INF5150 Obligatory Exercise
Drop 1, Autumn 2005, Group 3

Bjørn Olav Samdal
Florian Müller
Ingar Vindenes
Shawn Edward Svendsen
Tom Lysemose

October 14, 2005

Contents

List of Figures

1 Introduction

In this course we are supposed to make a system that offers a so-called "*Multiple Blind Date Service*" (MBDS). A MBDS is a public service, offered by a third party vendor, for meeting a group of people for certain events that are identified by their type and time. A customer is thereby able to send a SMS to the dedicated MBDS number to join an event that he looked up before[1]. The *Joining-Messages* could look like "Go for a beer at 8 PM Saturday" or "Play bridge at 2 PM today". At an appropriate time the MBDS sends a text message to the customer, notifying him about the upcoming event and makes a suggestion how the participant could use the public transportation system to get to the event.

The service is built upon a distributed architecture. Beside the MBDS itself it uses a telecommunication interface to send and receive text messages to and from mobiles. It also uses a mobile positioning system, to track the actual position of the customers. The last system that is uses is the "Trafikanten" system, the local traffic company, to find out the individual routes for the customers.

In the following, we describe the different types of messages within the system to clarify the description of the MBDS; to make the interaction between the customer and the MBDS as easy as possible there are different types of messages that a customer can send to the service:

1. *Join-Message*: With the Join-Message a customer is able to sign up for an event. The service registers this for a future notification.
2. *Cancel-Message*: If a customer wants to sign off an event, he can send this type of message to the service.
3. *Get-New-Route-Message*: The customer is at any time able to request a new route description. The system will track the position of the customer and send him a new route description. One can find more to this type of message in section 3.10.

Therefore on the part of the MBDS the types of messages are:

1. *Confirmation-Message*: When a customer signs up for an event, or if he canceled an event, he gets a confirmation for this in form of a SMS message.
2. *Notification-Message*: The customer will be notified at an appropriate time before the actual event that he signed up for. With this message the customer will also get a description how he should use the public transportaion system depending on his momentary position.
3. *Route-Message*: The customer can ask for a new route description due to chaning his position etc. The MBDS will send a new route description for the next event to the customer based on the current position of the customer.
4. *Cancel-Message*: The customer gets an confirmation for their cancelation of participation, if a cancel-message has been sent to the system.

Within this overview one has to say that there are already some extensions mentioned here: The basic system only consists of the *Joining message* and the later *Notification message*.

[1] For example, the vendor could publish these events on his internet pages or in magazines.

2 Assumptions

To be able to model the MBDS effeciently one needs to make certain assumptions of the area of application. Within our model we made the following assumptions:

- We model the communication between the MBDS and a single customer.
- We assume that the used data classes have appropriate getter- and setter-methods.
- We describe how the data is coming into the system, but not how the data is stored.
- Therefor we assume that the events and the event locations, that are belonging to the actual events, are already created by the vendor's staff. This will, in fact, change later in the modifications section (see section 4.3) where the event location of a single event is set up a short time before the notification of the participants takes place. This system does not handle the erasure of expired events.
- To be able to track the position of a customer we assume that the mobile phone of the customer can be positioned at all times.
- To get the individual route descriptions (e.g. bus stops, bus schedules, delays) we have to use the local Trafikanten system.
- Consequently we assume that the customer is located within the range of the Trafikanten system and that the farthest event location for a customer takes him about one hour to reach.
- We also assume that the locations of the single events need no reservation. That is to say that the MBDS does not need to notifiy the owner of the event's location.
- The MBDS offers a fixed list of events. Therefore the customer cannot create their own event within the system.
- We assume that the Trafikanten system can handle requests that have the form like "*Give me the fastest route from A to B with the estimated time of arrival T*".
- We assume that all components within our system have access to the positioning and routing services.

3 The Multiple Blind Date Service

This section of the project contains several diagrams and explanation for what the system is intended to do. The following pages are made with respect to the part of assumptions, 2. The diagrams are meant to give an overall view about which parts are interacting, and what kind of information is passed along.

3.1 Definition of diagrams

This section is meant to give a short introduction to the diagrams we have made and why we chose to make them. Each diagram will have a more detailed text explanation in the single sections.

Usecase diagram, 3.2 This is the use case diagram that shows which relations each actor has with the system, and what functions are called. This was an early diagram we made to specify what actors are interacting with.

Context diagram, 3.3 The context diagram shows the data flow in the scope of an organizational system with its boundaries and which external entities interact with the system. Also the information flows between the entities and the system are shown. This was the first diagram that we made, and was useful for making the composite structure diagram.

Composite structure diagram, 3.4 This is a more detailed level of the system boundaries and gives an overview of the internal components (with interactions between) of the blind date service. This diagram was a requirement of the project.

Class diagram, 3.5 The class digram defines the relations between the classes and objects of the system. This diagram was a requirement of the project.

Activity diagrams, 3.6, 3.7 The two diagrams give an overview of the messaging between the user and the service. These diagrams were made to see easily what messages are passed and to make sure that we had everything covered for the message passing.

Sequence diagrams, 3.8, 3.9, 3.10, 3.11 These diagrams show in which order messages are passed between the parts of the system, along with what kind of data and arguments. This diagram was a requirement of the project.

3.2 MBDS use case

Figure 1 shows a use case diagram for the modelled multiple blind date system. There are three actors within the diagram: The UserMobile, the TrafikatenSystem and the PositioningSystem, which all interact with the MBDS.

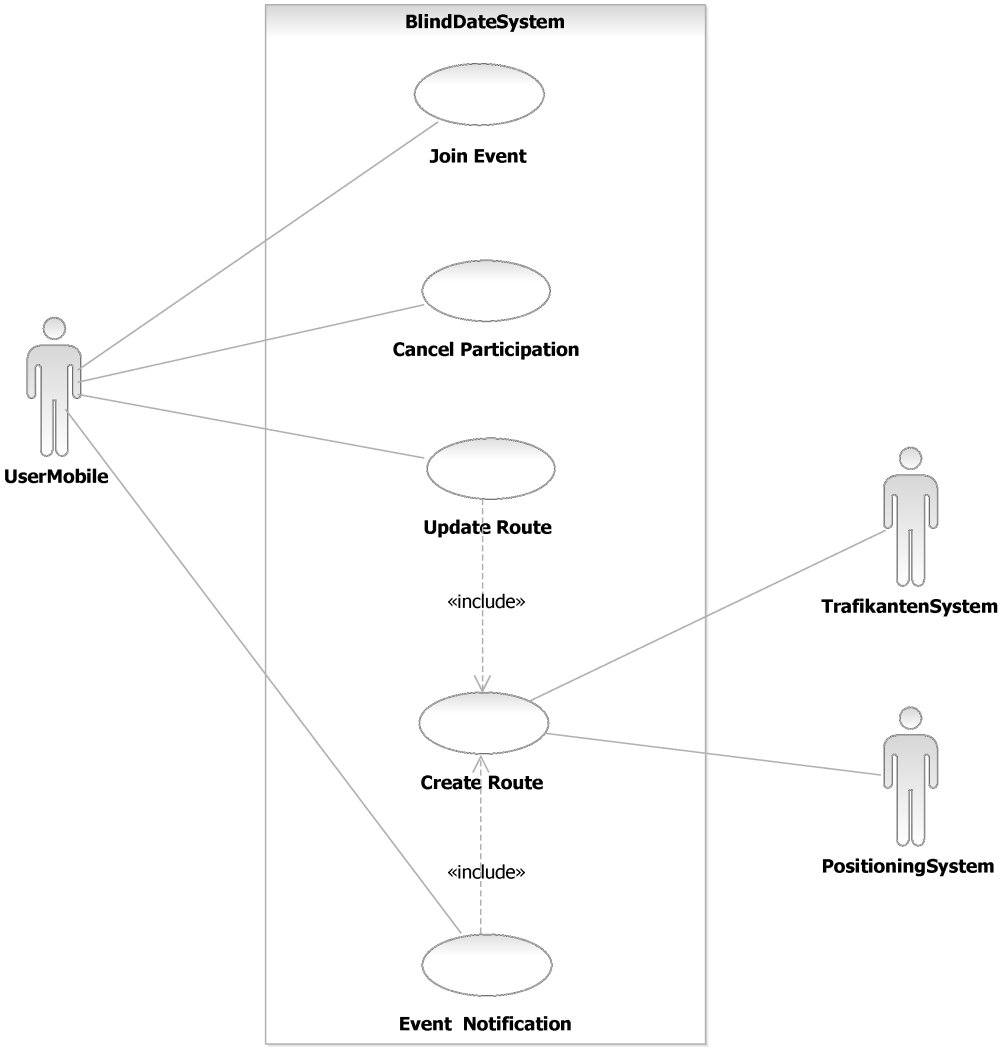

Figure 1: MBDS use case diagram

3.3 MBDSContext

Figure 2 displays the way the customer (user) interacts with the system. The MBDS is modeled as one whole actor in this case, even though the system interacts with the traffikanten and position-system.

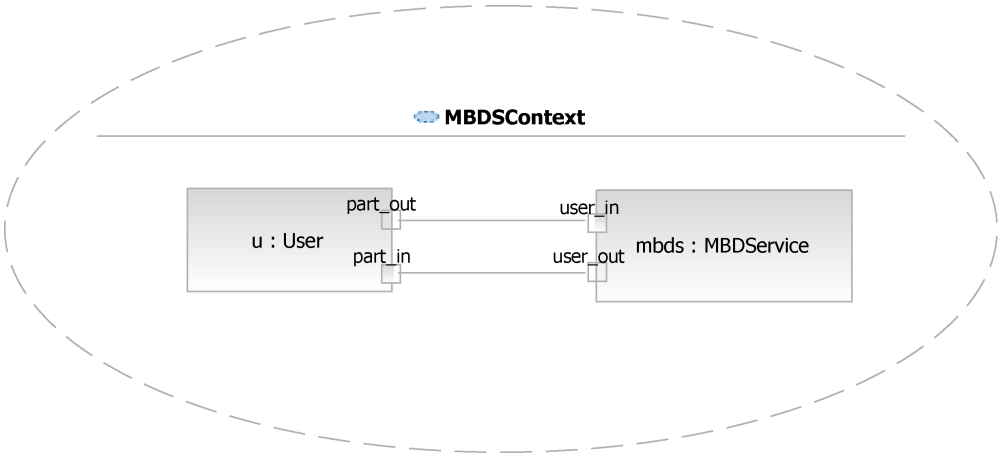

Figure 2: Multiple Blind Date Service context diagram

3.4 MBDS composite structure

Figure 2 displays the structure of the blind date system. This diagram also shows that the class *BlindDateSystem* is the only one which communicates with the user.

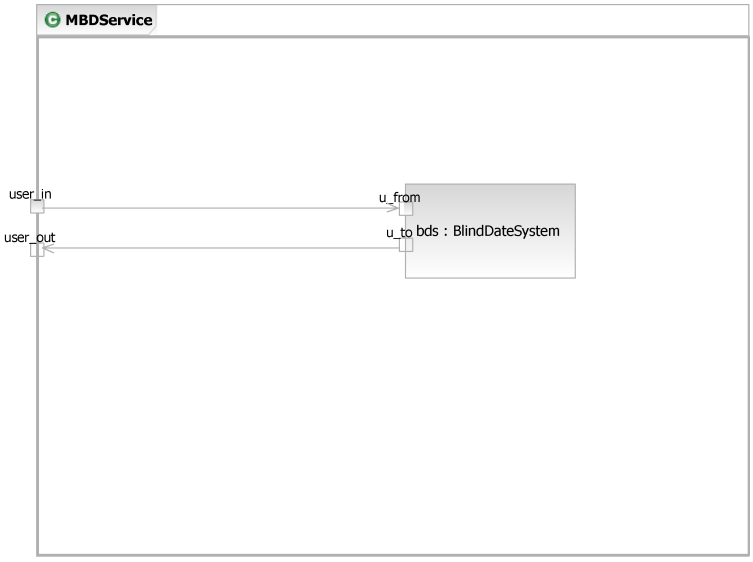

Figure 3: MBDService composite structure diagram

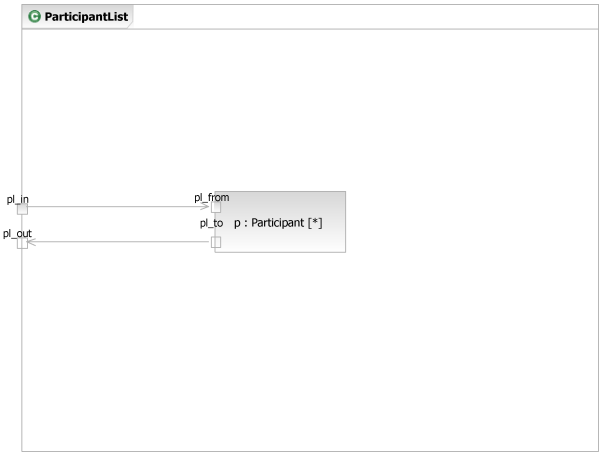

Figure 4: ParticipantList context diagram

3.5 Class diagram

Figure 5 shows the class diagram of the Muliple Blind Date Service.

The actual user of the system is represented as the class *UserMobile* whereas it only has the mobile number of the user as an attribute. This class is not connected in any way with the BlindDateSystem.

The pendant to the *UserMobile* class outside the mobile blind date service is the class *Participant*. This class is used to represent an user within the system.

There are some classes in the diagram that represent data types which are used within the whole diagram:

- **MobileNo**: MobileNo represents a mobile number of a user. The classes *MobileUser* and *Participant* have an attribute with this data type.
- **Position**: Position represents the data that contains informations about a certain position. E.g. an object of the type *Position* is returned by the positioning system to the BlindDateSystem.
- **TimeDate**: This class encapsulates a point in time and is used, e.g., to unambiguously store the date and time when an actual event takes place.
- **Route**: The *Route* class keeps the data that is returned by the routing service offered by Trafikanten. It keeps all the data which is needed to describe a whole Route, that is the place of departure and arrival, breakpoints etc.
- **PositionList**: List of positions, used to compute the optimal meeting place in one of the extensions.
- **EventTypeList**: List of supported event types, used by the class EventLocation.

The classes *RoutingSystem* and *PositionSystem* are representatives for the routing service of Trafikanten and the positioning service offered by the used telecommunication vendor. The *MBDS* holds a reference on each of these classes.

EventLocation stores all the information that is belonging to the single places where the individual events can take place. It has attributes to store the name, the position and the maximum number of participants of the place. It also holds a list of supported types of events.

The class *Event* is in close relation to the class *EventLocation*. *Event* represents the actual events which are offered by the vendor. For that purpose it has attributes to store the type, the time and the date of the event and also keeps a list of participants that have joined this specific event.

The class *Joining* is responsible for the mapping between the participants and the events. When a user joins a specific event an instance of this class is created and appended to the list of participants. For each participant of the event one instance is created, each having a single reference to a participant.

The central class of the system is the class *BlindDateSystem*. It is responsible for communication between the actual user and the multiple blind date service itself. The class BlindDateSystem keeps a list of all events that are offered by the vendor. It also has a reference to an instance of a *ParticipantList* class.

The *ParticipantList* class keeps a list of all users that have at least once used the Mobile Blind Date Service. It offers a method to get a reference on a participant through the participants mobile number.

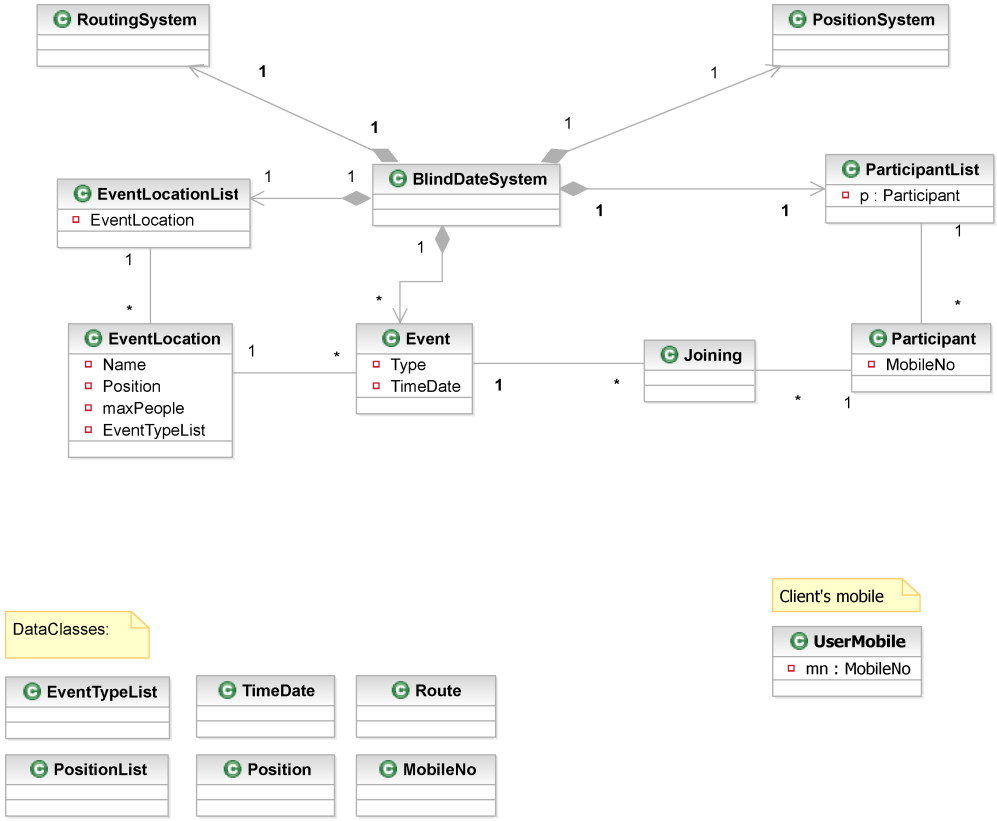

Figure 5: MBDS class diagram

3.6 Overview of incoming messages

Basically there can occur four different types of messages that the BlindDateSystem can receive:

1. **JoinEvent message**: JoinEvent occurs when a user wants to participate in an event.
2. **CancelEvent message**: CancelEvent occurs when a user want to sign off an event that he previously has signed up to.
3. **UpdateRoute message**: When a user wants to get a route description of the next event that he has signed up for, he can send an UpdateRoute message to the BlindDateSystem. He will get a route description, that takes his current position and the next event that he/she has signed up for into account.

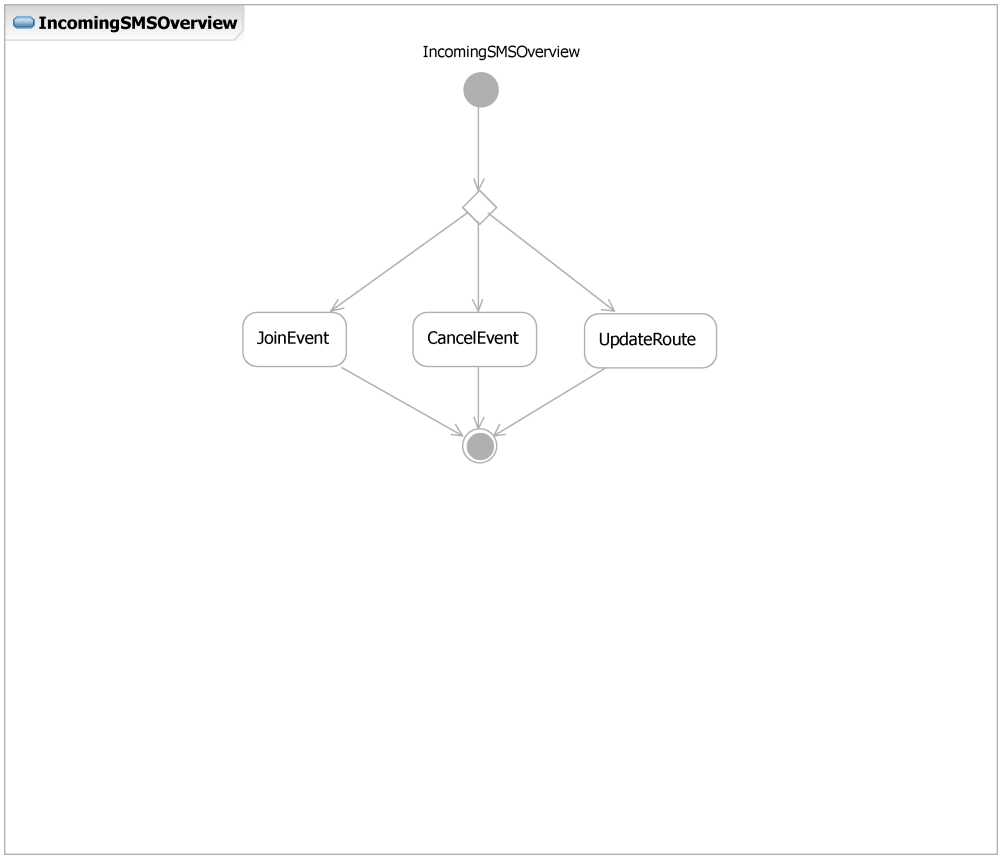

Figure 6: Overview of the incoming SMS messages

3.7 Overview of outgoing SMS messages

Figure 7 shows all possible types of messages that the BlindDateSystem can send. There are five different types of messages:

1. **SendJoinConfirmation message**: If a user wants to join an event and he/she was successfully added to an event by the BlindDateSystem, he/she will get a confirmation sent by the BlindDateSystem.

2. **SendCancelConfirmation message**: A user that has successfully signed of an event will receive a cancel confirmation message by the BlindDateSystem.

3. **SendEventNotification message**: The user will get a notification message about an upcoming event that he/she has signed up for. The event notification message includes the name and the time of the event and the route based on the users position which he/she has at the time the notification message is created by the BlindDateSystem.

4. **SendNewRoute message**: If the user wants to have a new route description (e.g. because he/she has changed his position), he/she can send an update route message to the BlindDateSystem. Hereupon the system will send an updated route description based on the current position of the user and the position of the next event.

5. **SendErrorMessage**: The BlindDateSystem will send an error message to the user in some cases. E.g. the user could try to cancel an event that does not exist etc.

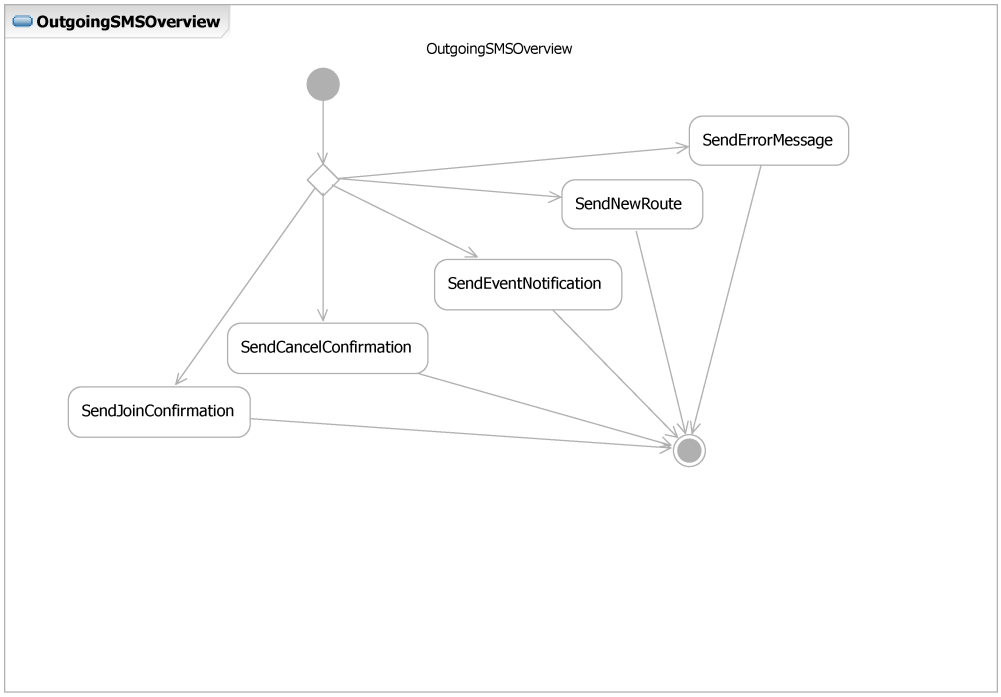

Figure 7: Overview of outgoing SMS messages

3.8 JoinEvent

The joint event (figure 8) interaction, described in the JoinEvent sequence diagram, is initialized by the user sending a joinEvent message to our system. When our BlindDateSystem receives the message it checks if the user is registered in our participant list. If not, the user receives an error message. At this point it should, under no circumstances, be the case that the user receives a message of successful joining of the event. Thus, this describes a negative trace.

If the user is enlisted in our participant list, the BlindDateSystem tries to identify the event matching the event identification provided by the user. If no match is found an error message is send. On the other hand, if a matching event is found, the joining may now be added in our system. This is further described in the AddJoining diagram.

Before entering this interaction (figure 9) we presuppose that both an event and a participant has been identified. The objective is then to add a joining of the participant to the event.

First, the event is "asked" to add the participant to its list of participant who has already joined. If, for example, the number of participants exceed the maximum number of participants of the event location (assuming there has been assigned an event location), the event may refuse to add a new participant. If so, the user receives a error message (joiningRefused). If the event "accepts" to add the participant, then the participant is asked to add the event to his list of joined events. If the participant accepts this, then a message of "successful joining" is sent to the user. However, the participant may reject to add the event, if for example he has already joined an event taking place at the same time. In this case, the joining is "rolled-back" by deleting the participant from the participants list of the event, and an error message is sent to the user.

3.9 CancelEvent

Figure 10 describes the activities that take place if a user sends a *Cancel Event message* to the *Multiple Blind Date Service*.

In order to cancel an event, the user has to include the events name into the *Cancel Event Message*. The number of the users mobile phone is automatically transferred with the message. Therefor the cancelEvent message within the sequence diagram contains two parameters: event and mobileNo.

3.9.1 sendCancelMessage

In order to cancel an event of a participator the BlindDateSystem asks in the first place for the participant. Afterwards it looks for the event, which the user wants to cancel. If the participant or the event do not exist then an error message will be send to the user. If both the participant and the event do exist then a message will be send to the actual event e, saying that participant p wants to cancel his participation. The ParticipantsList will aslo get a message, saying, that the participant p wants to cancel event e. In this implementation the user will get a confirmation even then, if the event and the participant exist, but the participant not joined the event. Figure 11 illustrates this.

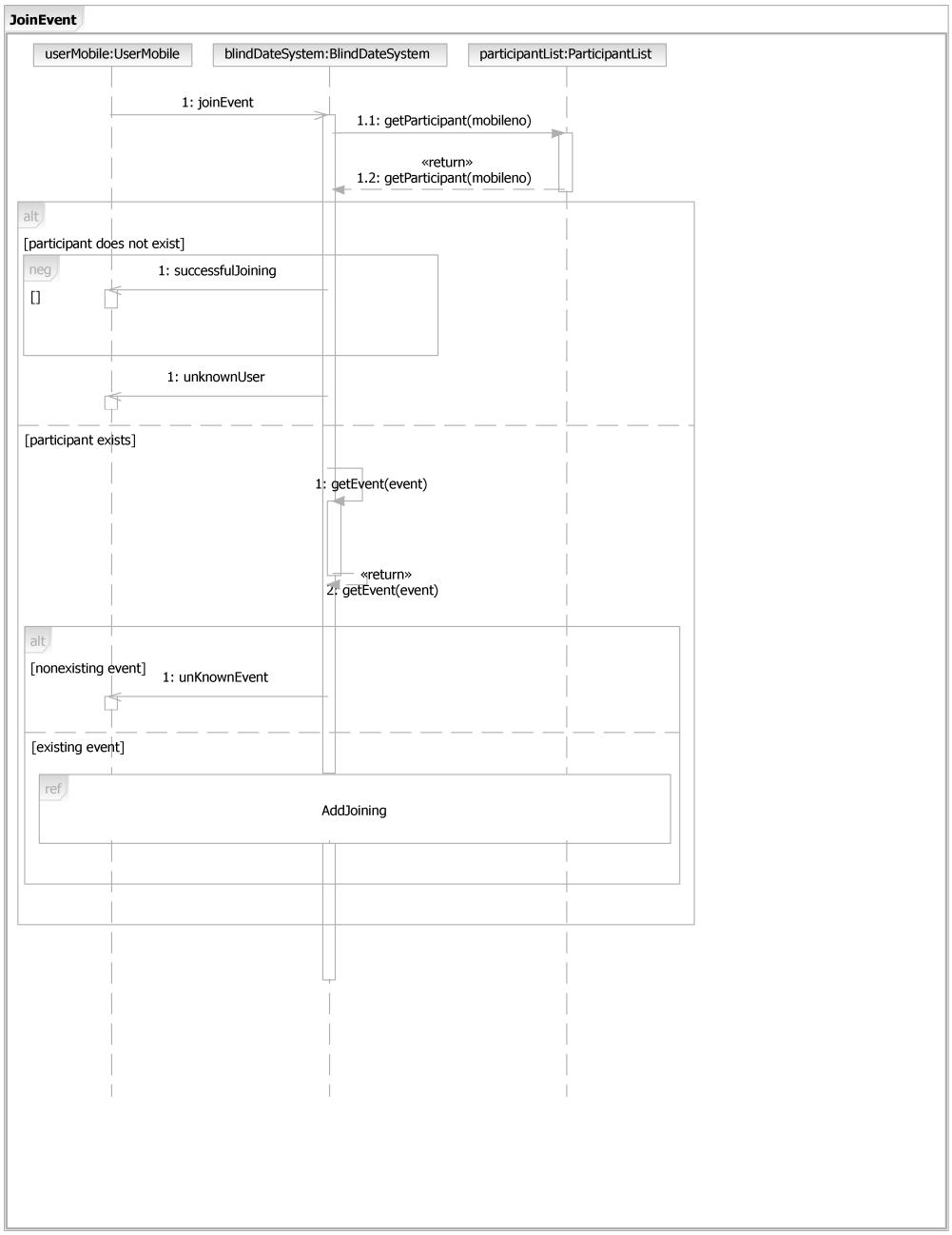

Figure 8: JoinEvent sequence diagram

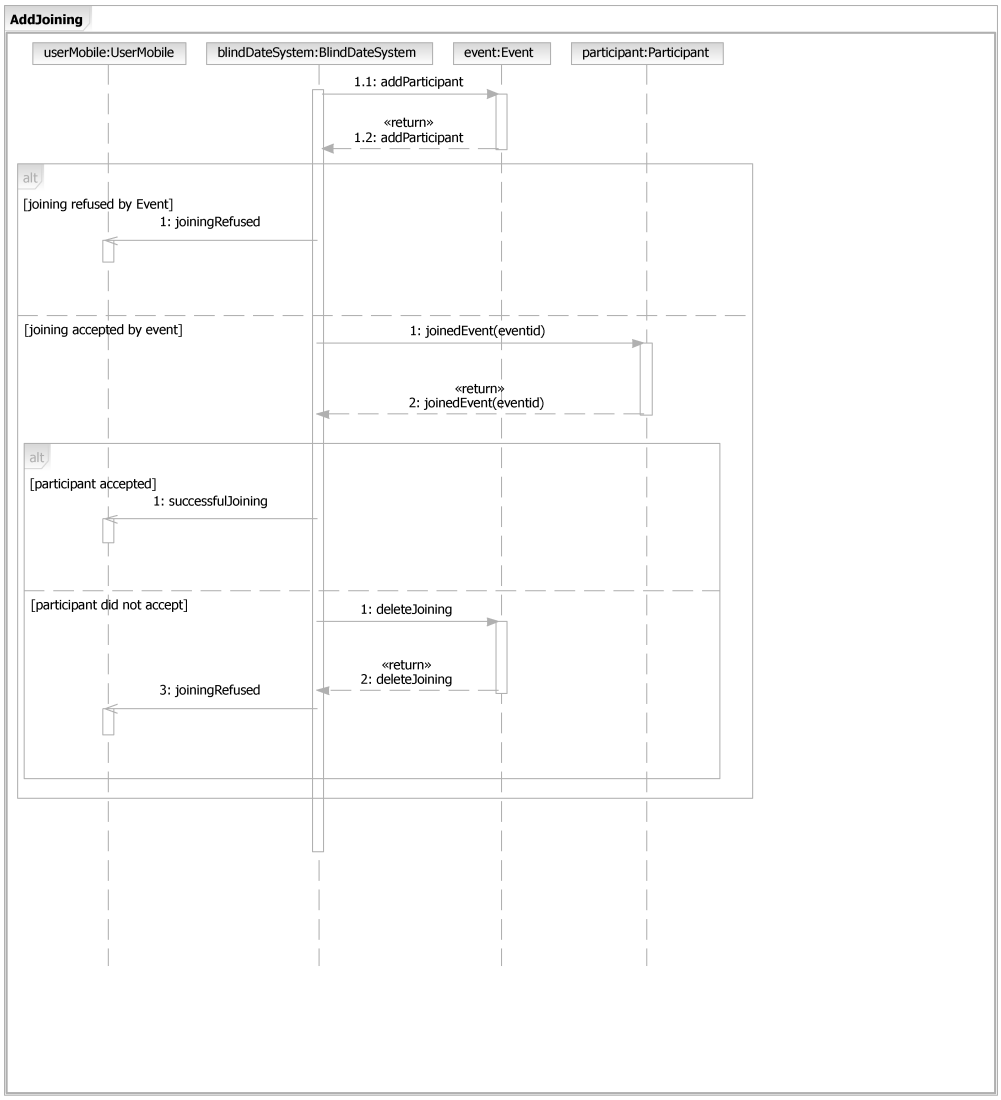

Figure 9: AddJoining sequence diagram

Figure 10: CancelEvent sequence diagram

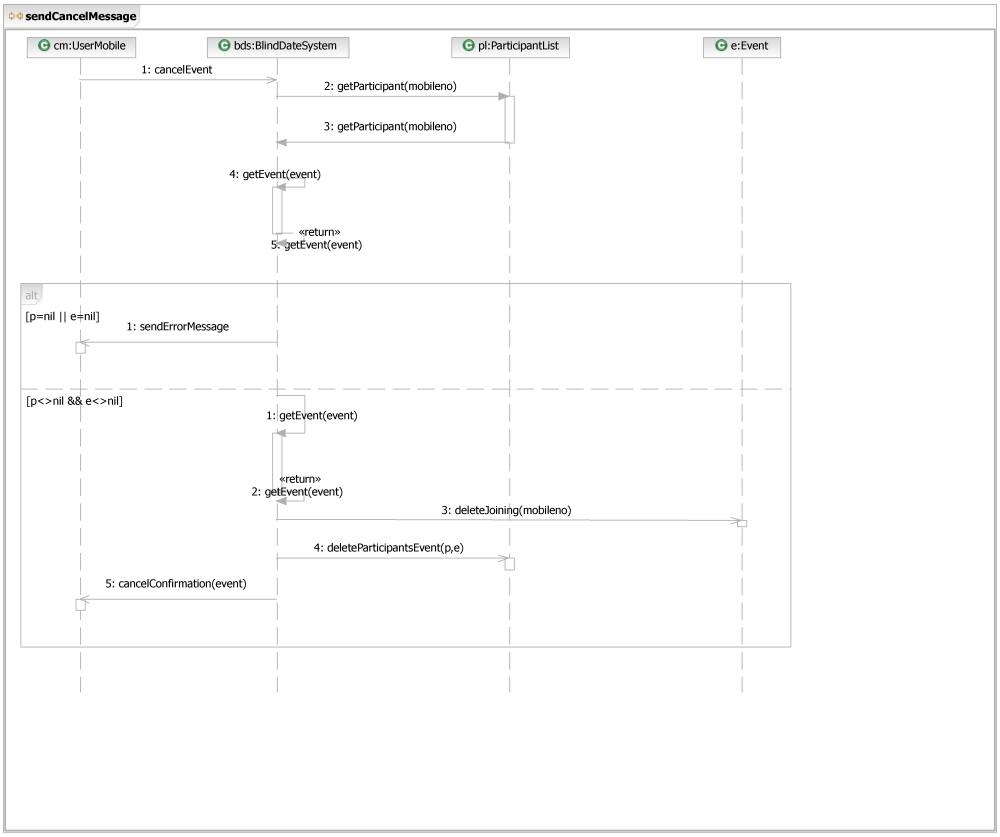

Figure 11: sendCancelConfirmation sequence diagram

3.10 UpdateRoute

The user will receive a message containing the route and name of the meeting place at a fixed time before the event takes place. If the user want this information before the system has sent the scheduled route message or the user has changed its position since the route message was sent, the user needs the update route functionality. If a user ask for an updated route when he/she haven't signed up for any events, the system will return an error message. Otherwise the system will return a message containing a route description, the event name and the time of the event.

3.10.1 getNewRoute

Figure 12 is a sequence diagram showing, with a very high level of abstraction, that when the user wants an updated route the user communicates with the BlindDateSystem class. When a message with the GetNewRoute keyword is received, the the referenced sequence diagram "sendNewRouteMessage" is initialized.

UpdateRoute - sequence diagram with high abstraction.

3.10.2 sendNewRouteMessage

Figure 13 shows the details of the referenced sequence diagram "sendNewRouteMessage". After receiving a GetNewRoute message the BlindDateSystem class will ask the ParticipantList for the first event that the user has signed up for. If the user has signed up for an event we get the EventLocation, otherwise we will return an error message to the user (we only want to send route descriptions to people who has signed up for an event). If there is no EventLocation allocated to the event we will also return an error message.

When we have the EventLocation we can access the PositionSystem to get the position of the user, and then use this position together with the time and position of the EventLocation to access the RoutingSystem to get a route description. The route description is sent to the user together with the name and time of the event.

3.11 SendNotification

3.11.1 InitializeNotification

Figure 14 describes the notification process of the system.

Our system requires us to send a route description to all the users who has signed up for an event about one hour before the event takes place. We do not specify who is initializing the notification process. This could for example be a timer or an employee.

If there is no EventLocation assigned to the event, we will send an error message to the initiator. Otherwise we will retrive the position of the EventLocation and the position of the participators. The position of a participator is tracked by the PositionSystem. For each participant we use the RoutingSystem to make a route description, which we send to the participant.

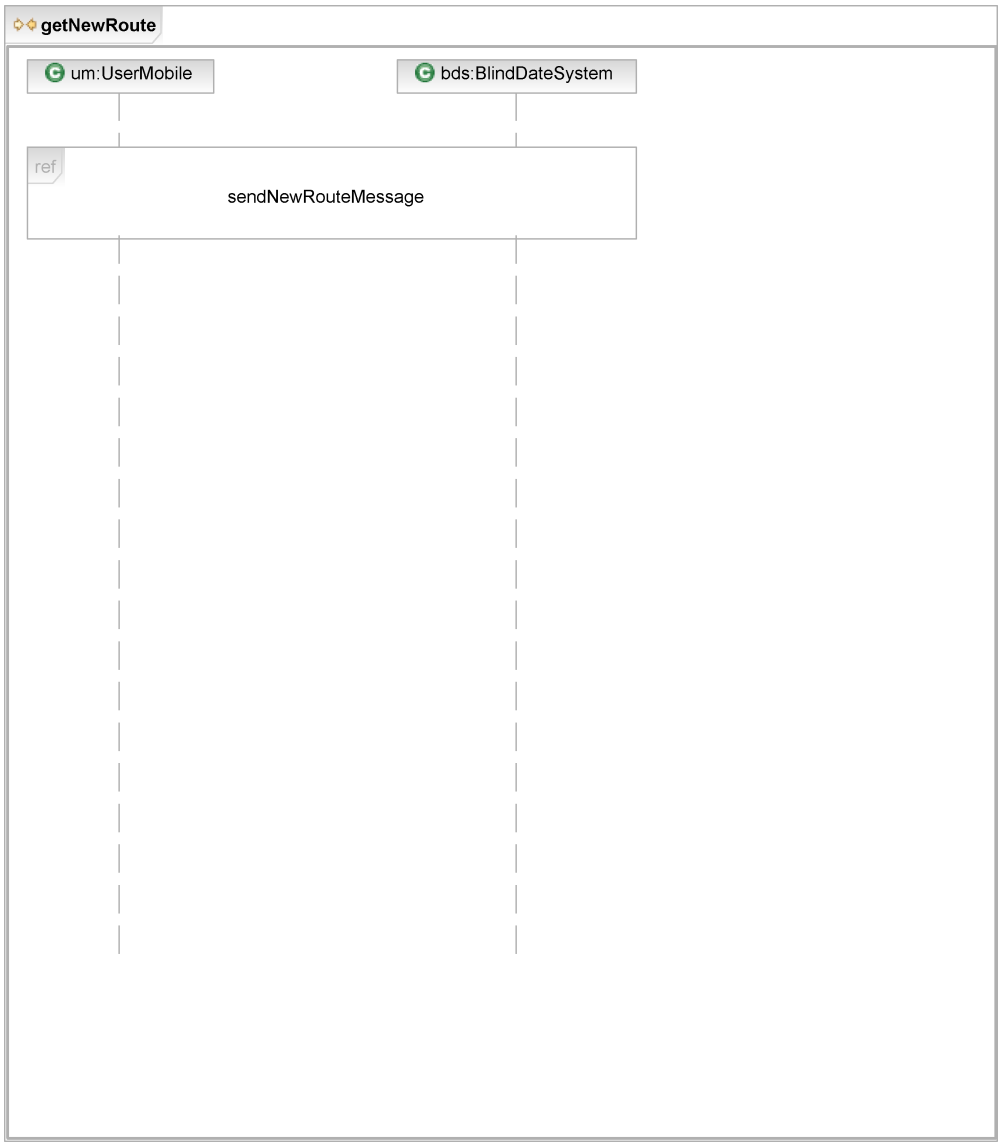

Figure 12: getNewRoute sequence diagram

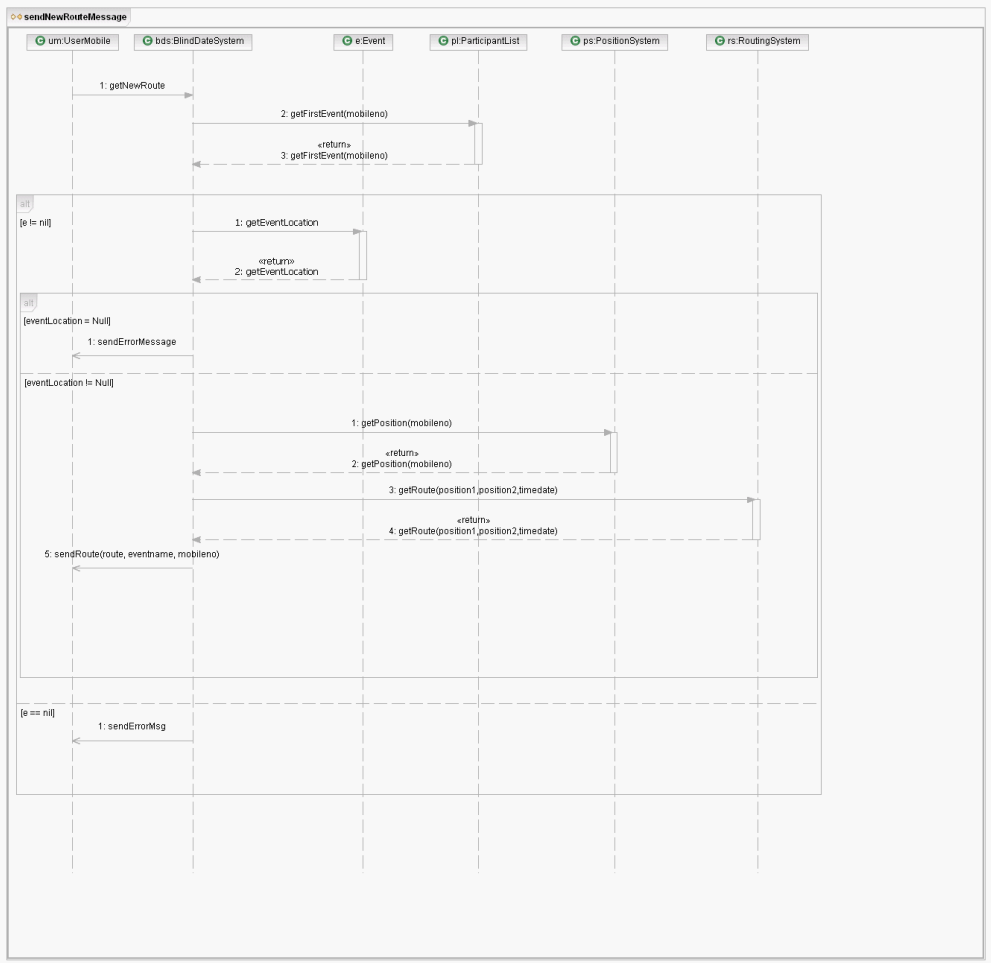

Figure 13: sendNewRouteMessage sequence diagram,

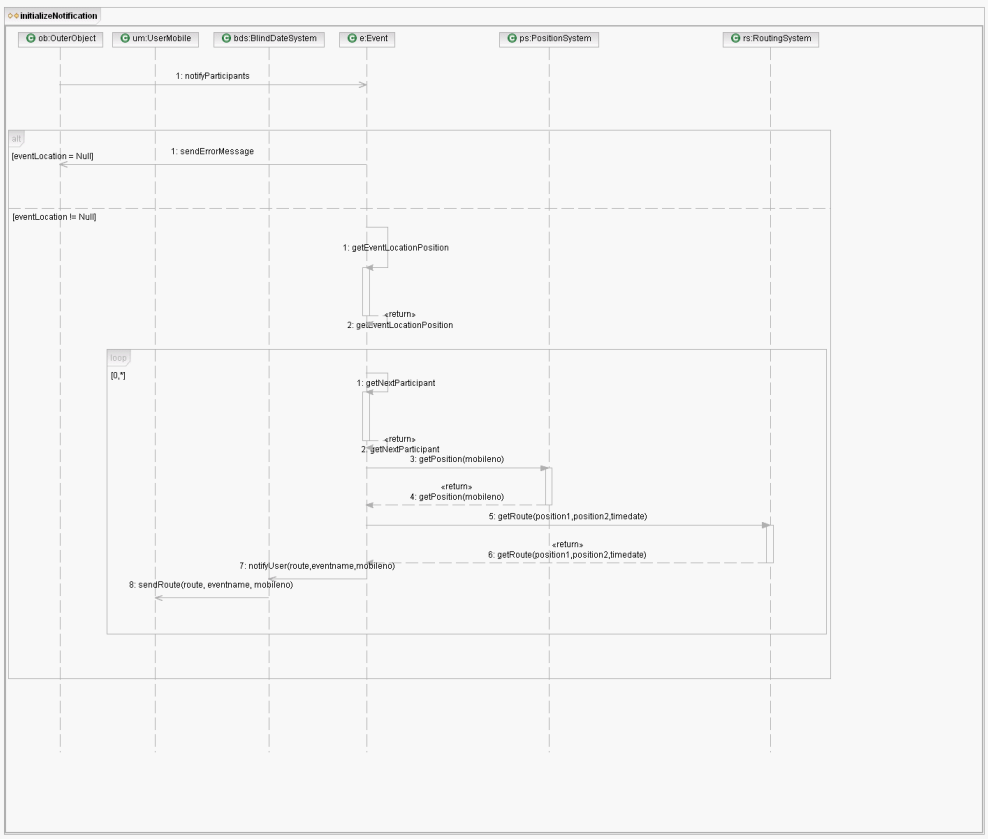

Figure 14: InitializeNotification sequence diagram

4 Alternative solutions to the service

4.1 Time of the event is always NOW+30 minutes

This is a modification of the original join event (figure 15). As in the join event interaction, our system first checks whether the user is a registered participant. After this is done, the list of events is looped through (we assume this list is not empty), finding the next event. In this looping, any event, which takes place in less than 30 minutes, is ignored.

If both an event and participant has been identified, the interaction continues with the AddJoining interaction (see description above).

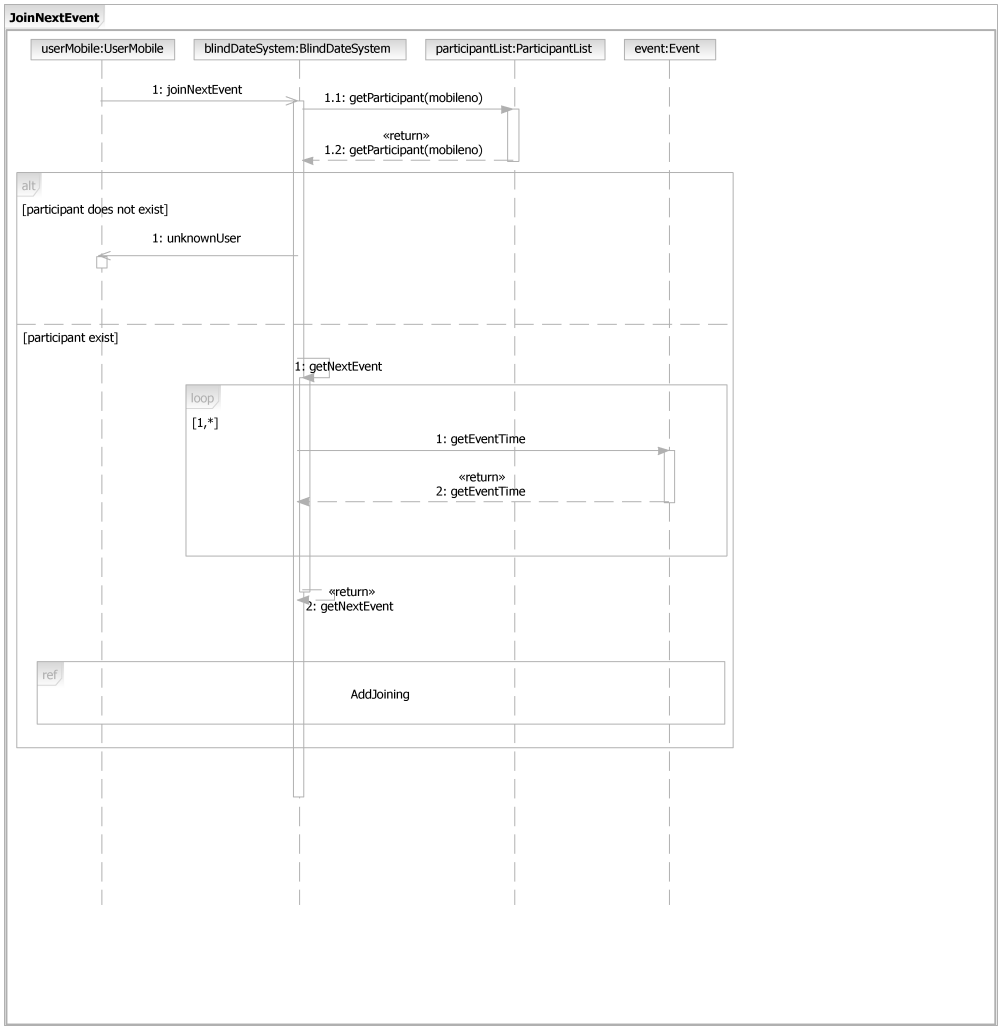

Figure 15: JoinNextEvent sequence diagram

AddJoining is described in figure 9.

4.2 Meeting place is given in the original event description

As one can see that one or more event location(s) in the basic version of our system are already binded to single events, this modification does not have any impacts on the design of our system architecture.

4.3 Optimal meeting place for the participants

Figure 16 shows the notification process for the situation where an optimal meeting place is made up just before the actual notifications are sent to the participants.

4.4 Our own modifications

During the discussion about how to implement the MBDS we noted down some extensions to the system. These extensions are meant to show how the system could be extended beyond the original specification, to serve further needs for both the client and the provider.

4.4.1 Implemented extensions

These extensions are implemented in our main diagrams.

CancelEvent Since the participant is able to join a event, he / she should also be able to cancel the participation. The canceling of a participation is done mainly to give the posibility of canceling an event alltogether by the provider due to low intrest (under required number of participants).

More specifications and diagrams in section 3.9

GetNewRoute Often people are moving around a lot, especial in the city, and therefore we thought it would be a useful extension to be able to request a updated route to the event place. In the original design the participant are positioned at a specific time in ahead of the eventstart, and sent a message containing the eventlocation along with the route description on how to get there by public transportation. Now the user can request an updated traveling route by querying the MBDS, with *GetNewRoute*. the new route is calculated from the position the participant is located at the time of requesting a new route.

More specifications and diagrams in section 3.10

4.4.2 Extensions implemented in own diagrams

This / these extensions are implemented in own diagrams to show how they would be implemented if used.

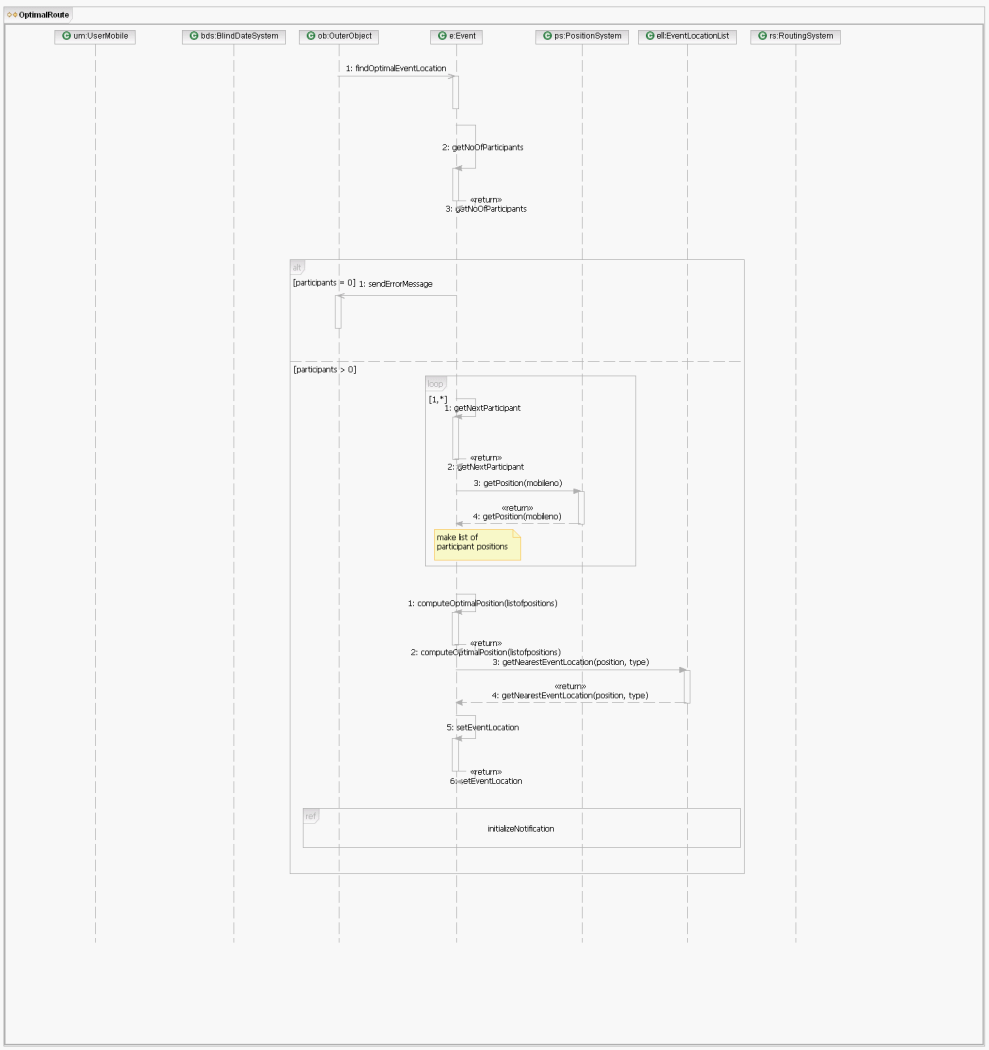

Figure 16: OptimalRoute sequence diagram

GetEvents One likely extension to the system would be that the possible participants would send a SMS to the MBDS to get a list of current events available. This extension would be implemented as a 5.th possible receive message from the participant, and would return a list of events. The list of events is still fixed, but now the flexibility of the generation of events is greatly enhanced.

Figure 17 describes the messaging that takes place in the case of an user wanting a set of offers. The system makes a string of the events that are active in the system and a message is passed back to the user with the information. See figure 18 and figure 19 for updated Activity diagrams, to get the overview for inbound and outbound SMS messages.

4.4.3 Thought about extensions

Possible extensions we came up with during discussions. These is not implemented in any way, just referenced as suggestions for further improvements of the system.

Reserve event location The first thing we came up with was that it should maybe be wise to reserve location for the events taking place. This would then imply that the provider of the service would have a deal with several locations, so that the provider could chose after seeing how the interest is for the event. This extension would not require anything more in the diagrams except maybe another function for *reserveEventPlace()*.

Male / Female attribute in participant Being a blind date service we thought of the extension to specify the sex (male / female) of the participant. This extension would be most suited for particular events, and not "Play bridge at ..." events. The changes / additions to the diagrams would be to save more information about the participant, and have a function trying to match the number of males and females.

Participantprofile In the light of the extension described above it could be implemented a webservice that stored a profile for each participant. This would be suited for participant that participate more than once in this MBDS. Either you could have participants preregister one the web only (identify them with their mobile number), or have both preregistered and unregistered ones. Most likely it would be best to have all preregister, preferably on the web with extended information on the participant. This would probably scare of some potential customers, but then again the ones signing up would be "more commited" to show up.

Participant identification on event location I practical thing for the service would be to have some way of identifying the participants when they arrive at the location for the event. Either by some sign or some specification in the same message containing the location, like "meet at bar" or "meet outside entrance".

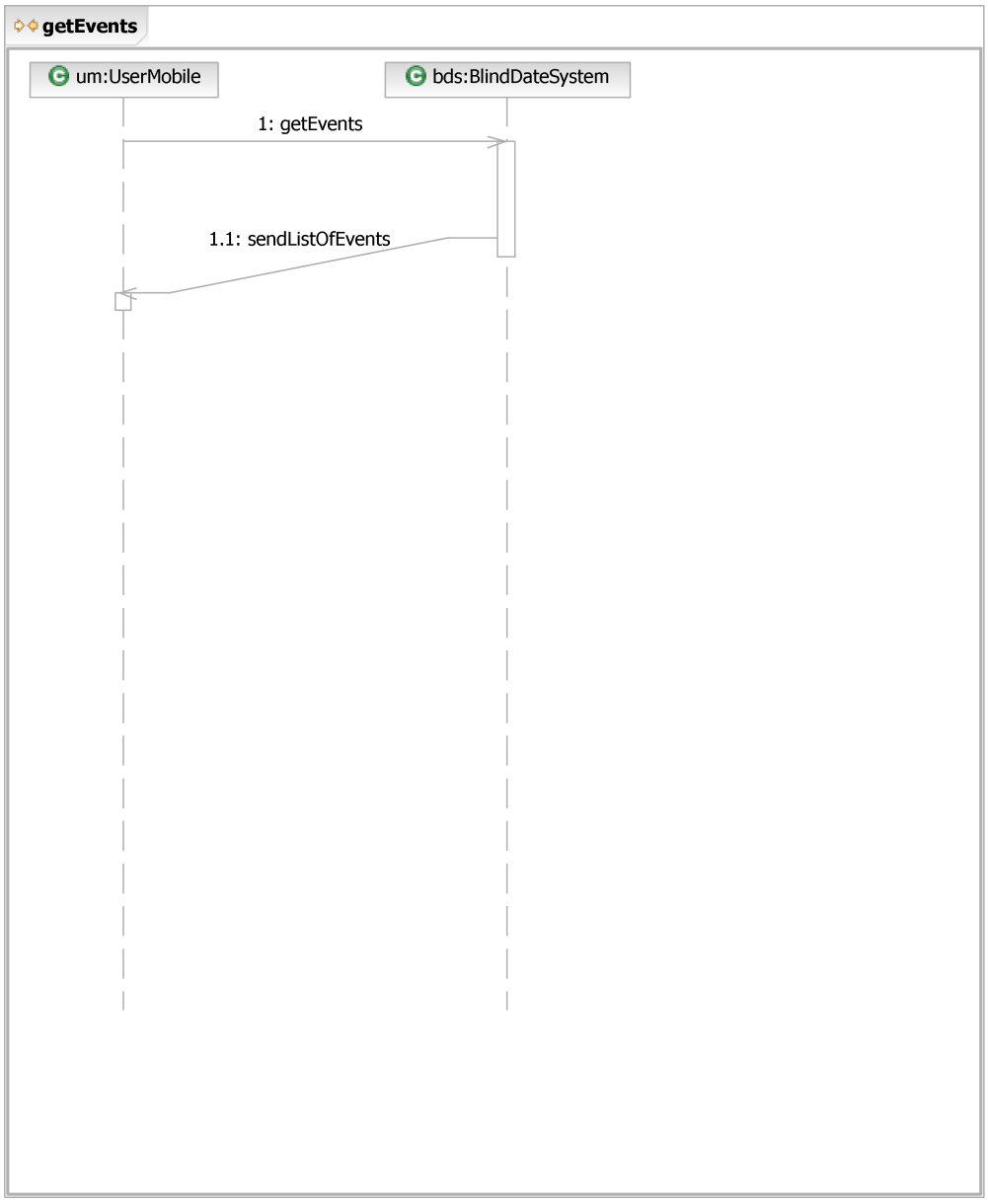

Figure 17: Extension: get list of events from the provider

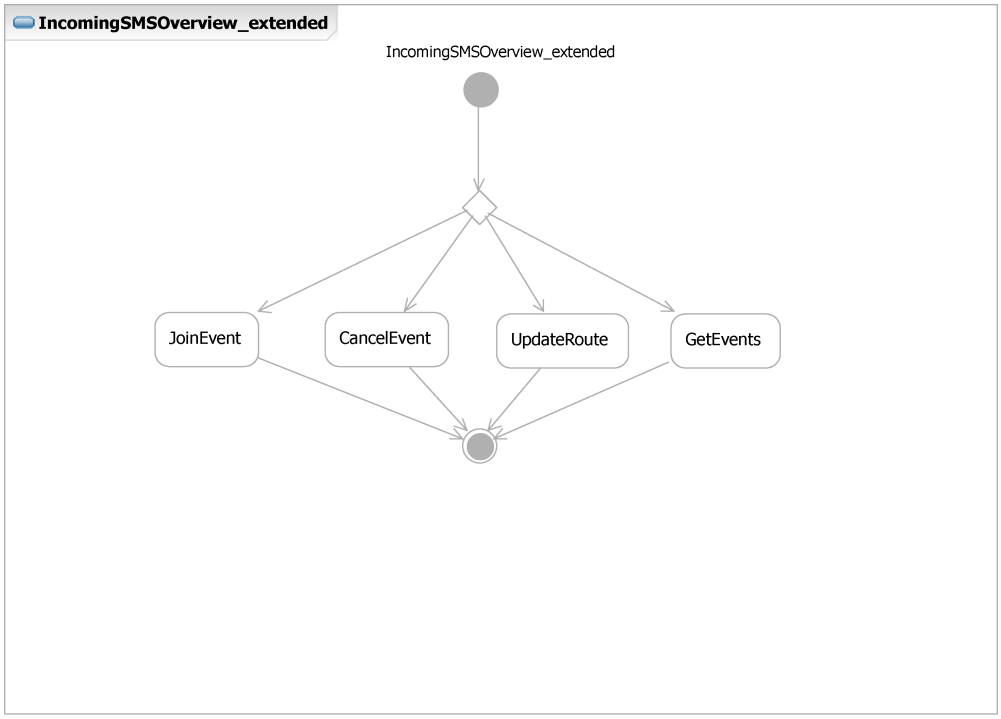

Figure 18: Extension: IncomingSMSOverview activity diagram

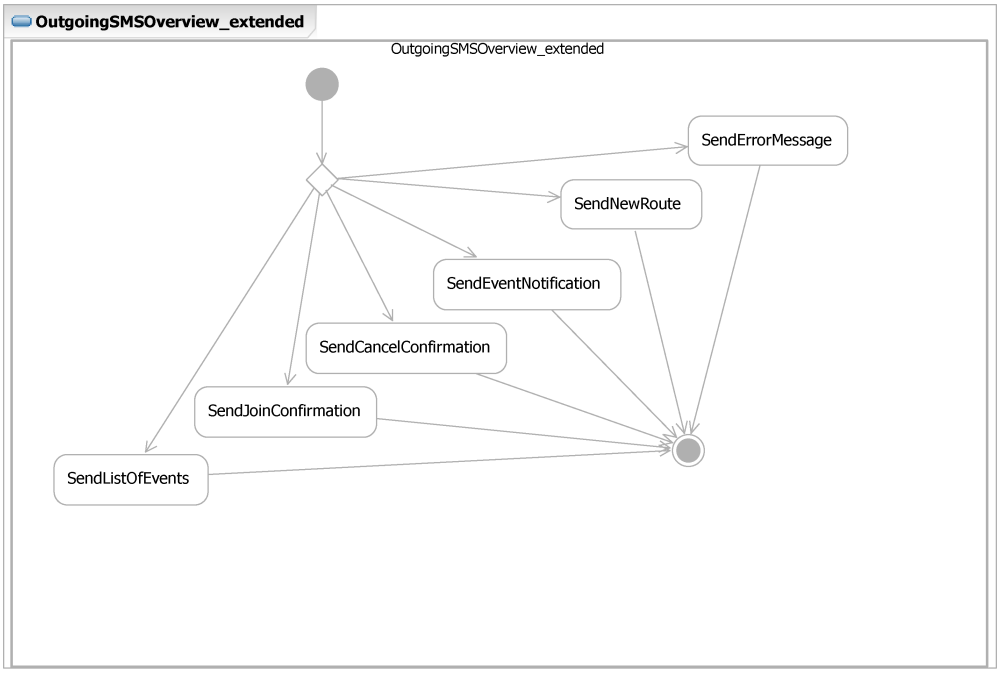

Figure 19: Extension: OutgoingSMSOverview activity diagram

5 Refinement proofs

In this section we will provide proofs that our various modifications can be understood as refinements of our original specification.

Let S_{org} denote our original specification, which consists of the following diagrams: JoinEvent (JE), NotifyParticipants (NP). Since the relation between the JE and NP diagram is defined as an *alt*, we have only one interaction obligation in S_{org}.

Let *pos(diagram)* and *neg(diagram)* denote the positive traces and negative traces of a diagram, respectively. Thus, the semantics $[\![S_{org}]\!]$ of S_{org} will be $\mathbf{[}S_{org}\mathbf{]}= \{(p_{org}, n_{org})\}$, where:

$$p_{org} = pos(JE) \cup pos(NP) \tag{1}$$

$$n_{org} = neg(JE) \cup neg(NP) \tag{2}$$

Since there is no negative traces in the NP diagram $neg(NP) = \emptyset$ and thus $n_{org} = neg(JE)$.

5.1 Event Time Modification

In this subsection we prove that the "NOW+30" modification, which we denote by S_{jne}, is a refinement of the original specification. In this new modification, it is no longer possible to join any other event than the next one. Hence, the old JE diagram is no longer valid, and consequently only describes negative traces. Consequently, both the positive and negative traces of the JE diagram, do now describe negative traces. The new positive traces are described in the JoinNextEvent (JNE) diagram, and the positive traces of the NP diagram is kept.

Hence, the semantics of S_{jne} is given by $\mathbf{[}S_{jne}\mathbf{]}= \{(p_{jne}, n_{jne})\}$ where:

$$p_{jne} = pos(JNE) \cup pos(NP) \tag{3}$$

$$n_{jne} = neg(JNE) \cup neg(NP) \cup neg(JE) \cup pos(JE) \tag{4}$$

By the definition of interaction obligation $(p_{org}, n_{org}) \rightsquigarrow (p_{jne}, n_{jne})$ iff

1. $n_{org} \subseteq n_{jne}$ and
2. $p_{org} \subseteq p_{jne} \cup n_{jne}$.

Since $n_{jne} = neg(JNE) \cup neg(NP) \cup pos(JE) \cup neg(JE) = neg(JNE) \cup pos(JE) \cup n_{org}$, requirement 1 is fulfilled.

Since $p_{org} = pos(JE) \cup pos(NP)$, and both $pos(JE) \subseteq n_{nje}$ and $pos(NP) \subseteq p_{nje}$, it follows that $p_{org} = pos(JE) \cup pos(NP) \subseteq n_{nje} \cup p_{nje}$. This establishes the fulfilment of requirement 2.

Since both requirement 1 and 2 are fulfilled, we have that $(p_{org}, n_{org}) \rightsquigarrow (p_{jne}, n_{jne})$ and consequently that S_{jne} is a refinement of S_{org}.

In informal terms, we see that S_{jne} is a refinement of S_{org}, since it is obtained from S_{org} by defining all traces of JE as negative (narrowing) and adding the traces from diagram JNE (supplementing).

Strictly speaking, by negating the old JE diagram we have introduced the empty trace as a positive one. In the above proof we have ignored this fact and consequently we have not proved refinement in the most strict sense of the term.

5.2 Static Meeting Place

This modification is already implemented in our original specification. Hence, refinement is trivially proved in this case, since a specification is a refinement of itself.

5.3 Optimal Meeting Place

In this subsection we analyze the optimal route optimization. Let S_{opt} denote the specification of this modification.

Since we keep the old option of manual event location decisions, S_{opt} is obtained from S_{org} by adding the description of the OptimalRoute (OR) diagram (using the alt operator). Since we use the alt operator when adding the new diagram, we still have only one interaction obligation. Hence, since there is no negative traces in the OR diagram, we only add positive traces to our original specification (the positive traces of OR). Adding positive traces is supplementing, i.e. S_{opt} is obtained from S_{org} by supplementing.

Consequently, S_{opt} is a property refinement of S_{org}.

5.4 Own Modifications

All our own modifications are simple adding of functionality. Since none of these modifications define any negative traces, we only add positive traces to our original specification, i.e. former inconclusive traces are now defined as positive (supplementing). We still have only one interaction obligation, since the new diagrams are related to the original ones by the alt operator.

This shows that our specification of our own modifications are obtained from the original specification by supplementing, which is property refinement.