PersonisJ: mobile, client-side user modelling

Simon Gerber, Glen Pink, Michael Fry, Judy Kay, Bob Kummerfeld, and
Rainer Wasinger

School of Information Technologies, University of Sydney, Sydney, NSW 2006,
Australia
`{sger6218,gpin7031,mike,judy,bob,wasinger}@it.usyd.edu.au`

Abstract. The increasing trend towards powerful mobile phones opens
many possibilities for valuable personalised services to be available on
the phone. Client-side personalisation for these services has important
benefits when connectivity to the cloud is restricted or unavailable. The
user may also find it desirable when they prefer that their user model be
kept only on their phone and under their own control, rather than un-
der the control of the cloud-based service provider. This paper describes
PersonisJ, a user modelling framework that can support client-side per-
sonalisation on the Android phone platform. We discuss the particular
challenges in creating a user modelling framework for this platform. We
have evaluated PersonisJ at two levels: we have created a demonstrator
application that delivers a personalised museum tour based on client-side
personalisation; we also report on evaluations of its scalability. Contribu-
tions of this paper are the description of the architecture, the implemen-
tation, and the evaluation of a user modelling framework for client-side
personalisation on mobile phones.

1 Introduction

Personalisation has the potential to offer many benefits, particularly in reducing
information overload by enabling a person to be more efficient in finding the
information they need or want. Personalised systems can also be valuable in an
active role, alerting the user to useful information. But there is a tension between
such personalisation and privacy; the user model that drives personalisation is
based upon the user's personal information. Moreover, there is evidence of con-
siderable community concern about the proper protection of such information.

One way to address such concerns is to perform the personalisation at the
client-side, with the user's model stored on their own system. This is in con-
trast to the widespread *server-side* personalisation. Consider, for example, an
e-commerce website such as 'www.amazon.com', where customers must register
with the site in order to shop. The site can log every action they take while
they are logged in, such as the items they view, add to their shopping carts and
ultimately buy. This is used to create a user-profile which is held on the server.
This means that the website's owners are in control of this user model and the
way it is used. In client-side personalisation, the user model is controlled by the

user and the personalised applications that also run on their machine, under their control.

We now consider the issue of mobile personalisation. Mobile phones are providing an increasingly important interface for people to access information. There are currently over 3.5 billion mobile phone subscribers [4] and there is an increasing trend for these to have data subscriptions (for example, 40% of mobile users in Japan have a data plan). Interestingly, studies of how people are actually using their mobiles reveal that many use their phones for internet access, even while in their own homes or with another computer nearby [15].

At present, personalisation of the information delivered to mobile phones is typically performed at the server-side, by services in the *cloud*. This has been a necessity due to the limited computational power and memory of mobile phones. However, with widely-available consumer phones becoming increasingly powerful, it is becoming feasible to support client-side personalisation for these devices.

To support mobile personalisation on phones, we need new tools to support the creation of new applications. PersonisJ is one such tool, providing a framework for developing context-aware, personalised applications on a mobile phone. It can support reuse of user modelling information by arbitrary personalised applications running on the device. PersonisJ is unlike other personalisation shells or context-aware frameworks in that it treats the mobile device as a platform, rather than simply as an actor in a larger framework [10]. It also has to operate under very different constraints from previous user modelling frameworks, because it must take account of the power constraints for programs running on a mobile phone.

The next section reviews related work. Then we describe the architecture and implementation of PersonisJ followed by our validation of it by demonstrating its use in the MuseumGuide application and our evaluations of its scalability. Finally, we discuss the implications of the work and future directions.

2   Related Work

The relatively recent emergence of powerful mobile phones has created new possibilities for mobile personalisation and the associated needs for personalisation. For example, studies point to the need for personalisation of mobile search interfaces [5]. There has been considerable exploration of mobile personalisation for a range of contexts and types of application. For example, personalisation of information available has been based on the user's social context [9] and location [2]. In e-commerce, personalisation has been widely deployed, with an increasing role for mobile, m-commerce [8]. Some interesting forms include systems that enable retailers to push recommendations to the mobile customer [13] and to offer both personalised product details and in-store customer advice [14]. Another important class of applications [11] is the mobile guides which can cover roles as diverse as museum guides, navigation systems and shopping assistants [12]. For example, the PEACH [17] system delivered personalised information about the art in a museum on PDAs.

Early research in mobile personalisation has been dominated by a view of the mobile phone as the client of (and portal to) a powerful server which was responsible for the personalisation, often restricted to a particular space such as a university or hospital [3]. In the mobile personalisation work described above, the architecture of the systems places the personalisation at the server side. We have not found reports of mobile client-side personalisation, or even mobile applications that reuse frameworks for client-side personalisation.

One of the barriers for client side personalisation is the lack of a framework for the user modelling. A recent review of generic user modelling systems [10] points to the considerable work on such frameworks for server side personalisation.

PersonisJ is strongly influenced by the PersonisAD context-aware modelling framework [1]. Distinctive features of this modelling framework are: the same mechanisms model users as well as devices and places; it supports distributed user modelling, particularly important for pervasive computing applications; it provides *scrutable* modelling, meaning that it was designed, from its foundations so that it could support a user's scrutiny of their user model and the way that it is used. It is also able to perform lightweight user modelling, making it a promising foundation for the phone where power consumption is a major concern.

We now describe key elements from PersonisAD that are important for PersonisJ. PersonisAD represents a model as an hierarchy of *contexts* which can contain *components*. It is based on the *accretion/resolution* representation. Applications interact with PersonisAD via three primitive operations. The first looks up a Model for a particular person, device or place. The application can then can use a *tell* operation to supply evidence, and an *ask* operation to request the value of a component. This value is dynamically determined at the time of the *ask* based on a two part process: an *evidence filter* selects just the evidence allowed for the application which did the *ask*; a *resolver* which then interprets the set of evidence. For example, a playlist application might use a resolver for a person's favourite genre from a list of evidence that includes the songs they have most recently played. Notably, flexibility and power in the reasoning comes from the availability of a range of evidence filters and resolvers. Some aspects of PersonisAD are not suitable for use on a mobile device. Notably, it is a distributed server application and *always on* so that clients can make TCP/IP connections. While the essence of PersonisAD gives a conceptual foundation for this work, the demands of creating a framework for client side personalisation on a mobile phone have meant that we have created PersonisJ from scratch and independently from it. Unlike PersonisAD, written in Python, PersonisJ is in Java to run as a native application on Android.

3 Architecture and implementation

We now describe the PersonisJ framework. We begin with the conceptual level, which has much in common with PersonisAD described above. Then we present the high level architecture. The actual implementation was on the Open Hand-
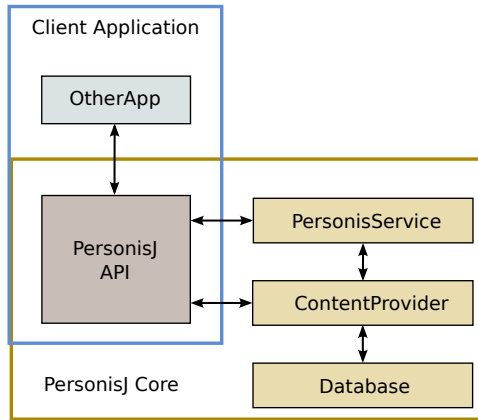
Fig. 1: PersonisJ Architecture

set Alliance's Android platform[1]. Pure implementation details are relegated to footnotes.

At a conceptual level, PersonisJ represents user models as an hierarchical structure of *contexts*, which can contain the *components* to be modelled. So, for example, it may have a context for the user's visits to museums and within this it may have components modelling the museums they prefer. Each component accretes evidence. This is essentially a value with metadata indicating how and when it was received. The metadata supports flexible evidence filtering and resolution and is a basis for supporting scrutability. The hierarchy of contexts and components constitute an *ontology* and a sub-tree within the hierarchy is a *partial ontology*.

Figure 1 shows the key modules of the PersonisJ architecture. The *PersonisJ API* enables an arbitrary application to access the user model, albeit only after the application has been granted read and/or write access to this model. The *PersonisJ Core* provides the *API*, *Database*, *ContentProvider* and *PersonisService*. We now describe how each of these have been designed to represent the model and to secure access management.

The *Database*[2] has a table for each context, component and piece of evidence in the model. The *Database* is not accessible outside the *PersonisJ Core*. Instead, access is mediated by the *ContentProvider*, which specifies a unique "content URI" for content it exposes to the client application.

When a client interacts with the model, via the *PersonisJ API*, it must use a generic *ContentResolver* to act upon data, using content URIs. It is critical that client applications cannot directly modify anything within the PersonisJ database; for this reason, custom security permissions have been created for PersonisJ (READ_CONTENT, WRITE_CONTENT, and TELL_PERMISSION).

---

[1] Android, URL: http://www.android.com
[2] SQLite, on the Android platform

These permissions are not intended to restrict unauthorised access; rather, it forces applications to explicitly declare how they wish to interact with PersonisJ. All applications intending to access the PersonisJ database must state the permissions they require upfront[3]. To avoid direct third party application access to the database, the WRITE_CONTENT permission has been given the Android protection level of "signature", meaning that only applications signed with the same certificate as the PersonisJ code have direct write access to PersonisJ content. This has the effect that only two of the three above defined permissions are publicly available; third-party applications intending to read from the database use the READ_CONTENT permission (which corresponds to the "ask" operation) and those intending to contribute to the database do so via the PersonsisService module using the TELL_PERMISSION (corresponding to the "tell" operation).

To allow client applications to interact with PersonisJ, the *PersonisJ Core* provides the *PersonisService* in Figure 1. Because this resides within the same application as the *ContentProvider*, it can freely write to the PersonisJ *Database*.

The *PersonisService* is also in charge of handling imports. It is not possible to expose the ability to create new contexts and components via the *ContentProvider* without also exposing the ability to modify or delete them. This problem is solved by allowing client applications to pass the *PersonisService* a description of the partial ontology they require encoded in JavaScript Object Notation (JSON). The *PersonisService* then creates any necessary contexts and components.

PersonisService has one other responsibility. Any time it processes a 'tell' operation it will broadcast a message[4] indicating which component was changed. It will then walk back up the context tree to the root, broadcasting notifications for each parent context in turn. This allows client applications to 'listen in', and discover if a particular component has received new evidence or, more generally, if any component beneath a particular context has changed.

The ontology is normally specified by applications, however location is so fundamental that PersonisJ provides a predefined *location monitor* context in the phone model, with components for the co-ordinates of each location value. When turned on, this uses the phone's GPS to record any change in location as Evidence in the PersonisJ model.[5] PersonisJ also supplies a 'default resolver' that can resolve co-ordinate evidence into the phone's own framework.

PersonisJ provides an Application Programming Interface [API] that applications can use to interact with the system.

---

[3] Security permissions are enforced by the Android OS, and the user is made aware of the required permissions when installing the application.

[4] an *Intent* in Android

[5] The location monitor utilises the location API provided by Android. To conserve battery power, it does not turn on the GPS of its own accord. Instead, it hooks in to its operation whenever another program turns it on. A simple piece of UI allows the location monitor to be enabled or disabled. While the GPS is disabled the location monitor does not reside in memory and uses no additional battery.

The 'ask' operation returns a single value for a component. An important aspect of PersonisJ is that each component may have a list of evidence. In order to resolve multiple pieces of evidence into a single value the PersonisJ API provides a *Resolver* interface, which interprets a set of evidence, returning a single value. The API includes a selection of 'default' resolvers for numerical values, booleans, strings and dates. The actual resolution of values is executed within the client application, enabling them to provide their own *Resolver* implementations.

The evidence passed to the *Resolver* can optionally be passed through an *EvidenceFilter*, which chooses the pieces of evidence to be used by the *Resolver*.

PersonisJ Core has optimisations to ensure efficient operation of resolvers that require just the last piece of evidence, or the last few pieces of evidence, for a component. These are very useful and commonly needed.

The API also exposes the ability to import and export partial ontologies. It provides a method, which can nominate any context, to gain the exported ontology. This operation occurs within the calling process. Optionally, a URL can be provided to the export function. After encoding into JSON the generated string will be passed to the PersonisService which, in a background thread, will transmit the data to the specified URL via an HTTP POST. The import function works in a similar fashion, to upload the partial ontology defining a new part of the model with *PersonisService* performing this operation as external applications do not have write access to PersonisJ itself.

4 Evaluation

This section reports the two approaches we have used to validate the PersonisJ framework. First, we used it to create a personalised client-side application called MuseumGuide , which is able to notify a user of nearby museums and then download content for any museum that the user is keen to visit. Then we conducted scalability evaluations.

4.1 MuseumGuide application

Consider the following scenario:

Alice and Bob, with their young family, are on a driving holiday in Sydney. Their phone has a model of the family's entertainment preferences including: low cost; suitable for children; kids are interested in ancient Egypt. The MuseumGuide , running on Alice's phone and aware of their location, sends an alert that they are near the Nicholson Museum. They decide to go to the museum and on arrival, download a personalised museum tour, based on a detailed model of the family's interests.

We now describe our implementation of MuseumGuide , making use of the PersonisJ framework to model a family's entertainment interests, as outlined in the scenario, and their more detailed interest model.

Figure 2 shows the MuseumGuide architecture, including the PersonsisJ API module described earlier on (see Figure 1), providing the application with access
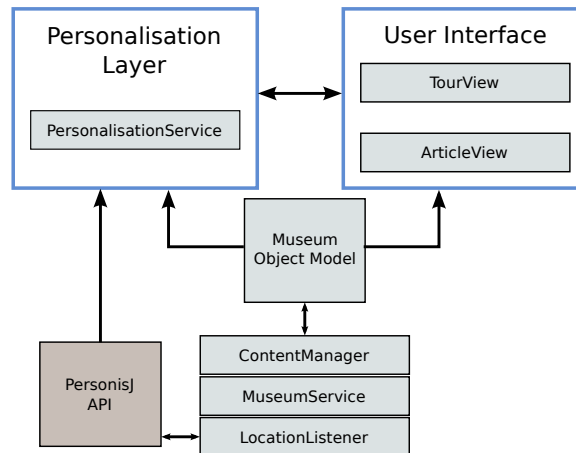
Fig. 2: MuseumGuide architecture

to the PersonisJ database. It operates as a client-side application, and like other Android applications requires the user to confirm when they are installing the application that they are happy with granting it the requested permissions, in this case, access to the Internet, access to the PersonisJ ask and tell operators, as well as several other permissions. MuseumGuide uses a BroadcastReceiver, named `LocationListener`, to listen for component changes broadcast by PersonisJ. If the changed component is the Location component the alert is passed to `MuseumService`, as shown in the figure. This compares the new location against a stored list of museums. It sends a notification as shown on the left in Figure 3. When viewed it starts an Activity that checks to see if there is content that can be downloaded (or updated) for the museum in question. If so, a prompt is displayed asking the user to confirm whether it should go ahead with the download (middle of Fig. 3).

The content is downloaded from a URL that is generated based upon the name of the museum. Once loaded, the background service invokes the `ContentManager` which imports it to MuseumGuide and then sends a second notification (right screen in Figure 3). Responding to this notification brings MuseumGuide into the foreground.

At this point the content is ready for viewing. A `PersonalisationService` is situated, architecturally, between the ContentProvider and its UI. This service is responsible for personalising the content before it is displayed. Figure ?? shows one such personalised article. In this example, text content suitable for a young child was chosen for display.

We have created a BroadcastReceiver that receives broadcasts when any component changes. This ensures MuseumGuide is notified anytime the location changes, to compare it against the locations of museums.
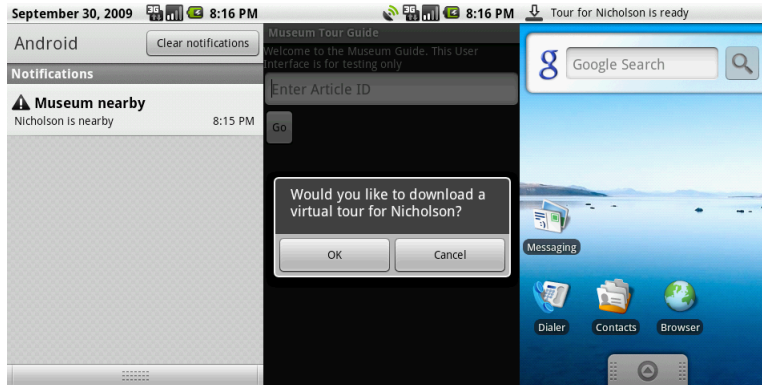
Fig. 3: MuseumGuide interface

At this point MuseumGuide checks to see if content is available for the museum and, if so, whether it has already been downloaded and whether it is up to date. If not, the program will prompt to fetch the content, as in Figure **??**. We used content for the Nicholson Museum, located on the University of Sydney campus, taken from an existing museum guide [6]. Now that the content has been imported it can be displayed.

The PersonalisationService exists as a separate Service to enforce a separation between the content, the user-interface and the personalisation of the content. The service takes a museum and article identifier as input and returns personalised output, based on a simple personalisation algorithm based on the age of the user.

Based on the above description of the implemented application, we have demonstrated that PersonisJ allows applications to register for updates in a manner that is no more complicated than requesting an update from existing Android system services. Once the logic for resolving a value from evidence has been encapsulated within a Resolver class it requires only a single call to `ask` in order to retrieve it.

We can see that PersonisJ makes it possible to create a context-aware application with only a few calls to a high-level API. We note that obtaining location updates from Android directly requires slightly less code. However, PersonisJ provides a higher level of abstraction.

4.2 Scalability evaluation

Before beginning our evaluation of PersonisJ we note that the DalvikVM[6] does not provide any 'Just in Time' [JIT] compilation. This has three important consequences. Firstly, it means managed code will always run slower on Android than its native equivalent. Secondly, it means that method level optimisations

---
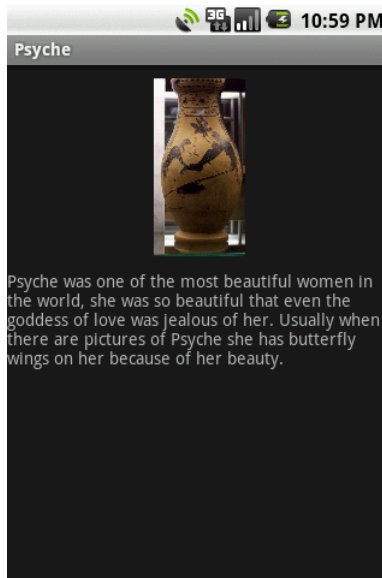[6] the Java Virtual Machine used in Android

Fig. 4: personalised article

are important, as one cannot rely on minor inefficiencies being 'optimised away'. Finally, the lack of JIT compilation means we can be relatively naive about our performance testing, knowing that the code we write is, more or less, the code that Android executes.

Performance was measured using the profiler built into the Android framework, which has a resolution of micro seconds. The results are shown in Table 1. The critical column is the final column which shows the relative performance for the tested actions. This was calculated as the ratios of the average time per call normalised to one, then rounded to the nearest ten. The 'time' and 'average time' columns were included for completeness, but should not be relied upon as a measure of real-world performance.

Tests 1 to 4 involve actions selected from the sample actions in Table 1. This gives a baseline for PersonisJ performance against some simple operations. Tests 5, 6 and 7 reflect some basic PersonisJ API operations. Test 5 is the `tell` operation. We would expect it to be a relatively quick API call as it does not do much work itself, but rather encapsulates a call through to the PersonisService class in the core framework. Test 6 is the corresponding `ask` operation. Test 7 is an example of an API call that would be used to drill down through an ontology. Finally, Test 8 retrieves the same information from the PersonisJ database as Test 7, but does so using a ContentProvider directly and is optimised to use only one query instead of two.

These results indicate that PersonisJ's fastest operation is clearly the `tell` operation. This is expected, as a tell operation is asynchronous. The `ask` op-

Table 1: Performance Tests: The result of 200 calls to various Android and PersonisJ methods, profiled using 'traceview'. Results are sorted by relative performance. An asterisk * before the test number indicates a test involving PersonisJ

Test Description	#	Time (ms)	Time/Call	Relative Performance
String.length()	1	7.606	0.038	1
HashMapIterator.next()	2	70.822	0.354	10
HashMap.put()	3	142.474	0.712	20
PersonisContext.tell()	*5	5763.410	28.82	760
LayoutInflator.inflate()	4	13904.226	69.521	1830
'Raw' context query	*8	14719.000	73.595	1930
PersonisContext.ask()	*6	24472.835	122.356	3220
PersonisContext.getChildContext()	*7	28371.960	141.860	3730

eration, in contrast, is a lot heavier and can be expected to differ significantly based on the supplied parameters. The ask operation tested in Test 6 was as lightweight as possible. Only the latest piece of evidence was examined and a default String resolver was used. The String resolver performs no additional computation, as all evidence is stored internally as a String. Even so, this simple `ask` took approximately three times longer than a `tell`. Test 7 is an example of a fairly common class of method. The ability to retrieve a model, context and Component by name is part of the required setup any client application must perform before it can even start to call `ask` or `tell`. However, once acquired, these methods do not usually need to be called again until the next time the application is killed and then relaunched. Test 8 acquires the same data as Test 7, but does so using only a single query. Its performance is roughly half that of Test 7. From this, we can determine that when only a few columns are involved it is much more efficient to return all columns in the initial query. This translates into an easy optimisation that can be applied to all PersonisJ calls that require ContentResolver queries.

We can also see from the results that the PersonisJ operations are the same order of magnitude as inflating a trivial user interface, with just one button, from XML (Test 4). Any user interface of practical use would far take longer, We can see that an *ask* on the model (test 6) takes of the order of 0.1 seconds. We can therefore conclude that acquiring complex information from PersonisJ would have not add a noticeable delay to the loading of a typical interface screen.

The PersonisJ API takes up 44.7kb when converted to dex format. This further compresses to 20.414kb in a .jar file. The PersonisJ application is 37.24kb as an APK package. It takes up 108kb of space on the phone when uncompressed. These sizes are negligible when compared to most Android applications. The PersonisJ Core Framework has essentially no UI and no other packaged resources, such as images. This makes it smaller, when first installed, than all but the simplest of applications. However, PersonisJ does create a database which will grow continuously over time.

The data requirements of PersonisJ are not dictated by PersonisJ itself, but rather the client applications. A full analysis of disk space requirements is therefore neither possible nor particularly helpful. An empty PersonisJ database is 10KB (10240 bytes).

In summary, the time performance of PersonisJ is adequate for the tasks required of it. Simple operations and resolvers that do not require much evidence are fast enough to be performed in the main UI thread without affecting application performance. More complex resolvers and contextual reasoning should be performed in a background thread, but are unlikely to take longer to run than setting up a typical UI screen. With respect to space performance, the size of the PersonisJ API modest. The PersonisJ database, however, could grow too large over a period of only a few months. Future versions of PersonisJ will address this with options to move the database to other storage media, pruning old evidence, or backing up old data to secure, personal storage over a network.

5   Conclusions

The goal of our design for the PersonisJ was to explore how to design a personalisaton framework that could support client-side personalisation on a mobile phone. This was motivated by two potential benefits: first, this enables the phone to deliver a personalised service even when the phone is not connected to a data network; and second, it stores the user model on a device that is controlled by the user.

With the user model restricted to the mobile phone, the issue of privacy is affected by the effectiveness of the security. Our description of the PersonisJ architecture included the careful design to address security issues, with PersonisJ mediating all accesses on the model. Descriptions of the actual implementation in [7] deal with these matters in detail. There is another critical element for the security in terms of the behaviour of the applications that a user loads onto their phone. Our MuseumGuide application, made use of a security framework [16] which enables the user control what such an application is permitted to do.

Client-side personalisation provides an important foundation for life-long user modelling, in which the user is able to create, edit, reuse, and extend their user model throughout their digital life experiences. We have described PersonisJ, a user modelling framework that can support client-side personalisation on the Android phone platform. We have reported evaluations of PersonisJ at two levels: in the MuseumGuide demonstrator application that delivers a personalised museum tour based on client-side personalisation; in evaluations of its scalability. Contributions of this paper are the description of the architecture, the implementation, and the evaluation of a user modelling framework for client-side personalisation on mobile phones.

References

1. Mark Assad, David Carmichael, Judy Kay, and Bob Kummerfeld. PersonisAD: distributed, active, scrutable model framework for Context-Aware services. In

Pervasive Computing, pages 55–72. 2007.

2. Mark Bilandzic, Marcus Foth, and Alexander De Luca. Cityflocks: designing social navigation for urban mobile information systems. In *DIS '08: Proceedings of the 7th ACM conference on Designing interactive systems*, pages 174–183, New York, NY, USA, 2008. ACM.

3. Guanling Chen and David Kotz. A survey of Context-Aware mobile computing research. Technical report, Dartmouth College, 2000.

4. Karen Church and Barry Smyth. Understanding the intent behind mobile information needs. In *Proceedings of the 13th international conference on Intelligent user interfaces*, pages 247–256, Sanibel Island, Florida, USA, 2008. ACM.

5. Karen Church and Barry Smyth. Who, what, where & when: a new approach to mobile search. In *Proceedings of the 13th international conference on Intelligent user interfaces*, pages 309–312, Gran Canaria, Spain, 2008. ACM.

6. Marek Czarkowski. *A Scrutable Adaptive Hypertext*. PhD, University of Sydney, March 2006.

7. Simon Gerber. Personisj: A platform for context-aware, client-side, mobile personalisation, 2009.

8. A. Goy, L. Ardissono, and G. Petrone. Personalization in e-commerce applications. pages 485–520, 2007.

9. J. Kjeldskov and J. Paay. Public Pervasive Computing: Making the Invisible Visible. *Computer*, 39(9):60, 2006.

10. A Kobsa. Generic User Modeling Systems. pages 136–154, 2007.

11. A Kruger, J Baus, D Heckmann, M Kruppa, and R Wasinger. Adaptive Mobile Guides. pages 521–549, 2007.

12. Antonio Krüger, Jörg Baus, Dominik Heckmann, Michael Kruppa, and Rainer Wasinger. Adaptive Mobile Guides. In Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl, editors, *Chapter in: The Adaptive Web: Methods and Strategies of Web Personalization*, pages 521–549. Springer, Berlin, Heidelberg, New York, 2007.

13. Stan Kurkovsky and Karthik Harihar. Using ubiquitous computing in interactive mobile marketing. *Personal Ubiquitous Comput.*, 10(4):227–240, 2006.

14. Jun Li, Ismail Ari, Jhilmil Jain, Alan H. Karp, and Mohamed Dekhil. Mobile in-store personalized services. In *ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services*, pages 727–734, Washington, DC, USA, 2009. IEEE Computer Society.

15. Stina Nylander, Teres Lundquist, and Andreas Brannstrom. At home and with computer access: why and where people use cell phones to access the internet. In *Proceedings of the 27th international conference on Human factors in computing systems*, pages 1639–1642, Boston, MA, USA, 2009. ACM.

16. Glen Alan Pink. Safe execution of dynamically loaded code on mobile devices, 2009.

17. O. Stock, M. Zancanaro, P. Busetta, C. Callaway, A. Kruger, M. Kruppa, T. Kuflik, E. Not, and C. Rocchi. Adaptive, intelligent presentation of information for the museum visitor in PEACH. *User Modeling and User-Adapted Interaction*, 17(3):257–304, 2007.