A prototype for defining and enforcing privacy policies

Prosjekt rapport 985

Kristin Skeide Fuglerud
Joachim Lous
Bjørn Nordlund

September 2002

Norsk Regnesentral
ANVENDT DATAFORSKNING

Rapport/Report

Tittel/Title:
A prototype for defining and enforcing privacy policies

Dato/Date: 02.09
År/Year: 2002
ISBN: 82-539-0494-0
Publikasjonsnr.:
Publication no.: 985

Forfatter/Author:
Kristin Skeide Fuglerud
Joachim Lous
Bjørn Nordlund.

Sammendrag/Abstract:

Knowledge of the location of a person's mobile phone can be used by service providers to tailor better services for their customers, and can generate new business opportunities. To make use of this information, the privacy of the users needs to be enforced. Information about a person's location is private information, and a key question is: "Who should have access to what location information under which circumstances?"

This report describes the development of a prototype for defining, administrate and enforcing a privacy policy for the location information generated from tracking a person's mobile phone. The policy will be used to decide who should have access to this location information.

The development was done using an agile software development methodology and a genetic algorithm for user interface design. These methods are further described in this document.

The resulting prototype is described, and the report ends with some proposals for further work.

Emneord/Keywords: Privacy, location based services, mobile, privacy enhancing technology, extreme programming XP, user interface design

Tilgjengelighet/Availability: Åpen/Open

Prosjektnr./Project no.: 802020

Norsk Regnesentral / Norwegian Computing Center
Gaustadalléen 23, Postboks 114 Blindern, 0314 Oslo, Norway
Telefon 22 85 25 00, telefax 22 69 76 60

Satsningsfelt/Research field: Personalisering og Personvern

Antall sider/No. of pages: 16

1 Content

2 Introduction

This report describes the development of a prototype in the research program mininfo (www.mininfo.no) at Norwegian Computing Center. The prototype intends to be used to conduct user studies, and further development can be done based on input from the user studies and other research conducted in the mininfo program.

The main purpose of this report is to give a state report of what has been done, and address some issues that needs further work. People that want to study or develop a system for defining and enforcing privacy policies, or join the research program mininfo, either as researcher or a partner could have use of reading this report.

3 Background

While mobile services are emerging, location based services (LBS) have long been on the doorsteps without breaking trough. A lack of a "killer application" or a critical mass can be among the reasons these services is not breaking trough. However, according to [1] there is an increasing concern among citizens about how their personal data is being used, and this is probably a major reason for the slow take up of these services. Therefore there is a need for technology that can provide privacy protection and at the same time allow for personalized electronic services. The user needs a simple way to decide and control *who* should be allowed to access *which parts* of their data or applications, as well as for instance *when*, *how often* and in *which way*.

One way of meeting these challenges could be to develop a system for defining, administering and enforcing privacy policies. Through a privacy policy the individual will be able to define in advance who will be given access to his/her location information, when, and to what accuracy. Snekkenes has in [2] proposed a framework for defining, distributing and enforcing privacy policies.

In this project we wanted to study the concept of user defined privacy policies further. Therefore we have developed a prototype system for handling access policies for personal location information. The prototype has been developed as a part of a research program funded by the Norwegian research council. The prototype is based on the proposal by Snekkenes [2]. The system is developed specially for controlling location information from a mobile phone, but the intention is to extend the system further to handle more general information units. The system can be described as Privacy Enhancing Technology (PET), or maybe we should call it Privacy Preference Technology (PPT), as the focus is on supporting the user in stating and enforcing his/her own privacy preferences.

In general users have high expectations to usability of new electronic services. From the user's point of view, the important part is the user interface (UI), which must provide a simple, easy-to-understand way to define an access control policy. Through the prototype we will be able to explore users' perception and attitude to the privacy policy concept and the system's ability to pass on the underlying ideas. Different aspects of the UI, such as use of language and images, may seriously affect the user interpretation of the technology and how it works. One of the great challenges in developing this UI will be to enable a user with little or no training in logic to define a policy that properly reflects the user's intentions. Having the

prototype we can run experiments investigating actual user attitudes and behavior regarding privacy of LBS.

3.1 The vision of the privacy policy application

We want to build an application for defining and enforcing privacy policies; we will call the application for the privacy policy application (PPA). The vision is to make it possible to utilize location information from personal mobile phones while at the same time giving the persons owning the mobile phones control of their own location information.

The PPA aims to be the industry standard for such systems characterized by these terms:

- **Effective:** The architecture should be scalable and provide the best possible throughput and least possible delay.
- **Usable:** The user interfaces should be easy to understand while at the same time provide enough functionality to define a concise and clear policy both understandable for the user and for the location provider using the policy to grant access to information.
- **Secure:** We need to have a highly secure system when handling personal information such as location information. The security requirements to the PPA system are no less that to transaction systems handling economic or medical information.
- **Neutral:** The PPA should be neutral when it comes to which service providers it handles.
- **Robust:** A service like this in use must have a 100% uptime. Although our prototype is deployed in a test environment without such robustness, it should be no problem deploying it in an environment like this without changes in the PPA.
- **Logical:** The PPA should be logical to understand by developers and service providers with some experience with system design, and new developers should come up to a productive level within short time studying the PPA. Especially should the contract models for client systems interacting with the custodian be easy to understand.

3.2 Architectural overview

The overall architectural model is based on [2], which describes a possible sequence of interactions for a request for a service based on location data. The model is outlined in **figure 1** and includes the following entities:

- **Personal Location Privacy Policy:** Statement of what can be released to whom and when. Each located object will have an associated policy.
- **Policy custodian directory:** Public directory of Policy Custodians.
- **Policy custodian:** Where the policy is stored and possibly also enforced. The set of permitted operations may include read, write, modify, and query etc. depending on the identity of the requesting entity.
- **Location provider:** Entity providing the location data. Any release of data should be subject to the policy.

- **Service provider:** Entity that is combining location data with other data to produce some service. An example of a service provider could be a company using information about a person's location to generate a graphical map of the area the user is in, with a red spot indicating where the user is.
- **Service consumer:** Entity to which services is presented for consumption. An example can be a taxi driver wanting to know where to pick up a passenger.
- **Service initiator:** Entity that would like and/or accept that the service is produced. An example of this can be a person wanting a taxi and accepts that the taxi driver gets location information about him.
- **Service requestor:** Entity that makes a request to the service provider for the service to be produced. An example of this can be the same as above, a person wanting a taxi, thus requests the service provider to give the taxi location data.
- **Located object:** The entity whose location data will be required to deliver the service.
- **Owner of located object:** The entity that owns the located object

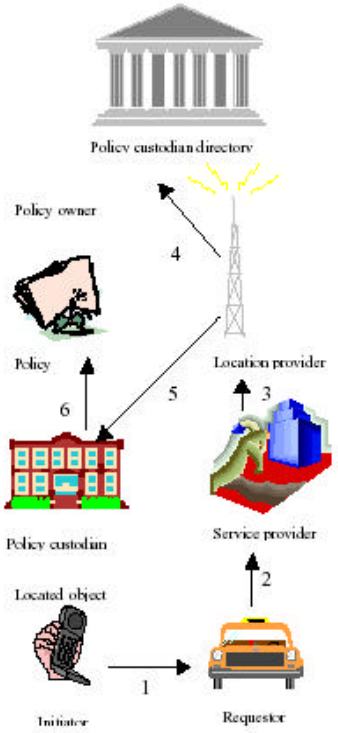

Figure 1. Service request: Sequence of interactions. From [2]

3.3 Limitations of application

The prototype described in this report is an implementation of the policy custodian in **figure 1** with interaction interfaces 5 and 6 in the described model. The policy owners can access the custodian with different clients to define and manage their policy, interaction-step 6 in the figure, and the policy is exposed for the location provider, interaction-step 5 in the figure. For simplicity (the) service provider, service initiator and service requestor are being handled as one entity, *the observer* in the prototype. The prototype can be further developed to handle all the entities.

4 Methodology

In this chapter we will shortly describe the different methods we have used in the process to build the prototype.

4.1 eXtreme Programming

Extreme programming (XP) [3-5] is a member of a family of software development methodologies called agile ("lightweight") methodologies [6].

XP is a "customer centric" method. Optimally one should have one representing the customer "on site" during development. In our case our customer is the Norwegian research council funding the program, but they are relatively far away from the type of customers the XP method is designed for.

We wanted to use the XP method because of its adapting nature. We believe that a good system design must come as a result from iterative development where the system design continually must adapt to new requirements and design ideas. Except for the part about the customer, XP seemed like a good fit for this project. XP emphasizes that a project should use the part of XP that works, and not necessarily use all parts in every project.

The lifecycle of XP is described in the picture below (**Figure 2**).

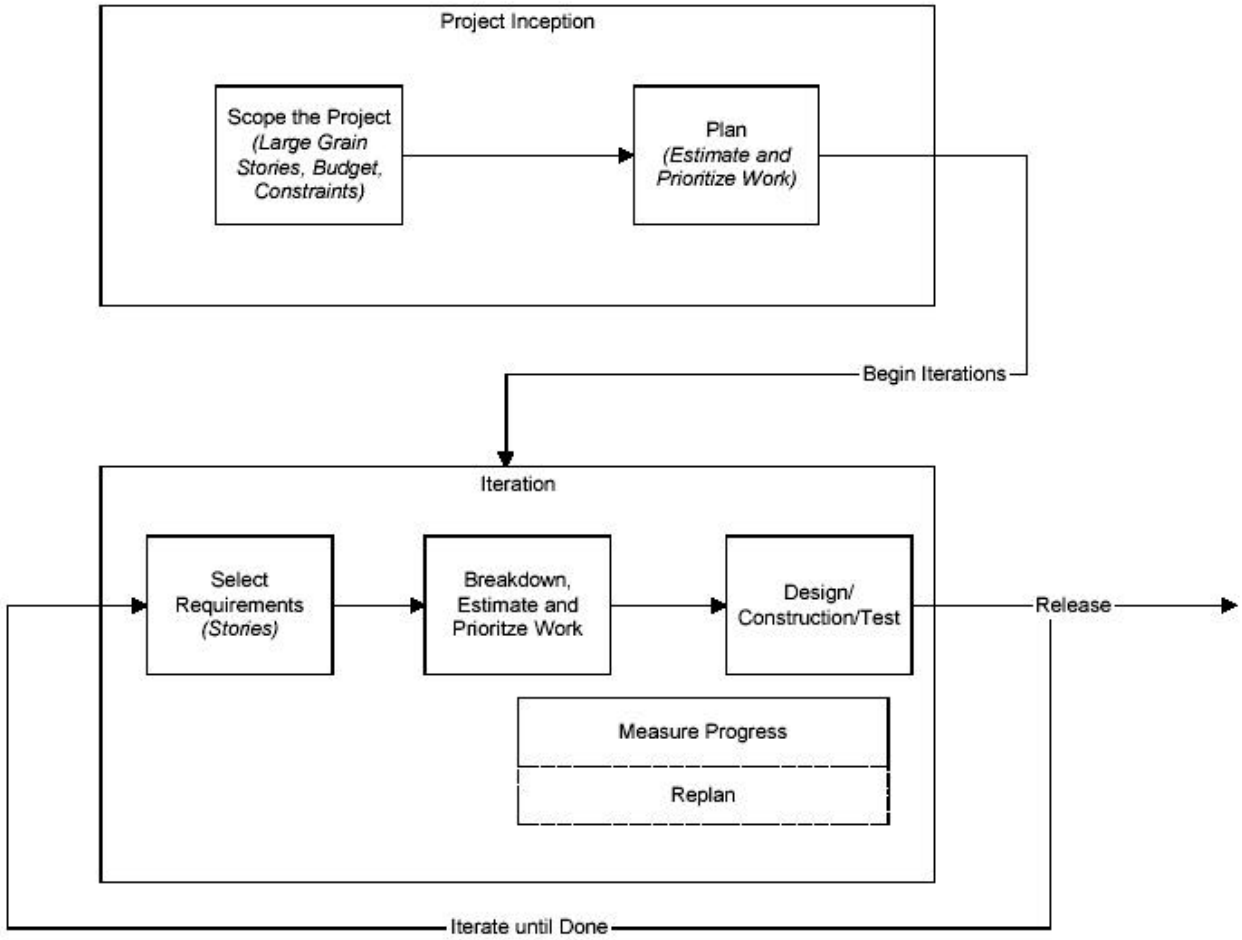

Figure 2. The XP lifecycle

First the project team members create user stories. User stories are written down on an index card, and should be a few lines describing some aspects of the system. This could be a functional requirement, a user experience, a system requirement or a limitation of the system. Then the project team meets in a release-planning meeting and selects which user story to implement in the current iteration. The customers should do the selection after the developers have provided some estimates of the cost.

Next the developers meet in an iteration-planning meeting and develop task cards. Task cards are based on the user stories. These are written on index card and should describe a particular programming task. The programming task should ideally take from 3 to 5 ideal hours to finish. That is if the programmer did nothing else but coding. Our experience is that you often have to multiply this by two to get a realistic measurement on how long it takes, but that is not the important aspect.

Then the iteration starts. Design is done by the rule of simplicity. Choose the simplest possible design that solves the current problem. Every programmer can change the design, but there is some quality assurance. Pairs of programmers should do all development. When two programmers sit on the same computer it is called pair programming. The idea is that any two programmers in the projects can take decisions

concerning the whole system architecture. This way efficient refractory of the classes can take place. The result should be better system design.

When designing for the problems at hand new design needs to be done when new problems arise. XP solves this by refractory. When starting to code so early in the development process one has to be open for changes. Refractory of code means that you move parts of the code to another place in the program, for example you move a function from one class to another. Good designed programs can easily be refractored, and refractored programs have usually good designs.

When a program is being refractored there is a need to test if the program behaves in the same way as before the refractory. XP emphasizes unit testing. Unit testing is tests of small units of the system; typically a unit is a function in the program, or part of a function like an algorithm.

Before each iteration the customer defines some acceptance tests the system should meet. The customer specifies scenarios to test when a user story has been correctly implemented. A story can have one or many acceptance tests, what ever it takes to ensure the functionality works. Iteration is finished when the acceptance tests are passed. Then a new iteration can begin.

Clearly we could not say we used XP in this project because the customer has such a central role in a XP project. But we did base the development process on some of XP's best practices, and we think it gave good results.

4.2 Using a Genetic Algorithm (GA) in user interface design

Since the concept of privacy policy is fairly new and unknown to most people, we wanted to explore several different design ideas, and since we were following the ideas of XP and quick prototyping, we did not have time for long explorations and evaluations. Therefore we selected a parallel design process based on a genetic algorithm as presented by McGrew in [7]. An important advantage with this method is that it provides an effective way of exploring and evaluating several design concepts virtually at the same time. Also the process is open for all team members, and thus supports assimilation of knowledge from the whole team including several disciplines into the design.

The main ideas of the process are to make new versions or "generations" of design proposals based on combining the best elements of the initial design ideas i.e. "Survival of the fittest design elements". The grain of the designs when following this method is adjustable. We chose a fairly high granularity level approach, and left the details to the implementers. The method takes into account that there might be good design elements in bad or unattractive design proposals. Also the process continually allows for new ideas by a concept of mutations.

4.2.1 Description of the process:

Step 0: Generate an initial population of design proposals
Step 1: Natural selection (possible via a fitness function)
Step 2: Generate next generation using following breeding mechanisms:

- cross over: select elements that you like from two of the proposed solutions and combine them in a new proposal
- mutation: put in any new idea that you get during the process

Continue to iterate step 1,2 until convergence. Since each participant is able to make intelligent decisions about each design proposal and elements in it, the design proposals usually converge fast (McGrew [7] suggests 3-5 generations).

From an academic point of view the main problem with this method probably is the process of selecting what design elements to "breed" upon. In [7] it is proposed to evaluate the different designs according to some usability criteria such as heuristics or usability guidelines that is applicable to the design problem at hand. This is called a fitness function.

In our case the selection was done very simply. Each participant presented his/her design, and all the participants discussed pros and cons. After this it was up to each member of the design team to select the design s/he liked most.

5 Requirements

Part of the purpose of the prototype is to uncover requirements and challenges, so there was no requirements document made at the beginning of the project. This is in keeping with the XP philosophy: requirements will *always* change, so generating requirements and adapting to new ones are a part of the development process itself. The requirements have the form of user stories, which are transformed into programming tasks and then acceptance tests as the project progresses.

In retrospect though, the general trend that emerged from the user stories produced was to aim for

- A demonstrator that would implement a simple custodian service and
- A personal policy editor client.

Regarding the custodian, more focus was placed on illustrating the general structure and end-to-end propagation chain for policy data, than on supporting realistically detailed data in the actual policies or integrating with external sources of identity info. For the user client, the focus was mainly on presenting the quite complex data in the policy to the user in an accessible way, using relatively simple and familiar GUI elements.

6 Design

6.1 A typical user walkthrough

First a new user will need to register. S/he chooses a custodian s/he trusts, and register with a username and a password. S/he then downloads the windows client and installs it on his machine (This will not be necessary when we have a web version of the client ready).

When s/he logs in with the username and password a default policy will be loaded. This policy has some typical observer groups, situations and visibilities already present, and the user can choose to use some of them, or change them according to his/her needs.

The user can then start to add elements to build the policy. A policy consists of the four elements Located object, Observer, Situation and Visibility bound together by a rule. Literally one can express it as *for a located object 'Located object', an observer 'Observer' can se my location in a situation 'Situation' with visibility 'visibility'*.

We address the need for dynamically update the policy by the possibility to change the 'situation'. By using the situation editor one can switch between different situations to change what and to whom location is being exposed.

6.2 High level design

The general design follows a classic multi-tier model: an SQL database for persistence, an object layer implementing the fundamental business logic, a webservice implementing higher-level functions and exposing them through the external SOAP API, and finally various types of clients for actually presenting and manipulating data in different ways (figure 3).

The two client actually realized are a relatively simple Windows-based policy editor/viewer, and a small WAP service for switching between predefined 'contexts' or 'situations' from a mobile phone, affecting which part of the policy is currently to be enforced. Both are directed at end-users.

No client was made for business users to query users' policies, since such clients will usually be integration layers towards location-providers' internal systems, not separate human-operated applications.

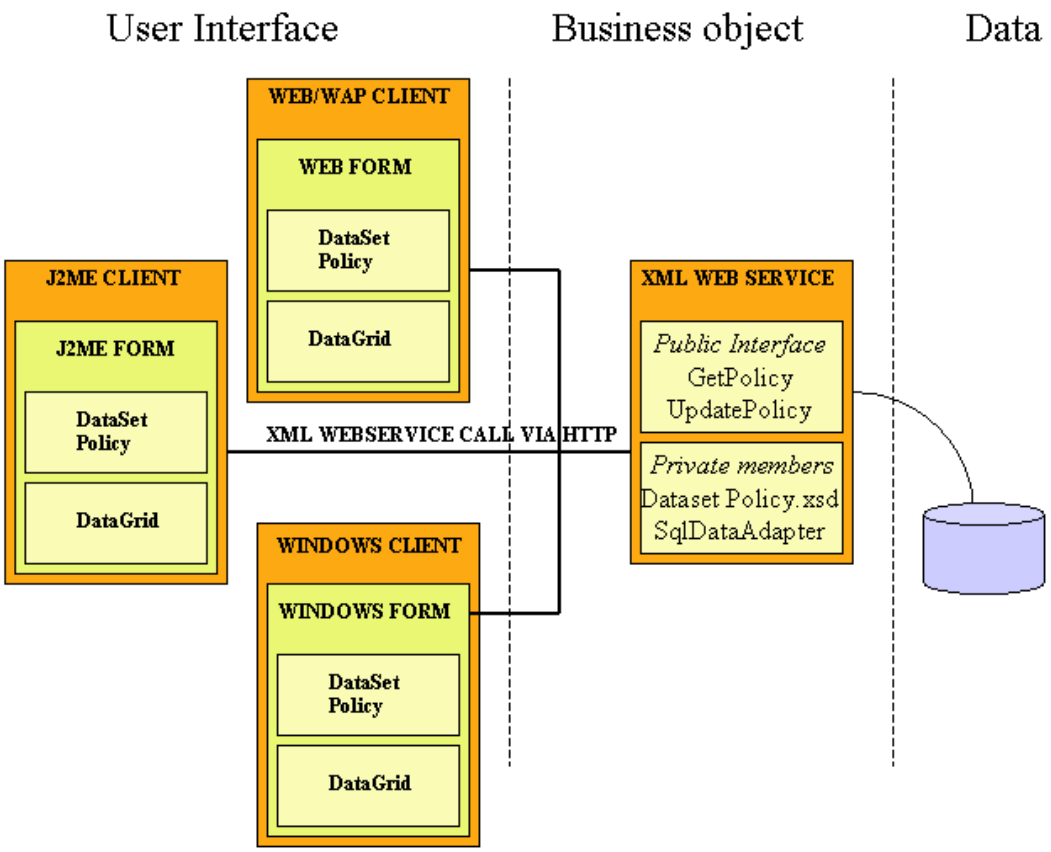

Figure 3. Multi-tier model

6.3 Low level design

The policy model is centered round the concept of using a set of rules to select an allowable visibility level for a given information type, depending on a set of circumstances.

In short, finding out what information an observer is allowed corresponds to:

```
visibility = f(infoOwner, infoType, requesterRole, ownerSituation)
```

where all values are enumerations, some of them wholly or partly user-defined. A separate database table of rules defines which tuples will result in which visibility level. A rule has no other function than to couple together named entities, defining how a specific tuple will result in a specific visibility.

- **Visibility:** A database table defines what a restrictions a named visibility level actually implies. The current table definition only allows a small demonstration set of properties for a visibility level, but this can change without altering the general lookup concept.

- **InfoOwner:** 'Owner' in the database. The policy owner must of course be identified to select which rules apply.

- **InfoType:** 'LocatedObject' in the database. The user may have different restrictions on different information, so what is being requested must be part of the rule. So far we have only been concerned with location data, so the InfoType is currently only used to select which of the user's locatable objects is being queried. This is reflected in the current naming of the database table.

- **RequesterRole:** Who is asking – the information requester must match one of a predefined set of Roles. Roles can be user defined, system-wide, or provided by an external service. How requesters are identified with and authenticated as a particular role is outside the scope of this demonstrator. Roles have a "parent" Role, so one can build hierarchies where roles inherit privileges, and only exceptions needs to be specified.

- **OwnerSituation:** What 'context' or 'situation' is the user in. Again, this is defined by selection from a user-editable list of named alternatives. This facility encompasses all other dependencies for information disclosure. One typical use is to allow a user to quickly switch policies by manually selecting a context, for example using a trivial WAP client from a mobile phone. Typical alternatives would be things like hidden, or *at work*, in effect activating one of several complete policies. But one can also see this facility as an enabler for automated context-dependency. For example, an external service can be authorized to change your context for you automatically, in response to your position or calendar or other information, according to rules you have specified. This way, the details of implementing dynamic policies is farmed out of the core rule system, making for less computation in the custodian service, and creating a new niche for value added services.

7 Results

In chapter two we presented a conceptual solution for defining and enforcing privacy policies. We have implemented software that can be run by the policy custodian (the custodian service) illustrated in **figure 1**. Also we have made the interaction layer to the system for the location provider, interaction step 5 in the figure. And we have made two clients for the policy owner to manage their policy, interaction step 6 in the figure. There is one client to define the policy (the admin client), and another client to update the dynamical part of the policy (the situation editor).

7.1 The Custodian Service

The policy custodian runs the custodian service. This is implemented as a web service in the PPA. The location providers must communicate with the custodian service trough the web service to retrieve policies tailing witch information they can give away to service providers.

The main methods available for the location providers are lookup-functions, returning a visibility to enforce based on provided parameters and the information owner's context (known already by the custodian and not disclosed). The set of functions for this use is not yet complete.

The owners of the location information need to communicate with the custodian service trough the webservice to administrate the policy.

The methods available for the owner clients allow adding and editing located objects, roles, situations, and visibilities, as well as rules for tying them together.

The exposed webservice API from the service can be obtained from the webservice itself as a WSDL (Web Service Description Language) file.

7.2 The Admin Client

The admin interface is where the user defines the policy. The admin interface can potentially be implemented on any platform as long as it supports the webservices standard. We have chosen to develop the client for the windows platform, since the rest of the development has been done on this platform in this project, and thus the developers is most familiar with it. There is a screenshot of the admin tool in **figure 4.**

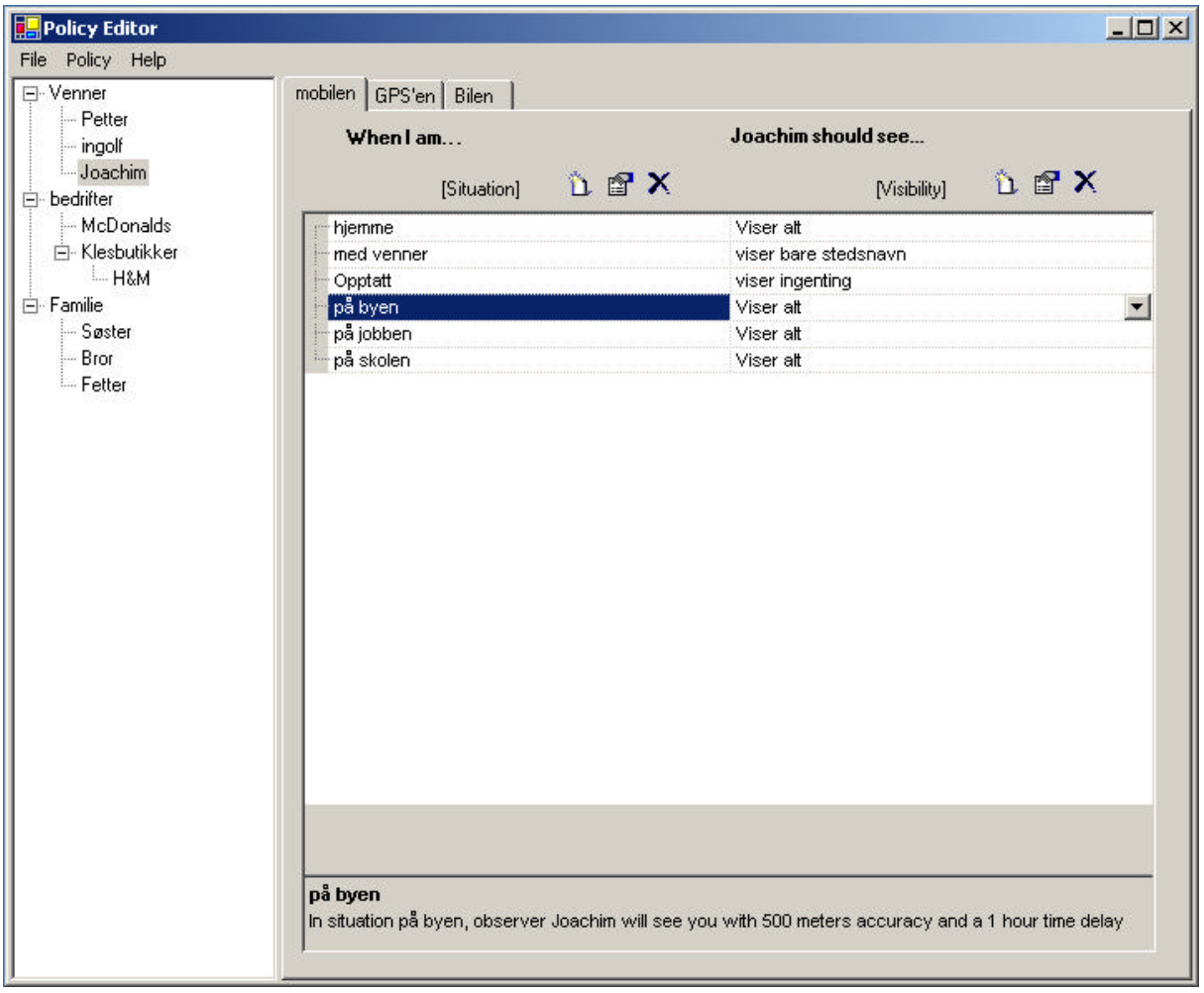

Figure 4. Policy editor

The policy is defined to be dynamic based on the users situation. When the users situation changes, the rule for what observers should see also changes. The user currently has to manually update his situation via the

situation editor described in the next section. Different software agents could also update the situation automatically.

7.3 The Situation Editor

The situation editor is used to update the situation the user is in from a set of predefined situations. Example on a situation is 'at work'. When the user sets his situation to 'at work' this could mean that colleagues are granted access to his location. When he is no longer in that situation, say when he is 'home' he probably doesn't want his colleagues to see his location. The situation editor for WAP is presented in **figure 5**. The situation editor is primarily made to show the concept and needs further development to work properly.

Figure 5

8 Other considerations, further work

We have only implemented the policy custodian part of the system described in figure 1. Our main focus has been on developing the user interface and the practical consideration of making the policy available to the other stakeholders in the system. First we will mention some issues regarding the different parts of a policy and extensions to the current prototype and then we will mention some more general aspects of the system. The motivation for adding this chapter is to serve as input to further research around the prototype.

8.1 Extensions of the current policy

As stated earlier, the policy consists of rules binding entities together. One rule is a unique tuple of the entities owner, located object, observer, situation and visibility. This is a simple model leaving out many of the difficult aspects. Although the scope of this prototype was to make just this simple model, we want to mention some of the difficult parts here. There is specially the tree entities observer, situation and visibility that we believe needs to be considered more closely.

- **Observer:** Observer needs further analysis. In the prototype, observer is only a name. This name has to be mapped to some real world identity, like for example an e-mail address, phone number, social security number or organization number. Observer could also be divided into different categories. For example a friend is different from a business, and should be handled different by the PPA. And as presented in [2], the initiator, the requester and the service provider can be different entities, and this should possibly be reflected in our model.

- **Situation:** In our prototype the owner manually selects the situation. For the policy situation is just another identified identity, but mapping the situation to that identity can be a complex system working on its own, collecting data from multiple sources like location, calendar, pc-usage, active smart cards, and calculate the situation using complex algorithms.

- **Visibility:** In this prototype the visibility is a name with some parameters, like the level of granularity for the accuracy of the location information. The thought is that to some observers you want to give away your location information, but with a much lower accuracy than to others. For example one could give away only the name of the city. Different parameters for this entity should be investigated further.

Other more general aspects of the PPA have also been trough some initial thoughts during the project and we will mention some of them in the rest of this chapter.

8.2 General aspects

8.2.1 User Interface

More work on the user interface has to be done. In a pilot test we discovered, not surprisingly, that the concept of making a policy might be difficult or confusing. Particularly the pilot test indicated that we have to look further into how to distinguish and clarify the difference between making a set of privacy rules for a situation, and selecting a set of rules when in a specific situation. The prototype will be further tested in a user test. The result from this test will certainly lead to some changes, and a new user test should be conducted again.

8.2.2 Security

We considered these alternative security mechanisms during development:

1. Basic: the username and password are sent in plain text. Easy network monitoring tools can intercept usernames and passwords.

2. Basic over SSL: Username and password is sent over the net using SSL encryption. This works well for Internet scenarios. Using SSL degrades performance.
3. Client Certificates: Use of secure identification of clients in Internet scenarios. Requires each client to obtain a certificate from a trusted CA. Certificates can be mapped to user accounts witch are used by IIS for authorizing access to the webservice.

We chose the first option with plain text transmission of username and password. This is clearly not good enough security for this system and should be further developed in the next versions of the PPA.

8.2.3 Legal and ethical aspects

As alleged in the introduction of this document, the PPA can be described as Privacy Enhancing Technology (PET). But as is pointed out in [8], even if PET's provide individuals with a means of controlling their personal information, these tools do not necessarily ensure privacy protection. Therefore a simplified view of privacy, understood mainly in terms of control over personal information, is criticized. It is important that ethical and juridical matters are studied further and these must be analyzed in conjunction with the development of new technology.

8.2.4 Generalization

The prototype is designed for location information generated from mobile phones. We believe that the policy could be modernized to work with other types of information as well. One part of the custodian prototype that needs to be changed to support other types of information is the visibility. When for example the radius seems like a good visibility parameter fro location data, it gives little meaning when talking about restaurant preferences.

9 More information

This report and additional software and information will be available at the mininfo website: http://www.mininfo.no

For further information about the prototype, contact Bjørn Nordlund (bjornno@nr.no) or Joachim Lous (Joachim@nr.no).

10 References

1. Bernard Clements, et al., *Future Bottlenecks in the Information Society: Report to the European Parliament, Commitee on Industry, External Trade, Research and Energy (ITRE).* 2001, Institute for prospective technological studies. Research teams from JRC and ESTO: Seville. p. 196.

2. Einar Snekkenes. *Consepts for Personal Location Privacy Policies*. in *EC'01*. 2001. Tampa, Florida, USA: ACM|.

3. Kent Beck and Martin Fowler, *Planning Extreme Programming*. 2001: Addison-Wesley, ISBN: 0-201-71091-9.

4. James Newkirk and Robert C. Martin, *Extreme Programming in Practice*. 2001: Addison-Wesley, ISBN: 0-201-70937-6.

5. *Extreme Programming: A gentle introduction* [online] Don Wells. http://www.extremeprogramming.org. Dated: August 17, 2002.

6. Martin Fowler, *The New Methodology* [online] http://www.martinfowler.com/articles/newMethodology.html. Dated: June 2002. (Access date: August 2002).

7. John F. McGrew. *Breaking the Human Computer Interface Design Bottle Neck: A Parallel Design Process Based on a Genetic Algorithm*. in *Tutorial 14: "Designing Interfaces for Handheld Computers" at CHI 2001*. 2001. Seattle, USA.

8. Herman T. Tavani and James H. Moor, *Privacy Protection, Control of Information, and Privacy-Enhancing Technologies.* Computers and Society, 2001. **March 2001**.