Technical Report No: 2002/08

Meeting the software engineering challenges of interacting with dynamic and ad-hoc computing environments

Henrik Gedenryd

Simon Holland

David Morse

2002

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

Meeting the software engineering challenges of interacting with dynamic and ad-hoc computing environments

Henrik Gedenryd, Simon Holland, David Morse
Computing Department
The Open University
Milton Keynes MK7 6AA, UK
{h.gedenryd, s.holland, d.r.morse}@open.ac.uk

Abstract

We argue that the normal circumstances for pervasive computing technologies will be dynamic and ad-hoc settings, in that the available technical resources will evolve and/or change frequently, rather than having been installed by design. We describe a second-generation software architecture for Ambient Combination [14], engineered to meet the software engineering challenges of achieving transparency of use under such conditions.

The architecture uses advanced software composition techniques closely related to aspect-oriented programming, along with computational reflection, to represent domain objects and their properties at a problem-oriented, high conceptual level. This architecture achieves a very good separation of concerns, while also providing the flexibility and extensibility needed to address the open-ended nature of these situations. This also enables the architectural building blocks to be flexible distributed across different machine configurations in an uncomplicated and robust manner.

1. The dynamic, ad-hoc nature of pervasive computing environments

A characteristic feature of pervasive computing environments will be that normal use will involve multiple computing devices working in concert [21, 22]. Another characteristic feature will be their dynamical nature: People and devices will move between different settings several times a day. Individual settings will have various people and devices flowing in and out of them continuously.

Today, configuring computers and peripherals to work together is a major problem of systems operation, perhaps even the main chore for an information systems department. And if a user today employs more than one computer to perform a task, then this counts as a rather exotic special case, deviating strongly from normal use. Even individual computers today need a certain amount of configuration work before they can be used productively.

The interaction of these two factors will turn the problem of device configuration for pervasive technologies into an insurmountable task. If configuration work were to be needed every time a user would want to use a different pair or set of devices in concert, this would present formidable disruptions to a natural work flow. Hence even small individual configuration requirements will accumulate to a point where productive use is replaced by a constant tinkering to achieve interoperability. (To a great extent that is true already today.)

However, even if we assume that technical configuration is made completely invisible to the user, transparency of use would still remain a distant goal. Dynamicity and interoperability would still present significant usability problems, for example in determining what devices, actions and other options are available at any moment, and how to specify either of them. The problems aggravate when the possibilities are distributed over more than one device: how do you determine what capabilities arise from combining two or more devices, and how do you specify such combined actions and the entities that they involve?

1.1. Lightly and strongly ad-hoc settings

Let us consider some scenarios to make these problems more concrete. Different settings can be placed on a scale according to how highly dynamical they are, and thus ought to exhibit these problems to different degrees. For example, the integrated audiovisual capabilities of future homes provide a setting that is probably only lightly dynamical: devices will need to interoperate and use will be intensive, but there will be relatively few movements and reconfigurations, in terms of the number of such changes per day.

Another type of physical environment offers both lightly and thoroughly dynamical scenarios to illuminate the problems we address. This is the media-augmented meeting room, which would have audiovisual capabilities permanently installed in it, for example board-size display surfaces, loudspeakers, and so on. (We will apply the convention used in [28] and use the terms tab, pad and board to refer to computer surfaces on different scales.)

In such a room, traditional one-person presentations would occupy the lightly dynamical end of the spectrum, and it serves to illustrate the basic aspects of the problems involved. The speaker is likely to be bringing her presentation materials with her on the one or more pads she has used to prepare them. The dynamics here lie in enabling the existing and new devices to work together, allowing the speaker to control them and to move material across or onto the wall boards in a flexible manner.

The problems of configuration, interoperability, and so on, all lie within the domain of technology, whereas the domain of people's interest in this case would be the material being presented. The speaker's interest is to communicate her message; dealing with the presentation support materials should not require her to deal with anything beyond this, or having to deal with it at a too low, technological level. She should be able to stay focused on her presentation task, and any technological aspect beyond her communication problem should remain in the background.

As anyone will recognize who has ever spent the first ten minutes or more of a presentation or meeting waiting for a portable computer, LCD projector and whatnot to begin working properly, the configuration problem is not trivial even for this lightly dynamical setting. However, it is at the other end of the scale that we will find the most difficult challenges. In a one-way presentation, it is the passive role of the audience that makes the setting only lightly dynamical. At most, audience members may need to get copies of the presented material onto their own notepads and make notes on them. Additionally, such a presentation is indeed a planned and (hopefully) well-prepared event, and it is therefore highly predictable. So much, in fact, that typically its format has percolated not merely into the interior design and fixtures of the room, but often even into the architecture of the building, as in the case of lecture halls and theatres.

Thus we should consider circumstances that are less formal and instituitonalized. At the opposite end of the spectrum we find a group work session. It may well be unplanned and spontaneous, and rather than meeting to communicate something that has been established in advance, the participants meet to get work done. Perhaps to discuss ideas or have a brainstorming session, perhaps to produce a report or some other joint work product. In this setting, unlike in a conventional one-way presentation, there are no formalized communication structures, so everybody is potentially an equal participant. Also, none of the participants may have brought with them any material that has been specifically prepared for this occasion, and that will serve as a structuring resource for the events that will occur (such as a prepared presentation). However, everyone present is likely to have brought along their working materials, agendas, etc., that may become useful during the course of the meeting. And even more importantly, the participants will be producing new materials in the session – sketches, notes, outlines and so on – some that will facilitate the work process as it takes place, but which won't be revisited later, and others that will turn out to be useful later on.

We will use the term "strongly ad-hoc" to characterize conditions such as these, since they tend to emerge spontaneously and be improvised on the spot. As a consequence, the actual circumstances of a strongly ad-hoc setting are neither predictable or planned, nor can they be prepared in advance. In very spontaneous cases, the participants may not even realize that they are improvising collaboration until afterwards.

Under circumstances such as these, the pervasive technology should support a highly dynamic work process, where anyone can draw and write on their own and others' work pads, move things between any of them, and project material onto wall surfaces if such surfaces are available. They should be able to bring resources with them into the situation and instantly use them productively, as though they were ordinary paper notepads and so on. The participants' interests are the topic and contents of the meeting and they should be able to remain in this domain, even when using the various devices.

1.2. Unpredictable vs. highly anticipated environments

The emphasis on the dynamic and unpredictable nature of these settings distinguish it from previous related work. The pioneering work on a smart video-conferencing room [7, 8] drew heavily on the known, static conditions of the room, and the designers went through several iterations of their design. The result was made highly automatic and transparent by allowing the technology to act "intelligently" on the specific circumstances of that particular setting. We believe that attempts to make the technology act "intelligently" would be inappropriate (as well as unfruitful) when the circumstances are highly dynamical and unpredictable by nature, and therefore cannot be anticipated and built into the system well in advance [11, 26]. Instead, we think that to work under highly unpredictable conditions, the technology should just stay out of the way, trusting that people will know what they want, and allowing them to "just do it".

There is one factor in particular that suggests that the emphasis on ad-hoc conditions may be important. If these computing technologies become truly ubiquitous, only a marginal fraction of them will have been set up by usability experts, or even by people with any basic usability awareness. To see why this is true, consider the technology that made "interaction design" truly ubiquitous. In the early 1990's, who would have thought that a few years later the worldwide web would have primary school teachers, your own fifth-grader, and the secretary of the local football club laying out screens, choosing icons and placing panes and widgets in windows? And similarly, the ubiquitous "multimedia designers" of today are office workers creating PowerPoint presentations. Analogously, if computing technologies ever become truly ubiquitous, they will be set up by people with no special training whatsoever, and they will be unplanned and unanticipated. From one point of view, they will be set up by no one at all: for all intents and purposes they will emerge by circumstance rather than by forethought, as new devices come to be placed there, one at a time.

Such rooms will not be intelligent by design, and this is why we believe that ad-hocness will be a prominent characteristic of truly “ubiquitous” computing technologies.

2. An architecture for representing actions and combinations

The above perspective of dynamic and ad-hoc settings has emerged from our previous work on direct combination [15] and ambient combination [14]. This is an interaction technique where the user specifies a pair of entities (two entities is the typical case; one or more is the general case). The system then presents a set of actions and options that are relevant for those entities. When the user selects an alternative, this will cause the action to be carried out.

The goal of our work described here was to develop a next-generation solution, which would go from a proof-of-concept prototype into a more flexible and realistic solution. In particular, it would need the ability to be applied to usability problems that are of general relevance, rather than problems chosen for the purpose of demonstrating the technique in itself. In the process of exploring and evaluating various potential scenarios, the conditions of dynamicity and ad-hocness (i.e. being open-ended, unprepared and unpredictable) emerged as the underlying factors behind the most interesting difficulties and problems that we encountered. The rest of the paper will describe how we addressed these problems.

2.1. The first generation architecture

Although it is not a very hard problem to find the alternatives that apply to a certain pair of entities, there is a challenge in finding an implementation architecture that is specified at a proper level of abstraction, and also extensible in that new and unanticipated devices can be added in a robust manner. This is best illustrated by the problems we encountered in our first prototype implementation. The natural way to approach this problem is by applying object-oriented techniques, so that each domain object is represented by one kind of object, and combinations are computed by asking the objects involved to specify what combinations they afford. This leads to the design of an “ontology” of the entities involved, in this case, pervasive computing devices and the virtual entities they manipulate (documents, media content and so forth).

So far so good, but many or most of the problems in the first-generation prototype came from the well-known limitations of inheritance, in that it doesn’t scale beyond being able to express a fairly small set of similarities and distinctions between different kinds of object. For the objective described here, this limitation arises in trying to finding a good way of representing valid combinations and their results. Most valid operations apply not just to a single pair of entities, but to a range of combinations.

To be precise, there is a range of valid entities on both sides of the combination, and one wishes to avoid having to specify the same operation in multiple places, which leads to a combinatorial explosion, and this is where the limitations of inheritance come into play. Even if you design the class hierarchy expressly to deal with this problem, single inheritance will only allow you to handle a few dimensions of sharing combinations, then it doesn’t scale any further. Similarly, if multiple inheritance were used, its known problems would apply roughly as they do elsewhere (cf. below).

2.2. A high level of representation

In our second generation we tried to address the breakdowns of the first generation. Instead of inheritance and classes, we use a scheme that is compositionally complete [13]. That is to say, it can be shown formally that this scheme is able to define all compositions that are theoretically possible (whereas e.g. single inheritance is restricted to the linear composition of a class and its superclasses). In this particular context it means that since the scheme imposes no restrictions on what can be composed, unlike inheritance it should not have a problem with scaling and handling the combinatorial explosion beyond any particular point.

This is a novel scheme and Ambient Combination was the third problem area it has been applied to. It bears great similarities to the schemes of aspect-oriented programming [10, 17], but it is also closely related to the prototype/delegation system used in Self [6, 27]. With respect to Self, the differences that make ours compositionally complete are relatively small, and immaterial to the present discussion. It also bears noting that this domain has proven to be less demanding than the previously addressed domains, and we have so far not needed to use any of the more advanced forms of composition that are available.

In this composition scheme the structuring principle is to decompose more complex entities into functional components (figure 1). Instead of arranging objects into a purportedly unbiased or neutral inheritance hierarchy, this takes an explicitly instrumentalist approach to organization: Objects are structured in a manner directly adapted to the function or purpose of the application at hand (while allowing for several different purposes to be reconciled, even potentially conflicting ones). That is to say, an object is decomposed into parts corresponding to its different functions (or purposes or roles) in the application. (We note that role-based design is a popular technique even with inheritance-based languages which do not provide language support for functional decomposition.)

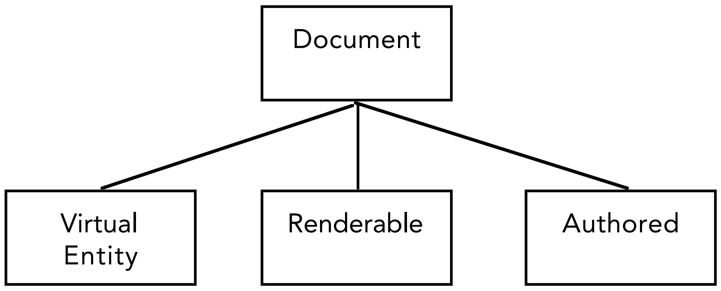

Figure 1. Decomposition into functional components

Applying this principle will almost by definition lead to a highly problem-oriented representation of the domain objects, and which is stated at a high conceptual level. In the present case, it comes to closely match the problem of finding applicable combinations. Consider the question of what it is that makes two entities combinable. For example, a document can be both printed and shown on a screen. Why is this the case? It is because a (digital) document has some properties that make it suitable for representation on a piece of paper. Let us call this the property of renderability (cf. figure 1). Likewise, a printer has a property, or set of properties, that enables it to print such things on paper. Thus, as one of its potential purposes or functions, a document is Renderable, and the primary function of a printer is to render digital entities that are Renderable. It is also the match between these properties that makes them combinable in a print operation: A Renderer can show anything that is Renderable. Lastly, a document can be shown on a screen as well because a screen can render viewable things just like a printer.

2.3. Performing the combination by computational reflection

This is the most high-level and conceptually direct explanation of “combinability” that we have been able to find, and the upshot is that our second-generation architecture for combination directly reflects this explanation. In the architecture, the combinable properties of an entity are given by a number of subcomponents that each directly represents such a property. For all intents and purposes, these components *are* the combinable properties.

Because the combinability in this way is reified and directly expressed in the structure of an object, the combination can now be performed by computational reflection or “meta-level programming” [18, 24]. In the inheritance-based version, an object would exchange messages with the potential combinee to see whether the two could be combined. This required the combination search to be implemented for each combinable class, reusing code in the restricted range of cases where inheritance made it possible. This is considered as the base or non-meta level, in that the code is implemented as usual in an object, or strictly in the various classes of the objects involved.

In contrast, in reflective code an object operates on a *description* of itself, i.e. on a meta-level or “looking at itself from the outside”. The description is normally given in the class of an object, in our scheme it is given by the composition graph. Thus the second-generation algorithm works by simply examining the subcomponents of the combined objects, listing the combinable properties it finds in each, and then identifying any matching pairs – these are the available combinations (figure 2). For example, the Renderer property of a printer matches any Renderable property and contains the operation “show the renderable object on my rendering surface”, and possibly more operations.

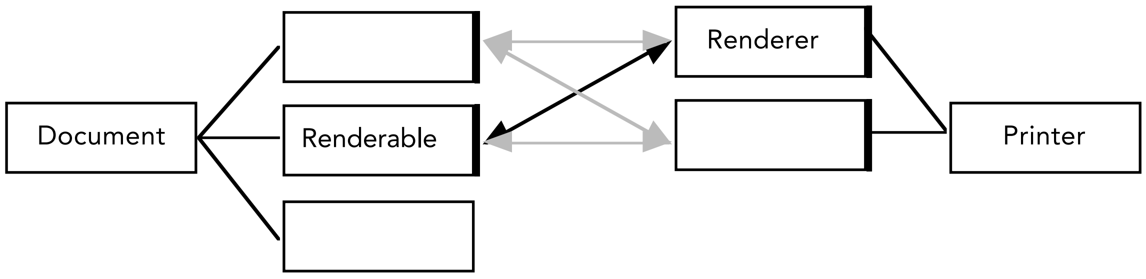

Figure 2. The combinable properties are marked with a black bar.

This algorithm separates the algorithm of the combination search, which is implemented once and only once in its own functional component, from the information about combinations, which is given as combinable properties in each object and is not tangled with any search code. This means that no code needs to be added, nor be changed, to add new combination possibilities, you just specify new types of object with old and/or new combinable properties. In contrast, the first version tangled the search code with the combination information, and distributed it over the combination search methods that existed in each of the involved classes.

2.4. The technical difference with respect to inheritance

As decomposition into functional components is the normal way to decompose and relate objects in this scheme, it would also be used for an implementation of the basic functionality, even if there were no combination facility: All functionality related to rendering would be given in a Renderable subcomponent of a Document (or anything renderable), and functionality regarding e.g. authorship would reside in an Authored subcomponent. Thus, combination information will match the normal structure of the implementation rather than requiring a class hierarchy designed specially for combination.

However, it also means that not all functional components will be combinable. Instead you make some out of all the regular functional components be Combinable – you add a Combinable aspect to them, cf. figure 3. (Here, an aspect is effectively just another term for a functional component.) In this aspect you provide the content that is needed for performing combinations: You specify what corresponding Combinable it matches with (this is a one-way relationship), for example, Renderer combines with Renderable, and what methods in the basic functional component should be available for combination (e.g. the method for "render").

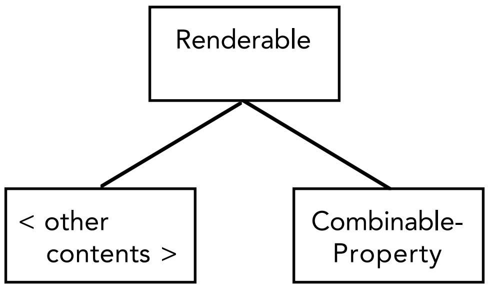

Figure 3. A functional component is marked as combinable by having a CombinableProperty aspect (which corresponds to the black bar in figure 2).

And herein lies the crucial difference with respect to inheritance, for which the explanation needs to become somewhat technical: Combinable is a property of Viewable, but it is not also a property of the Document. In other words, it is the Viewable property that is combinable, not the Document. On a technical level, the Combinable property is encapsulated within Viewable, so that something Viewable will not also become Combinable. This encapsulation means that there can be multiple functional components having Combinable properties, but that the specific contents of each of these would be kept apart.

This is the part that multiple inheritance cannot handle, in fact it is a case of its best-known weakness, the diamond-shaped inheritance problem [5]. For example, if we would emulate the above scheme and have e.g. Document inherit both RenderableEntity and AuthoredEntity, which in turn both inherit from CombinableEntity, you wouldn't be able to separate which one to take the combination information from (in actual fact, both of them should be used, each separately).

The cause of this problem for multiple inheritance isn't that it lacks aspects or some such, but simply that inheritance doesn't provide any mechanism for encapsulation of superclasses. It would even be a contradiction in terms since the difference between inheritance and regular composition is that the contents of a parent also becomes the inherited content of any child – this is the opposite of encapsulation. (This is also the technical significance of the *parent* property in Self.) Thus a crucial difference between functional decomposition and inheritance is that functional components can be encapsulated – in reality it is the normal case, and you have to specify that a component should not be encapsulated. Lastly, in answer to the objection that language X or Y could obtain a similar effect, we note that any Turing-complete language can do so, including raw machine code of course. The leverage comes from having direct support for it in the language constructs and tools.

3. Keeping technology in the background by using Perspectives

The next feature of the architecture is the use of *perspectives*; perhaps the last remaining secret of Xerox PARC [4, 16]. This mechanism allows an object to be regarded from several different perspectives, *as though* it were different objects with different implementations. Conceptually, this elaborates the instrumentalist approach of decomposing an object into different subcomponents implementing different functionalities, in that it allows even potentially *conflicting* functionalities or purposes to be reconciled within a single object. (There may or may not be sharing between perspectives, similarly there may or may not be conflicts between perspectives.)

Perspectives may be useful to draw on multiple sources for combination information, for example to support multiple competing standards for similar functionality. It is also useful for providing differentiated access, for example based on personal preferences or different security clearances. Here, however, we will illustrate its use in realizing a central usability objective: allowing the user to deal directly with her objects of interest, by pushing the pervasive technologies into the background.

To attain directness and transparency of use, a user should not be forced into taking the technological point of view of her environment. Instead, actions and options ought to be represented within the realm of the user's activity, in a way that is relevant to her concerns. We will refer to these respectively as the technology and user perspectives. In straightforward cases this means that for example an active badge would not present itself as a piece of identification technology, but instead directly represent the person wearing it. Similarly, in one early combination scenario, two people with Tabs meet, and a combination of the two tabs might offer exchanging business cards or contact information (at a technical conference), or offer to schedule a meeting at a time that suits both people (in an office corridor). Here it would be less appropriate to base the combination on the two items as technological items, than on what they mean to the user: "give her my business card", versus "beam my digital business card via infrared transmission onto her Palm Pilot". This corresponds to the distinction between the two perspectives.

This applies to the implementation as much as to user presentation: Offerings related to e.g. contact information should be offered by any artifact that can represent a person, say active badges as well as Tabs. However, the contact information should not be associated with these technologies even in the implementation, but rather with the people that they represent. This is why object perspectives are useful: they enable actions and combinable properties to be organized in the most appropriate manner.

Beyond the straightforward uses, there are a few complicating factors. In certain cases a user *does* want or need to deal with the technology "as technology" in some sense. One may also want to give the user a choice of what perspective to apply. Another factor is that the resulting perspective may depend on the combination that is being made. However, this factor may work in the user's favor – sometimes there will be no applicable combinations for one or more perspectives, so these perspectives will simply not come into play.

In any case, the consequence of these issues is that combinations sometimes need to be collected for multiple perspectives at once. Our solution is that technological perspectives typically serve as *representatives* for entities in the user domain. Thus, references to for example a pad are normally not treated as such, but instead as referring to the entity it displays, say a document – an object in the user's domain. Here, the pad is a representative for the document, and in this way the technology perspective is kept out of the way most of the time. However, combinations are always performed for both the pad and the document. If there are matches for both, the user-level object is given priority but both options are given.

For illustration, consider a scenario that is not too complex – a regular presentation. You have your presentation on a pad, plus more than one wall projection surface (board) at your disposal.

You want one wall to show a certain slide, say an overall diagram, whereas the other wall surface should be continually updated to reflect the material on your pad as you flip through the slides. (A similar thing is sometimes done with two projectors today, but it requires careful planning and execution and often involves a second person controlling the projectors.)

For both walls, the combination would involve your pad and the wall surface in question: “put this … there”. However, the first desired action is “*show this page* on that surface”, the second is “*mirror this surface* on that one”. The first reference is to the represented document, your presentation. Here the technology (the pad) as such is mapped into the concern of the user (the presentation), which is the normal case. The second reference, however, treats your pad as technology, that is, as a computer with a display. The two references point to different objects in the object model (the physical pad and a virtual document), and the technical operations carried out underneath in each case may be widely different. In both cases, the user combines the same two objects. This ambiguity is directly handled by the perspective/representative mechanism, in that it performs the combination query using all the involved perspectives, and finds (at least) two possible combinations: one between the page and the surface, and one between the pad screen and the wall surface. The former combines two technology perspectives, the latter one technology and one user perspective. While there may be simpler ad hoc schemes for this particular example, the perspectives mechanism provides a solution to the general problem.

4. Allowing a flexible distribution of the software elements

The last piece of the architecture is the one that compiles and coordinates information about the entities that are available. In its simplest form this amounts to having a Combination Server. It would have the following principal functional units (cf. figure 4):

1. The environment database. This is where devices register their presence, and where object tags are converted into full representations of the entity in question.

2. The combiner, which computes combinations from reflective descriptions of the combined objects.

3. The option selector, essentially a simple menu which presents the resulting options to the user, and collects a response and returns it.

4. The command performer. Technically, there is a unit responsible for carrying out the chosen option, although this unit is not strictly relevant here.

However, for several reasons it is desirable to have a greater flexibility than such a plain server solution. In an infrastructure-rich environment, typically in office environments, combination clients may profitably be thin, i.e. small, cheap, and depending on others for doing the heavy lifting, whereas in roaming environments (i.e. outdoors) it would be desirable to have a “thicker” combination wand that may carry the object models and databases as well as perform the combination operation, and thereby does not rely on other devices for its function.

Another desirable option is the ability to draw on multiple distributed information sources. Under realistic circumstances it is likely that the representation of the environment will be distributed, so that some sources will hold information about available entities, and others serve as sources for deeper information about the capabilities of entities that may appear, for example in manufacturers’ databases. To deal with heterogeneous information, you couple distributed information sources with different perspectives. This yields the ability to draw on incomplete, inconsistent or even conflicting information.

For these reasons a generalized, more freely distributable and reconfigurable architecture is desirable, having minimal ties between the various functions that must be performed and the specific machines or setups that perform them. To make the units easily redistributable, they should be separable and simple to link into different configurations, e.g. across different machines. Because of the highly portable cross-platform software used (see below), moving the functions is not a problem, so the problem comes to hinge on easy linkage. In this respect, transporting object tags to the environment database and transferring options to a user's device for selection are both very simple tasks.

The remaining difficulty is to allow the combination operation to be performed in a distributed manner. Since the first version performed combinations by executing code within and between the objects involved, this would have placed rather high demands on a distributed solution. In effect, a full CORBA-like facility would be needed to allow objects residing on different machines to communicate via message-passing. Needless to say, if any device that could be potentially involved in a combination would need to have a CORBA solution, then this would have made the entry barrier much higher. Furthermore, such distributed computation would place rather strong requirements on the speed and stability of connections. The alternative approach of transporting the objects representing the devices to be combined (including the code to be executed) would have relaxed the physical communication requirements, but the other problems of executing code in distributed objects from disparate sources would remain.

Fortunately, the reflective solution of our second generation made it possible to avoid these problems entirely. Since the combination is performed on the reflective descriptions of the involved objects, all that is needed is the ability to find out whether descriptions match. That is, to perform a combination on a distributed basis, all that is needed is the ability to transfer static descriptions of the properties that should be combined, and what you need from such a description, in turn, is for one Combinable property to recognize the property that it can be combined with; typically a tag of some kind is sufficient.

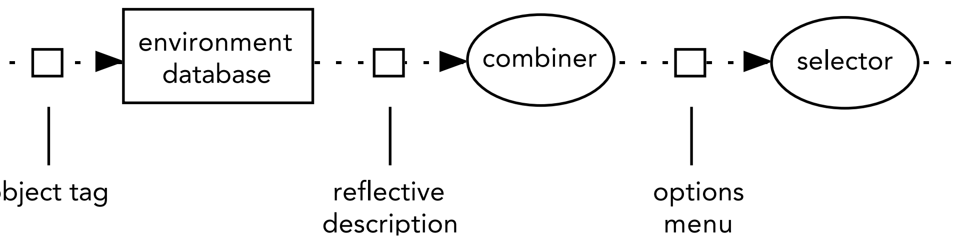

Figure 4. Simple data formats between each step of the combination process mean that the different parts can be easily distributed.

In all, this guarantees that combination queries can be easily redistributed into different configurations. Moreover, the baseline requirement for a device to participate in combinations is quite low. At the lightest end of the spectrum, it only needs to somehow provide a description of itself. The reflective information need not be provided by the device itself, it could be retrieved by a trivial query from a remote database, from a device identification of some sort. Moreover, there are no requirements for synchronous communication or even very good response times, because devices don't need to "talk to each other" to determine whether they can be combined.

4.1. The technical solution

Since our emphasis is on usability issues rather than on the technology of pervasive computing, technically sophisticated solutions are not an end in itself. Our initial distributed solution uses HTTP for the transport, and encodes the reflective descriptions in XML. The latter is in general a poor choice for pervasive solutions, in that mere text and particularly the verbose format of XML consume more bandwidth resources than necessary. However, the ability to use something as simple as XML to encode the reflective descriptions does serve well to illustrate our point: A very low degree of technical sophistication is required of a distributed, reflection-based combination architecture.

The implementation uses Squeak [2], an open-source version of Smalltalk. Unlike for example Java and C++, powerful reflective capabilities are part and parcel of the Smalltalk language as well as the environment and tools, which makes the reflective solution very straightforward to implement. The extreme portability of Squeak, and its platform-independence down to the bit level, means that all parts of the system (including the server functionality) can be directly transferred to iPaq-class devices with no porting effort. The server uses the core of the Comanche web server framework [1], hooking our object model directly into the HTTP request mechanism.

5. Related work

Rekimoto has previously noted the importance of inter-device operations for pervasive computing [21]. However, Pick-and-Drop [21, 22], the InfoStick [19] and DataTiles [23] do not have the open-endedness that Ambient Combination allows, they only deal with a restricted range of built-in inter-device operations. Consisely put, Ambient Combination is to Pick-and-Drop what Direct Combination is to Drag-and-Drop on the desktop.

Media-enriched meeting rooms have been addressed by several research projects. The relation of our work to that of Cooperstock et al. [7, 8] has already been discussed. The iRoom project [12] involves a media-augmented room and multi-device interactions. However, they focus on the technological infrastructure and do not aim to achieve transparency of use, or to push the technology into the background. For example, their room is operated via a PC-based web browser where you control the technology by clicking on links on specially prepared webpages, with links such as *Turn on all projectors* and *Switch Smartboard 1 to laptop drop*.

Abowd [3] pointed to the need for separation of concerns and software engineering techniques for pervasive architectures, and described how the CyberDesk project encountered problems by not being able to properly disentangle context gathering from context-triggered behavior. Previous work on architectures for ad-hoc settings include the Proem project, which aimed to provide infrastructure for building special-purpose ad-hoc collaboration applications [20]. Our aim is to support *strongly* ad-hoc conditions; specifically, if there is specially collaboration software that has been prepared in advance, then the collaboration is arguably less ad-hoc already. The Aura software architecture [25] also addresses dynamically changing resources, but it does this by explicitly representing the user's task, and it centers on the technical challenges, in effect on the level of infrastructure, whereas our concern is on the level of user interaction.

Our composition scheme is closely related to Aspect-Oriented Programming and other approaches to software composition [9, 10, 17]. These in turn are related to earlier work on object-oriented perspectives [4, 16]. The scheme also bears great resemblance to the prototype/delegation scheme of Self [6, 27].

6. Conclusions

The architecture we have presented uses novel software composition techniques to meet the requirements of dynamic and ad-hoc environments and to address some classical software engineering problems:

- The composition scheme guarantees that new functional properties can always be kept completely separated from existing ones. Because of its compositional completeness, the scheme can even guarantee that no property will ever need to be combinable with more than exactly one other property, which is the ideal case. This ability to attain an ideal separation of concerns directly addresses the scaling problem.

- Since combinations are computed reflectively from high-level properties, no information about other hardware (or any such dependency on foreign devices or entities) need to be hard-coded into any participant in the combination scheme. Thus, there is a strong separation of technical data from functional properties, and likewise the combination algorithm is well separated from the data it uses (combinable properties). Moreover, the separation of combination data from the computation also makes the system easily distributable across different machine configurations.

- Perspectives are used to improve usability by keeping the technology in the background. This can also be made to deal with heterogeneous information that is likely to arise in complex, ad-hoc settings.

- You can add new and unanticipated devices in two respects: Firstly by equipping them with existing functional properties, which would allow them to be combined with existing devices that combine with these properties. Secondly by creating all-new properties that add entirely new capabilities, and which can still live side by side with any older properties.

- Combination functionality can be added entirely non-invasively, by attaching Combination aspects to the basic functionality. The result is that combination information is supplied in a form that resembles an annotation scheme. Thus one does not have to design class hierarchies expressly for combination, but can instead use the regular decomposition approach of this scheme, i.e. functional components, in a way that is no different from what would have been the case if combination hadn't been considered at all. This also makes the architecture robustly extensible.

However, we wish to conclude by pointing out what enabled us to meet these software engineering concerns, which are all rather technical by nature: our decidedly nontechnical aim of attaining a conceptually direct and maximally understandable representation of the problem domain.

References

[1] *The Comanche web server framework*, : http://minnow.cc.gatech.edu/swiki.

[2] *Squeak*, : http://squeak.org/.

[3] Abowd, G.D. *Software Engineering Issues for Ubiquitous Computing*. in *Proceedings of the 21st International Conference on Software Engineering (ICSE-99)*. 1999.

[4] Bobrow, D. and T. Winograd, *An overview of KRL, a knowledge representation language.*

Cognitive Science, 1977. **1**(1): p. 3-46.

[5] Carr, B. and J.M. Geib. *The Point of View notion for Multiple Inheritance.* in *Proceedings of ECOOP/OOPSLA 1990*. 1990.

[6] Chambers, C., *et al.*, *Parents are shared parts of objects: inheritance and encapsulation in Self.* Lisp and Symbolic Computation, 1991. **4**(3): p. 207-222.

[7] Cooperstock, J.R., *et al.*, *Reactive environments.* Communications of the ACM, 1997. **40**(9): p. 65-73.

[8] Cooperstock, J.R., *et al. Evolution of a reactive environment*. in *Proceedings of CHI-95*. 1995.

[9] Czarnecki, K., *Aspect-Oriented Decomposition and Composition*, in *Generative Programming: Methods, Techniques, and Applications*, K. Czarnecki and U. Eisenecker, Editors. 1999, Addison-Wesley: Reading, MA.

[10] Elrad, T., R.E. Filman, and A. Bader, *(Eds.), Special issue on Aspect-Oriented Programming.* Communications of the ACM, 2001.

[11] Erickson, T., *Some Problems with the Notion of Context-Aware Computing.* Communications of the ACM, 2002. **45**(2): p. 102-104.

[12] Fox, A., *et al.*, *Integrating Information Appliances into an Interactive Workspace.* IEEE Computer Graphics and Applications, 2000. **20**(3): p. 54-65.

[13] Gedenryd, H., *Universal Composition*, . 2001: Manuscript.

[14] Holland, S., D.R. Morse, and H. Gedenryd. *Ambient Combination: a New User Interaction Principle for Mobile and Ubiquitous HCI*. 2002. Submitted for review.

[15] Holland, S. and D. Oppenheim. *Direct Combination*. in *Proceedings of ACM CHI'99*. 1999.

[16] Kay, A., *The Early History of Smalltalk*, in *History of Programming Languages–II,* T.J. Bergin and R.G. Gibson, Editors. 1996, ACM Press: New York. p. 511-578.

[17] Kiczales, G., *et al. Aspect-Oriented Programming*. in *Proceedings of the European Conference on Object-Proented Programming ECOOP'97*. 1997: Springer Verlag.

[18] Kiczales, G., J.d. Riviéres, and D.G. Bobrow, *The Art of the Metaobject Protocol*. 1991, Cambridge, MA: MIT Press.

[19] Kohtake, N., J. Rekimoto, and Y. Anzai, *InfoStick: an interaction device for Inter-Appliance Computing*, in *Workshop on Handheld and Ubiquitous Computing (HUC'99)*. 1999.

[20] Kortuem, G., S. Fickas, and Z. Segall. *Architectural Issues in Supporting Ad-hoc Collaboration with Wearable Computers*. in *Workshop on Software Engineering for Wearable and Pervasive Computing, International Conference on Software Engineering ICSE-2000*. 2000.

[21] Rekimoto, J., *Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments*, in *Proceedings of UIST'97*. 1997. p. 31-39.

[22] Rekimoto, J., *A multiple-device approach for supporting whiteboard-based interactions*, in

Proceedings of CHI'98. 1998.

[23] Rekimoto, J., B. Ullmer, and H. Oba, *DataTiles: A Modular Platform for Mixed Physical and Graphical Interactions*, in *Proceedings of CHI '2001*. 2001. p. 269 - 276.

[24] Smith, B.C, *Reflection and Semantics in a Procedural Language*, . 1982, MIT LCS TR-272.

[25] Sousa, J.P. and D. Garlan, *Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments.* submitted for publication, 2002.

[26] Suchman, L., *Plans and Situated Actions: The problem of human–machine communication*. 1987, Cambridge, MA: Cambridge UP.

[27] Ungar, D. and R.B. Smith, *Self: the power of simplicity*. Lisp and Symbolic Computation: An International Journal, 1991. **4**(3): p. 45-55.

[28] Weiser, M., *The Computer for the Twenty-First Century*. Scientific American, 1991(September): p. 94-101.