA*s (registered in the *PA profile* in the *list of services in use*) for pending tasks.
- After the user authenticated to his *PA*, the *PA* stores the current user location in the *PA profile*. The flag *user_is_online* is set. Any request by some agent to retrieve the current location of the user according to what is stated in the diary is automatically overruled by this tag. The *PA* then returns the address of the *UA* through which the user is connected online instead of what is stated in the diary.

In all following sequence charts we identify the *User* with the corresponding *User Access Component*. Transactions like "request info – get info" will be described as a single method call. All of the below described processes might fail to complete due to failures in one of the system components or message passing problems. Exception handling will be addressed in more detail in the system design phase.

Figure 18 describes the Use Case *registerAsUser*.

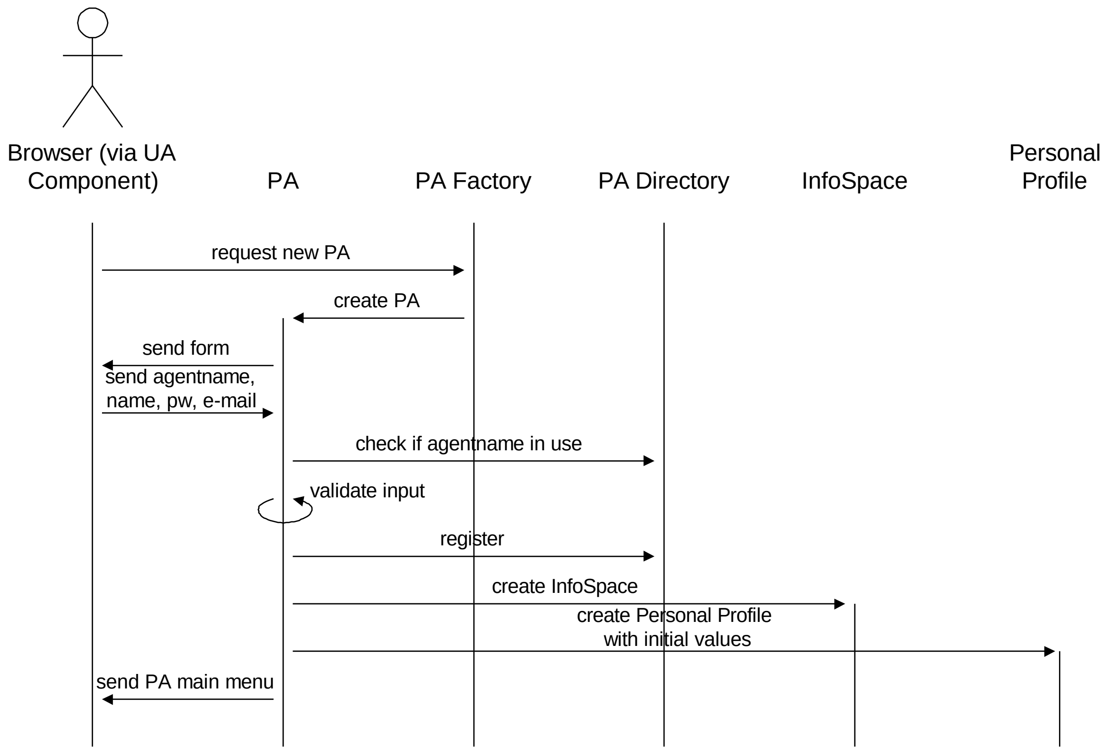

Figure 18: Use Case registerAsUser

Annotations:

- Note that the new *PA* needs to know how to contact the user. The *UA* must communicate its address to the *PA factory* which in turn needs to hand over this information to the new *PA*.
- The new *PA, information space* and *profile* will be initially instantiated at the place from where the request to create the *PA* came from (the place where the respective *UA* resides).
- There are a number of possible exceptions in this use case:
  - The *factory* might not be reachable
  - The instantiation of the new *PA* might fail
  - The *PA* might be created but the remaining operations of the initialization process might fail. In that case there must be a mechanism to ensure that the nameless, unregistered agent is killed
  - The agent name the user chooses might be already in use by some other agent. In that case the *PA* must prompt the user to choose another name.
  - One of the other parameters (user-name, user e-mail, password) might not be valid. In that case, the user must be prompted to change his input.

Figure 19 describes the event flow for the use case *removeAsUser*.

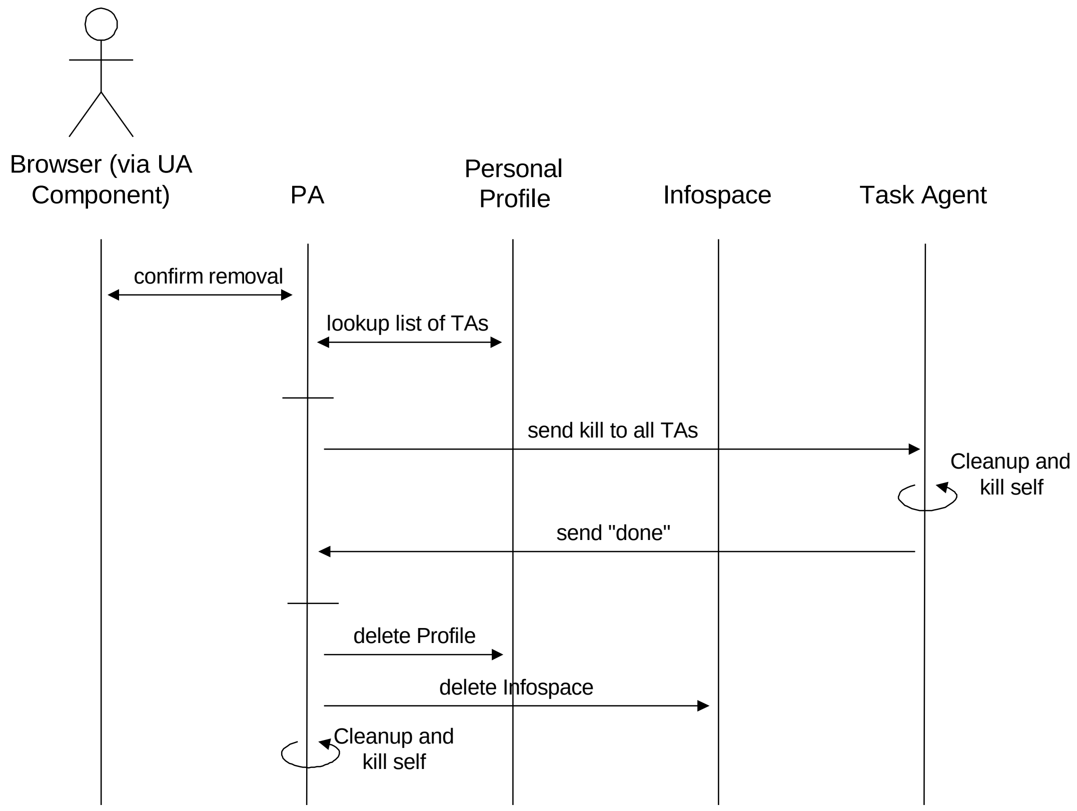

Figure 19: Use Case removeAsUser

Annotations:

- Above process might be entered after the *PA* did send the *PA main menu* to the user and the user selected the *removeAsUser* option.

The event flow for the use case *manageProfile* is described in the design document of WP-E (deliverable DE3).

Some issues about handling the diary are still to be discussed:

- One of the major use cases for interaction with the profile is to specify how to reach the user at which points in time. There need to be mechanisms that allow the user to specify these points in time in terms of rules like:
  - RULE_1: Monday to Friday, 9:00-17:00 -> device=e-mail(hgs@fast.de); else -> device=fax(++4992004718@followme.fast.de)
  - RULE_2: Sunday, 01.03.98 -> device=phone(++4992004755@followme.fast.de)

  An agent having to deliver a report on Sunday, 01.03.98 will request the appropriate device from the profile. The profile needs to apply above rules in the correct order and with correct priorities (the device in above example should be the phone).

  The diary object must ensure, that at any given point in time the parameter of where to reach the user is always well defined.
- The user might define that a specific agent should deliver a specific report at specific points in time (use case *scheduleReporting*). To specify which device should be used for delivering reports should be optional:
  - TA 007: deliver report A daily at 18:00

- TA 008: deliver report B every Friday at 15:00 to device fax(++4992004718@followme.fast.de)

Figure 20 describes the event flow for the use case *subscribeService*.

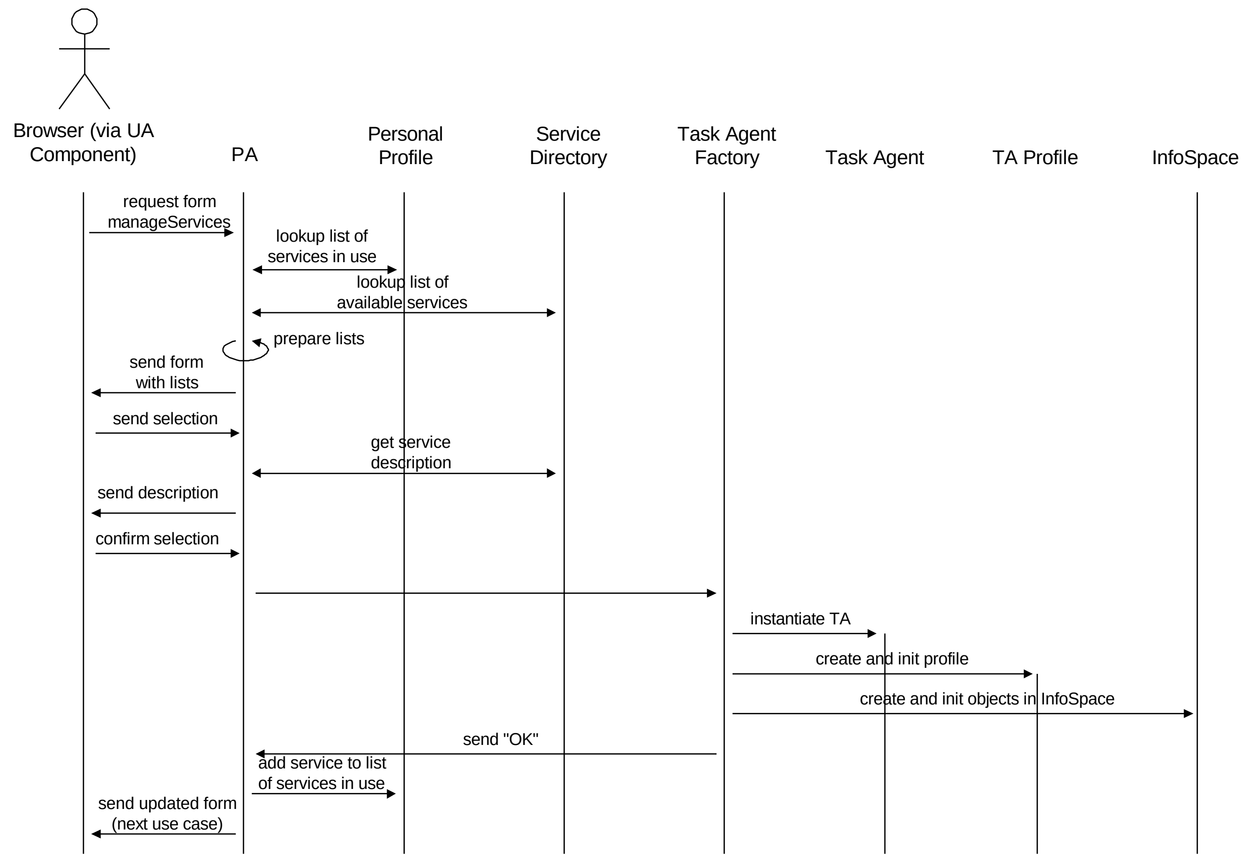

Figure 20: Use case subscribeService

Annotations:

- The "*form with lists*" send to the user consists of two lists: the list of services in use, which is retrieved from the *personal profile* and the list of available services not yet in use, which is the list of all available services retrieved from the service directory minus the list of services in use.
- The process of instantiating a *TA*, initializing a *TA profile* and creating and storing service related objects in the *information space* is described in the design document of WP D (deliverable DD3) in cooperation with WP C.

Figure 21 describes the event flow for the use case *unsubscribeService*. The use case can be entered by the user by highlighting one item in the list of the services in use (which is send to the user during use case *manageServices* as described above).

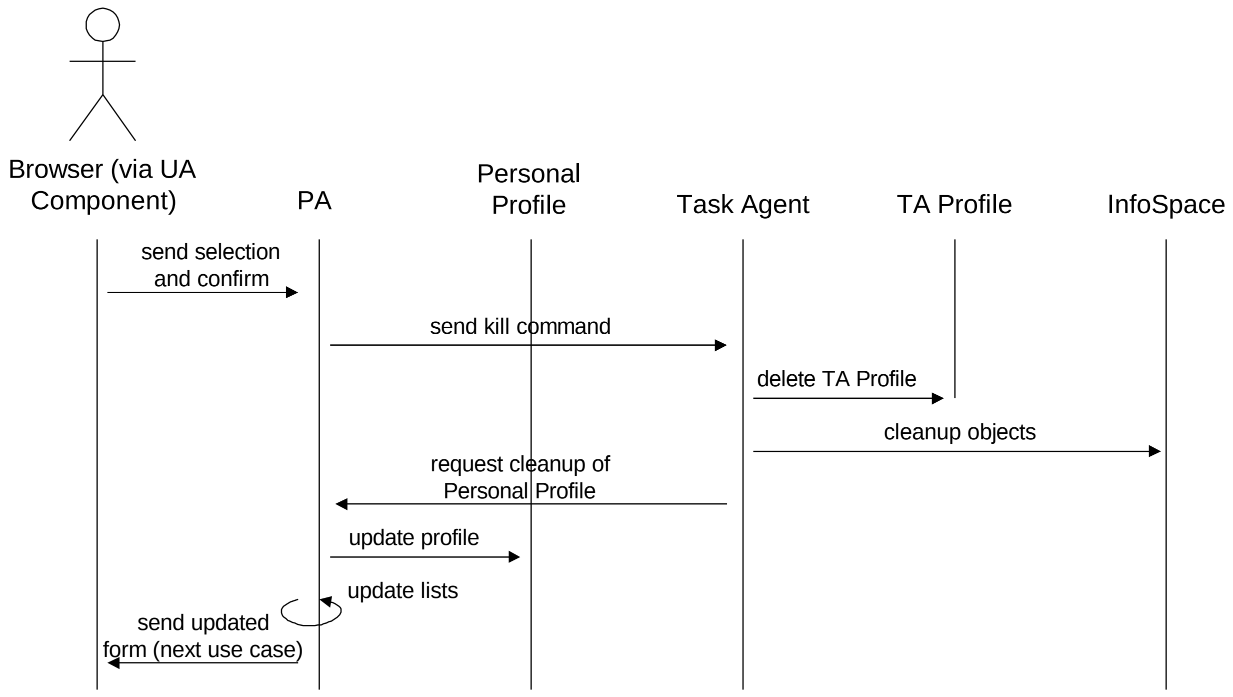

Figure 21: Use case unsubscribeService

Annotations:

- The process of killing a *task agent* is described in the design document of WP D (deliverable DD3).
- WP D and WP E define whether *TA*s have direct access to the *personal profile* or all changes to the *personal profile* are mediated through the *PA*. Same holds for changes to the *information space* (addresses WP C).
- Cleaning up the *personal profile* includes deleting all task related events scheduled in the diary and deleting the respective service from the list of services in use.
- Killing a *TA* includes de-registering at some *trader* / *TA directory*. Issues on registering/de-registering at traders are addressed in the design document of WP B.

Figure 22 describes the event flow for the option *scheduleReporting* in the use case *maintainService*. The use case *maintainService* can be triggered by the user by highlighting one item from the list of services in use (see Figure 20) and choosing the option *maintainService*.

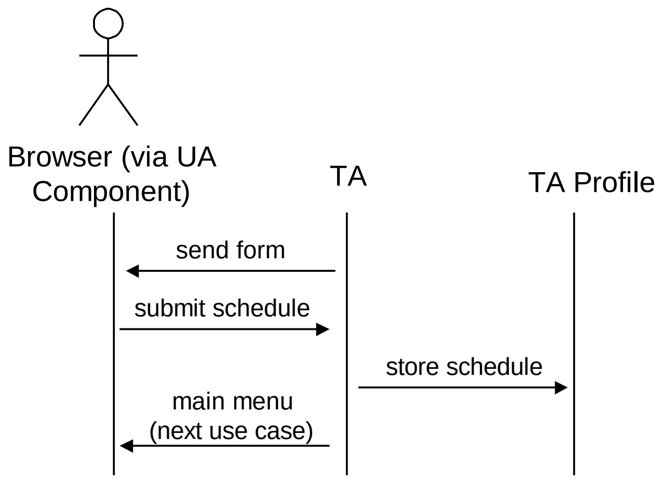

Figure 22 : Process scheduleReporting in use case maintainService

4.2 Stock domain specific event flow

Within the stock management pilot application, the *task agent* will actually consist of two parts:

- A user specific part responsible for dealing with the content of a users portfolio and cash account.
- A component that is shared among all users on a single host responsible for retrieving stock values from stock information providers. This component is introduced due to the fact, that stock information needs to be updated very frequently. Thus it would be a lot of bandwidth overhead if every single user would retrieve stock values at a high frequency. Instead, users subscribe to the shared component referred to as *value server*. The *value server* consists of the active *value server agent* component, the *value server profile* (storing data about subscribers and information sources) and a data store facility (to store the share values).

The data structure used to describe an entry in a user's portfolio consists of the following attributes:

attribute	***example Values***
company name	Oracle
ticker symbol	ORC
source of values (provider)	Yahoo Financial US
source of values (stock exchange)	New York Stock Exchange
stock exchange business hours	16:30-22:00 GMT
value provider update times	16:30-22:00 GMT, constantly (= 1 sec.)
delay	15 min
currency	US $
last value	152.50
timestamp of last value (GMT)	10.10.1999, 16:15
value at purchase	145.30
timestamp of purchase	01.01.1999, 12:00
amount of shares purchased	200
lower limit	130.00

attribute	***example Values***
upper limit	165.00
annotations by the user	should sell this at $ 165.00, but not before 01.12.
link to related industry news	http://biz.yahoo.com/n/o/orcl.html

Figure 23 describes the event flow for the use case *buyStocks*.

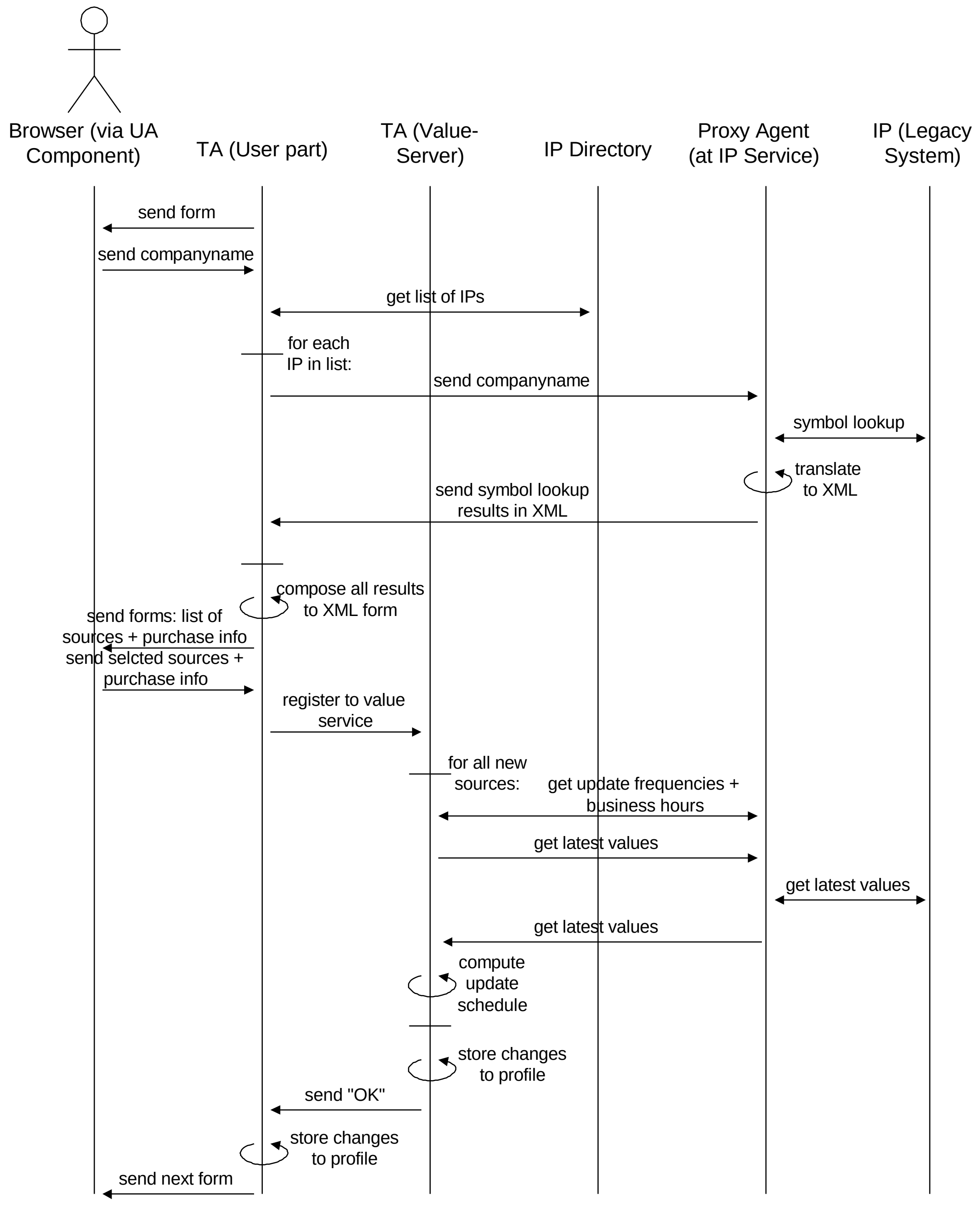

Figure 23: Use case *buyStocks*

Annotations:

- Interaction with the user might continue immediately after he submitted the list of selected sources and the stock share purchase information. The internal processes of registering to the value server could be regarded as independent of the ongoing user interaction.

Figure 24 describes the event flow for the use case *sellStocks*. This use case may be entered by the user when viewing the portfolio contents. The user highlights a particular stock position and chooses the option to sell these stocks. If the user sells all stocks within one position, the user component of the *TA* needs to unsubscribe from the value server component of the *TA*. In that case, if the particular user is the last subscriber to a specific stock value feed, the value server component of the *TA* needs to stop serving the values of this particular stock.

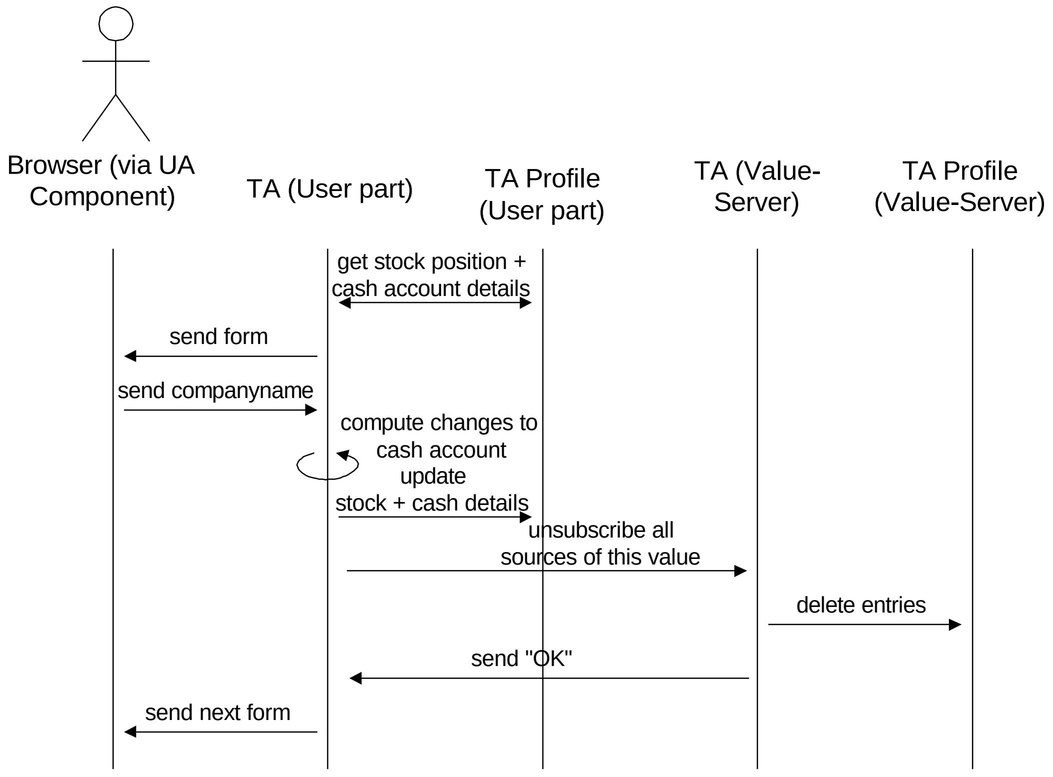
Browser (via UA Component)
TA (User part)
TA Profile (User part)
TA (Value-Server)
TA Profile (Value-Server)
get stock position + cash account details
send form
send companyname
compute changes to cash account
update stock + cash details
unsubscribe all sources of this value
delete entries
send "OK"
send next form

Figure 24: Use case sellStocks

Figure 25 describes the system internal process of retrieving stock values from information providers.

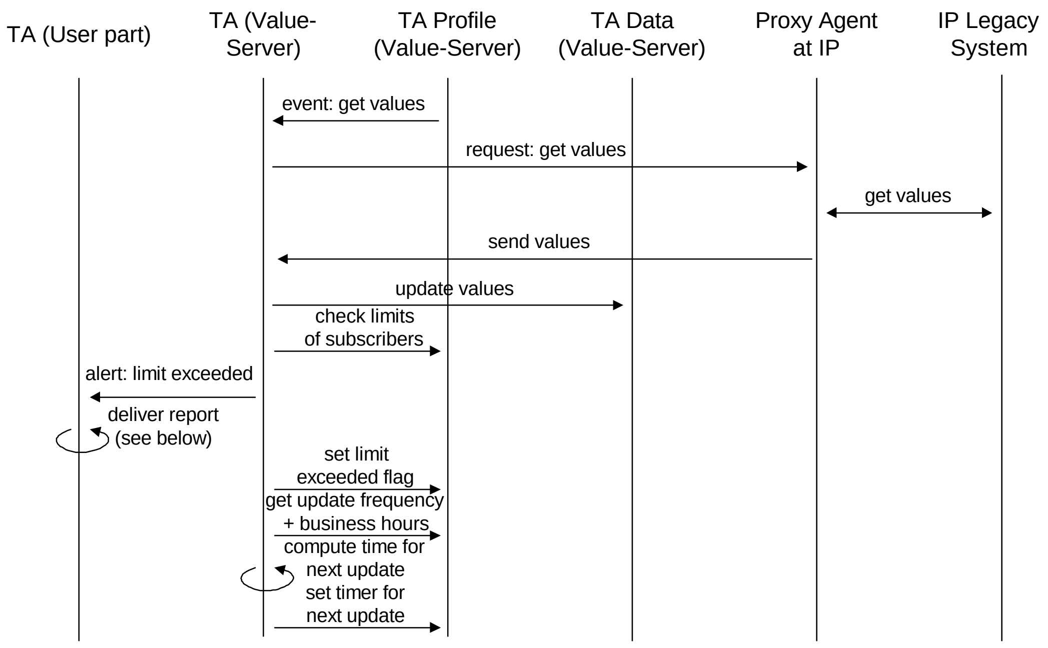

Figure 25: Process of stock value retrieval

Annotations:

- The ”*limit exceeded*” flag is set whenever a limit was exceeded. It deactivates the trigger and thus prevents the trigger from firing indefinitely. The trigger is reactivated when the user entered a new limit.

Figure 26 describes the event flow of the use case *deliverReport* for the stock domain in case the reporting is triggered by the event of exceeding a user defined stock value limit. It is triggered by a listener event send by the value server component of the *TA*. The value server did send an alarm including the actual stock value together with other related attributes like the source of the value and the timestamp for the value. The user did not specify a location defining where the reports should be sent to. Thus the *TA* needs to contact the *PA* to lookup where to send reports according to what is specified in the diary section of the *PA profile*.

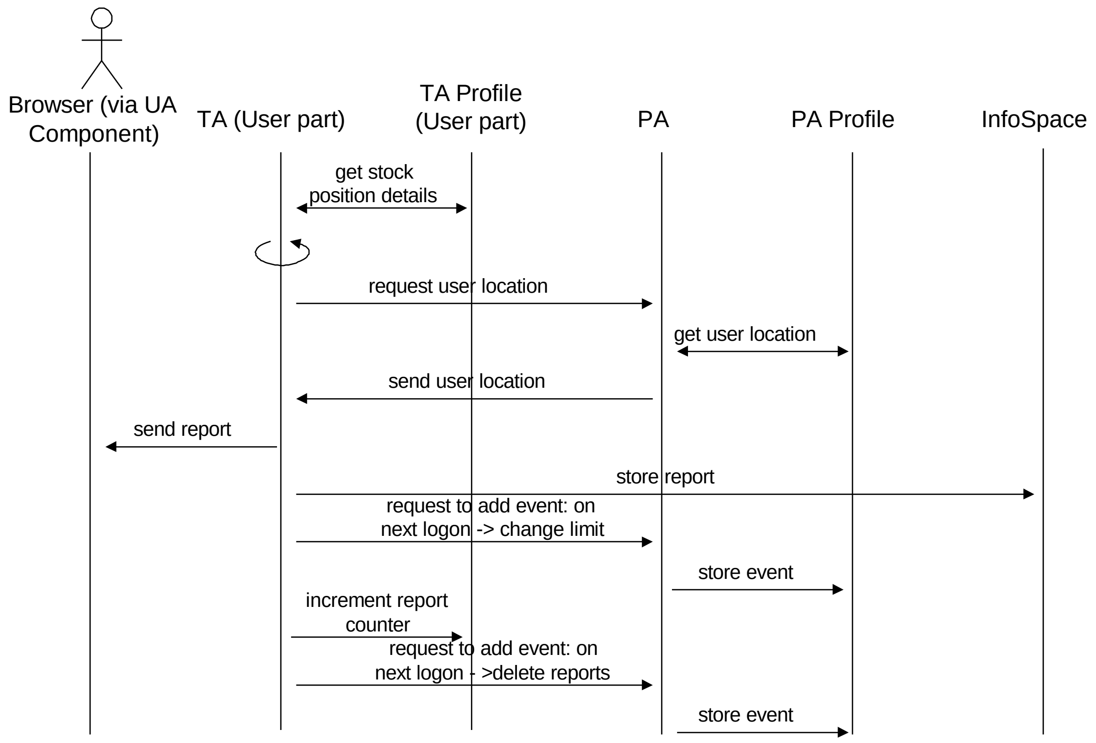

Figure 26: Use case *deliverReport* (event triggered)

Annotations:

- Layout information (XSL-files) for reports are part of the *TA* distribution downloaded from the respective *TA factory* and is stored in the *information space*?
- Once a limit was exceeded, the *TA* instructs the *PA* to create an event tag in the *PA profile*, that will cause the *PA* to immediately call the *TA*s *setLimit* method when the user logs on to the *PA*.
- To prevent storing large amounts of outdated reports in the *information space*, each *TA* may only store a limited number N of reports. Any time the *TA* stores a report in the *information space* (for later reference by the user), a report counter is increased. When N reports have been stored in the *information space*, the *TA* stops delivering reports and instructs the *PA* to create an event tag in the *PA profile*, that will cause the *PA* to immediately call the *TA*s *manageInfoSpace* method when the user logs on to the *PA*.

Figure 27 describes the event flow of the use case *deliverReport* for the stock domain in case of a regular, scheduled report. It is triggered by a diary event stored in the diary section of the *TA profile*. The user did not specify a location defining where the reports should be sent to. Thus the *TA* needs to contact the *PA* to lookup where to send reports according to what is specified in the diary section of the *PA profile*. When the report is not triggered by schedule but directly by the user (choosing the immediate reporting option from the *TA main menu*) the event flow is the same than described in Figure 27 except that the call to send a report originates from the *UA* instead of the *TA profile*. The process of determining the user location is still the same except that the fact that the user is online connected overrides the entries in the diary.

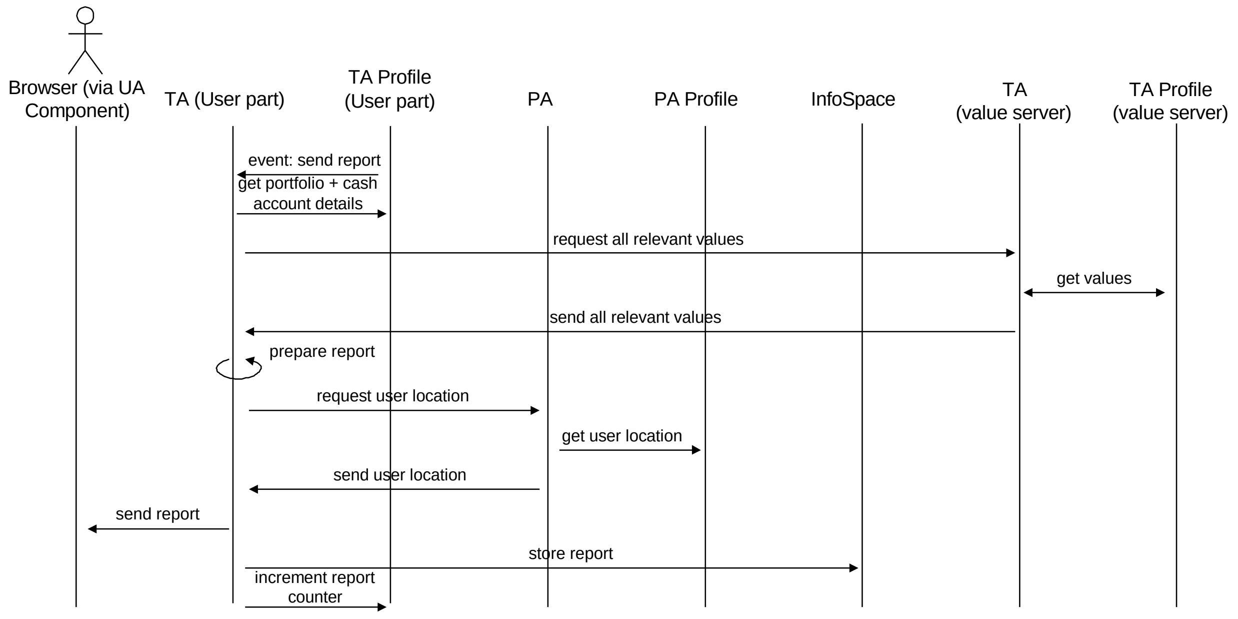

Figure 27: Use case *deliverReport* (scheduled)

The event flow for the use case *manageCashAccount* is described as follows:

- The user requests to manage the *cash account* by selecting the respective option in the *setTask-Parameters/useService* menu of the *TA*.
- The *TA* retrieves all relevant information (*cash account* transaction history and up to date values) from the *TA profile* (or data file).
- The *TA* prepares a form using some XSL-stylesheet.
- The user may view the history or enter other new transactions (like adding/withdrawing money; paying fees...). He must specify a transaction- type by choosing from set of available types.
- The user submits the filled out form to the *TA*.
- The *TA* checks if all input is valid (using a DTD) and stores the changes to the *TA profile*.

Issues not covered in detail in the first implementation of the pilots (due to a more static approach):

- In the stock domain the directory of available *information providers* (*IP*s) is not queried on the fly. For various technical reasons user interaction is required when selecting information sources (i.e. symbol lookup - this part could be redesigned any time whenever there is a worldwide standard for ticker symbols). Selecting *IP*s (information sources) is done by the user during the use case *buyStocks*. However, additional *IP*s might become available anytime. Thus there is a need for a mechanism that allows stock *TA*s to check for new *IP*s every now and then so that they can inform their users about these new sources. Therefore the *TA profile* provides a parameter that specifies the frequency at which the *TA* will automatically check the *IP directory* for new *IP*s (i.e. once a week). To determine which *IP*s are new (since the agent last checked the directory), one of the attributes of an IP-entry in the *IP directory service* must be the date at which the *IP* was advertised in the directory.

- *TA*s might as well check for changes to interface descriptions of *IP*s (either changes to attributes like update frequency or business hours or advertisement of new methods) on a regular scheduled basis.
- *TA*s might check their creators (*TA factories*) for updates on a regular scheduled basis. When updates become available, *TA*s could be automatically updated by recompiling, exchanging code, linking new classes or extending/exchanging mission profiles.

4.3 Event notification domain specific event flow

The basic data structure within the domain of event notification is of type *event*. An event can be basically described by:

- Event type (What?)
- Event location (Where?)
- Date of event (When?)

Investigations in modeling these three aspects showed that they can be of almost arbitrary complexity. To illustrate this, we give a few examples of textual event descriptions:

- Trade fair 19.03.-25.03.1998; 19./20.03. 10-20 h at location A; 21.03.-25.03. 9-19 h at location B
- English courses June to August at the university; every Friday at 18:00-20:00; not 19.06.; room numbers to be announced individually one week in advance
- FC Bayern fan club pub meeting at pub XY; all year every 1st Monday of the month at 18:00; open ended
- Bus schedule for bus station A; workdays 6:00-23:00 every 20 min starting at 6:04; weekends 9:00-18:00 every 30 min starting at 9:04

Our approach towards handling this complexity is to start with a rather simple model that covers most but not all types of events. This model is designed to evolve along with the rest of the system by introducing a versioning concept. Events and task agents (that perform queries on events) will have an attribute labeled *version*. Version 1.0 agents will be capable of querying version 1.0 events. Version 1.1 agents will be capable of querying both version 1.0 and version 1.1 events.

Data model for events (version 1.0):

attribute	***example Values***
event title	Programming Agents in Java
event description	FAST-seminar where participants will learn how to create agents using Java
event category (up to three levels)	Education_Training_Knowledge.Courses.Computer
start_date, start_time, end_date, end_time	12.03.1998, 14:00, 12.03.1998, 16:30
location of event (postal address)	FAST e.V., Arabellastr. 17, 81925 Munich, Germany
location of event (geographic location)	x = 29,27; y = 28,63
event organizer (postal address)	Mr. Stein, FAST e.V., Arabellastr. 17, 81925 Munich, Germany, Tel. ++498992004755, Email hgs@fast.de
web-link to event (by organizer)	http://www.fast.de/seminars/JavaAgents

attribute	***example Values***
target audience	anyone interested in Agents and Java programming
price	students: DM 200,- ; standard DM 500,-

Event title and description are both strings. Their contents are not restricted to specific values. Agents will not query these attributes. They are intended to serve as additional information for human readers.

Event categories are described in up to three levels of detail. As the system evolves more levels might be introduced. Event categories are typed data attributes that can be queried by agents. Their values are restricted to predefined value sets. So far, the following categories have been identified by our partners in the Bavaria Online network:

First level	***second level***	***third level***
Formal Education, Training, Knowledge	Educational System	…
	Fine – Arts Educational System	…
	Courses	Courses about Languages
		Courses about Health
		Courses about Computer
	Consulting sources	…
	Others	…
Art and Culture	…	
Sports and Recreation	…	
State, Politics, Administration	…	
Philosophies and Religions	…	
Health and Socials	…	
Natural Science and Technology	…	
Economy, business, trade, industry, craft	…	
Agriculture	…	
Others	…	

Within version 1.0 of the event data model, events cannot be grouped. There is no way of modeling meta-structures that, i.e. would group a series of workshops. As far as the workshops have different topics or take place at different locations and thus need to be individually described, they are treated as separate, unrelated events.

Periodicity also cannot be expressed in the version 1.0 model. Periodic events like, i.e. a meeting every Monday, are modeled as separate events with specific location in time like 'meeting on 09.03.1998'.

However, the timeline of an event can be split into an arbitrary number of time intervals or sessions. This is only possible when each session takes place at the same location and separate sessions do not require separate descriptions. Only events that under normal circumstances require the user to attend all of the sessions in order to attend the event should be modeled using the session approach. As an example, a German course with 10 separate sessions could be modeled that way. The event will only match the user's query if all of the session dates fit the user's time constraints. In case of a series of workshops, where the user could be interested in attending only some of the advertised sessions, the session model won't fit. Instead each workshop will be modeled as separate event. The group structure of the series of workshops is not coded.

The physical location of an event is coded in two ways. First, it is coded as postal address which serves as textual information for the user. The postal address consists of the following attributes:

- Address name (i.e. FAST e.V. or Ludwig-Maximilian University Munich)
- Name of department
- Street
- House number
- Floor and room number
- Postal code
- City
- Country

Second, it is coded using geographical coordinates (longitude and latitude). This is used by the system to enable queries like ‘within a radius of 50 km around my current location’.

The event organizer is coded as a postal address with the following attributes:

- Organization name (i.e. FAST e.V. or Ludwig-Maximilian University Munich)
- Name of department
- Name of contact person (Mr. Stein)
- Street
- House number
- Floor and room number
- Postal code
- City
- Country
- Phone number
- Fax number
- Email address

An optional web link can be provided to link the user to additional information either on the event or on the organizer of the event.

The attribute target audience specifies who could or is allowed to participate in the event.

The attribute price specifies the costs for participation.

For technical reasons, two additional attributes will be attached to every event (done by the system providing the data storage facility for events):

- Name of the organization hosting the event data storage facility (e.g. the name of the Bavaria Online node)
- Date of last update of the event data entry in the storage facility

Above described data structures of type event can are queried by task agents according to a user's needs. When defining data queries, the user may specify a title for the query and the following parameters:

- The user may specify the following parameters to define which events are of interest to him:
  - Event category: The user needs to select on or more entries from the list of available categories. The specified level of detail is up to the user. Examples given below:
    - Example 1: Event category: All
    - Example 2: Event category: Formal Education, Training, Knowledge.Courses.Courses about Languages
  - Event location: The System will provide a clickable map interface with zoom levels. The user may pin a location and specify a radius, thus defining the area in which events of interest should be located. The input is interpreted by the system in form of geo-coordinates (longitude, latitude). The respective interface will be provided by a third party geo information system. *TA*s will communicate with this component via Java interfaces.
  - Event time: The user specifies a time interval into which events of interest need to fit. The interval is specified by start and end calendar dates (no daytime). In addition, the user may provide two kinds of patterns: weekdays and daytime intervals. See the following example:
    - Calendar interval: [01.04.98; 31.06.98]
    - Weekday pattern: {Monday, Tuesday, Sunday}
    - Daytime pattern: [16:00; 18:30]
- The user may specify the following parameters to define when the system should deliver reports:
  - A time interval during which reports should be delivered
  - A set of weekdays at which reports should be delivered
  - The daytime at which reports should be delivered
  - The device to which the reports should be sent (optional)

Unlike in the stock domain, there is no value triggered reporting within the event notification domain. Therefore queries need only be executed prior to report delivery. There will be a fixed offset of two hours between query execution time and report delivery time.

The user may declare two additional parameters that specify the minimum and maximum time gap between the execution time of the query and the point in time where the events take place (*minTimeOffset* and *maxTimeOffset*). These parameters are specified in terms of days. Say, e.g., the minimum parameter is set to 7 days and the maximum parameter is set to 14 days. Every time the query is executed according to the user defined schedule, only events will be returned, that are located at least 7 days and at most 14 days in the future. These parameters are optional. If they are not explicitly specified, the minimum parameter is by default zero days and the maximum parameter is by default set to 365 days.

The system checks for outdated queries. Whenever either the interval of the reporting schedule or the interval specifying which events are of interest are outdated, the system will inform the user and prompt him (on his next interactive session) to either drop the query or specify new intervals.

Figure 28 describes the event flow for the use cases *defineQuery* and *maintainQuery* simultaneously. Prior to the described process, the *TA* did look up the titles of the already defined queries in the *TA* *profile* and display it to the user. The user then did either choose to define a new query or to change pae pae parameters for an existing query. In the first case, the form sent *b*y the TA is blank or contains default values. In the second ca*se*, the TA looks up the currently defined query parameters *in the TA* profile and displays them within the form so the user may

e them.

(Note that when the user changes the parameters of an existing query, the TA must delete the time-stamps of the last visit to information provider sites. The purpose of this parameter is described in the process description of executing a query – see

b

elow).

EMBED Visig.

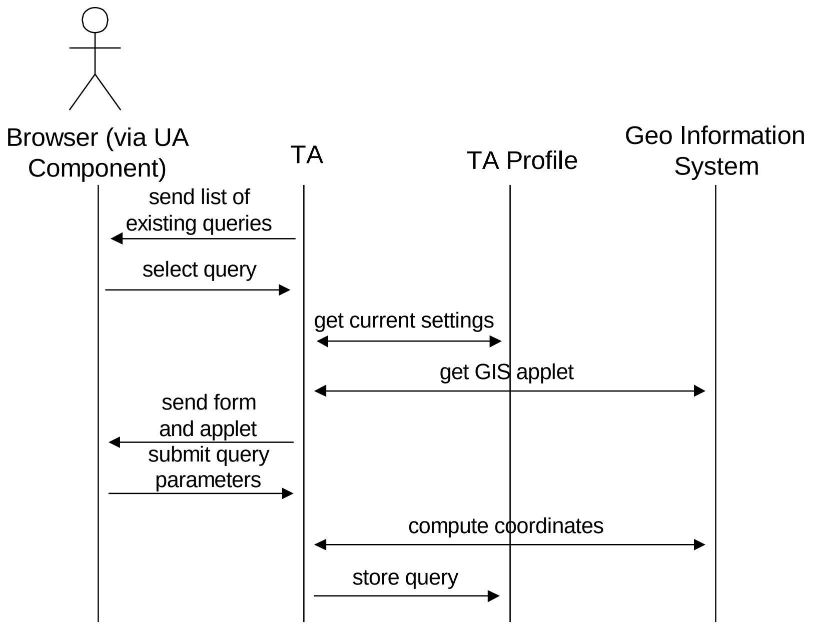

Figure 28: Use cases defineQuery and maintainQuery

The process of defining a reporting schedule for a single query is identical to what has been described for the generic use case *scheduleReporting*. The use case is entered either automatically after the user did define a new query or by the user requesting to change the schedule of an existing query.

To delete a query, the user selects the respective query from the list of existing queries (see above) and submits his delete request. The *TA* then deletes the respective entry from the *TA profile*.

Figure 29 describes the process of the system executing a scheduled query. This process is triggered by the *timer component* of the *TA profile* according to what the user specified in the reporting schedule. When the *service interaction interface* at an *information provider* site receives a query statement by the *TA*, it passes the query to the *information provider*'s database engine. The database engine has to match the query parameters against the database entries. This means in detail:

- Matching of event categories.
- Matching of time patterns: Events must fit in the specified time intervals (calendar date and day-time) and match the pattern for weekdays.

- Matching of parameters *minTimeOffset* and *maxTimeOffset* (see above) relative to the execution time of the query.
- Matching of geographical location: The database must compute the distance between the location of the event and the location specified in the query. The distance may not exceed the distance specified in the query.
- Matching of last update times: The *TA* provides the date if its last visit of the respective information provider site in the context of a specific query (*lastVisit* parameter). The timestamp of the last update of an event entry in the data base needs to be dated after the date of the last visit. This mechanisms ensures that events are not queried twice (unless there was an update to the database entry).

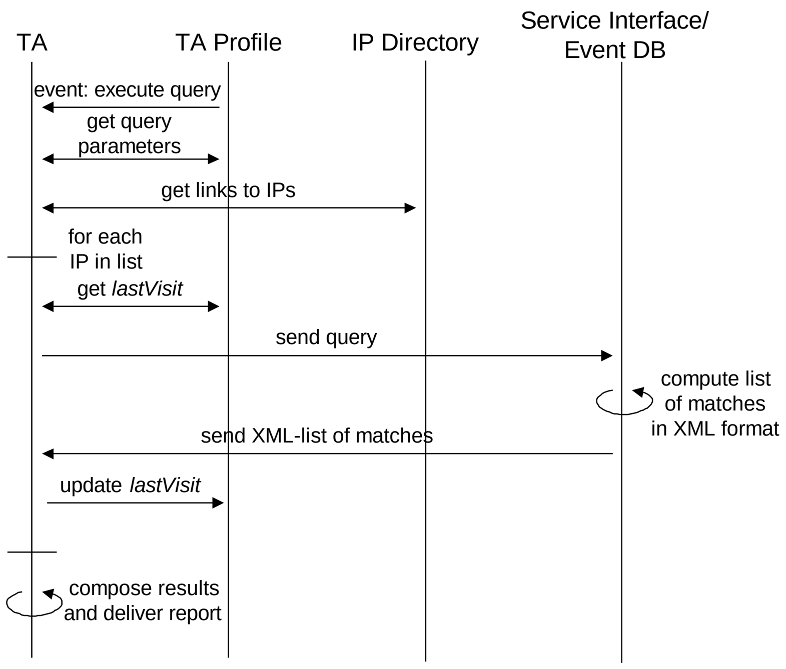

Figure 29: Event flow of query execution

Delivery of reports is very similar to what is stated for the stock domain. When the user did not explicitly specify a device, the appropriate device is looked up in the *PA profile*. The report is sent to the respective device and stored in the user’s *information space*. The report counter (see stock domain) is incremented.

5. Deployment of System Components

Deployment of system components is organized in close cooperation with our partners in the *Bavaria Online* network. For deployment and evaluation of the first prototype of the applications, we identified five *Bavaria Online* nodes that committed to actively participate. Contact to technicians running the nodes was successfully established. The nodes are all located in close geographical locations which is of significant importance with respect to the event notification system (people might not be interested in events to far from their hometown). The nodes are located at the following towns:

- Schwindegg (project contact point)
- Muehldorf
- Landshut
- Ebersberg
- Weihenstephan

During system development, all components will be hosted by *FAST*. Hosts at *FAST* will be set up to emulate the role of the *Bavaria Online* nodes. Once first versions of the system are up and running, the respective components will be moved to the selected *Bavaria Online* nodes. In detail, components will be distributed as follows:

- In the stock domain, *information providers* will be well known stock information providers like Yahoo US, Yahoo Germany and some other German providers. Content from these sites is provided either in HTML or in ASCII format. S*ervice interaction interfaces* to these sites will be hosted by *FAST*. Agents acting on behalf of their users and residing on *Bavaria Online* nodes will query these *interfaces* to retrieve information.
- In the event notification domain information is provided by local event organizers (i.e. cinemas, schools, etc.). They communicate their event advertisements to the *Bavaria Online* nodes by means defined by the individual *Bavaria Online* nodes themselves. Basically, *Bavaria Online* programmers will develop a front end tool that assists *information providers* in inserting their data into a data storage facility hosted by the *Bavaria Online* nodes. The data storage facility may either be a standard SQL-based database (for those nodes that already have such a system in use) or a system of XML-files (for those nodes that do not have any database in use). In the first case there will be a mapping of the contents of the SQL database to a system of XML-files. *Service interaction interfaces* that offer FollowMe agents access to the event data stored in XML-files will be hosted by the *Bavaria Online* nodes. In addition, *Bavaria Online* programmers will provide mechanisms that allow the creation of event listings in file formats like MS Word or HTML for use with other applications like newsletters or web forms (see Figure 30 and Figure 31).

- All user specific components will be hosted by the *Bavaria Online* nodes (e.g. *personal assistants*, *task agents*, *information spaces*, *agent profiles*, *user access modules*).
- *FAST* will be in the role of the *service providers*. All related components (e.g. the *service directory*, *information provider directories*, *TA factories*) will be hosted by *FAST*.
- *FAST* will host the third party *geo information system* required for the event notification system as described in 3.2.2.
- *PA factories* and *information space factories* are part of any FollowMe *place* and thus will be available at any *Bavaria Online* node.
- For locating a user's *PA*, we implement a global *PA directory* located either at *FAST* or at one of the *Bavaria Online* nodes.

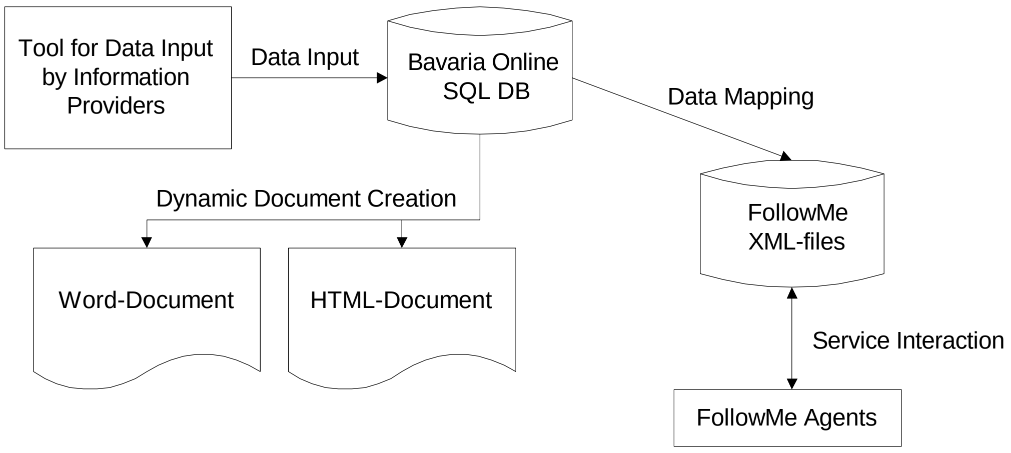

Figure 30: Event notification data storage with SQL DB

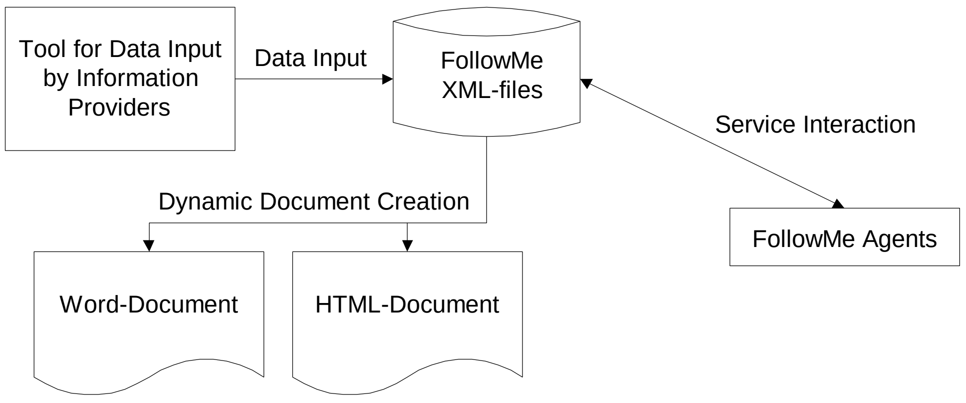

Figure 31: Event notification data storage without SQL DB