Büscher M., Christensen M., Hansen K.M., Mogensen P., Shapiro D. Bottom-up, top-down? Connecting software architecture design with use, In Voß, A., Hartswood, M.; Ho, K., Procter, R.; Rouncefield, M.; Slack, R.; Büscher, M. Configuring user-designer relations: Interdisciplinary perspectives . Springer Verlag, Accepted for publication in 2007.

Bottom-up, top-down? Connecting software architecture design with use

Monika Büscher[1], Michael Christensen[2], Klaus Marius Hansen[2], Preben Mogensen[2], Dan Shapiro[1]

[1] Department of Sociology, Lancaster University, UK. {m.buscher; d.shapiro}@lancaster.ac.uk
[2] Computer Science, Aarhus University, Denmark. {toby; klaus.m.hansen; preben}@daimi.au.dk

1 Introduction

Participatory design has traditionally focused on the design of technology applications or the co-realisation of a more holistic socio-technical bricolage of new and existing technologies and practices. 'Infrastructural' design issues like software architectures, programming languages, communication, security, and resource models do not seem to be in need of, nor amenable to, participatory design. Yet we should expect, and research has indeed shown, that there are deeply consequential relationships between use and software architectural design. If designers hide the 'sensing', 'reasoning' and computation technologies do, for example, people can find it difficult to perceive, understand and creatively exploit technological affordances (e.g. Belotti and Edwards 2001). In addition, the causes of failure and breakdowns can be hard to detect and even harder to address (Belotti et al. 2002).

Moreover, the emergence of ubiquitous, ambient and component-based computing has taken computing out of comprehensive systems into a multitude of devices, services and resources. In some sense this makes the computer disappear or become invisible (Weiser 1991), and it enables increased flexibility and 'bricolage' of disparate elements, but it also introduces extra difficulties, for example when it comes to determining which computing devices, services or resources are the most appropriate to use in specific situations. To engage ubiquitous computing technologies effectively and creatively, people need support in making computational processes, states and potential perceivable or 'palpable' as and when they may need or wish to do so, and in ways that are appropriate for the particular situation they are in, their level of computational literacy and their interest. New software architectures are needed to support palpability. To address this, various European research teams have come together in the 'Palpable Computing' (PalCom) project (www.ist-palcom.org). The project is creating a range of palpable ambient computing prototypes in healthcare, emergency services and landscape architecture. Its principal aim, however, is to create an open architecture for palpable computing. The open architecture will consist of a set of specifications as well as a reference implementation of these specifications.

This is an ambitious goal. The demand for appropriateness and the complex, multi-layered translations between material computational processes and the functionality and interfaces

that users experience, mean that it is a goal that is impossible to meet completely. However, some significant progress can and must be made if ubiquitous computing is to be an attractive and useful prospect. Clearly, design for palpability is not simply a matter of revealing what was previously hidden. Palpable computing is a new design initiative that envisages ubiquitous technologies whose states, processes and affordances can be made available to the senses, or 'palpable', and that are therefore more easily understood, appropriated and controlled. To address palpability, we take six dimensions of the vision of ambient and ubiquitous computing, and challenge them by considering their opposites. Users will often need to find a position that lies between the extremes:

ubiquitous/ambient computing	complemented with	**palpable computing**
invisibility		visibility
construction		de-construction
scalability		understandability
heterogeneity		coherence
change		stability
automation		user control and deference

When a supposedly seamless and transparent set of connections breaks down, for example, users should be able to make them visible and inspect what has gone wrong. Similarly, users should be able to deconstruct an ambient assembly of devices and services, both to inspect it for repair and to use its elements for new assemblies. While ambient environments should be able to scale up to large numbers of participating elements, they should also remain understandable. Coherence must be forged from heterogeneous materials, such as disparate digital and physical devices and information, while recognising and where necessary preserving the particularities of each. Changes – for example, of location, resources, context and activity – are normal in an ambient environment, but sometimes users need to be aware of the change and sometimes they need to experience highly stable adaptivity. Users do not want to be constantly pestered with choices and they need to be able to delegate 'routine' decisions, but it is inevitable that the system will often guess wrong, so users must always be able to retrieve control – and must have the information to help them to know when they might want to do so.

To support the situated negotiation of these core dimensions of ubiquitous computing and to allow people to make computational 'sensing', 'reasoning', potential and actual activities palpable, new forms and depths of interactivity are required. This is where the ambition and, perhaps, inescapable unattainability of the ultimate design goal of palpable computing lies. Some form of human-like social and contextual perception and skill on the part of the technologies seems to be essential, yet, research within computer-supported cooperative work (CSCW) and related fields proves that it is impossible to produce anything but very

limited and flawed versions of such perception and skill (Suchman 1987, Dreyfus 1992). When people interact with each other, they are able to negotiate contradictions and complementarities with ease, using nuanced skill, perception and judgement to act appropriately as the situation demands. They are able, that is, to act with social and contextual skill. For computers, however, this is extraordinarily difficult. Palpable applications and services 'must' be able to make, and support the making of, optimal choices concerning each of the dimensions outlined above in situated use, and a palpable software architecture 'must' support the construction and operation of applications and services that can do so. But we know in advance that it will not be possible to achieve this completely, and that various compromises and simplifications will have to be made.

Many designers of computer applications, spanning groupware and CSCW systems (Dourish 2003, Bansler and Havn 2006), ubiquitous computing (Chalmers 2003), context aware and ambient systems (Belotti et al 2002), grid technology (Hartswood et al, submitted), and system security (de Paula et al 2005) share similar concerns. Component-based computing potentially makes creative (de-)composition possible and, more explicitly than any socio-technical step before, turns users into designers. It 'dissolves' the privileged position of the designer who knows 'the system's range of actions in advance' (Dourish 2003). Research and design have begun to address these challenges with flexible architectures that support a range of tailoring techniques (MacLean et al 1997), with maps and models that reflect, and allow users to modify, system behaviour (Dourish 1995), and ways of revealing system properties through 'seamful design' (Chalmers and Galani 2004). The PalCom research builds on this work. In particular, we seek to move beyond the appreciation that it is impossible for designers to predict what kinds of translations of computational states or processes would be appropriate for different users in different situations. While developing reflective, agent, and component-based support for palpability (Rimassa et al 2005, Ingstrup and Hansen 2005), PalCom also supports strategies that rely less on the skill of designers to anticipate how and when someone (whose level of computational literacy and situated needs for inspection are unknown) might wish to examine computational processes, and more on support for 'reflexive' – in the sense of direct, two-way, feedback-rich – forms of human-computing interaction. Our design incorporates the evolution of standards (Belotti et al. 2002) and a variety of discovery and inspection tools.

To pursue these software architectural design goals in a way that fits design into emerging practice, an ethnographically informed, participatory design approach is essential. However, stretching the iterative cycles of participatory design to involve users in the design of software architectures poses a number of difficulties. First in line is the indirectness of users' experience of computer architectures.

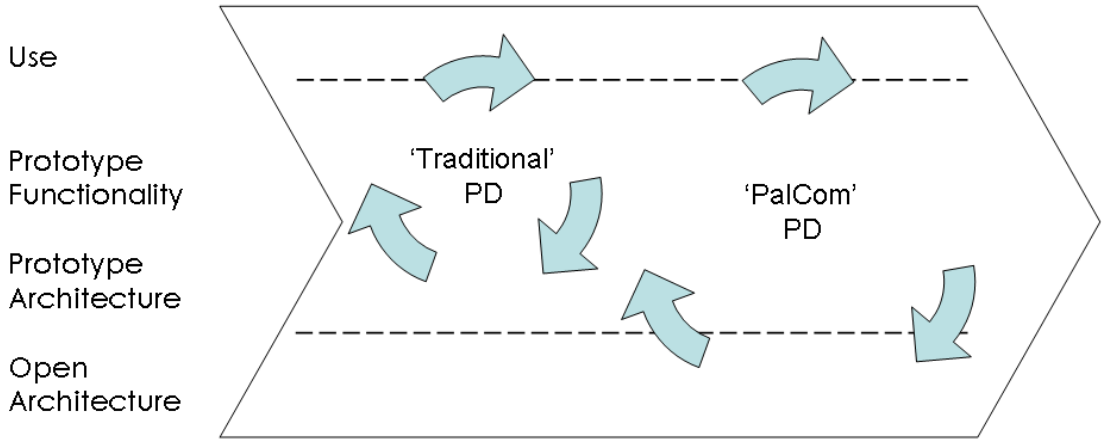

Figure 1 Stretching 'traditional' participatory design methods to inform software architecture innovation

In 'traditional' participatory design, user participation usually informs the design of hardware and software that seek to support the users' work directly. Users are able to engage with mock-ups and prototypes of the objects they are co-designing directly, often in a hands-on manner. Where software architecture is concerned, this engagement is indirect. Even though users of palpable applications and services will rely on features of the software architecture to make computational states, processes and affordances palpable, they will rarely interact directly with it. In their pioneering exploration of challenges for user-centred design and evaluation of infrastructure, Edwards et al (2003) focus on the indirectness of users' experience with computer architectures and raise important questions:

- Is it possible to more directly couple design of infrastructure features to the design of application features?
- How can this more direct coupling exist when the applications the infrastructure supports don't yet exist or cannot be built without the infrastructure itself?
- Could the context of either users or the use of these unknown applications have an important impact on the features we choose?
- How can we avoid building a bloated system incorporating every conceivable feature, while ensuring we have a system that will not be constantly updated (and so repeatedly broken) throughout its lifespan? (Edwards et al 2003)

Our experience with participatory design shows that in-depth, long-term engagement with the context of users and use is essential for good design. We involve users deeply and equally as co-designers in long-term processes of socio-technical co-innovation. This is motivated by the fact that long-term use (and design-in-use) of prototypes that is as realistic as possible, in settings that are as realistic as possible, allows users to bring hands-on practical creativity to the use of new technologies. This is a condition for the emergence of viable future practices

which, in turn, should inform the design of the technologies under development. Thus, to bring participatory design to the design of software architectures, we must also ask:

- How can we make use experience of software architectures more direct?

In this chapter we describe how we bring participatory design to the design of the PalCom open architecture. The schema in Figure 2 gives an overview. Four sets of people with different primary interests and skills (users, work analysts, application developers and software architects) connect through observations, participatory workshops, and experiments. Collaboration is often face-to-face and hands-on, as users, work analysts, developers and architects travel to each others' sites of work, to bring prototypes into real-world use. In the context of participatory and ethnographically-informed design, there is nothing new in users, work analysts and application developers working closely together to inform and challenge the evolving design. Bringing software architects into this process is less usual, however, and a technique we have created to support this is the formation of a group of 'travelling architects' (Corry et al., 2006). Prototypes embody, and serve as the focus for, innovation in terms of practice, applications and services, and architecture.

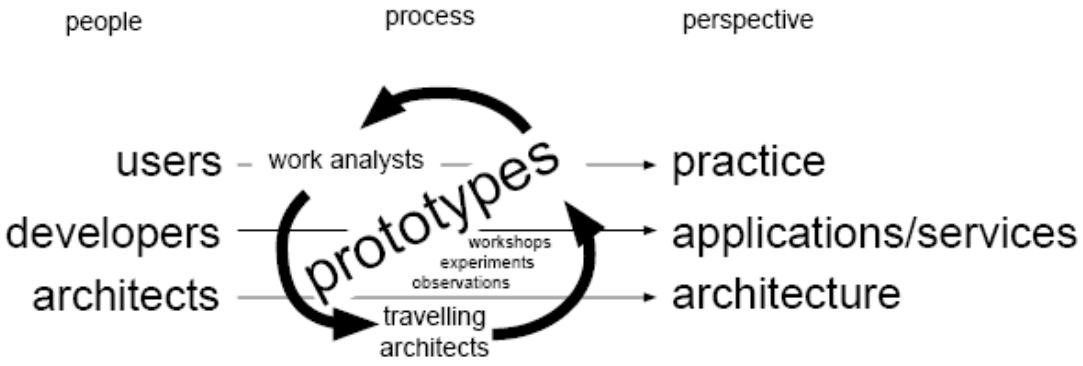

Figure 2 Schema of the participatory design process

This method introduces at least four participatory elements to the design of the open architecture. First, the analysis of work practice and of corresponding possibilities for technical support suggest requirements for an underlying software architecture. Second, practitioners' experiences of using evolving application prototypes expose strengths and weaknesses in the software architecture design, and suggest further requirements. Third, the presence of 'travelling architects' – gaining first-hand experience of users' work settings and of their encounters with prototypes – opens new direct pathways to the software architecture and empowers software architects to participate in a wider range of discussions and negotiations around the design. Fourth, the application developers within the project are themselves users of the evolving open architecture, opening up an opportunity for a participatory design relation amongst the computer scientists in the project.

Sections 2 and 3 below explore the intersections between these four elements by focussing on a central example, tracking the evolution of the concept of ‘assemblies’ through a series of reflections from different perspectives, revealing how perspectives and experiences from use, application prototype design and software architecture design intertwine in the participatory design of the PalCom open architecture. Section 2 explores how the concept of ‘assemblies’ arose in the course of close collaboration with one set of prospective users of an application prototype. It formulates some core technical challenges, describes scenarios derived from work practice of the prototypes in use, and considers some implications for the open architecture. Section 3 explores the ways in which the concept of ‘assembly’ was taken up in the open architecture itself. In Section Four, we draw out some key insights from this reflective process.

2 Challenges to coupling infrastructure, applications and services

Ubiquitous computing has always posed technical challenges for software architectures (Weiser 1993). This stems in part from a complex interplay of requirements from particular applications and particular use, and in part from general properties of these kinds of computing systems such as resource constraints, use of wireless connectivity, and mobility of devices and users. Considered from a technical point of view, many of the six dimensions of the challenge for palpable computing (invisibility/visibility, scalability/understandability, construction/de-construction, heterogeneity/coherence, change/stability, automation/user control and deference) are amenable to established object-oriented software engineering practices. In this chapter, we will focus on the negotiation of visibility/invisibility and construction/deconstruction in an effort to achieve a creative understanding of computational affordances on a small and large scale, although we also touch on the other dimensions. *Invisibility* of the internals of objects, for example, is usually supported by information hiding and considered a major technique in managing dependencies in software systems (Parnas 1972). *Construction* (or *composition*) is the raison d'être of component-based architectures in which applications ideally may be composed from available software components (Szyperski 1998). *Understandability* may be said to be coupled to (static) typing in programming languages where program elements are statically assigned a set of permissible data values.

On the other hand, some of the complementary concepts in the challenge pairs give rise to interesting issues in languages, middleware, and software architecture. *Visibility*, for example, may be in conflict with information hiding (Ørbæk 2005), in that controlled ways of ‘opening up’ software systems are needed. In particular, if exceptions arise in the use of palpable computing systems, visibility of what has gone wrong and possibly why becomes important. Actually, in a dynamic pervasive computing world, failure cannot really be seen as exceptional and thus we instead try to design for contingency handling (a concept covering more than just failure handling) rather than exception handling. *Change* of, e.g., location is also a challenge

in that references to resources from software components need to be re-established. *De-construction/de-composition*, in particular when the de-construction is not an exact inverse of a previous construction, emerges as a major and radical new issue. In general, it may be said that much effort has been expended in middleware development in order to make application programming as transparent as possible to location/distribution, time, failures etc. whereas it was realized at the outset that palpable computing would have to go beyond this in addressing the challenge pairs, e.g., in terms of having to support visibility of components (and their locations) in order to support de-composition. Indeed, it has quite often been remarked that 'transparent' in computer science – meaning concealed and invisible – is quite contrary to its everyday use where it means open and accountable. One example is that of distributed systems where "distribution transparency" means exactly that it is not known to components of the system that they reside on different hosts (cf. e.g., Stroud (1992))

2.1 Gaining a sense of how coupling could be achieved in a world where applications/services don't yet exist

These technical issues have given input to ethnographic work as well as participatory design work in PalCom. It should be noted that although the sequence here places technical constraints and opportunities first, it does not imply a cause/effect relationship from technical issues to fieldwork/PD. The analysis of technical issues is deeply and continuously inspired by observations of existing practice and develops opportunities and problems for ubiquitous computing systems in the light of such observations.

One suite of application prototypes we are developing as drivers for software architectural design seeks to support landscape architects in landscape and visual impact assessment (LVIA) e.g. of windfarms. The major difficulties in this work lie in evaluating the impact of planned but not-yet-existing developments on views and experiences of landscape (Büscher 2006). This involves identifying and finding key viewpoints, carrying out and documenting a complex and rigorous process of evaluation.

Figure 3 Landscape architect Lynda trying to see whether a proposed wind farm would be visible from touristicallly or otherwise significant viewpoints or when passing: While driving, with maps, computer models, GPS

In a 60x60 km study area of undulating hills it can be extremely difficult to keep track of the location of a proposed (but not yet physically present) windfarm and envisage its visual and experiential impact on people's experience of the landscape. The 'sitepack' prototype is designed to support landscape and visual assessment. It allows users to assemble photo and

video camera(s), location devices, displays, the car, and other components, including computation services that convert location signals or track specific locations.

To illustrate some key ideas and challenges around which our participatory design process revolves, and to give readers a sense of some concrete prototypes, we present a set of brief envisioning scenarios. They take activities observed in real work practice within a typical working day for a landscape architect, re-imagined in the context of new support tools. The scenarios, although quite challenging, simplify the reality of work practice, in that they assume only one landscape architect is on site. In reality, there will often be two in the car and sometimes there will be more than one car.

Assembling the SiteTracker

In the morning, before driving off to find and assess views within the 60x60 km study area, landscape architect Lynda opens her sitepack and assembles a 'SiteTracker'. The SiteTracker consists of a small video-camera, a motor capable of turning 360 degrees, a digital compass, a GPS, and a display device. The camera, motor and compass are mounted on the car (Figure 4) in order to assist in determining whether the proposed wind farm is visible from various stretches of the road. Using a service running on her desktop providing an overview of available devices, Lynda assembles the SiteTracker and marks the GPS coordinates for the centre of the proposed wind farm.

Figure 4 Sketches and mock-ups of the site-tracker
from participatory design workshops with the landscape architects.

The SiteTracker service is set running on the display device and Lynda brings the physical assembly of devices to the car. She mounts the GPS in the front window, the display on the dashboard, and the video camera and compass on top of the motor that, in turn, is mounted on the roof of the car inside a protective casing (Figure 5).

Using the SiteTracker when driving

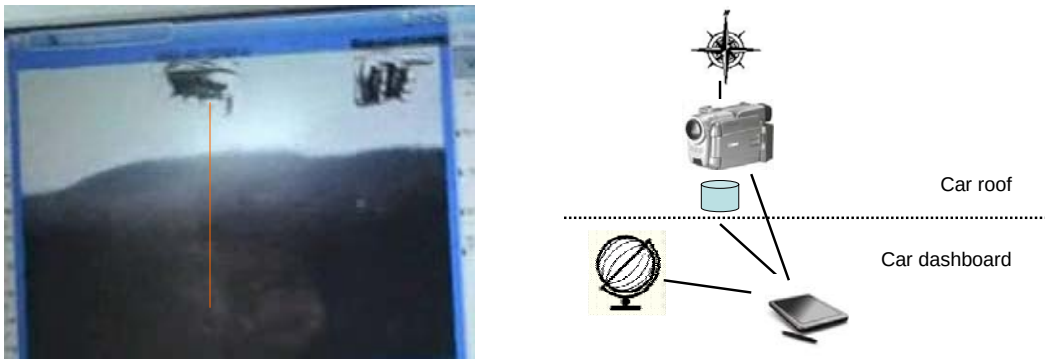

Figure 5 First SiteTracker prototype and its components. The hands and the line track the location of the centre of the wind farm and other important landmarks

While driving, the GPS constantly provides location information, and the digital compass directional information of where the video-camera is pointing (with faster updates than the GPS). On this basis, the SiteTracker service turns the motor, and thereby the camera, to point towards the proposed wind farm, and the resulting video footage from the camera is shown on the display with an overlay showing exactly where the centre of the wind farm would be, seen from the position of the camera on the roof.

Figure 6 Current Site Tracker prototype

Recconnaisance

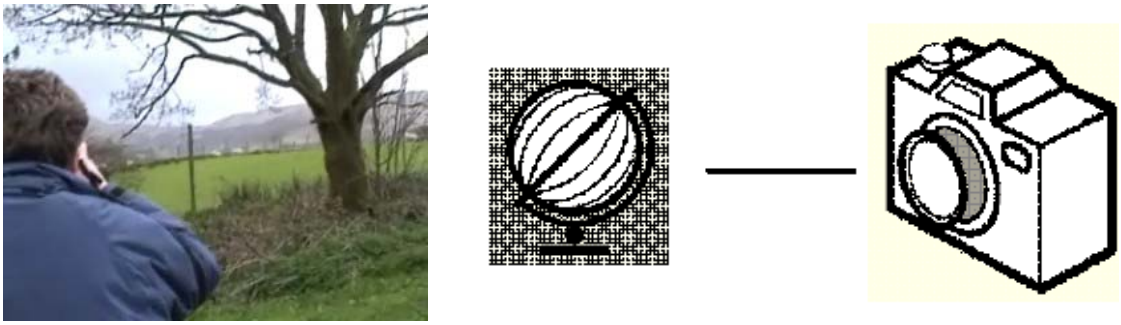

Figure 7 The 'GeoTagger' indexes photographs with location and direction information

While driving, Lynda passes a number of places that will need to be documented later on when the weather improves (documenting a viewpoint usually requires clear, sunny weather to ensure satisfactory visibility). To help her remember viewpoints where pictures should eventually be taken, Lynda frequently stops the car, gets out a still camera, unclips the GPS from the dashboard (disassembling the SiteTracker) and re-assembles the GPS with the still camera to form a 'GeoTagger' (Figure 7), providing a light-weight solution for bringing up into towers, through woodland, up hills, etc. When taking pictures, the GPS coordinates and a rough direction from the GPS is stored alongside the pictures on the camera.

Documenting the site

Figure 8 The GeoTagger expanded with tripod

The sky clears and Lynda passes an important viewpoint. She decides to document the view by taking high quality panoramic pictures (at least 180 degrees) for photomontages for the official report. She unclips the SiteTracker assembly from the car roof, replaces the video-camera with a high resolution still camera and mounts the new GeoTagger asssembly on a tripod. The tripod provides tilt information. Using the GeoTagger service she can now take panoramic pictures, where locations, accurate directions and tilt of the camera are stored along with the pictures (Figure 8). On return to the car, she disassembles the GeoTagger and re-assembles the SiteTracker to continue the survey.

Visiting a Landowner

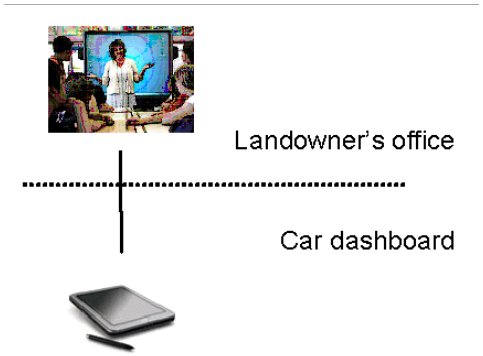

Figure 9 Assembly at the landowner's office

Later the same day Lynda visits one of the landowners possessing knowledge about the local usages of the terrain, wildlife, bio-diversity, etc. She dismounts the display from the dashboard, stores some pictures, maps and video footage on the display's storage media and walks into the landowner's offices.

In order to present draft layouts and findings, the display is now made part of new assemblies (via the landowner's network), utilising local devices: for example, by accessing material via the small display device, but showing and navigating through it using a large screen available in the office. Changes, annotations, etc. are stored on the display device.

When Lynda leaves the premises, all material that was shown on external devices (unless explicitly agreed otherwise) is taken back with the help of a 'take back service', so that no potentially confidential material is left on external devices.

Remote access and control of devices

Figure 10 Site Tracker prototype and Site Tracker assembly expanded with remote control (mobile phone). Snapshot from a participatory field experiment

On a second survey, it turns out that changes have happened since the last visit. Firstly, parts of industrial forestry have been felled, so much more of the wind farm will be visible from an important viewpoint. Secondly, now trees and hedges have leaves (the first visit was during winter), meaning that the hedge along the roadside can no longer be seen through.

As a consequence, Lynda has to extend the tripod so that the the GeoTagger is situated some 2,5 m above ground, making it impossible to look through the camera and operate it. Therefore, Lynda extends the GeoTagger assembly with a mobile phone. The image recorded by the still-camera is now shown on the phone's display, and the phone's keypad is used as a remote controller to turn the camera and take pictures.

Remote collaboration

The visibility of the proposed wind farm is now more problematic than previously envisaged, and Lynda starts to wonder whether this may have an influence on the overall layout of the

turbines and viewpoints. With the GeoTagger still mounted on the tripod, Lynda now uses the mobile phone to create a data connection back to her home office in order to transfer pictures of the new visibility to show and discuss with colleagues. As the discussion unfolds, the colleague back home is able to remotely control the devices on the tripod (e.g. turning the camera and seeing the result).

Below, we summarize how the prototypes described here probe the PalCom open architecture. This corresponds to the first of the four participatory elements of the design of the open architecture introduced on page 5, how the analysis of work practice and of corresponding possibilities for technical support suggest requirements for an underlying software architecture.

2.1.1 Challenges

Continuous (Re-)assembling

All scenarios involve a continuous (re-)assembly and *(re-)construction* of services and devices. This is richly supported back in the office with appropriate prototype interfaces to make and show the device and service assembly. However, the work also calls for various disassemblies and reassemblies in the field with more impoverished resources, which must nonetheless optimise both making, and representing to the user (making *visible*) the assemblies that are in play. It needs always to be clear to what assembly (if any) a particular device currently belongs, what assemblies are in play, on what device a particular service is running, etc.

On-the-move

In all the scenarios, the assemblies in question will be in motion. This means that even if an assembly remains constant itself, its context changes frequently. An assembly must react appropriately to resulting *changes* – by, for example, notifying users if potentially useful additional resources become available, such as the processing power in devices in a car that has come in radio range or by switching communication channels when one drops out. This calls for appropriate choices and behaviour, and appropriate documentation of such choices and behaviours, on the part of the assembly.

Shifting modes of cooperation

The scenarios entail shifts in the actors in collaboration as well as in the modes of collaboration. This may require a change in the behaviour of an assembly, even though neither its constituents nor its physical environment has changed. It may, for example, raise challenges regarding privacy and confidentiality of actions and materials as well as challenges in relation to who operates what assemblies, support for collaborative work, the question of whether users are part of the assemblies, and how to make those relations *visible*

and *understandable*. The ‘character’ of an assembly depends on such intangibles as the people involved and their purposes.

Quality of Service

Different assemblies may be able to do the same things, but with different capacities, for example with different levels of accuracy. A high degree of accuracy is not required in all the scenarios, but it is very important in all of them that the user knows and is made aware of the given accuracy. If a landscape architect is taking photographs in poor weather, for general work planning purposes rather than as photographs for official records, she may decide that relatively inaccurate direction information, derived from GPS alone, is adequate. But she should not be misled, either in the present or when reviewing materials at a later date, that just because a compass direction is given, it has the accuracy of a digital compass reading. It may be appropriate to operate with an implicit assembly with regard to accuracy, where the assemblies ‘choose’ among several potential services offering location information, depending on which one is most accurate at the moment (this changes as one moves), but paying attention always to represent the accuracy available in the current state.

Un-anticipated use

In the scenarios above, we have anticipated a number of assemblies coming into effect during a rather short period of time. What is also expected from this family of situations is that it will produce a set of un-anticipated and unpredictable usages of the existing services and devices, thereby providing for unanticipated or emergent (serendipitous) assemblies and contexts. This in turn informs architectural discussions about whether ‘types’ may emerge during runtime or will be known at design time, whether it is just a matter of naming a particular assembly for one’s own later re-use, or whether it is a matter of sharing a new type of assembly among colleagues, etc.

3 Assemblies

In Section 2, ‘assemblies’ were discussed as a concept arising in and from practice and prototype design, and some consequent challenges for the open architecture were considered. But is the concept of assembly itself also relevant for the open architecture and, if so, how? In this section we use the development of the concept of palpable assemblies as a representative illustration of the ways in which the competencies of ethnographers, users, software developers, and software architects interact as part of co-design. In doing so, we analyze four instances of how the assembly concept has evolved, each explored from three different perspectives: software architecture, application development, and use. In section 3.1, we describe how a basic notion of assemblies was introduced to the open architecture. This prompted reflections on the relationship of assemblies to the more conventional software architecture concept of service composition, discussed in section 3.2. The challenges thrown

up by this highlighted the issue of assemblies as resource composition, discussed in section 3.3. Lastly, the developing centrality of assemblies foregrounded the need for means to browse and inspect them – to make them palpable – discussed in section 3.4. At all of these stages, there was a consequential interplay between the perspectives of end-users and work analysts, application developers, and software architects, corresponding to all of the four participatory elements of the design of the open architecture introduced on page 5.

3.1 Basic Assemblies

Landscape architects – like many professionals – routinely put together 'assemblies' of devices (the car, cameras, tripods, GPS, compass, maps, etc.) for particular jobs. To leverage some of the potential of computing technologies into this practice and to drive architectural design, users and work analysts (in collaboration with application prototype developers and software architects) began to talk about engagement with assemblies, components, and devices.

3.1.1 Software architecture perspective

The concept of an assembly was embraced by the architects and attempts were made at translating this concept directly into software architecture. A decision was taken to make the assembly a first-class object of the software architecture. A 'first-class object' in this context is a construction that users of the software architecture (e.g. application developers) may directly use to construct their programmes, e.g., through a set of specific classes in an object-oriented framework. The assembly was designed as a 'service' that had the responsibility of coordinating other services. In the context of the architecture, a 'service' is functionality (running in a process) that announces itself on the network and that can be accessed through messages to another process. In the scenarios above, examples of services by this definition are the GeoTagger, the displays, the 'take back service', etc. Furthermore, the assembly had responsibility for monitoring the state of the assembly in terms of the availability of the constituent services.

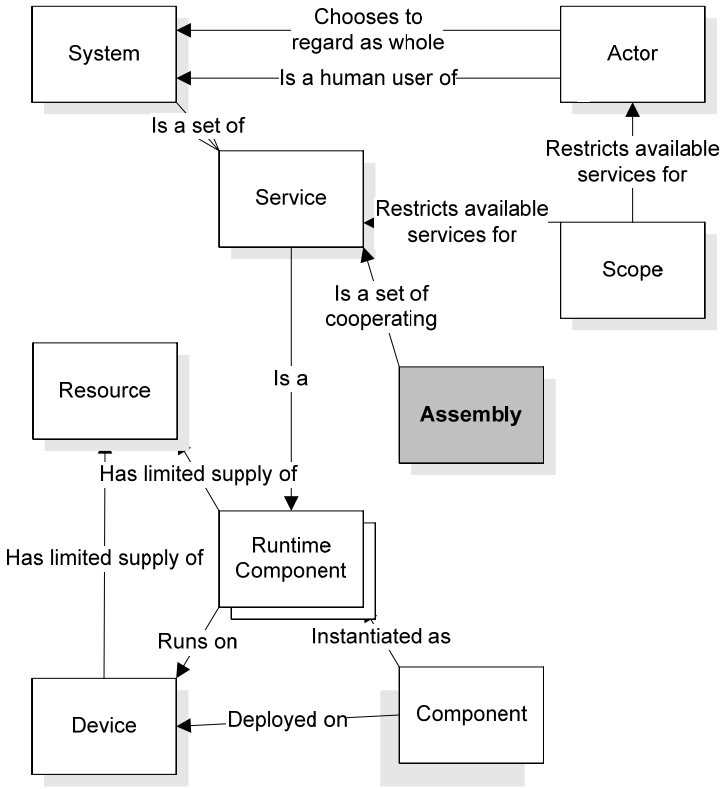

Figure 11 The central architectural concepts and their relations taken from the first complete version of the open architecture for palpable computing (from first project internal architecture overview deliverable in 2004). The boxes illustrate concepts and the arrows define qualified relationships between concepts. The concepts related to assemblies are highlighted.

In summary, Figure 11 shows the central concepts of the first basic palpable computing architecture. An Assembly is here seen as a set of cooperating Services which are each Runtime Components that in addition to being able to run on a device also provide the service capabilities outlined above. This design may be seen as a rather direct translation of the use-oriented concept into architectural concepts where the assembled parts are considered to be units of communication and functionality, or services, in a distributed system.

3.1.2 Application developers' perspective

Landscape architects' work on site is only one of several application domains explored with the aim of informing software architectural design in the PalCom project. The challenge for the prototype work is not to design 'perfect' special purpose prototypes in support of work in each application domain, but rather, to support the dynamic configuration and reconfiguration of a set of interacting devices into assemblies supporting a wide range of different usages, and thereby to challenge and inform the design of the software architecture. This means that the participatory design of the application prototypes themselves and concerns with their usability are a second order priority. A delicate balance has to be struck to develop realistic and functional enough application prototypes that allow users to appropriate and shape a socio-technical future where palpable computing is available, but that do not 'waste' valuable resources needed for the exploration of architectural design requirements. Prototypes may

remain ‘sketchy’, complex and fragile for longer than one would otherwise accept. They maybe ‘wrapped’, that is, run on a laptop simulating, e.g. a mobile phone, rather than instantiated inside an actual mobile phone, and consist of more parts and actions than is obvious to the user.

In the first iteration of the GeoTagger the assembly consists of a digital still camera, a GPS, a display device (e.g. laptop or PDA) and a mobile phone. When the camera takes a picture, it automatically notifies its surroundings of this. At the same time the GPS is constantly emitting world coordinates for its current location. A software component assembled with the camera and the GPS writes the current location information into the (meta-data part of the) image received from the camera. The updated image is then displayed on e.g. the PDA and simultaneously sent to a web server (typically located back at the office), utilising the Internet capabilities offered by the mobile phone,.

The SiteTracker, similarly, consists of four devices: a GPS, a display, a video camera and a digital compass. The GPS constantly provides location information, and the digital compass directional information of where the video camera is pointing. The resulting video footage from the camera is shown on a display with an overlay showing exactly where the point(s) of interest would be. The GPS that takes part in this assembly may be the same GPS as the one that is part of the GeoTagger assembly – it is acting as a service in different contexts.
From a use perspective, going from GeoTagger to SiteTracker or vice versa is a matter of disassembling and re-assembling a number of devices.
Development of the application prototypes takes place in parallel to the development of the open architecture, and for this reason the open architecture described in section 3.1.1 was in fact not the first architecture developed for these prototypes. For the first iteration of the GeoTagger and the SiteTracker, a prototype software architecture implementation, called Corundum (Ørbæk 2005), was developed by the application developers themselves. Inspired by the understanding of, and vision for, use developed through fieldwork and participatory engagement with users, the prototype software architecture implementation behind these first application prototypes focused on supporting five main concepts:

- assembly – a set of communicating services
- service – announces itself to its surroundings and communicates asynchronously with other services
- process – contains services and components and holds a hierarchical map
- component – a module residing on disk, can be loaded into a process
- hierarchical map – a tree-structured name space used to hold (most of) the non-transient data of a single process.

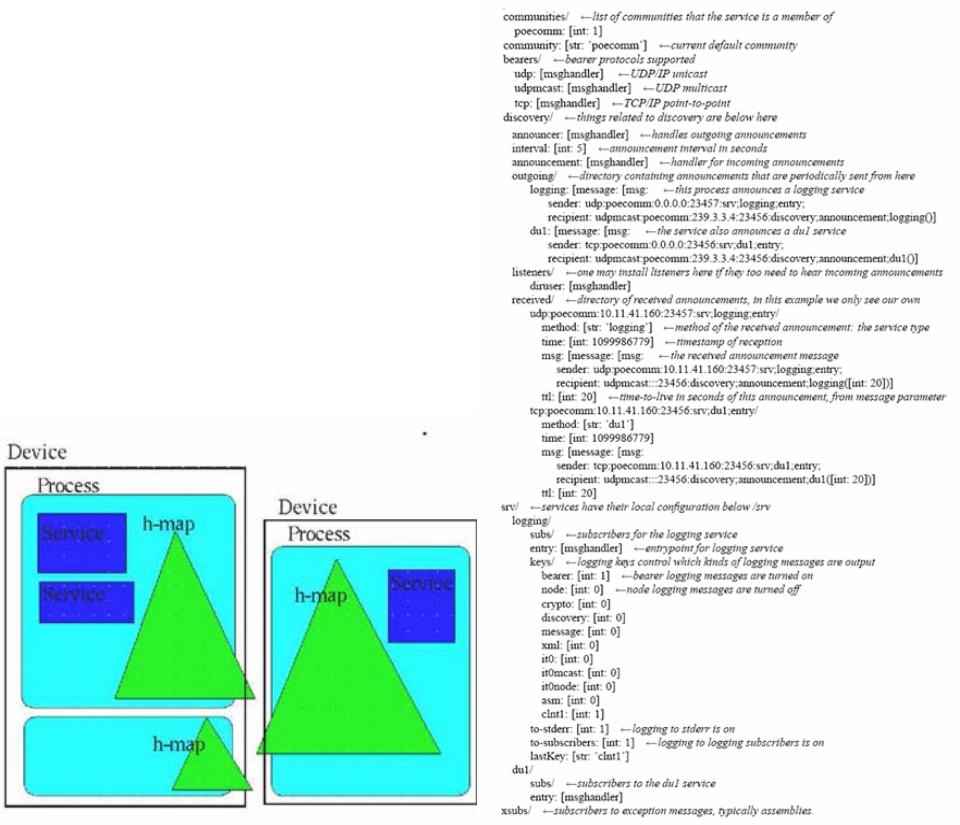

Figure 12 Hierarchical maps: 'Two devices, one hosting two processes each with their own h-map. The h-maps extend outside the devices to illustrate that they are accessible from the outside'. The listing on the right is a commented dump of the h-map of an isolated instance of a simple service (du1), in a situation where it cannot see other services. It is one of the simplest real-world examples. (Ørbæk 2005)

All components and services ran on Corundum (Bardram et al. 2004) which, 'encourages an *extrovert programming style*, where components and services expose what they can do (potential uses, events accepted and sent), what they are doing (eg. logging), and what they have been doing (history). This is all done via the h-map which is globally visible, and accessible from outside the process over a network.' (Ørbæk 2005). The Corundum framework differs from the first version of the open architecture described in section 3.1.1 in several respects. The devices that take part in the assembly are seen as a set of communicating services contained in processes on a network. Each of the services can be externally configured through manipulation of an externally visible hierarchical map. An assembler service also uses this hierarchical map when dynamically (re-)configuring the paths of communication necessary for a specific assembly configuration. Each process potentially consists of a number of services and components.

One of the main points here is that the technical infrastructure of these early prototypes is deeply influenced by a use perspective – devices have a number of external interfaces that

users configure in order to assemble the devices and to make them communicate. However, the story is, of course, more complicated than that – in order to make prototypes like the above do anything beyond the most trivial the need arises for more pure software components and services. In the case of the GeoTagger, there is a need for a piece of logic, for example, that combines the image and the coordinates. Since services are distributed and able to dynamically discover and use each other, this service can in principle reside on any of the participating devices. However, making an informed decision – by the user or (semi-) automatically by a run-time system – about which device to run such a service on requires some degree of software architectural support for visibility and inspectability of available resources (processing power, available memory, network bandwidth etc.) – all matters that were to become central to the Open Architecture.

Use perspective

Turning back to the fieldwork, the prototypes were put to use with two landscape architects to carry out some initial experiments (Figure 13). The SiteTracker, for example, produces useful, dynamic composite pictures that accurately track specified points in the landscape. This is first achieved in a static context. Subsequently, when the experiment is repeated in a moving car, the prototype continues to work accurately. Unfortunately, the soldering on the connection to the digital compass breaks after just a few minutes of driving. We experiment with the compass internal to the GPS, but it does not provide updates fast enough and the experiment has to be abandoned.

Figure 13 SiteTracker

In the course of the experiment a number of difficulties arise that are inspiring for redesign. We only outline difficulties and design implications for the software architecture, as our focus in this paper is on the participatory process, not the detailed design of the software architecture or the application prototypes (for more detail on the architecture design, see Andersen et al 2006).

Difficulty	**Implications for design or design process.**
Calibrating the compass and the GPS is awkward. Calibration seems to be fragile and requires frequent repetition of the calibration process.	Ways of detecting trouble caused by faulty calibrations and practices of testing the accuracy of calibrations should be supported.

The wrapped setup – with cables and laptop – is clumsy, and it is difficult to see anything on the screen in the sunlight.	To enable real users to experiment with the prototype in as realistic as possible use situations, a less complex design is required
The translation from GPS to Ordnance Survey (OS) coordinates is faulty. The cause is unknown. The problem is fixed by driving to a known point of interest and recording the position in OS coordinates.	The detection of faults in the computation, and ideally their causes, should be supported.
When trying to re-assemble the SiteTracker after a break, it turns out that a LAN/Wi-Fi type network has to literally be put in place before an assembly can be made. This is because, in order to exchange messages, services and assemblies at this stage require the presence of a network connection that supports UDP. On the Windows laptop this is only present if the laptop is connected to such a network infrastructure in the physical surroundings. Therefore, it is not possible to assemble using just the single laptop, the camera, the GPS and the compass.	Software architecture should not require connection to LAN/Wi-FI infrastructure in it's physical surroundings, as such infrastructures will typically not be present when on the move. Generally, the software architecture should be able to scale from working in infrastructure rich environments to the infrastructurally simpler environments..

A second round of experiments with a modified SiteTracker prototype takes place a few weeks later. This time, however, the developers run into a whole series of problems right from the start. These, too, reveal pertinent design issues:

Difficulty	**Implications for design or design process.**
When connecting a device it is sometimes necessary to find the virtual com port to which it connects in order to make the service communicate with the device via this port. The virtual com port number is dynamically assigned whenever such a device tries to connect – the com port may change depending on how many devices are currently connected.	When devices connect they should automatically acquire the necessary resources for establishing the connection. On the other hand, in case of a failure, there should be support for making such connection resources visible.
Currently an assembly is invoked via an XML specification in a file, that can be located on any one of the devices involved in the assembly, and the meta assembler – the service responsible for setting up and maintaining the assembly – then looks around and sets up communication between services, it does not start them.	There is a need for an overview of the assemblies available for launch and a mechanism to invoke an assembly in such a way that it automatically attempts to start the required services.
Connectivity is still required before an assembly can be made, even if only one computer is involved. Is this a design flaw in the Corundum framework? A constraint from Windows? An IP problem? If the computer on	The software architecture should support tools for monitoring communication paths and network traffic.

which a service is running does not have network connectivity, it is impossible to transmit messages.	
The SiteTracker loads points of interest from a configuration file on startup. The easiest way to add or change points of interest is currently to manually change the configuration file and then to restart the SiteTracker service. Corundum actually supports on-the-fly changes, but there are no tools to support actually doing it.	Better tools for inspection and change of the state of a running service. Use exposes a missing link between prototype and architecture.
There is no mechanism or user interface to see or select what configuration file the SiteTracker actually reads from when started. In this instance there are two different files, one with UK OS and one with Danish position information.	The options and selections should be inspectable. There could be a need for detecting and visualizing the physical context in which services and assemblies exist. This may also be subject to (semi-) automation, e.g. on the basis of location.
The translation between GPS and OS is wrong, but we do not know where it goes wrong.	It would be nice to be able to take a service out of its current assembly and network context and simply test it by 'poking' it with some input seeing if it comes back with a proper output.
It would sometimes be useful for the SiteTracker user interface to visualize the coordinates sent to it from the GPS service.	The basic state of any service should by default be able to be shown in a graphical user interface and it should be possible to dynamically combine and change user interfaces while services are running.
In the experiment the tracker 'hands' (Figure 5) jumped from one side of the display to the other. This could have several causes – the field of view could be too big, the point of interest could be behind, the coordinates could be wrong. In a later trial it turns out that this issue was caused by the assembly not being properly assembled – i.e. communication paths were not properly setup and the SiteTracker service was using outdated and flawed position and orientation data for its calculations.A further test in Aarhus reveals similar problems, but here the tracking is correct. This suggests that there are also conversion failures.	Again this calls for tools and architecture support for getting an overview of running services, their paths of communication and whether or not they are participating in a running assembly.

In general, the difficulties encountered in the use experiments show that there are more activities taking place, with more potential for things to go wrong, than were anticipated, which impact at the level of the architecture as well as the level of the prototypes.

3.2 Assemblies as Service Composition

As we saw in section 3.1.1, an assembly is defined as a set of cooperating services. In this section we consider the practical implications of this and how it should be realised, again from software architecture, application prototype, and use perspectives.

3.2.1 Software architecture perspective

A main conclusion from the use perspective regarding basic assemblies was that the open architecture should support introspection and visibility in various ways (section 3.1). Though this was always known in principle, experience from use enabled it to be given specific content. Section 3.3 explains how the concept of 'resources' partly helps meet the challenges. The software architecture was evolved to support this through a refinement of the idea of assemblies as sets of services, eventually leading to the realization that the software architecture also needs to support a more complete concept of assemblies.

As part of this refinement process, the investigation of the basic concept of 'assemblies as services', led to exploration and refinements of the assembly concept based, among other things, on what services are traditionally thought to be in software architecture (Szyperski, 1998). One example of this would be the classification of services as 'stateless' or 'stateful'. A stateless service does not retain a client-specific state (such as the latest GPS coordinate of a specific client) between uses of the service whereas stateful services may do so.

Such a distinction is important for (among other things) reasons of scalability (and understandability) of service composition and use in software architecture: if a service is stateless it may be replicated so that different clients access different instances of runtime components and conversely many clients may use a resource-intensive service concurrently. 'Scalability' is an example of an 'architectural quality' (Bass et al. 2003) that exemplifies architectural concepts and practice that are important in designing software architectures. Most architectural qualities correspond to architecturally significant 'external qualities' of ISO 9126 (ISO/IEC, 2001). In contrast, the qualities that participatory architectural design is also concerned with are qualities-in-use (effectiveness, productivity, safety and satisfaction as seen from the point of view of ISO 9126). As a result of field studies and workshops, the assembly concept as outlined above was thought to support desirable qualities-in-use. On the other hand, little stress was put on external qualities such as performance or scalability in the Corundum prototype and in the h-map implementation. Thus these qualities remained to be explored in the context of software architecture. The participation of 'travelling architects' in some of these use experiments (the third of the four participatory elements of the design of the open architecture introduced on page 5) helped to communicate the importance of the approaches adopted in Corundum, and to effect their transfer to the Open Architecture.

A further refinement of the assembly concept was the realization that assemblies (at this state of the project) could be thought of as primarily and mainly *service compositions.* As a consequence, it was considered to remove assemblies as a first class concept in the architecture: if assemblies were only (dynamic) service compositions, their realisation could have been expressed in terms of reference compositions of components/runtime components. However, for reasons discussed in more detail in section 3.3.1 below, it was decided that it was necessary and beneficial to leave the assembly concepts as a central and first-class part of the open architecture. The architectural refinement of the concepts of assemblies and services was then used in application prototyping as discussed below.

3.2.2 Application developers' perspective

The second major iteration of the prototypes involved a move towards a more fine grained service-oriented architecture as defined by the open architecture. Figure 14 shows a schematic of the first version of the SiteTracker assembly, combining a GPS service, emitting basic GPS location information, a compass service, emitting compass direction, and a SiteTracker service combining video images with location and directional information .

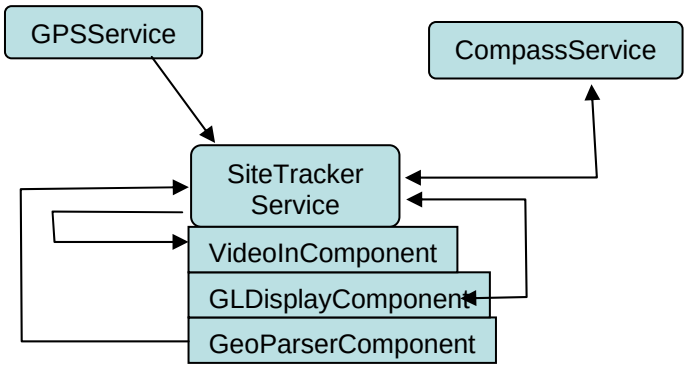

Figure 14 Simplified view of the original SiteTracker services, components and communication paths, showing that there are three services involved: GPS, Compass and SiteTracker; and that the last loads and uses three components inside it (arrows depict paths of communication)

Also integrated into the SiteTracker service was a so-called 'GeoParser' component. This component took raw GPS protocol strings (nmea-0183), parsed and converted them into a coordinate system that was appropriate to do the mathematics involved in locating the points of interest in the video image. Experience in use and other considerations (outlined below) suggested that this structure needed to be changed. The second version of the SiteTracker (Figure 15), for example, breaks the GeoParser functionality into two: a basic GPSParserService for parsing the GPS protocol strings (emitting coordinates in latitude and longitude) and a GeoConversionService for converting between different geographically

related coordinate systems. Furthermore, these functionalities were no longer loaded directly into the main SiteTracker service but instead acted as separate services in their own right.

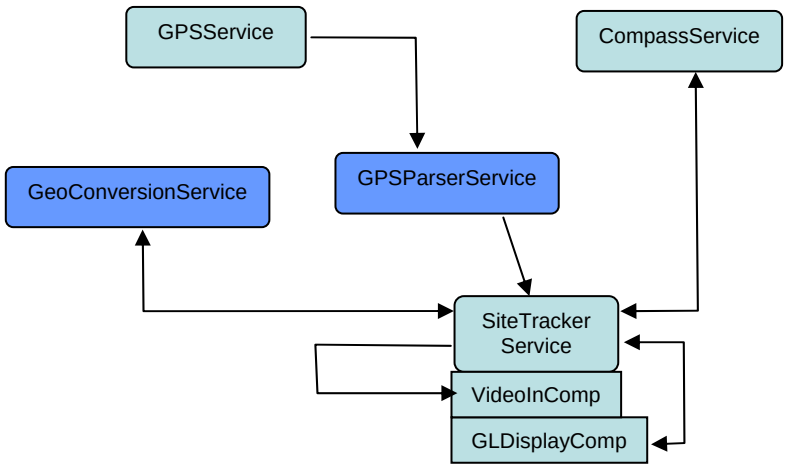

Figure 15 Simplified view of the second version of SiteTracker services, components and communication paths.

There are several reasons for this small but significant change in the software architecture of the prototype. First of all, in a general architectural context we wanted to further explore the scalability and service composition qualities of the prototypes. For example, if the main SiteTracker service is running on a resource constrained device, the conversion and parsing services can be deployed on separate devices in the network in order to achieve better load balance. Also, since the parsing and conversion services are more or less stateless, other services can dynamically attach to them and make simultaneous use of their functionality – saving having to load the component in more than one place and making efficient use of available computing power on the network. In the latest version of the SiteTracker this is put to practical use when a landscape architect wishes to supplement the augmented video image of the SiteTracker service with a digital map showing their current position and the positions of points of interest. This map service also needs to parse and convert coordinates and therefore looks up running versions of these services on the network and assembles itself with them in order to show the updated information.

As users of the SiteTracker, the landscape architects will not see any changes in functionality through this underlying change of architecture. However, as users of the software architecture, they (and the software developers), experience a significant improvement in relation to how flexibly the parts of the system can be composed, de-composed and deployed. This enhances the end users' experience, in that the assemblies and constituent services lend themselves to a richer set of options in relation to end-users composing their own assemblies.

Furthermore, the use experiences gained from the first experiments, as explained in section 3.1, strongly indicated the need for better tools for inspection and awareness in relation to services and assemblies and their context. At this point in time we therefore, firstly, initiated development of a basic tool for the browsing and composition of running services and assemblies in the network – an 'Inspector'. Secondly, the need for inspection of any single service on the network – potentially from a remote location – inspired initial work on the design of a framework for remote inspection and control of services.

3.2.3 Use perspective (application developers as users)

The developers of the application prototypes are closely engaged in the design of the open architecture. In fact, as we saw, they themselves designed a first prototype implementation of a PalCom open architecture, *Corundum*, in parallel to more comprehensive and conceptual efforts on the part of the software architects. They also use this and subsequent iterations of the PalCom open architecture as part of their development and implementation work and they are, therefore, an invaluable resource in the participatory design process. The goal of the open architecture is to support people from different walks of life, with different levels of 'computer literacy' and engaged in different situated activities in making computational states and processes palpable. The challenge is to enable the production of appropriate reflections of computational states and processes (Dourish 1995) or otherwise 'sensible' data. Software application developers are highly IT literate users. By examining their current practices of making computational states and processes palpable, and by engaging them in a participatory design process, important insights for the design of the PalCom open architecture can be gained. This corresponds to the last of the four participatory elements of the design of the open architecture introduced on page 5.

Figure 16 Developing applications and services on prototypes of the PalCom open architecture

On the right hand side of Figure 16 we see the SiteTracker and other prototypes working at a 'Future Laboratory' with users from another application domain – different emergency response services (police, fire brigade, medical teams) – at the emergency services training ground in Aarhus, Denmark. Future Laboratories enable users to 'colonize' and shape a

socio-technical future by asking and allowing them to accomplish realistic work with functional prototypes in as realistic as possible work settings (Büscher et al. 2004). The commitment to serious hands-on simulation and exploration of real world work enables embodied, practical creativity and reflection as well as participatory evaluation. Here, we have staged a car pile-up, paramedics are putting bio sensors and locators that will be part of the SiteTracker assembly on victims, someone is taking pictures of the victims. That data is sent to the trauma doctor in a prototype acute medical coordination centre. The trauma doctor needs to decide to which hospital victims should go, taking into account the nature of their injuries and the special skills at the different hospitals. Amongst other applications and services, the SiteTracker and GeoTagger are used to take pictures of the individual patients at the scene of the accident and display them on one of the screens in the Acute Medical Coordination Centre.

Future Laboratories foster the emergence and evaluation of future practices, which is particularly important when it comes to involving users in the design of software architectures envisaged to support the use of ubiquitous computing as it emerges over the next decades. Our series of Future Laboratories is still in progress and will be the subject of future publications. But the fact that Future Laboratories with end users require functional prototypes means that developers have to create, assemble and test them extensively, in effect inventing and evaluating emerging future practices of developing software applications – carrying out 'Future Labratories' of development work.

Below we present an analysis of events on the day before the Major Incidents Future Laboratory, when developers were making the prototypes work, coding, assembling, and addressing difficulties by making their causes 'palpable' wherever this is possible with the support of the prototype PalCom open architecture.

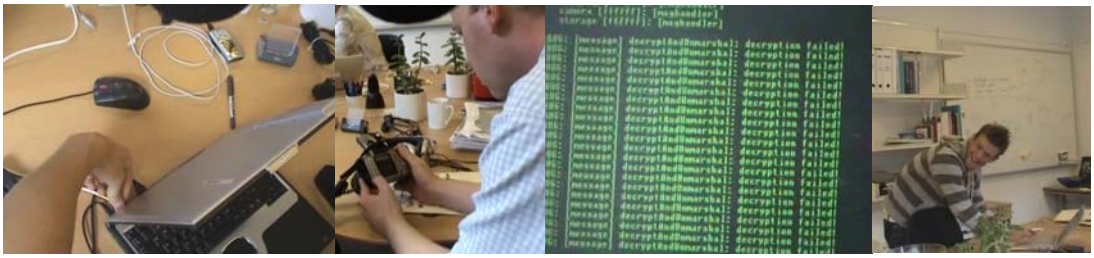

Figure 17 Assembling the SiteTracker

Jesper is assembling the SiteTracker. He looks, waits, then exclaims: 'What?' and reads out loud: 'no cameras are currently connected', reaching for the network cable as he speaks (Figure 17). Michael saw that the camera had stopped responding, and turned the switch to wake it up, but there are also messages about failed 'decryption'. They speculate about these errors until they hear Esben laughing behind them. To debug, Esben changed the Java version of the Corundum architecture prototype so it does not encrypt anymore. Because

Jesper is receiving messages from services on the Java architecture and his C++ version of the architecture tries to decrypt them, they are getting errors, but this is not what is causing the lack of connection between the camera service and the display service.

Jesper's hand reaches for the network cable again. Similar trouble before was caused by network problems. He leaves the cable plugged in, though, and does a number of things:

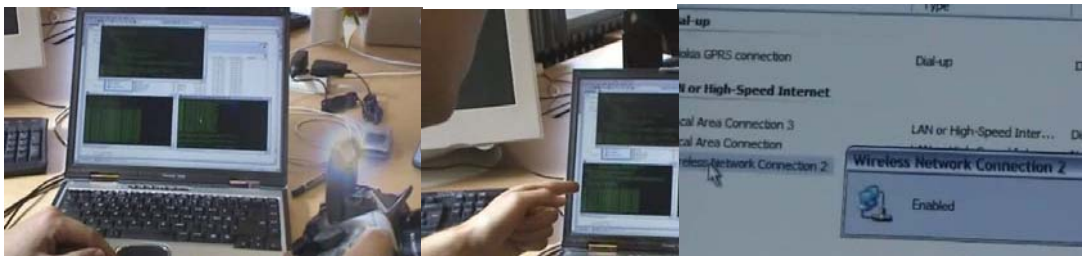

Figure 18 Trying to figure out what is wrong

He picks up the camera and takes a picture (Figure 18), and notes that where it should say 'get file', nothing happens, while the assembler says 'assembly is possible' and is, indeed, assembling. The meta assembler is adding subscriptions to services. Only the day before Jesper took five pictures and it went 'tick tick tick, they arrived with 'get file messages' …'. The network is still the prime suspect. Jesper unplugs the cable and switches to the wireless network, but to no avail. While Esben's sensor services are working nicely, Michael and Jesper are frustrated. They download and install a loopback adaptor, to create a 'one machine guaranteed functioning network' to check conclusively whether the problem is network related. This takes about 30 minutes. But again, they have no luck.

Michael suggests checking each individual subscription. They start the 'Inspector' and examine what is going on. It does not help, and desperation sets in. They restart the machine. Consider to drop the prototype from the experiments at the emergency exercise.

They wonder if there are too many images on the camera (which has also been a problem in the past), but again, no. Jesper explains their current understanding of the problem:

> *... for some reason the assembler doesn't finish the job. It doesn't set up the subscription between the two services. It can see both, it attaches to both and the next step is actually to set their subscriptions up and for some reason it fails that. So when I press the button, due to the fact that the camera does not have any subscribers it does not send a picture out on the network and then the service that is supposed to display it never receives it.*

The machine is back up and running, and it works … once. The second time nothing happens. Jesper suspects that he tried to take another picture too soon and waits a moment before he takes another one, and it works again.

When the GeoTagger and the SiteTracker work during the Future Laboratory with the emergency personnel, the developers notice a strangely long delay between taking the picture and it showing up on the display. A week later, at another demo, they figure out some of what is wrong. For example, when the camera takes the picture, it is so busy it stops sending even a heartbeat – a simple message saying 'I'm alive' to the other services. This breaks the assembly. As soon as the heartbeat is back, the assembly is re-established – but this takes time – and only when it is done can the overview service display the picture. The solution is to run the heartbeat in a separate thread or as part of the communication layer instead of sharing a thread with the camera data and processes.

This story informed a day long 'fieldstorm': a data session with application developers and software architects where the aim is to generate ideas for technologies that could support the work of developers in making the causes of failures (and possibilities for creative assembly) palpable.

The discussion brought out a list of methods of finding out what is going on

- The developers insert print commands into the code to produce messages (like 'assembly is possible', 'assembling', 'no cameras connected'
- People make amplifiers/translators for themselves (like the inspector)
- There is something like 'pattern recognition'. Flows of messages 'look right' when things are working and 'wrong' when something is wrong
- There are other sensory clues (e.g. the sound of Mac storage in infinite loop)
- There is categorisation: specific message types 'belong' to specific processes
- There is a strong sense of sequence and timing, which helps sense whether things are going well or badly
- There is a temptation to re-create 'good' (i.e. well known) environments where things worked even when that is not necessary
- People pose hypotheses of what might be wrong and falsify
- There is a temptation to test things one can easily test, especially under time pressure, and to ignore potential causes that are outside one's scope
- A lot of the process of encountering and dealing with trouble is made public by 'talking out loud to the machine' ('no cameras currently connected')

Finding out is a mixture of 'intuition, detective work, collaboration and trial and error'. The skills that some developers bring to the matter of computational potential are remarkable. However they are not just special talents, but also the acquired and honed result of everyday

practical engagement with computational technologies. Perceptual acuity and analytic proficiency can be trained. They rely on the reflexivity of interaction with computational matter.

The term ‘reflexivity’ as it is used here is inspired by notions of the spontaneous, ‘kneejerk’ reflexive reaction to physical stimuli, and the mutually defining, reflexive character of moves in human-human interaction highlighted by Ethnomethodology (Garfinkel 1967, Lynch 2000). It does not imply deliberate reflection. In interactions with each other, but also with technologies and the material world, people treat appearances ‘as “the document of,” as “pointing to,” as “standing on behalf of” a presupposed underlying pattern’ (Mannheim, quoted in Garfinkel 1967). In human-human interaction, this ‘documentary method of interpretation’ is sequentially organised and reflexive, that is, each move – each utterance, silence, gaze or embodied behaviour – is shaped by preceding and subsequent events. Each move prospectively informs the next and retrospectively shapes what has happened before. Each move documents a particular understanding of what is going on, and, as such, shapes the interaction as a whole – e.g. as an informal conversation, a meeting, or a medical consultation.

Although in human-matter interaction only one partner is sentient, engagement relies on similarly reflexive, sequentially organised moves and documentary methods of interpretation. Materials ‘document’ their processes or states and their ‘understandings’ of moves that their human or non-human counterparts make in the interaction. In everyday encounters with materials much of our human response to material moves becomes reflexive in the sense of automatic. The acts of perception, interpretation and response are unnoticed, what is perceived is a ‘flow’ of activities. However, human-matter interaction in science, medicine, sport, craft, engineering and many other activities amply documents that perception can be trained, that ways can be found to make materials whose moves are outside of the human ‘naked’ sensorium speak in a way that people can sense. The developers’ methods of making computational states and processes ‘speak’ by translating, amplifying and eliciting documentary evidence are instantiations of such practices.

It is a major aim, and a major challenge for our participatory design and research efforts to support advanced as well as ‘ordinary’ users’ practices of making computational states and processes palpable. Paying close attention to developers’ practices is one strongly informing strategy for palpable design. Participatory design with developers and end-users, based around hands-on engagement with prototype architectures and prototype applications and services, suggests that support for ‘reflexivity’ (as well as reflection) is a productive avenue for design.

3.3 Inspection and Awareness of resources

The troubles occasioned in using the prototypes repeatedly showed the need for computational states and processes to be made palpable and 'brought into the light' so that their operation could be understood and engaged with. In this section we indicate, in very brief outline, some of the ways in which this is being achieved.

3.3.1 Assemblies as Resource Composition

Some of the challenges outlined above may be handled by supporting a more detailed, fine-grained, and dynamic modelling and use of resources (e.g., the load level of a CPU such as in the camera example above) in palpable computing. The need for handling resources in a detailed way in palpable systems led to the inclusion of the concept of first and second order resources in the open architecture (Figure 19). Second order resources encompass a diverse set of concepts (among others services, actors, and communication channels). Second order resources, in turn, contain first order resources which are resources found in hardware and software layers below the palpable computing open architecture. Examples include memory, storage, and battery power. An assembly consists of collections of first and second order resources, and communication. It has a description (of how it is assembled and how it behaves when running) and is run on a computational node running the PalCom open architecture.

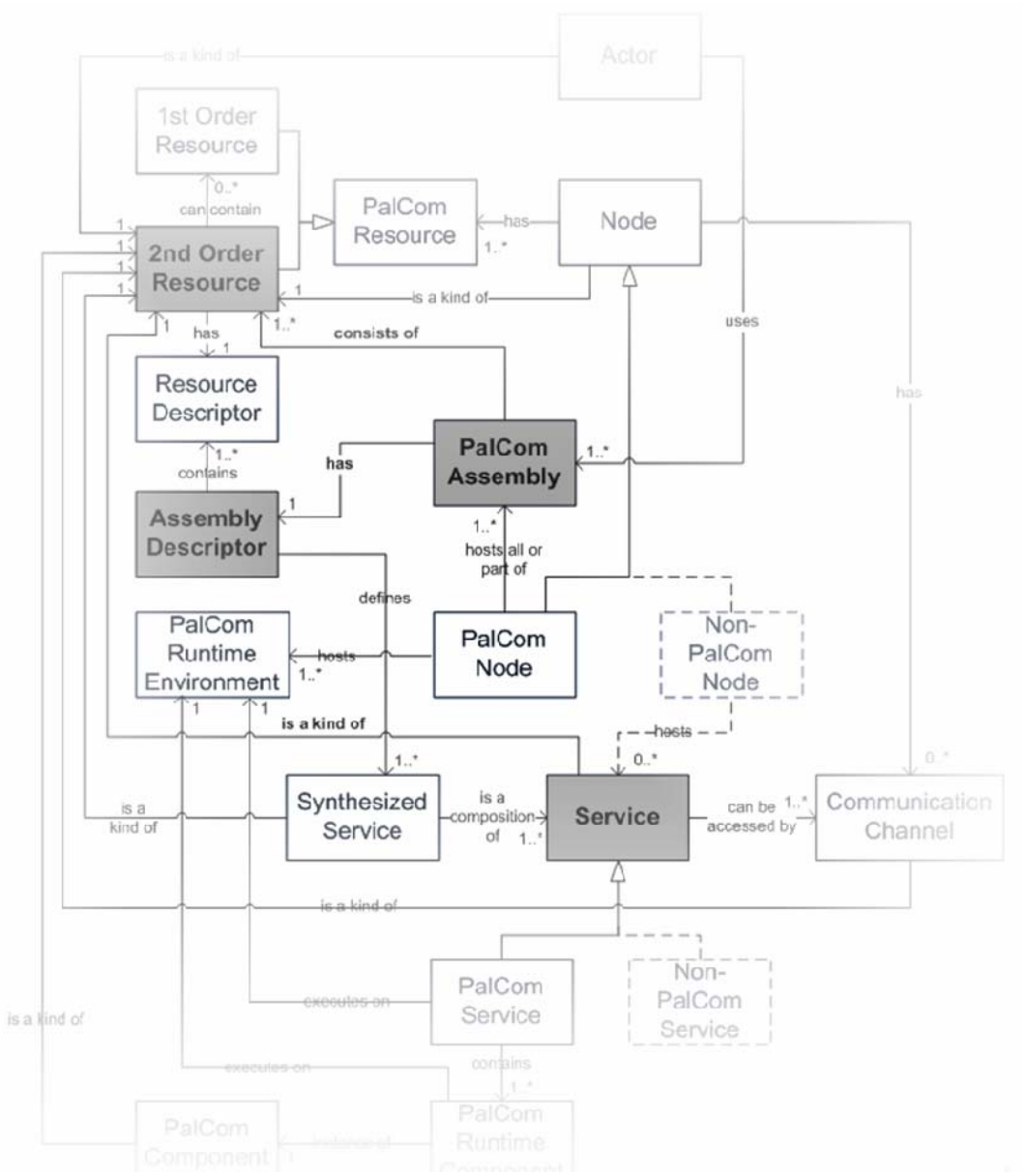

Figure 19 Central concepts and relationships from the second complete version of the palpable computing open architecture (from 2006 Deliverable, here focused on the concept of assembly, with edges blurred). Assemblies and related concepts are highlighted. The concepts are further developed and refined from the concepts shown in Figure 11.

3.3.2 Browsing Services and Assemblies

The experiences gained from developing and debugging the prototypes as well as the lessons learned and changes made in relation to the architecture, have together led to the development of a tool for browsing, combining and inspecting services and assemblies. This latest version of the tool is a reimplementation of the first 'Inspector' prototype of such a tool mentioned in section 3.2. The new version combines browsing and composition functionalities with capabilities for inspection of single services and assemblies. To do this the tool builds on top of the framework, also mentioned above, for remote control and display of services.
Via its graphical user interface, the tool shows all services running on devices in the networking context and lets the user inspect possible ingoing and outgoing connections of each service. Furthermore, the outgoing and incoming interfaces of services can be combined into assemblies and all currently running assemblies can be browsed and inspected.

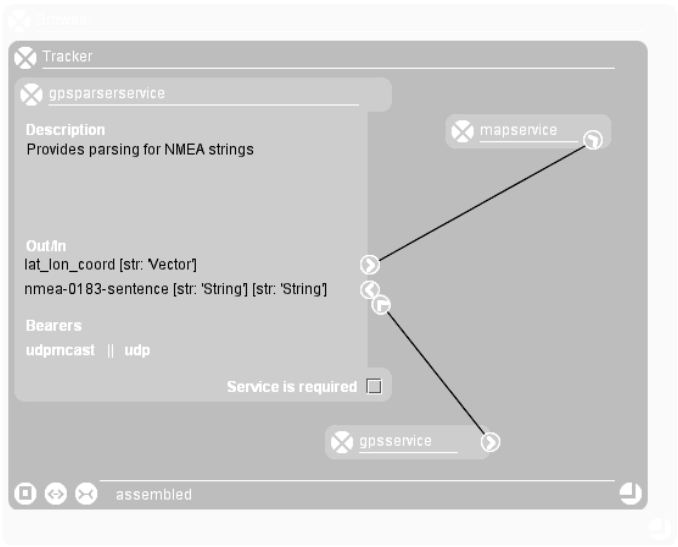

Figure 20 A screen shot of the current prototype implementation of a service and assembly browse and inspection tool

This is the functionality supported by the current prototype implementation of the tool, and plans are in the near future to extend the tool with abilities to e.g.:

- further inspect and change the state of single services – possibly with the option to isolate the service and test it in its own 'sandbox',
- visualize required and used resources for services and possible reconfigurations of resources in relation to instantiation of assemblies,
- monitor and filter data sent between services collaborating in assemblies, and
- further inspect the properties of the context in which services and assemblies exist.

On the one hand the construction of the tool and the functionalities added to it follow directly out of a simple set of demands stemming from the development and debugging of the prototypes described in the previous sections. However, the point is that in order to make the tool truly workable for everyday users from different walks of life, with different skills and engaged in different use situations, the underlying software architecture has to support such functionalities. By design, any service, for example, has to support inspection and allow for the change of its state at runtime. Different forms of monitoring, browsing and changing the behaviour of assemblies in context can be supported through the assembly concept with 1st and 2nd order resources, encompassing e.g. other services and communication channels..

In a broader context the development of the tools and the architecture supporting them is a way of attempting not only to reveal the materiality of digital entities, such as services and assemblies in a network, but also to provide a way of supporting the interplay and dialogue with such materials. Such support for reflexivity has a number of software architectural implications. In addition to the support for introspection of dynamic resources, i.e., the *present*, the dialogue with computational material can also be based on assemblies that have been used previously, i.e., the *past*, and with possibilities for assemblies in a given computational context, i.e., the *future*.

Supporting users in reusing past (templates for) assemblies points to the need for distributed storage. Given the inherently ad hoc network of devices and services in palpable computing, this requirement may lead to significant changes in the communication layer of the open architecture. If users of a set of services and devices in a given context should be supported in making informed choices about possible futures of assemblies, distributed storage should be augmented with more powerful semantic models of the capabilities of available resources/services. For example, given a set of services (such as a camera and a compass service), it should be possible for the open architecture to, e.g., support an application that suggests looking for other services (such as a GPS service) to create a SiteTracker assembly.

4 Pulling things together

We started out with the puzzle of what the relationship could and should be between software architecture design, application prototype design, and experience of use, and of whether these are amenable to an integrated participatory design approach. The material presented in this chapter shows that these ‘distant’ elements of a large project are indeed mutually informing, and can be made very productively so with some conscious focus and targeted methods.

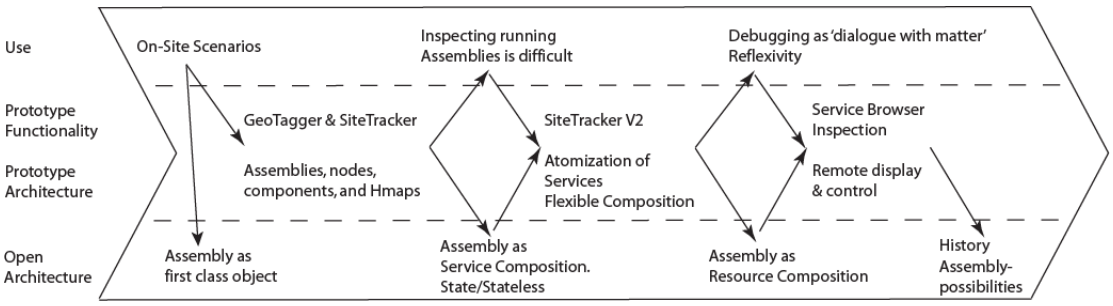

Figure 21 Map of the start of the participatory design of the PalCom open architecture

Figure 21 summarises our participatory open architecture design iterations up to this point. The cross-connections that emerged took various forms, some more general and some direct. We began with a set of scenarios, formulated through ethnographic fieldwork, participatory analysis and design workshops, prototype design and experiments. The scenarios envisage how landscape architects would assemble and use sets of devices and services whose processes and affordances can be made palpable, supported by the open architecture under design. Focusing on the real needs of skilled practitioners produced initial requirements for palpable computing that were more specific and problem oriented than could be expected from an attempt to consider 'ambience' in the abstract.

The application prototype designers could not wait for a considered open architecture to be ready, and so programmed an 'extreme' version of their own. Because this was available for early experiments with the prototypes in use, further problems, limitations and needs were exposed. Due in part to the cross-participation of personnel, going beyond conventional communication between software design sub-disciplines, these lessons were incorporated into the design for the Open Architecture, where they were generalised to the demands of other settings, and integrated with other practical and theoretical imperatives. Some of the lessons had a relatively specific focus, such as the atomisation of various services for the SiteTracker. Some were more far-reaching, such as the need for inspection and browsing services and the forms these could usefully take. At least one was quite structural: the adoption of the concept of 'assembly', originating in use, as a first-class object in the Open Architecture.

Architectural and application prototypes are now being taken to the test in more demanding realistic use situations. This requires as full as possible functionality and the application prototype developers engage in realistic testing and experimenting. This gave the opportunity to introduce a participatory design element among the computer scientists in the project themselves, by studying the ways in which application prototype designers made use of the evolving Open Architecture. As well as exposing further problems and needs, this demonstrated how in practice designers make software palpable, interrogating intangible materials in ways that bring them to sight and to voice. Observations of their practices of

debugging reveal a reflexive ‘dialogue with computational matter’ that relies on rich sensory feedback. Where such sensory data is unavailable, the developers devise means of translating, amplifying, manifesting computational processes. This provokes the need, and shows some of the possibilities, for ‘reflexive design’, and this is being addressed in the third iteration of the application prototypes. These allow users – in the first instance the developers themselves – to inspect, monitor and perceive, computational processes and affordances.

Our experience shows that the lessons learnt in ‘traditional’ PD, namely that by involving users more innovative and more viable socio-technical change can be brought about, are equally true when it comes to architectural design. The point we are at at the moment is a gateway to more direct end-user experience of how the open architecture does (or does not) support people in making computational affordances and processes palpable. By observing and by engaging application developers as users in a participatory architecture design process, we have chosen a perspicuous setting where we can study and experiment with current practices of making computational processes and affordances palpable. In doing so, we gain concrete insight into the constraints and possibilities for software architecture design for palpability. If developers cannot make things palpable with the support the prototype architecture provides, then end users would also fail.

At the heart of our approach is the observation that engagement with matter is reflexive. What this means is that we go beyond reflection. Reflection assumes that some designer somewhere can anticipate the situation and the computational literacy of the person needing a representation of computational processes. Whether the user’s ‘status’ is chosen by the user or ‘detected’ through context sensors, reflection assumes that designers can pre-prepare appropriate representations. While we ourselves engage in reflective design, we are certain that it is ultimately impossible to achieve appropriateness in this way. In parallel, we therefore also chose a radically different approach: by documenting material processes as ‘objectively’ and at as ‘atomic’ a level as possible, we provide ‘sensible’ data. We believe that there are already standards of producing such documentary evidence emerging, not only in our own work. People may not be able to perceive such documentary evidence with their ‘naked’ senses and not without training and acculturation. We build tools that can amplify, translate, manifest such documentary evidence. This, in turn will enable training and acculturation.

5 References

Andersen, P. et al. (2005) Open Architecture for Palpable Computing Some Thoughts on Object Technology, Palpable Computing, and Architectures for Ambient Computing. Ob-ject Technology for Ambient Intelligence Workshop, Glasgow, U.K. *Proceedings of ECOOP 2005*.

PalCom External Report 50: Deliverable 39 (2.2.2): Open architecture. Technical report, PalCom Project IST-002057, December 2006. Available at: http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-39-[2.2.2]-open-architecture.pdf

Armstrong, J. (1993) Concurrent Programming in Erlang. Prentice Hall

Bansler, J.P. and Havn, E. (2006). Sensemaking in technology-use mediation:Adapting groupware technology in organizations. *Journal of Computer Supported Cooperative Work* 15-55.

Bardram, J., Christensen, H. B., and Hansen, K. M. (2004). Architectural Prototyping: An Approach for Grounding Architectural Design and Learning. In Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), pages 15–24, Oslo, Norway.

Bass, L., Clements, P., and Kazman, R. (2003). Software Architecture in Practice. Addison-Wesley, 2nd edition

Bellotti, Victoria, Back, Maribeth, Edwards, W. Keith, Grinter, Rebecca E., Henderson, Austin, Lopes, Cristina (2002): Making sense of sensing systems: five questions for designers and researchers. In: Terveen, Loren (ed.): Proceedings of the ACM CHI 2002 Conference on Human Factors in Computing Systems Conference. April 20-25, 2002, Minneapolis, Minnesota. p.415-422

Belotti, V. and Edwards, K. (2001) Intelligibility and Accountability: Human Considerations in Context aware systems. Human-Computer Interaction, Volume 16, pp. 193–212

Büscher M. (2006) Vision in motion. *Environment and Planning A* 2006, volume 38(2) February, pages 281 – 299

Büscher M., Mogensen P., Agger Eriksen M., Friis Kristensen (2004) J. Ways of grounding imagination. *Proceedings of the Participatory Design Conference (PDC)*, Toronto, Canada, 27-31 July 2004 pp. 193-203.

Chalmers M, Galani, A. Seamful Interweaving: Heterogeneity in the theory and design of interactive systems. Proceedings of DIS 2004: 243-252.

Chalmers,M. (2003) Seamful Design and Ubicomp Infrastructure Proc. Ubicomp 2003 Workshop 'At the Crossroads: The Interaction of HCI and Systems Issues in UbiComp'.

Christensen, H.B., Hansen, K.M., Schultz, U.P., Ørbæk, P., Bouvin, N.O. Architecture Presentations: Experiences from Pervasive Computing Projects at Computer Science Department, University of Aarhus. Technical Report, 2004. Available from http://www.ist-palcom.org/

Corry, A.V., Hansen, K.M., Svensson, D. (2006). Travelling architects – A new way of herding cats. *Quality of Software Architectures (Lecture Notes in Computer Science 4214)* Berlin: Springer, pp. 111-126.

de Paula, R., Ding, X., Dourish, P., Nies, K., Pillet, B., Redmiles, D., Ren, J., Rode, J. and Silva Filho, R. (2005). In the Eye of the Beholder: A Visualization-baseed Approach to System Security. Int. J. Human-Computer Studies.

Dourish P. Developing a reflective model of collaborative systems. ACM Transactions on Computer-Human Interaction 1995 2(1):40-63.

Dourish, P. (2003) The appropriation of interactive technologies: Some lessons from Placeless Documents. *Journal of Computer Supported Cooperative Work* 12: 465-490.

Dreyfus, H. L. (1992) *What Computers Still Can't Do: A critique of artificial reason*. Cambridge, MA: MIT Press.

Edwards K, Belotti V, Dey AK, Newman MW. Stuck in the middle: The challenges of user-centred design and evaluation for infrastructure. *Proceedings of CHI 2003.*

Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A. The many faces of publish/subscribe. ACM Computing Surveys, Volume 35, Issue 2, June 2003, pp 114-131

Garfinkel 1967. *Studies in Ethnomethodology*. Polity.

Hartswood et al. submitted. Abstractions, Accountability and Grid Usability. NCESS.

Henning, M., Vinoski, S. (1999) Advanced CORBA programming with C++. Addison-Wesley

Ingstrup M, Hansen K M. A Declarative approach to architectural reflection. 5th IEEE/IFIP Working Conference on Software Architecture WICSA 2005

Ingstrup, M. and Hansen, K.M.. A Declarative Approach to Architectural Reflection. *WICSA 2005*

ISO/IEC (2001). Software Engineering - Product Quality. Part 1: Quality Model. ISO/IEC 9126-1.

Lynch, M 2000. Against reflexivity as a academic virtue and source of privileged knowledge. *Theory, Culture and Society* Vol. 17(3), pp. 26-54.

MacLean A, Carter K, Lövstrand L, Moran T. User-tailorable systems: Pressing the issue with buttons. Proceedings of CHI 1990: 175-182.

Parnas, D.L. On the Criteria To Be Used in Decomposing Systems Into Modules. Communications of the ACM, Vol. 15, No. 12, pp. 1053-1058, December, 1972

Rimassa G, Greenwood D, Calisti M. Palpable computing and the role of agent technol-ogy. Proceedings of Multi-Agent Systems and Applications IV, 4th International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2005

Stroud, R., Transparency and reflection in distributed systems, In *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring,* 1992.

The PalCom project http://www.ist-palcom.org/ 2006 February.

Szyperski, C., Component Software – Beyond Object-Oriented Programming. Addison-Wesley. 1998

Weiser M. The Computer for the Twenty-First Century. *Scientific American*, pp. 94-10, September 1991.

Weiser, M. 1993 Some Computer Science Problems in Ubiquitous Computing," Communications of the ACM, July 1993. (reprinted as "Ubiquitous Computing". Nikkei Electronics; December 6, 1993; pp. 137-143.)

Ørbæk P. Programming with hierarchical maps. Technical Report DAIMI PB-575, DAIMI, 2005. http://www.daimi.au.dk/publications/PB/575/PB-575.pdf