Stuck in the Middle: The Challenges of User-Centered Design and Evaluation for Infrastructure

W. Keith Edwards, Victoria Bellotti, Anind K. Dey,[] Mark W. Newman

Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 USA
+1 650.812.4405
{kedwards, bellotti, mnewman}@parc.com

[]Intel Research Lab at Berkeley
2150 Shattuck Avenue, Suite 1300
Berkeley, CA 94704 USA
+1 510.495.3012
anind@intel-research.net

ABSTRACT
Infrastructure software comprises code libraries or runtime processes that support the development or operation of application software. A particular infrastructure system may support certain styles of application, and may even *determine* the features of applications built using it. This poses a challenge: although we have good techniques for designing and evaluating interactive applications, our techniques for designing and evaluating infrastructure intended to support these applications are much less well formed. In this paper, we reflect on case studies of two infrastructure systems for interactive applications. We look at how traditional user-centered techniques, while appropriate for application design and evaluation, fail to properly support infrastructure design and evaluation. We present a set of lessons from our experience, and conclude with suggestions for better user-centered design and evaluation of infrastructure software.

Keywords
Interactive software, technical infrastructure, toolkits, design, evaluation

INTRODUCTION: THE INFRASTRUCTURE DILEMMA

Infrastructure is software that supports construction or operation of other software. It comprises systems ranging from toolkits (including those for building graphical user interfaces [14] or collaborative applications [18]), to network services (including document management systems [5]), to other sorts of platforms.

In all of its many manifestations, infrastructure is one or more software layers that provide *new technical capabilities* for applications. It enables applications that could not otherwise be built or would be prohibitively difficult, slow, or expensive. It also reflects the desire to engineer reusable and well-architected software systems and can serve many different applications, some of which may be unforeseen at the time of its implementation.

While infrastructure itself is not visible to the user, it *affords* certain styles of application and interface. In other words, the technical capabilities of the infrastructure may enable new interaction techniques, and lend themselves to expression in new application features or styles.

For example, an easy-to-use Macintosh toolkit, called the Toolbox [2], facilitated creation of graphical applications. Developers could quickly create widgets like scrollbars and dialog boxes, which previously demanded extensive programming. As the Toolbox acquired new features (such as ToolTips), developers exploited them. The Toolbox not only allowed the explosion of graphically consistent applications for the Macintosh, it also served as a disincentive for alternative interface styles. When it is so easy to create applications using the Toolbox, why spend the time and effort to do something different?

Such tight coupling between infrastructure and application features is not limited to graphical toolkits. Consider an infrastructure that allows applications to "tag" documents with useful properties (such as owner, last edited time, references, etc.), and stores these properties persistently. Such a system may allow certain styles of document-based applications to be more easily created (for example, applications that provide property-based organization rather than fixed, directory-style organizations).

Figure 1 illustrates the relationship between infrastructure and other aspects of an end-user system. The infrastructure provides higher-level abstractions that shield application developers from the demands of interacting with lower-level data, hardware devices, and software concepts. The infrastructure itself is separate from, but must anticipate the demands of possible applications, users and use contexts for the data, or hardware, or software constructs to which it provides access.

The key problem addressed by this paper is that even though the technical features of the underlying infrastructure are visible in—and to a large degree, even *determine*—the features of the applications created using it, we often lack criteria for designing or evaluating the features of the infrastructure itself.

CHI 2003, April 5–10, 2003, Ft. Lauderdale, Florida, USA.

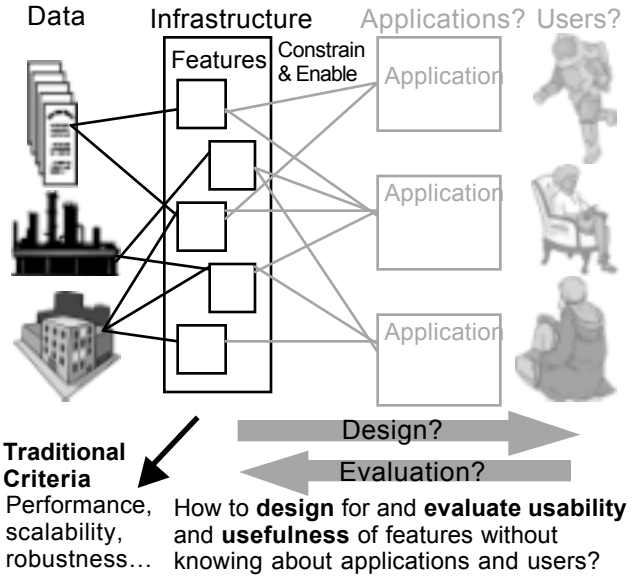

Figure 1. The Infrastructure challenges: How can the designer **design** and **evaluate** features without knowing about applications and users?

The Problem

Typically, computer scientists design infrastructure based on a desire for reuse [15], and evaluate infrastructure based on "classical" technical criteria: performance, scalability, security, robustness, and so on. These are all crucial metrics, and they must be accounted for if we wish to build workable systems. But there is a distinction between technical *workability* and *value for end-users*.

Unlike infrastructure, *user-visible applications* have long traditions of user-centered design and evaluation. Techniques such as participatory design, ethnography, and others all play a part in deciding what features go into an application, how well those features add value for users, by addressing needs with an acceptable user experience.

There are beginning to be some efforts to relate systems architecture, principles, and engineering processes to concepts of usability through some new concepts. These include architecturally sensitive usability scenarios [3] (end-user requirements that demand special architectural considerations); software engineering patterns [1, 13] (creational, structural and behavioral patterns in software engineering define known relations between contexts, problems and solutions); and agile software methods [12] (a collection of techniques such as Extreme Programming [4] that explicitly acknowledge that defining end-user requirements a priori is a practically impossible goal; a fact of which we are acutely aware).

There is, however, little reported experience in the HCI literature on the special process of, and problems that arise in, determining what particular user motivated features should go into the *design* of any given infrastructure system, nor for determining the success or failure in the *evaluation* of those features.

The problems of designing and evaluating infrastructure are not qualitatively the same as those for user-visible applications. Rather, the separation between infrastructure features and user-visible features, and the *indirectness* of the coupling between them, brings new challenges to the user-centered design and evaluation of infrastructure.

Designing Infrastructure for Usability

When developing infrastructure for interactive systems, designers add features to their infrastructure with the intention of enabling certain interaction experiences. However, there is a degree of separation here between the desired experience and the mapping of that experience onto technical features. For example, trying to anticipate the needs of all possible applications will lead to bloated and complex software that will be difficult for developers to understand and maintain. However, failure to anticipate important functions or behaviors that may be required by likely applications will lead to infrastructure that does not support the needs of those applications. Such infrastructure will either fail, or be tinkered with by developers who may not understand exactly what they're tinkering with and the consequences of doing so.

Currently, the determination of features for an infrastructure system is not particularly user-centered and is largely based on designers' experience and intuition, having built both prior infrastructure and applications. Designers must *abstract* from a desired set of application features to a specific set of technical features in the infrastructure. This leads to a number of questions:

- Is it possible to more directly couple design of infrastructure features to design of application features?
- How can this more direct coupling exist when the applications the infrastructure will support don't yet exist or cannot be built without the infrastructure itself?
- Could the context of either the 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?

Evaluating Infrastructure for Usability

Imagine a toolkit for graphical applications that provides a set of radical new features (e.g., support for visually rich displays, or very high resolution). The features in this toolkit are intended to support applications with a compelling, new user experience. But how can we tell if this user experience is really compelling, or even "right?"

We can only evaluate the user experience afforded by the toolkit and its features by building applications that use it, and then evaluating them. While the toolkit itself can be evaluated on its technical criteria, the aspects of it that are designed to support a particular user experience can only be evaluated in the context of use and thus must be evaluated indirectly—through applications built with the toolkit. This leads to the following questions:

- How do we choose which applications (for which users and uses) to build to evaluate the infrastructure?
- What does the manifestation of the infrastructure in a particular application say about its capabilities (or even desirability)? How useful is this "indirect" evaluation?
- Are the techniques we normally use to evaluate applications acceptable when our goal is to evaluate the infrastructure upon which those applications are based?

Discussion

Clearly, there is cross-talk between the problems of design and evaluation. Iterative design implies at least some evaluation. And thorough evaluation cannot happen in the absence of some designed artifact to be evaluated.

Our aim in writing this paper is to begin a discussion on the questions enumerated above, and the problems inherent in design and evaluation of infrastructure to support interactive experiences. Our belief is that a new model of user-centered design and evaluation must evolve to support such systems; this belief is motivated by our experiences in designing and evaluating not only infrastructures, but also applications that exploit them.

We frame our discussion in terms of two case studies of our own work. Perhaps most interestingly, the systems described were motivated by human, not technical, concerns. These case studies illustrate real problems inherent in trying to create useful and usable infrastructure in the absence of good design and evaluation techniques.

CASE STUDY ONE: PLACELESS DOCUMENTS

The Placeless Documents system [9] is an infrastructure that enables fluid storage, exploration, and use of information spaces. It was designed specifically to enable applications that escape the rigidity of traditional filesystems and other data storage technologies.

In traditional filesystems, the user creates a more-or-less static collection of folders or directories, to reflect the structure of her work. These structures are often created early, at the beginning of a project, for example, rarely changing, due to the difficulty inherent in re-filing and re-structuring the hierarchy of folders and their contents.

In the Placeless model, *documents* (analogous to files) can simultaneously exist in any number of *collections* (analogous to folders). So there is no one place in which the user must store a document. Moreover, collections themselves are much more fluid and powerful than traditional folders. The contents of collections can change dynamically, for example, to contain all documents used within the last five minutes, or all documents created by user Paul. The flexibility of Placeless' collections allows them to be used for ad hoc organizations of a file space, as well as for multiple, simultaneous organizations, perhaps to support different uses of the same documents.

From the outset Placeless was intended to *become* your filesystem: traditional files and folders would be replaced by more flexible Placeless collections and documents. Additionally, Placeless would be the single repository for *all* types of documents, including email messages and web pages, not just files. In short, Placeless would unify currently disparate access mechanisms, and provide a flexible set of means to store, organize and use them.

User-Centered Design Challenges

A major problem that provoked much discussion during the Placeless project, was how to determine the technical features required to support the new applications that would use these flexible organizational capabilities.

Using Multiple Scenarios to Drive Design

We decided to use a scenario-based approach to determine the infrastructure features. We identified multiple contexts that could benefit from the Placeless infrastructure. These included "ideal" scenarios—completely replacing the desktop file-and-folder metaphor with a more powerful scheme—as well as mobile access to information spaces, collaborative document use, and so on. The applications that resulted from these scenarios included "better" versions of existing tools (such as file browsers and mail readers), and also some completely new tools.

From these scenarios we then identified specific application requirements and technical features required to fulfill them. Features to support fluid organization were required by virtually every scenario. These features included a flexible model for document metadata, coupled with a query mechanism in which dynamically evaluated queries could be embedded within document collections.

Additional possible infrastructure features were not necessitated by all of the scenarios; instead, certain features were identified that were only required by individual scenarios, or small sets of scenarios. For example, our collaborative scenarios depicted users with different roles, moving about and sharing documents. Supporting this scenario demanded a security model (to determine who can and cannot view your documents) and a distribution model (so that your documents can "refer to" documents owned by someone else. Our mobile use scenario required replication features (so that documents can coexist on multiple machines simultaneously).

Lessons for User-Centered Design

All of the features mentioned above were motivated by our scenarios, and a desire to provide a positive, robust experience to the user. But of these, only a few—the basic metadata mechanism and the query model—were truly crucial to supporting the core ideas of the project.

The Dangers of Feature Bloat

This experience points out what we believe to be a key danger when designing infrastructure. While secondary features such as security, distribution, replication and so on were all key to certain scenarios, technically interesting in themselves, and perhaps even necessary for a "real world" deployment of Placeless, they were not central to the value that Placeless promised to deliver. In brief, these secondary features had *no* value if the primary features were not successful; if applications using Placeless do not provide better information organization, it does not matter if they're secure or distributed.

One can even go so far as to argue that *until* one specific application domain is validated, secondary features can distract from the design of infrastructure. For Placeless, the absence of a focus on a particular application domain, and strict requirements derived from it, led to it becoming a "grab bag" of technical features. Many of these features could only have been justified or motivated (or evaluated) after the core premises had been validated.

Lesson 1—Prioritize core infrastructure features: *First build a minimalist infrastructure to test core design ideas and then, once those are validated, obtain user feedback to motivate additional features.*

In the case of Placeless, the core novel features (those that pertained to flexible organization) should have been pursued before other, secondary features were investigated (replication and so on).

User-Centered Evaluation Challenges

It was clear that to fully exploit the Placeless infrastructure new applications were required. Current applications—the Macintosh Finder, Windows Explorer, email, and so forth—did not provide access to novel features afforded by Placeless; only custom-built applications could perform this function.

We adopted a two-part strategy for evaluation. First, we felt that the capabilities of Placeless would be best tested under "real world" conditions. That is, by providing users with enhanced replacements for everyday tools, such as mail readers, we might determine how infrastructure like Placeless could fit into everyday work practice. Second, we built some small "throw away" applications, designed to demonstrate some aspect of the infrastructure, while not providing all the functionality of a "real" application.

Evaluation in Real Use Contexts

For the first strategy, we created two fairly heavyweight applications to test the infrastructure. The first was a "Placeless-aware" mail system that would not force users to create fixed folders of messages. Instead, structure would be emergent, according to the task at hand. Messages could exist in multiple folders, and entire folder hierarchies could be created on the fly according to ad hoc organizational criteria imposed by the user. The second application was a web portal-style tool designed to provide the benefits of Placeless organization to a shared document repository on a web server.

Both of these applications were designed to withstand real long-term use. Additionally, since we believed that users would be unwilling to forgo their existing applications and data completely, we built migration tools to allow users to use both existing, non-Placeless tools alongside our new applications, accessing the same data.

Evaluation via Lightweight Technology Demonstrations

The second path to evaluation was the creation of a number of lightweight "throw away" applications, purely to demonstrate the utility of novel aspects of the infrastructure. These, while engineered to be neither robust nor featureful enough for long-term use, were easy to build and required few engineering resources.

These tools used Placeless as a platform for experimenting with workflow and collaboration [16]; to address shared categorization schemes [10]; and for experimenting with other user interfaces.

Lessons for User-Centered Evaluation

While the chief lesson we learned could perhaps be summed up as, *"prioritize your evaluation criteria correctly,"* the sections below discuss aspects of this prioritization in detail.

Defer Work that Does Not Leverage the Infrastructure

The chief error made in evaluation was to aim immediately to evaluate the system in the context of real use. Such an evaluation, of course, required that the system be robust enough for day-to-day use as an information repository, support features (such as security) that might not be required in other contexts, and provide tools to support migration to the platform.

This last requirement was particularly troublesome. We believed users would reject new applications and tools if this meant giving up their existing tools, and perhaps even data. We strove to ease migration to Placeless-based applications by creating multiple backward-compatible means to access and store documents in Placeless. Tools were constructed to allow Placeless to be accessed as if it were a normal filesystem (using existing file browsers), or a web server (using existing web browsers), or an email server (using existing mail clients). Likewise, the content of documents stored in Placeless could reside in existing filesystems. The goal was to ease users' transition into Placeless-based tools, and obviate the need for a "clean slate" approach to the change.

While this was a rather significant—and to large degree, successful—engineering effort, this migration support did nothing to facilitate evaluation of the infrastructure's features, nor did it validate the class of applications the infrastructure was designed to enable. These tools allowed users to continue using, say, Windows Explorer and all of its features, but still provided no access to Placeless-specific features for evaluation purposes.

While required to support evaluation in a real use context, none of these tools in themselves satisfied the prime objectives of leveraging the platform and demonstrating its utility. Furthermore, they diverted engineering resources away from evaluation strategies that would have better proved the core value of the infrastructure.

Lesson 2—First, build prototypes that express the core objectives of the infrastructure: *Initial prototypes should leverage the fullest extent of the power of the infrastructure, since the assumption that this power makes a difference is the critical thing to test first.*

Usefulness and Usability Remain Essential Criteria

The two major applications, created to leverage Placeless, were meant to be evaluated during long-term, real-world use. However, both of these applications failed as tools for evaluation, although for different reasons.

The mail system failed simply because of *usability* problems. It did not matter that it offered useful tools for

organization if it did not also support other email task requirements. Any value of the Placeless-derived features of the mailer was overshadowed by a lack of other features that would have demanded more effort to perfect.

The web portal failed for a more prosaic reason: there was never a clear indication of precisely what the portal would be used for, nor a clear set of requirements for what the portal should do. Simply put, the system was not *useful*, or, at least, not useful for the general-purpose project management tasks that it was intended to support.

Lesson 3—Any test-application built to demonstrate infrastructure must also satisfy the criteria of usability and usefulness: *These criteria are difficult to satisfy, demanding requirements gathering, prototyping, evaluation, and iteration as with any application. The more ambitious the application, the more these criteria will come into play. Designers must trade off the time to be devoted to satisfying them against the time available for building and testing core infrastructure features.*

The Placeless mailer failed to meet the usability requirement, while the web portal failed to meet the usefulness requirement. While these are perhaps obvious criteria for any design, they can be overlooked when the goals of the developers are not to evaluate the applications themselves, but rather the infrastructure they exploit.

Assess Intrinsic Value of the Platform Early

While the heavyweight applications failed to serve much useful purpose, the lightweight, proof-of-concept applications succeeded in demonstrating compelling new user experiences. Since these tools did not depend on long-term use to provide feedback to the project, neither a migration strategy nor features that would make them usable in a "real world" setting was required.

Lesson 4—Initial proof-of-concept applications should be lightweight: *Early testing of core infrastructure features should not require building myriad application features purely for delivering a well-rounded, real-world-style application since such features do little but distract from the main goal of getting the basics of the infrastructure itself (rather than the applications) right.*

Discussion

These lessons, while perhaps obvious in hindsight, were learned the hard way by the developers of Placeless. Its features were driven by careful analysis of the usability problems with existing document storage systems, and a desire to create a compelling and sensible user experience.

The chief problem was finding appropriate outlets for evaluation. Without a focus on what the actual objectives of the test-applications were—namely, to test the power of the infrastructure and to reveal shortcomings—the design requirements ballooned. This expanding design, of course, demanded even more outlets for evaluation. While empirical observation of real use is essential in proving an application's worth, focusing too early on realistic use detracted from our main goals. Significant engineering resources are required to support such an evaluation.

Instead, a more lightweight approach to evaluation would have been better, especially given the experimental nature of the infrastructure. We should have begun with modest, easy-to-build applications that leveraged unique features of Placeless, and then—if those features were found to be useful—proceeded to a longer-term evaluation. Any "firming up" of the infrastructure, as well as testing application usefulness and usability, would have occurred at this stage. All this is summed up by the following meta-lesson; a composite of the earlier ones:

Lesson 5—Be clear about what your test-application prototypes will tell you about your infrastructure: *It is easy to get distracted by the demands of building applications in themselves and to lose sight of the real purpose of the exercise, which is purely to understand the pros and cons of your infrastructure.*

CASE STUDY TWO: CONTEXT TOOLKIT

The Context Toolkit [8] (built by a different team from that of Placeless) is infrastructure to support context-aware applications. These are applications that dynamically adapt to users' context: identity, location, environmental state, activities, etc. The Context Toolkit was built in the spirit of ubiquitous computing to allow programmers to explore this space and to provide users with more situationally appropriate interfaces to interacting with computing environments.

Traditional applications have little or no access to context information and cannot adapt their behavior to user, system or environmental information. These applications are sub-optimal in dynamically changing situations. Further, in a ubiquitous computing environment, with many services and information resources available to a user at any time, the use of context to help determine which information and services are relevant is particularly crucial. However, context has so far been difficult to acquire, and so has been used infrequently.

To address this problem, the Context Toolkit treats context as a first-class citizen, providing applications with easy access to contextual information and operations to manage it. This means application programmers need not worry about the details of acquiring context and can simply exploit existing mechanisms. So context is made as easy to use as traditional keyboard and mouse input, allowing programmers to concentrate on designing the application itself. Further, end-users of these applications are provided with the added value of having information and services automatically adapt to their current situation.

User-Centered Design Challenges

Survey of the State of the Practice

Our design of the Context Toolkit comprised four parts. First, we examined existing context-aware applications in the literature to derive a set of necessary features for our infrastructure. Second, we looked at existing support for building context-aware applications, and made a list of its features and examined the complexity of the applications they were able to support. Third, we performed informal interviews with builders of context-aware applications (mostly researchers) to determine what features they

would want in the Context Toolkit. Finally, we used our own experiences, honed after building numerous context-aware applications and a previous system to support building of these types of applications.

We determined that the minimal set of features we needed were: useful abstractions for representing context, a query- and event-based information mechanism for acquiring context, persistent store of context data for later use, cross-platform and cross-language compatibility to allow use by many applications, and a runtime layer that could support multiple applications simultaneously.

The model chosen for acquiring and representing context was the "widget," a notion taken from the field of graphical user interface (GUI) design. The widget is useful in encapsulating behavior so application designers can ignore the details of how a widget such as a menu or scrollbar is implemented, focusing instead on integrating that widget with their application. We created the notion of context widgets that were also intended to encapsulate behavior, allowing designers to focus on *using* context information, rather than *acquiring* it [8].

Supporting Design Through Simulation

There is an important distinction between the context world and the GUI world: the number of context sensors potentially surpasses by far the number of "standard" input devices. Thus, while it is feasible to create all the device drivers needed to build and operate a GUI-based application, the same is not true for context-aware applications. It is here that the widget metaphor breaks down. Application designers cannot avoid the details of acquiring context from sensors because often they are using sensors for which no standard device driver exists.

To alleviate this need, we were forced to add the ability to *simulate* context data, allowing designers to either fake the use of existing sensors or to fake the existence of sensors that do not yet exist (e.g., mood detection). In the evaluation section, we will discuss the impact that simulation had on evaluation of the infrastructure.

Lessons for User-Centered Design

Providing a Basic Infrastructure for Programmers

We designed and constructed infrastructure to support a set of carefully chosen features only to find that some of the features were unnecessary, while others were too difficult to get right or too limited to be useful to programmers. In hindsight, the support for multiple applications executing simultaneously and for multiple programming languages and platforms was wholly unnecessary. While these features would make any eventual infrastructure for supporting context-aware applications complete, they were not necessary for supporting programmers exploring the space of context-awareness. Programmers could build individual applications in a single language on a single platform and still perform interesting and useful exploration. Similar to the experience with Placeless, we designed for features that were not central to the value of the infrastructure.

In addition, the support we provided for some of the key features was too limited. We did not make it easy enough for programmers to build their own context widgets. Also, we did not deliver enough widgets to support the applications that developers wanted to build on top of our infrastructure. This reveals yet another crucial lesson:

Lesson 6—Do not confuse the design and testing of experimental infrastructure with the provision of an infrastructure for experimental application developers: *It is hard enough to anticipate applications you yourself might build, without providing means for others to push the envelope even further. By the time developers are let loose on the infrastructure it must be relatively stable. If they begin to demand—or even attempt themselves to make—changes underneath each other's live applications, this will lead to propagation of redundant, missing or changing features and consequential chaos and breakage.*

As implied in this lesson, the toolkit we built, while addressing many interesting research problems, was not completely appropriate for allowing programmers to investigate the space of context-awareness. While we spent much time trying to make the creation of widgets as easy as possible, a better approach would have been to spend time creating a toolkit with a simpler set of features (reiterating lesson 1) and deriving a base set of usable and useful widgets (analogous to the seven basic widgets in the Macintosh Toolbox) and letting designers explore the building of whatever applications were possible with those. This way, we could have elicited feedback on what additional widgets were *desirable* as well as on what features were *crucial* in making the toolkit easier to use.

User-Centered Design Challenges

Because the Context Toolkit was designed for exploration of context-aware computing, we decided to build some novel applications that would both leverage the toolkit's features and highlight the utility of context-awareness to users. While modification of existing applications (e.g. web browser, calendar, e-mail) would also have been useful, the lack of source code or sufficient hooks to alter these applications made this impossible. We also considered development of equivalent applications, but, as stated in the discussion of Placeless, these "equivalents" would not provide the stability or range of features necessary for user adoption. In addition, context-awareness provides the greatest benefit when the user's context is changing frequently. Therefore, we needed applications that operated in an "off-the-desktop" mode, for mobile users in dynamic environments.

We built many applications, from simple prototypes that demonstrated only the core features of the toolkit to complex systems that used most of the features. While of course being easier to build, the simple prototypes more succinctly illustrated the value of the toolkit and context-awareness in general than did the complex ones, as suggested by lesson 4. Furthermore, while intended as short-term demonstration prototypes, they ended up being the ones that were used over a long period. Examples include an In/Out Board [19]; a context-aware mailing list

that forwarded mail to only the current occupants of a building [8]; and a people tracking tool for the home.

Lessons for User-Centered Evaluation

The Value of Lightweight Prototypes

As with the Placeless lesson 3, the more sophisticated applications provided less return on investment. We built a conference helper to assist attendees in finding relevant presentations, taking notes, and retrieving them later [7]; a reminder tool that used contextual information (not simply time) to trigger reminders [6]; and an intercom system that followed users around a building and used context to determine when they could receive an incoming call [17]. These applications were naturally more difficult to build and harder to sustain and maintain for long-term use. While they were intended to more richly illustrate the toolkit features, they ended up more richly illustrating its shortcomings, both to programmers and end-users.

In our attempt to design a toolkit to support many different types of context, it was difficult to provide strict guidelines on how to use its programming abstractions. As the type of context being used varied and the ways in which it was used became more sophisticated, the ways that the toolkit could be used to support this increased. Thus, when building more complicated applications, programmers were confused about the "right" approach to take. This leads to the following lesson:

Lesson 7—Define a limited scope for test-applications and permissible uses of infrastructure: *Unconstrained interpretation of the purpose of infrastructure causes confusion both about what features to use, and how best to use them, making it difficult to assess the strengths and weaknesses of the infrastructure, independently from misunderstandings about how to exploit it.*

In addition, as more sophisticated applications were built, they often required the use of context for which there was no available sensing technology. In many cases, the desired sensors were either too expensive or in the realm of science fiction. With the added simulation support, there was little incentive for programmers to scale back their intended applications. They could build a simulated version of an application and do partial testing, but these applications could not really be deployed or put to any significant use to perform realistic evaluation [6, 11, 17].

As applications grew more complex, it became difficult for users to understand the mapping between context and application action. In particular, users could not determine how the simulated applications provided benefit because too much was faked, from the sensing technology to the situation that the users were in. This leads to:

Lesson 8—There is no point in faking components and data if you intend to test for user experience benefits: *By building simulations of aspects of both the infrastructure and the data it collects, you risk learning nothing about real user impacts and defeat the purpose of evaluation, wasting precious time and resources.*

All of the problems mentioned above ballooned as the sophistication of each application increased. Instead of being able to experiment with many applications, designers could only evaluate the simple ones, as only they could be put to authentic and sustained use. The limitations on the applications made it equally difficult for us to evaluate the ease of use and utility of the toolkit.

DISCUSSION AND CONCLUSIONS

So, what did we learn from our two examples of the challenges of user-centered design and evaluation for infrastructure. What we have is a list of apparently commonsense lessons—but while these lessons may seem obvious, it is only with the benefit of direct experience and hindsight that they have become clearly apparent:

- Lesson 1—Prioritize core infrastructure features.
- Lesson 2—First, build prototypes that express the core objectives of the infrastructure.
- Lesson 3—Any test-application built to demonstrate the infrastructure must also satisfy the usual criteria of usability and usefulness.
- Lesson 4—Initial proof-of-concept applications should be lightweight.
- Lesson 5—Be clear about that your test-application prototypes will tell you about your infrastructure.
- Lesson 6—Do not confuse the design and testing of experimental infrastructure with the provision of an infrastructure for experimental application developers.
- Lesson 7—Be sure to define a limited scope for test-applications and permissible uses of the infrastructure.
- Lesson 8—There is no point in faking components and data if you want to test for user experience benefits.

In themselves, these lessons do not fully answer the questions proposed earlier in this paper. We do believe, however, that they not only provide useful guidelines for infrastructure developers, but perhaps more importantly, form a starting point for further work on user-centered approaches to the design and evaluation of infrastructure.

Bridging the Infrastructure Design and Evaluation Gaps

To conclude our discussion we propose a way of looking at the infrastructure user-centered design process in terms of an augmented design-implement-evaluate iteration cycle. In Figure 2 we depict what our lessons tell us is an appropriate cycle for infrastructure design and evaluation.

Based on our experience, user-centered infrastructure design demands applications to demonstrate the power of the infrastructure. But the focus at first must be on applications with only simple proof-of-concept goals (core test applications). These should be selected based on fidelity to the features of the infrastructure (the degree to which they express the power of the infrastructure), with minimal demands for usability and usefulness evaluation (that is, only in as much as these criteria serve the goal of determining the power of the infrastructure). The design-develop-evaluate cycle for test applications, shown in the left of the diagram, is a very lightweight process, and the feedback to the infrastructure concerns core features only.

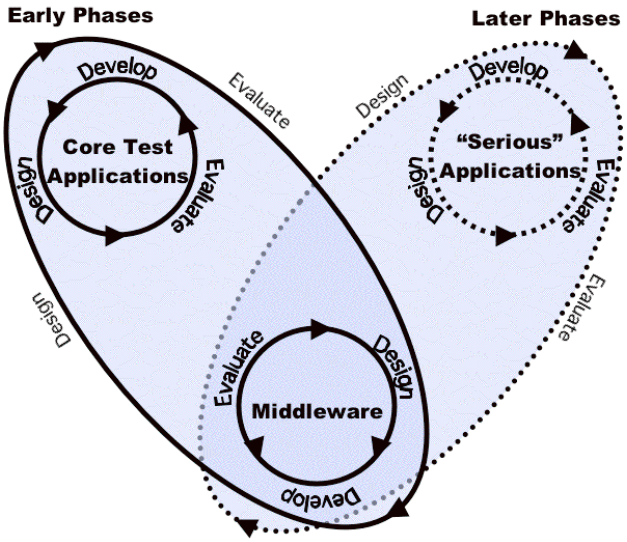

Figure 2: An augmented design-develop-evaluate cycle for infrastructure and its applications

Only in later cycles (shown to the right in Figure 2) should serious applications be built as these will have demanding engineering and usability requirements of their own that do not demonstrate the success of the infrastructure. There, the design-develop-evaluate cycles for the "serious" applications is a much more stringent process, since these are applications that will be deployed for real use. By this point, the design and evaluation cycles for the infrastructure should have honed in on the refined features necessary to support such applications.

In the terms of Figure 2, both the Placeless and Context Toolkit projects wasted effort on the parts of the figure represented by dashed lines. They were overly concerned with supporting, engineering and evaluating serious applications rather than with a user-centered evaluation of the infrastructure via simple, core feature-oriented test applications. While core test applications are not, in fact, the same as the serious applications that users really want, such applications are still essential, demonstrating as they do the power of the infrastructure and pointing to needs for the ultimate applications of the technology.

ACKNOWLEDGEMENTS

The authors would like to thank the members of the two projects used as case studies in this paper.

REFERENCES

1. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. and Angel, S. *A Pattern Language*. Oxford University Press, Oxford, UK, 1977.
2. Apple Computer, *Macintosh: Macintosh Toolbox Essentials*, 1993.
3. Bass, L., John, B.E. and Kates, J. *Achieving Usability Through Software Architecture.* Carnegie Mellon University. 2001, 2001.
4. Beck, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
5. Cousins, S.B., Paepcke, A., Winograd, T., Bier, E.A. and Pier, K., The Digital Library Automated Task Environment (DLITE). In *Proceedings of ACM International Conference on Digital Libraries*, (1997).
6. Dey, A.K. and Abowd, G., CybreMinder: A Context-Aware System for Supporting Reminders. In *Proceedings of Symposium on Handheld and Ubiquitous Computing (HUC)*, (2000), Springer-Verlag.
7. Dey, A.K., Futakawa, M., Salber, D. and Abowd, G., The Conference Assistant: Combining Context-Awareness with Wearable Computing. In *Proceedings of International Symposium on Wearable Computers (ISWC)*, (1999).
8. Dey, A.K., Salber, D. and Abowd, G.D. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human Computer Interaction*, *16* (2-4). 2001.
9. Dourish, P., Edwards, W.K., LaMarca, A., Lamping, J., Petersen, K., Salisbury, M., Thornton, J. and Terry, D.B. Extending Document Management Systems with Active Properties. *ACM Transactions on Information Systems (TOIS)*. 2000.
10. Dourish, P., Lamping, J. and Rodden, T., Building Bridges: Customisation and Mutual Intelligibility in Shared Category Management. In *Proceedings of ACM Conference on Supporting Group Work (GROUP)*, (1999).
11. Espinoza, F., Persson, P., Sandin, A., Nystrom, H., Cacciatore, E. and Bylund, M., GeoNotes: Social and Navigational Aspects of Location-Based Information Systems. In *Proceedings of Ubicomp*, (2001), Springer-Verlag.
12. Fowler, M. http://martinfowler.com/articles/newMethodology.html.
13. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.
14. Hudson, S.E. and Stasko, J.T., Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions. In *Proceedings of Symposium on User Interface Software and Technology (UIST)*, (1993), ACM.
15. Jacobson, I., Griss, M. and Jonsson, P. *Software Reuse: Architecture, Process and Organization for Business Success*. ACM Press, New York, NY, 1997.
16. LaMarca, A., Edwards, W.K., Dourish, P., Lamping, J., Smith, I.E. and Thornton, J.D., Taking the Work out of Workflow: Mechanisms for Document-Centered Collaboration. In *Proceedings of European Conference on Computer-Supported Cooperative Work (ECSCW)*, (Copenhagen, Denmark, 1999).
17. Nagel, K., Kidd, C., O'Connell, T., Dey, A.K. and Abowd, G.D., The Family Intercom: Developing a Context-Aware Communication System. In *Proceedings of Ubicomp*, (2001), Springer-Verlag.
18. Roseman, M. and Greenberg, S., GROUPKIT: A Groupware Toolkit for Building Real-Time Conferencing Applications. In *Proceedings of Conference on Computer-Supported Cooperative Work (CSCW)*, (1992).
19. Salber, D., Dey, A.K. and Abowd, G.D., The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *Proceedings of Conference on Human Factors in Computing Systems (CHI '99)*, (Pittsburgh, PA USA, 1999), 434-441.