Applications considered harmful for ambient systems

Simon Dobson

Department of Computer Science, Trinity College, Dublin, Ireland
simon.dobson@cs.tcd.ie

Abstract. The notion of *application* – a single, bounded piece of functionality presented to users – goes almost unquestioned. However in the context of highly adaptive and ambient systems it is not clear that pre-building and pre-packaging functions is useful, and it may be that a more dynamic model of providing functionality is required. We re-assess the traditional notion of packaged applications, and instead explore a dynamic model of component composition. The model is naturally adaptive in the sense that services "self-assemble" in direct response to user needs, environmental factors and information relationships. We explore some ways in which this might change the way we think about adaptability in ambient systems, and sketch some directions for the future.

Introduction

Applications are almost synonomous with computing. Users often conceptualise their computing environment as being composed of a suite of frequently-used applications (web browser, e-mail reader, word processor, CD player etc). An application offers a convenient artifact for marketing purposes, making it easy to highlight the functions on sale Even highly document-centric environments are based heavily on applications.

From a technical perspective applications also have desirable features. Applications are *pre-selected*, *pre-built* and *pre-packaged*: a development team picks the functions to be provided, combines existing libraries and components with bespoke code to create a stand-alone executable, and then packages this executable with appropriate documentation and supporting data. Even most free or open-source software is designed and delivered in this way.

Much of this activity has a single purpose: *to create a targeted, self-contained entity that can be presented to users*. Most applications have only minimal interactions with other applications on the user's desktop. This emphasis on the stand-alone is a little perplexing in the face of the prevailing mainstream development methodologies that stress the re-use of components. The combination of components is only really accepted up to compile-time: deployed systems remain largely static. This means that the benefits of component-based methodlogies – easy adaptation, customisation, incremental alternation, familiarity and so forth[12] – accrue to the developer but not to the user.

In desktop situations this observation might simply be an idle curiosity. In the case of ambient systems, however – where the computing infrastructure can sense and react directly to its environment – the idea that one *can* pre-select what a user can use, and how, seems rather tenuous. An ambient system must by definition be adaptive at its core, and needs a distraction-free "flow" that is profoundly different to a traditional environment[4].

In this paper we begin to address the idea that pre-packaged applications may not be suitable for ambient systems – may, indeed, be fundamentally ill-adapted and harmful to the creation of truly adaptive environments. For the purposes of discussion, we describe instead some early ideas for a model of dynamic run-time composition driven from an underlying context model that causes applications to "self-assemble" in reponse to contextual factors such as user, task and environment. This builds on our previous experiences in building ambient systems, both academically and commercially. The model is intrinsically adaptive, in the sense that the environment naturally tailors the construction of the services it offers. We explore how this might change the way in which we think about adaptability for ambient systems, and sketch some future directions.

The problem

Ambient systems are perhaps the most adaptive IT solutions it is possible to conceive: a computing and information system that senses its users' needs, tasks, information and environment and attempts to provide a seamless IT service tailored to them. A good overview can be found in [11].

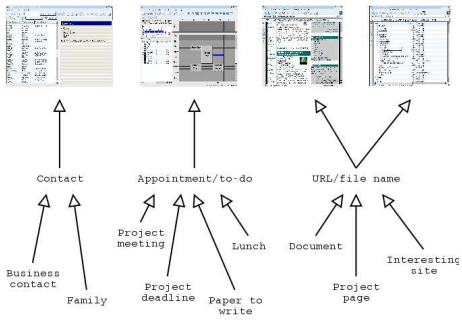

Fig. 1. Information and tools combined

The development process for a traditional environment is well-studied. Best practices centre around object-oriented analysis and design methodologies such as RUP[6], with their focus on domain-level understanding through *use cases*. The use case approach involves studying the tasks users will perform with the system being developed, and then using these articulated needs as a basis for design. The intention is that each use case captures a user task *as the user perceives it*, which is then translated into a conceptual design, concrete design, and finally to code. Tool support can help ensure consistency at each stage of the process. Some developers – and users – find such top-down approaches prescriptive. Rather than trying to articulate a complete application up-front, "agile" methods such as XP[2] seek to extend applications piecemeal as new capabilities are required, adding functionality within a lightweight framework. These methods differ for the developer, but not for the user: the aim is to create a largely packaged, largely stand-alone solution[1].

To see what impact this such packaging has, consider a typical desktop environment (figure 1). A wide variety of information from a variety of sources – manually-entered appointments and to-dos, web pages visited, documents generated etc – is manipulated through a small suite of common desktop tools. The tools typically provide a single view onto the information, with few if any cross-links. Even in a state-of-the-art environment it is impossible to (for example) relate a to-do item to the several meetings at which it is discussed, or relate a URL to the contact details of its author, or explore what URLs were visited in the course of writing a particular project deliverable – or any of a range of other possible views.

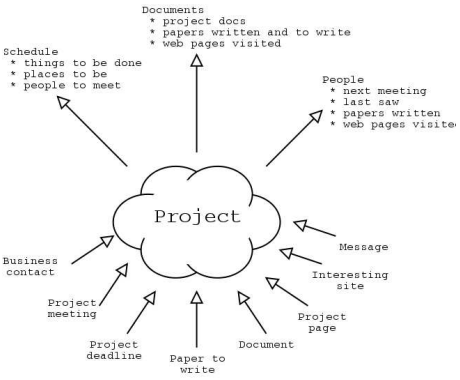

Fig. 2. A more conceptual view

Diving behind the scenes, we find that the information base being used by the desktop environment is split across a number of files, each with its own format, each separate from the others, and each closely coupled with a tool. This remains true even when – as with the environment being shown here (KDE) – all the tools use open (and generally public standard) file formats. The applications define the information structure. However, for most people information is not structured this way. One might equally – and more logically, from a user perspective – structure information based on the projects to which it applies (figure 2).

Now, an obvious response is: "Oh, well, you've brought us a suite of new use cases that different users *might* want. For those that can't learn how to work the *right* way, we *could* build a new set of views into the tools – or a new tool – to let them look at things this way too." However, this completely misses the point, which is that *no single set of tools will capture all the possible views*, especially when there is an extensible

[1] Even the word *user* seems vaguely pejorative, somehow: someone who is supposed to *consume* software rather than *create* or *modify* it....

information model with rich cross-links, and especially not in the presence of real-world cues and triggers such as occur in ambient systems.

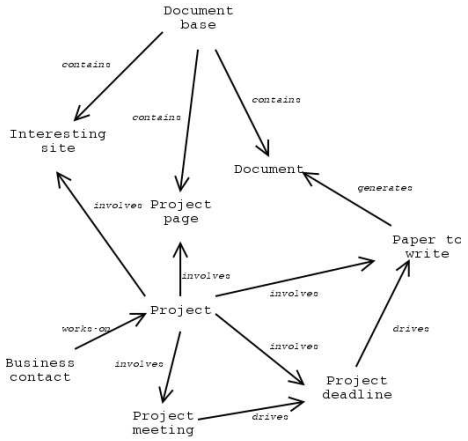

Fig. 3. Multiple views in a rich structure

The reason for focusing on ambient systems is simply that they show into sharp relief the whole notion of what it means to be "user friendly". An ambient system is intended to fit seamlessly into a user's daily activities – both physically (in the sense of being always available) and cognitively (in the sense of "doing the right thing" without distractions). This is not accomplished by simply allowing a user to access information wherever they are: the presentation of information must fit the use to which the information is being put[10].

An information base – the set of available documents, contacts, URL, appointments etc – can be viewed as a graph with facts as the nodes and relationships as the edges (figure 3). Using the tools of the "semantic web", and especially the Resource Definition Framework (RDF)[7], we can view and represent this graph as a collection of hyperlinks within and between XML entities. The tool-centric view from figure 1 breaks this conceptual graph into separate sub-graphs, destroys some of the edges that link the information, and then presents each using a separate tool. While there are advantages to this approach, a more sensible strategy is to retain the knowledge base as a single entity within a layered collaborative architecture[8,9].

Information is mostly neutral, in the sense that it can be viewed from many viewpoints. Each "projection" gives a user a different sense of what is important and how it relates to other information. Conversely, a tool encapsulates a single viewpoint (or occasionally a small number of viewpoints), generally somewhat isolated from others. Moreover, each tool dictates an interface that the user has to the information, and also typically makes assumptions about the capabilities of the interface being used to view it.

What are the alternatives to the static application? One well-known approach is the highly interactive graphical environments targeted at children and associated with Seymour Papert and (most especially) Alan Kay. Various environments have demonstrated that it is possible to build systems without building applications, by giving users a rich toolbox of components and a flexible way of combining and re-combining those components themselves[2]. Some developers would argue that most users do not want this degree of control over their desktops. While we might dispute this claim in general, it is undoubtedly true that standard presentations can be easier to learn and less distracting. However, this does not excuse presentations that confuse or hinder useful operations, and this is especially true for ambient systems in which useful, seamless operations are the *raison d'être.*

Using information to control information

However, we may make an observation that provides an interesting alternative to full dynamism. The information base we have been discussing is very rich, in the sense that information is heavily hyperlinked and the hyperlinks (and hence the anchors) have well-defined semantics. Because the information is well-typed we can use it to affect how it is used.

[2] "...we realized [when designing Smalltalk at PARC] that you really wanted to freely construct arbitrary combinations (and could do just that with (mostly media) objects). So, instead of going to a place that has specialized tools for doing just a few things, the idea was to be in an "open studio" and *pull* the resources you wanted to combine to you ... all the other objects that intermingle with each other should have very similar UIs and have their graphics aspect be essentially totally similar as far as the graphics system is concerned – and this goes also for user constructed objects."[5]

This is in itself quite an interesting departure: many IT systems act as simple filing cabinets and do not make much use of the information they store. A diary, for instance, generally limits itself to allowing the user to specify a single simple alarm before an appointment. However, a diary entry goes through a whole sequence of possible states in its relationship to the user – from *far-away*, through *needing-action* and *needing-immediate-action*, to *current* and finally to *completed*. What the user needs from the appointment, and any information associated with it, change accordingly. With a sufficiently rich knowledge base we might send reminders to other participants[3], collect together the documents relating to the appointment, generate an agenda from the to-dos of a previous related meeting, and so forth. – all functions that are impossible without understanding the *meaning* of an appointment.

If we addressed this problem directly, the result would be another application suite – perhaps well-suited to a particular domain. However, we want to be able to adapt behaviour to need without re-writing or re-compiling; ideally we want to be able to use whatever information and tools are to hand when the user needs the information. In order to accomplish this we need to provide intelligence over the knowledge base. It is relatively straightforward to connect a rules engine to a knowledge base and then use the normal tools of knowledge manipulation – notably Horn clause logic and resolution, but also including temporal logic[1] and other non-standard techniques – to represent the way in which information should be handled dynamically. Essentially we create "mini-workflows", difering from the traditional kind in that they represent fragments of a larger process and concern real-world events, with all that entails for accuracy and error handling.

Adaptability is usually discussed as a question such as "how can we match this application to these circumstances?". Ambient systems modify this question to be "what does the user need to accomplish her tasks?". In the new model we modify the question further to be "what components should I select to best match the user's needs?". Adaptability is intrinsic to this model: rather than presenting a pre-built application and then trying to customise it (figure 4(a)), we can instead select the information we want to manipulate and use whatever components are available for manipulating it (figure 4(b)).

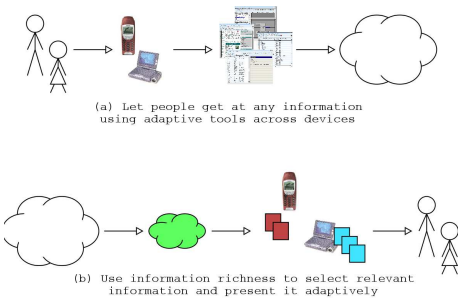

Fig. 4. Two approaches to adaptability

A number of factors can influence component selection including task, available information, deadlines, user identity, user location, component affordances. The point is that we want *all* these rules and the information in which they depend to be represented uniformly. This does not necessarily imply a single physical knowledge base, which can be unwieldy to search and reduces the chances of using existing components: it is possible to build federated knowledge bases whose elements present a common interface with queries being brokered across them. This unifies the tool- and web-based views of figure 1 and 2.

For example, suppose we have a knowledge base that includes simple project information (such as in figure 2). We can define a use case for (for example) a quarterly project meeting which will discuss progress during the period, re-assess the project timetable, review any documents and code submitted, and so forth. This in turn might give rise to a "workflow-let" of definitions and rules such as the following:

1. The "period" covered by the meeting runs from the last project meeting up to this one
2. Remind everyone that the meeting is coming up
3. Collect together any documents submitted in the period
4. Make the project manager aware of the project plan, for review before the meeting,
5. Collect together any management e-mails in the period
6. Circulate all documents and collate comments
7. Send urgent reminders the day before the meeting to anyone we haven't heard from since the first reminder

[3] Some collaborative diary systems do this, but generally only within a single organisation.

How does this help with applications? If we regard an application as a set of components, two core decisions for a developer are (a) what components should be included and (b) how they will fit into the process of a user's tasks. If we have sufficient richness in the knowledge base – both in terms of information relationships and relevance and other rules – we can replace both decisions by using the relevant information to drive component selection and presentation. For example, rule 3 might give rise to a web page with the documents linked, which then turns up on the user's desktop as "something to look at"; rule 4 might give rise to a link on the desktop plus a to-do in the diary (which can itself be highlighted on the desktop); rule 7 might generate an SMS message, or an e-mail, depending on the target user; and so forth.

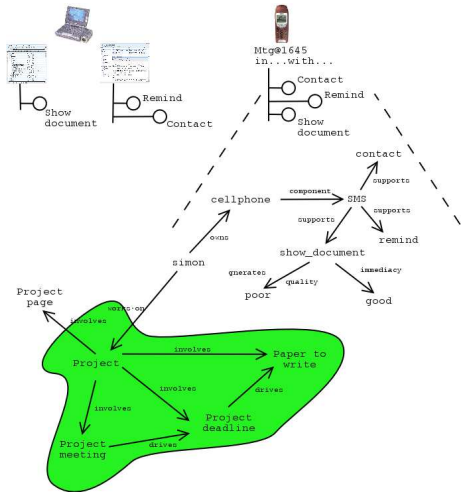

Fig. 5. A new model

In this new model (figure 5) a set of components make their affordances available as interfaces coupled with descriptions of their appropriateness to various environments. A set of rules is used to "activate" parts of the knowledge base (the circled parts of the diagram), marking certain tasks and certain information as relevant in the current context. In the example above we used a time-based relevance, but one could equally use a location-based relevance metric, or some more complex derivative of the network of relationships. These information and tasks then need to be matched against the available affordances. In general there will be several candidate components, which may then be brokered using another rule set. Note that the same mechanism – information and rules – can be used at all levels, making it easy to build complex derivatives (and also making it easy to tie oneself in knots).

Adaptability comes (firstly) from determining what is relevant and (secondly) from deciding what component should be used to present it. A user might encounter the project management application several times before a meeting and have it appear differenrly each time: once because new information has arrived, then because her palmtop device is supressing information she cannot reasonably interact with on the move, and finally because shae has installed a new component that provides additional functions.

The net result is a subtly different view of both applications and desktops or mobile devices. The desktop contains information presented through appropriate tools rather than tools themselves; the "project management application" is just a collection of information, accessed as appropriate. Moreover the "application" can co-exist with *other* applications completely seamlessly: the user has things to do and the components with which to do them, selected using rules of relevance from a shared information base.

We do not end up with "normal" applications under this model – and that might be a good thing. Instead we have a system that is driven by knowledge activated by some mechanism. We need not know the information, the mechanisms or the individual components *a priori*: what we are providing is an environment in which components can be described and capabilities accessed uniformly and extensibly. In effect the application *self-assembles* from the set of available components as required, and dis-assembles when the task is complete.

Many of the usual architectural features turn out to be available. For example, one may compose two components so that the result of one is passed to another by having the first insert its results into the knowledge base in such a way that they will be marked as active, which will then automatically invoke the second component on them. There are also some extremely interesting novel architectural possibilities: by way of illustration, a shared knowledge base and components on running multiple devices form the basis for a collaborative application without any additional development beyond high- and low-level concurrency control.

Several component-based engineering approaches (notably [12]) take a very dynamic view of components and emphasise dynamic interface acquisition. This is also a feature of many distributed software systems

including EJB and CORBA. There is a growing understanding of the issues involved in building large systems of this type and successfully evolving them. All this knowledge may be deployed to address adaptive applications under our model.

However, many aspects of successful development remain under-explored. The approach relies heavily on being able to structure the relevance of information over time: if information remains relevant the desktop will be unusably crowded (or the SMS message box very full). Controlling relevance is both a matter of defining suitable rule sets *and* of having them compose together constructively – not a problem that is well-understood. One solution might be to adopt some of the techniques suggested by Fogg[3] to structure how users deal with large information bases and task sets.

Equally importantly, we need to populate and maintain the knowledge base, and do so without requiring additional effort. A lightweight approach is to capitalise on existing components and provide instrumented versions of others. Experience suggests that some structures seem to appear of their own accord – many project managers maintain a directory and a mail folder per project, and any information inserted into either is immediately classified. However, some degree of user involvement is inevitable, at least in the early stages, to capture and classify information.

Conclusion

We made the argument that the traditional approach to packaging IT functions as pre-built applications may not be optimal for developing ambient systems. We outlined an alternative variation on accepted component-based software in which component composition occurs at dynamically at run-time in response to user needs, interface capabilities and environmental factors. We suggested some possible technical avenues to explore in order to realise this model.

We pointed out at the start that our model is still very early. We still need to address a number of vital issues, with perhaps the two most critical being the ways in which we can capture such complete contextual information and the effects of missing or noisy data on system behaviour. Initial experiments are being performed within a framework that allows us to extract contextual information from user interface gestures, explicit information and simple task-based models. We hope to use the lessons learned to formulate a theory and toolkit for dynamic component composition and the self-assembly of adaptive, extensible ambient systems.

References

1. James Allen and George Ferguson. Actions and events in interval temporal logic. *Journal of Logic and Computation*, 4(5):531–579, 1994.
2. Kent Beck. *EXtreme Programming EXplained*. Addison-Wesley, 1999.
3. B.J. Fogg. *Persuasive technology – using computers to change what we think and do*. Morgan Kaufman, 2003.
4. David Garlan, Daniel Siewiorek, Asim Smailagic, and Peter Steenkiste. Project Aura: towards distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2):22–31, 2002.
5. Alan Kay. Posting to the OpenCroquet mailing list, April 2003.
6. Philippe Kruchten. *Rational Unified Process: an introduction*. Addison-Wesley, 2000.
7. Ora Lassila and Ralph Swick. Resource Description Framework model and syntax specification. Technical report, World Wide Web Consortium, 1999.
8. Paddy Nixon, Simon Dobson, Sotirios Terzis, and Feng Wang. Architectural implications for context-adaptive smart spaces. In *Proceedings of the International Workshop on Networked Applicances*, pages 156–161. IEEE Press, 2002.
9. Paddy Nixon, Gerard Lacey, and Simon Dobson. Smart environments: challenges for the computing community. In Paddy Nixon, Gerard Lacey, and Simon Dobson, editors, *Managing interactions in smart environments*. Springer Verlag, 2000. Introduction to the book.
10. Donald Norman. *The invisible computer*. MIT Press, 1998.
11. Debashis Saha and Amitava Mukherjee. Pervasive computing: a paradigm for the 21st century. *IEEE Computer*, 36(3):25–31, March 2003.
12. Clemens Szyperski. *Component software: beyond object-oriented programming*. Addison Wesley, 1998.