

Stuck in the Middle: Bridging the Gap Between Design, Evaluation, and Middleware

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

IRB-TR-02-013

September, 2002

DISCLAIMER: THIS DOCUMENT IS PROVIDED TO YOU "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE. INTEL AND THE AUTHORS OF THIS DOCUMENT DISCLAIM ALL LIABILITY, INCLUDING LIABILITY FOR INFRINGEMENT OF ANY PROPRIETARY RIGHTS, RELATING TO USE OR IMPLEMENTATION OF INFORMATION IN THIS DOCUMENT. THE PROVISION OF THIS DOCUMENT TO YOU DOES NOT PROVIDE YOU WITH ANY LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS

Stuck in the Middle: Bridging the Gap Between Design, Evaluation, and Middleware

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

Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 USA
+1 650.812.4405

{kedwards, bellotti, mnewman}@parc.xerox.com

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

ABSTRACT

Middleware is software designed to support the development or operation of other software. A particular middleware system may afford certain styles of applications atop it, 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 middleware systems intended to support these applications are much less well formed. In this paper, we reflect on case studies of three middleware systems for interactive applications. We look at how traditional user-centered techniques, while appropriate for application design and evaluation, fail to properly support middleware design and evaluation. We present a set of lessons from our experience, as well as opportunities for further work.

Keywords

Interactive software, technical infrastructure, middleware, design, evaluation

INTRODUCTION: THE MIDDLEWARE DILEMMA

Middleware, loosely defined, is a class of software designed to support the construction or operation of other software. The embodiment of middleware can range from toolkits (including those for building graphical user interfaces [10] or collaborative applications [13]), to services (including networked services such as document management systems [2]), to other sorts of platforms.

In all its many manifestations, middleware is technical infrastructure: a software layer that exists to provide *new technical capabilities*, and on top of which applications are created. It allows for applications that could not otherwise be constructed or, alternatively, would be prohibitively difficult, slow, or expensive. Middleware is often the desirable result of efforts to engineer reusable and well-architected software systems. A single middleware system can serve many different applications, some of which may be unforeseen at the time of its implementation.

While middleware itself is not visible to the user, it may *afford* certain styles of applications and interfaces. In other words, the technical capabilities provided by the middleware may enable new interaction techniques, and may lend themselves to expression in new application features or styles.

For example, in the Macintosh, the presence of an easy-to-use toolkit for creating graphical applications, called the Toolbox [1], supported the rapid creation of graphical applications. With the Toolbox, developers could easily create on-screen widgets such as scrollbars and dialog boxes, which otherwise would have required extensive programming. Further, as new features became available in the Toolbox (such as ToolTips), application developers could easily take advantage of these. The presence of the Toolbox not only allowed the explosion of more-or-less graphically consistent applications for the Macintosh, it actually served as a disincentive to developing 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 middleware features and application features is, of course, not limited to graphical toolkits. Consider an infrastructure that allows applications to “tag” documents with useful properties (such as owner, last edited time, references, and so on), and stores documents along with their properties persistently. Such an infrastructure will need to represent

*Space reserved for
ACM copyright notice*

all the useful properties that might be required for a range of different document management applications, such as version management tools, workflow tools, and so on. It will also need to perform various general tasks to support these applications, such as ensuring that no two documents have the same identity, querying for documents, and so on. Figure 1 provides a schematic illustration of the relationship between middleware and other aspects of an end-user system. The middleware shields applications (and thus their developers) from the demands of handling data objects, which may include documents, databases, or live data from instrumentation in sensing systems. Middleware itself is separate from, but must anticipate the demands of possible applications, users and use contexts for this data.

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 have no clear criteria for designing nor evaluating the features of the infrastructure itself.

The Problem

Traditionally, computer scientists have evaluated middleware 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 workability and worth.

Unlike middleware, *user-visible applications*—applications with which the user interacts directly—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 address the needs of users, and—ultimately—the worth of the application itself.

There is, however, no comparable set of user-centered techniques for determining what features should go into a middleware system (the *design* of the middleware), nor for determining the success or failure of those features (the *evaluation* of the middleware). There is a much greater degree of removal from the end-user than for normal application development—it is not clear how notions framed in terms of user experience get translated into infrastructure technology, and vice versa. Nonetheless, this infrastructure technology will constrain what is possible for the applications that depend upon it.

We believe that the gaps between middleware and its design, and middleware and its evaluation, are fundamentally different than those of design and evaluation of traditional applications. In other words, we do not believe that the problems of designing and evaluating middleware are qualitatively the same as those for middleware, “only more so.” Instead, we believe that

the separation between middleware features and user-visible features, and the *indirectness* of the coupling between these features, brings new challenges to the design and evaluation of middleware.

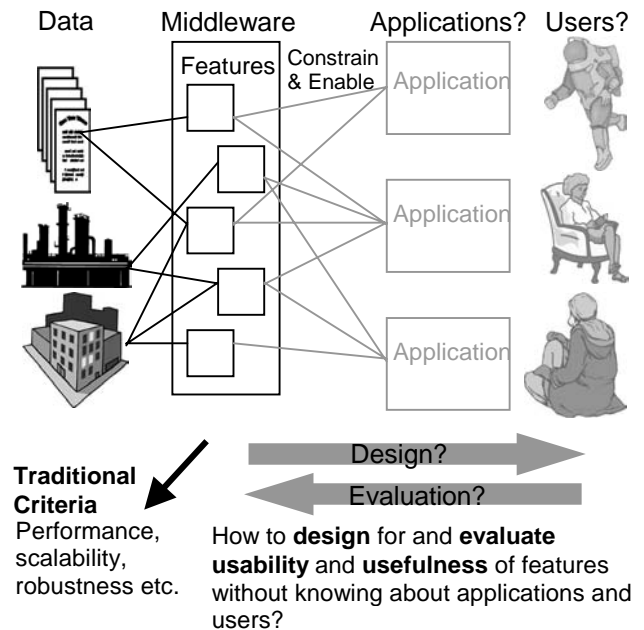


Figure 1. The Middleware gaps: How can the designer **design** and **evaluate** features without knowing about applications and users?

The Middleware-Design Gap

The process of designing a middleware system determines what features go in that system, along with their form and specific function. Of course, trying to anticipate the needs of all possible applications will lead to bloated and complex middleware that will be difficult to understand for those who have to use it, and difficult to maintain for those who have to develop it. On the other hand, failure to anticipate important functions or behaviors that may be required by likely applications will lead to middleware that does not sufficiently support the needs of those applications. Such middleware will either fall by the wayside, or be tinkered with by application developers (who may not understand exactly what they’re tinkering with and the possible negative consequences of doing so).

Currently, the determination of what features will go into a middleware system is not a particularly user-centered one and is largely based on the experience, sensibilities, and intuition of the designer. Technologists choose the feature set of a middleware layer based on their experiences in building both prior infrastructure, and in building applications. The process is one of *abstraction* from a desired set of application features to a specific set of technical features in the infrastructure.

There are a number of questions that arise from this situation:

- First, can we do better? Is it possible to more directly couple the design of infrastructure features to the design of application features?
- How can this more direct coupling exist when the applications that will be built atop the middleware don't yet exist...and may be impossible to build without the middleware itself?
- Could the context of either the users or the use of these unknown applications have an important impact on the features we decide upon?
- How can we avoid building a bloated, overly complex system incorporating every conceivable useful feature, at the same time as developing a system that will not need to be constantly updated (and thus repeatedly broken) throughout its life span?
- Are there better models for deciding on the features of "experimental" middleware, designed to support completely new interaction styles?

The Middleware-Evaluation Gap

The second problem is in the evaluation of a middleware system. For any infrastructure technology, there may be any number of possible manifestations of the technology as applications.

For example, imagine a new toolkit for graphical applications that provides a set of radical new features (perhaps support for visually rich displays, or extremely high resolution). How would one go about evaluating this toolkit? The toolkit itself is motivated by a desire to create applications with a compelling, new user experience. How can we tell if this user experience is, in fact, compelling, or even "right?"

One obvious conclusion is that 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 (performance, and so on), the aspects of it that are designed to support a particular user experience can only be evaluated in the context of users, not programmers, and thus must be evaluated indirectly—through applications built using the toolkit.

The questions that arise from this situation include:

- How do we choose which applications to build to evaluate the middleware?
- What kinds of users and contexts (types of uses) for these applications should we consider as appropriate for testing purposes?
- What does the manifestation of the technology in a particular application say about the capabilities (or even desirability) of the middleware itself? How useful is this "indirect" evaluation?

- Are the techniques we normally use to evaluate applications acceptable when our goal is to evaluate the middleware upon which those applications are based?
- Is it possible to evaluate the middleware outside of the context of a particular application?

Discussion

Clearly, there is cross-talk between these two problems. Iterative design is nearly always based in some evaluation, whether formal or not. And thorough evaluation cannot happen in the absence of some designed artifact that is to be evaluated.

Our hope in writing this paper is to illuminate these issues and begin a discussion on solutions to bridging the middleware gap. In particular, our goals are motivated by our own experiences in designing, building, and evaluating not only infrastructure technologies, but also user-visible applications built using those infrastructure technologies.

This paper frames our discussion in terms of a set of case studies, based on our own work. Perhaps most interestingly, all of the systems described in these case studies were motivated by human, not technical, concerns. That is, their goals were to directly improve a user experience, not simply improve a system along some technical metrics (such as scalability or robustness), which are even further removed from the user experience. These case studies illustrate, in specific terms, the problems inherent in trying to create useful and usable middleware in the absence of good design and evaluation techniques.

CASE STUDY ONE: PLACELESS DOCUMENTS

The Placeless Documents system [6] is an infrastructure designed to enable fluid storage, exploration, and use of information spaces. The system was designed with a specific goal in mind: to enable applications that break away from 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, meant to reflect the structure of his or her work. These structures are often created early—at the beginning of a project, for example—and rarely change—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 is forced to store a piece of information. More importantly, 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.

At the project's inception, the ideal goal was that Placeless would *become* your filesystem. That is, folders and files—in their current incarnations—would go away, to be replaced by the more flexible notions of Placeless collections and documents. Additionally, Placeless would be used as the single repository for *all* types of documents, including email messages and web pages, not just files. In short, Placeless would unify the currently disparate mechanisms for accessing all sorts of documents, and provide a flexible set of features for storing, organizing and using these documents.

Experiences of the Middleware Design Gap

Placeless was intended to support a fluid form of information organization not found in current applications. A major problem that consumed many hours of project meetings and discussions, over the 3 years of the project's duration, 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 a number of application contexts that could benefit from the fluid organizational structures afforded by Placeless. These scenarios spanned a number of domains, and included “ideal” scenarios—completely replacing the desktop file-and-folder metaphor with a more powerful scheme—as well as a number of others, including mobile access to information spaces, collaborative document use, and so on. The applications that resulted from these scenarios included “better” versions of existing ones (such as file browsers and mail readers), as well as completely new applications.

From these application scenarios we then identified specific requirements of these applications and, from these, the technical features that would be required to implement them.

Virtually every application scenario we identified required technical features core to the goal of supporting fluid organization. These included a flexible model for document metadata, coupled with a query mechanism in which dynamically-evaluated queries could be embedded within document collections.

Additionally, a set of other features was required, either by individual scenarios, or by broader sets of the scenarios. For example, many of our application scenarios depicted users with different roles, moving about and collaborating in the use of documents. Supporting such applications required 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 another person). Our mobile use scenarios

required replication features (so that documents can coexist on multiple machines simultaneously).

Lessons from the Design Gap

All of the features mentioned above were motivated by our application scenarios, and by a desire to provide a functional, 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 that motivated the project.

The Dangers of Feature Explosion

This experience points out what we believe to be a key danger when designing middleware. While features such as security, distribution, replication and so on, were all required by certain scenarios, were technically interesting in themselves, and were perhaps even necessary for any “real world” deployment of the infrastructure, they were not central to the value that the infrastructure promised to deliver. More directly, these secondary features have *no* value if the primary features are not successful—if Placeless-derived applications do not provide better organization of information, it matters little if they're also secure or distributed.

One can even go so far as to argue that *until* a specific application domain is selected and validated, such secondary features distract from the design of the infrastructure. In the case of Placeless, in the absence of a focus on a particular application domain, and strict requirements derived from that focus, the infrastructure became 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 proved.

Lesson 1—Prioritize core middleware features: *It may make more sense at the outset to build a minimalist infrastructure to test the core design ideas and then—once those have been evaluated—to use feedback to motivate any new features that might be required.*

Experiences of the Middleware Evaluation Gap

From the project's beginning, it was clear that fully exploiting the Placeless infrastructure would require new applications. Users' current tools—the Macintosh Finder, Windows Explorer, their email applications, and so forth—did not provide access to the novel features afforded by Placeless; only custom-built applications could provide an opportunity to evaluate these features.

We decided on a two-part strategy for evaluation. First, we believed that the capabilities of Placeless would be best understood under “real world” conditions. That is, by providing users with enhanced replacements for their everyday tools, such as mail readers, we would be able to understand how an infrastructure such as Placeless could fit into everyday work practice. Second, we undertook to build a number of small “throw away” applications, designed to demonstrate some aspect of the infrastructure,

while not necessarily providing all the functionality that would be required in a “real” application.

Evaluation in Real Use Contexts

For the first evaluation path, we created two fairly heavyweight applications to test the infrastructure. The first was a “Placeless-aware” mail system. This was a mail application that would not force users into creating fixed folders of messages. Instead, message structure would be emergent, according to the task at hand. Messages could readily exist in multiple folders, and entire folder hierarchies could be created on the fly according to some ad hoc organizational criteria the user wished to impose. The second application was a web portal-style tool designed to provide the benefits of Placeless organization to an actively shared collection of documents on a web server.

Both of these applications were designed to be robust enough for long-term use. Additionally, since we believed that users would be unwilling to forgo their existing applications and data completely, we built a set of migration tools designed to allow users to use both existing, non-Placeless tools alongside these new applications, and 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 to demonstrate certain features of the infrastructure. These applications, while engineered to be neither robust enough nor featureful enough for long-term use, were easy to build, required few engineering resources, and were meant to demonstrate the novelty of the core infrastructure.

These tools used Placeless as a platform for experimenting with workflow and collaboration [11]; to address shared categorization schemes [7]; and other experimental user interfaces.

Lessons from the Evaluation Gap

Evaluating an infrastructure such as Placeless, with its broad implications and lofty goals, was problematic. While the chief lesson we learned could perhaps be summed up as, “*prioritize your evaluation criteria correctly,*” the sections below discuss the 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, in particular, proved problematic. For a real use context, we believed that users would be unwilling to adopt wholly new applications and tools,

especially if this meant having to give up their existing tools, and perhaps even data. Our strategy, then, was to ease users’ migration to a set of Placeless-based applications by creating multiple backward-compatible means to access and store the documents in Placeless. Tools were constructed to allow the Placeless infrastructure to be accessed as if it were a normal filesystem (facilitating the use of existing file browsers), or a web server (facilitating existing web browsers), or an email server (facilitating existing mail clients). Likewise, the actual content of documents stored in Placeless could reside in existing filesystems. The goal was to ease users’ transition into Placeless-based tools, and to 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 in itself to facilitate an evaluation of the features of the middleware, nor did it prove the validity of the class of applications the middleware 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, which would be used in an evaluation.

While required to support evaluation in a real use context, none of these tools in themselves satisfied the prime objective of *leveraging the platform* or demonstrated the platform’s 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 with high fidelity for expressing the core objectives of the middleware: In other words, initial prototypes should leverage the fullest extent of the power of the middleware platform, since the assumption that this power makes a difference is the most important 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 the mailer presented useful tools for organization if it did not also fulfill users’ other expectations about the features a mailer should have. In this setting, it became impossible to evaluate the Placeless-derived features of the mailer because they were overshadowed by the lack of other features that would have demanded more effort to get right.

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 to the general-purpose project management tasks that it was intended to support.

Lesson 3—Any test-application built to demonstrate the middleware must also satisfy the usual criteria of usability and usefulness: *These criteria are difficult to satisfy, since they require all the same careful requirements gathering, prototyping, evaluation, and iteration as any other application. The more ambitious the application, the more these criteria will come into play.*

The Placeless mailer failed to meet the requirement of usability, while the web portal failed to meet the requirement of usefulness. While these are perhaps the most obvious criteria for any applications, they can fall by the wayside when the goals of the developers are not to evaluate the applications themselves, but rather the infrastructure on top of which they are built.

Assess Intrinsic Value of the Platform Early

The Placeless project made mistakes by believing in a “real world” scenario and building extensive support for this scenario, none of which leveraged the basic value of the infrastructure. This mistake was later compounded by a selection of heavyweight evaluative applications that failed to meet the useful and usable criteria.

Only the lightweight, proof-of-concept applications proved to provide an interesting evaluation of the power of the infrastructure in creating 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 were required.

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

Discussion

The lessons to be learned from these experiences, while perhaps obvious in hindsight, were not foreseen by the developers of the middleware at the time. The developers based the feature set of the infrastructure on a thoughtful 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 in finding appropriate outlets for evaluation of the technology. Without a focus on what the actual objectives of the test-applications were—namely, to test the power of the middleware platform and to reveal any shortcomings—the design requirements quickly ballooned.

This expanding design, of course, afforded even more outlets for evaluation. While the empirical observation of real use is often an essential tool in proving the worth of an application, a focus on realistic use was misplaced, at

least in the early stages of the project. Significant engineering resources are required to support such an evaluation.

Instead, a more lightweight approach to evaluation would have been appropriate, especially given the experimental nature of the middleware. Such an evaluation would have begun with a number of small, easy-to-build applications that leveraged the unique features of Placeless, and then—if those features were found to be useful—proceeded to a more long-term evaluation. Any necessary “firming up” of the infrastructure, as well as testing of application usefulness and usability, would have occurred at this stage. All of this condenses down to 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 middleware: *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 middleware.*

CASE STUDY TWO: CONTEXT TOOLKIT

The Context Toolkit [5], an ongoing research effort from a different team than the one that developed Placeless, is a piece of middleware intended to support the design and construction of context-aware applications. Context-aware applications are those that dynamically adapt to users’ context: identity, location, activities, environmental state, etc. The Context Toolkit was built in the spirit of “off-the-desktop” or ubiquitous computing. It was intended to allow programmers to explore the space of context-aware computing and to provide users with more situationally appropriate interfaces to and interactions with their computing environments.

Traditional applications have little or no access to context information and are left without the ability to adapt their behavior to user, system or environmental information. These applications do not serve users optimally in dynamically changing situations.

Further, in a ubiquitous computing environment, with many possible 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 therefore has been used infrequently, even in ubiquitous computing system prototypes.

To address this problem of acquiring context, the Context Toolkit treats context as a first-class citizen, providing applications with easy access to contextual information and operations to analyze and deal with it. This is intended to serve two purposes. First, application programmers do not need to worry about the details of acquiring context but are able to simply leverage off of existing mechanisms. In this way, context information is

made as easy to use as traditional keyboard and mouse input. This support allows programmers to concentrate their efforts on designing the application itself. Second, end-users of these applications are provided value-added behavior by having information and services that have automatically adapted to their current situation.

Experiences of the Middleware Design Gap

Survey of the State of the Practice

Our methodology for designing the Context Toolkit comprised four parts. First, we examined existing context-aware applications in the literature to derive a set of necessary features of a context-aware architecture. Second, we looked at existing support for building context-aware applications and made a list of the features and examined the complexity of the applications they were able to support. Third, we performed informal interviews with the builders of context-aware applications (almost all were researchers) to determine what features they would want in the Context Toolkit. Finally, we also 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 [5].

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 a collection of all the device drivers one might need to create 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 has been created.

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 (mood detection, for example). In the evaluation section, we will discuss the impact that simulation had on evaluation of the middleware.

Lessons from the Design Gap

Providing a Basic Infrastructure for Programmers

We designed and constructed an infrastructure to support a set of carefully chosen features only to find that some of the features were unnecessary, while some were too difficult to get right or too limited to be truly 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 middleware for supporting context-aware applications complete, they were not necessary for supporting programmers in 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 found ourselves designing 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 middleware. This reveals yet another crucial lesson:

Lesson 6—Do not confuse the design and testing of experimental middleware with the provision of an infrastructure for other experimental application developers: *It is hard enough to anticipate applications you yourself might build, without inviting and trying to support all means for others to push the envelope of possible features even further. By the time developers are let loose on the middleware it must be relatively stable. If developers begin to demand—or even attempt themselves to make—changes underneath each other’s live applications, this is a situation likely to lead to growing numbers 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.

Experiences of the Middleware Evaluation Gap

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

We built a large number of applications ranging from very simple prototypes that demonstrated only the use of the core features of the toolkit to very complex systems that used most of the toolkit features. From an evaluation standpoint, the very simple prototypes were quite different than the complex systems we built. They were easier to build and more succinctly illustrated the value of the toolkit and context-awareness in general as suggested by lesson 4. Furthermore, while they were mainly developed to be short-term demonstration prototypes, they ended up being the applications that were used over a long period of time. Some example prototypes include an In/Out Board [14]; a context-aware mailing list that forwarded sent mail to the current occupants of a building as opposed to all the people that spend time in a building [5]; and a people tracking tool for the home environment.

Lessons from the Evaluation Gap

The Value of Lightweight Prototypes

As with the Placeless experience expressed in lesson 3, it turned out to be the more sophisticated applications that provided less return on investment. We built a conference helper that assisted conference attendees in finding relevant presentations, taking notes and accessing the notes at a later date [3]; a reminder tool that used contextual information beyond the use of time-based alarms to trigger reminders [4]; and an intercom system that follows users around a building and uses context to determine when they can receive an incoming call [12]. 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 an attempt to be very generic and design a toolkit that could support many different types of context, it became very difficult to come up with strict guidelines on how to use the programming abstractions in the Context Toolkit. 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—Be sure to define a limited scope for test-applications and permissible uses of the middleware: Unconstrained interpretation of the purpose of the middleware leads to confusion both about what features are important, and also about how best to use them, making it very difficult to attribute successes or failures to real advantages and limitations of the middleware, rather than 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 [4, 9, 12].

From a user's standpoint, as applications became more complex, the mental models required to understand the mapping between their context and application action also became more complex, and often too complex. For the simulated applications, users could not truly get a good sense of how the applications provided benefit because too much was faked, from the sensing technology to the situation that the user was in. For the systems that used real data, the applications were often too new and different from what users were used to, making them less attractive for users to use and making them more difficult to evaluate. Finally, the collection of context has serious privacy implications. Without a mechanism to convey what information was being captured and how it was being used, users were hesitant to participate in using these applications. While user skepticism appears to be a special issue for context aware systems, the problem concerning simulation provides a more general lesson:

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 middleware 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 were exacerbated as the sophistication of the context-aware application

increased. Instead of being able to experiment with a wide variety of applications, programmers could really only evaluate the simple ones, as only they could be put into real, 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.

CASE STUDY THREE: SPEAKEASY

The genesis of Speakeasy [8], an ongoing research project at Xerox PARC that has benefited directly from the lessons learned in the Placeless project, is the observation that current problems with interoperability will only grow as networked devices and services become more predominant. Currently, for a device such as a PDA to be able to use a printer, the software on that PDA must be explicitly written to use not only the *type* of thing with which it will interact (“Printer”), but often also the specific *instance* of the thing (“Xerox NPS700”). With ubiquitous networking, more and more devices will be potentially reachable from one another. Will the networked world of the future be one of isolated islands of interoperability, or will we be able to fluidly interconnect arbitrary devices and services, with no advance planning, implementation, or software installation?

The technical intuitions that guided the infrastructure are that we will never be able to build in *a priori* support for all possible devices into any software; there will *always* be new devices, and new classes of devices, that we will encounter but which our software is not explicitly written to use. In the absence of such hard-coded support for the devices around us, what is the best we can hope for?

Speakeasy provides a set of primitives that allow devices and services to interoperate on a very low level. These primitives include notions such as data transfer and discovery. Importantly, the *semantics* of these devices is not encoded in these primitives; instead, knowledge about what a particular device *is* or *does* is left up to the user. For example, in the Speakeasy world, a PDA that is not running software that is explicitly written to use printers would still be able to communicate with any printer it encounters. But it would never do so unless told to do so by a user, who presumably understands what a printer is, when to use it, whether it’s in someone’s office that doesn’t want to be disturbed, and so on.

Experiences of the Middleware Design Gap

The design of the Speakeasy infrastructure was guided by the overall technical philosophy of the project—semantically neutral interfaces, coupled with the ability of humans to make reasoned decisions about when and whether to use a device—but it was also informed by iterative feedback from applications.

Scenario-Based Design

As in the Placeless case, the Speakeasy developers used scenario-based design to hone in on a set of possible embodiments for the technology. Among other things,

these scenarios were designed to answer one key question early in the course of the project: how were users expected to interact with the devices in their environments? Would they carry a full-blown laptop computer that would allow them to interconnect and control the devices they find around them? Would they use a handheld computer as essentially a “universal remote control?” Would they carry no computer at all, instead controlling their surroundings simply by physical interactions with them?

We decided to initially pursue the first two of these scenarios, believing that exploring a range—albeit, a narrow range—of usage settings would provide good feedback to the infrastructure development. To that end, we built two “browser-style” applications. These were tools that would provide the user’s “front end” to the Speakeasy world, allowing him or her to interact with and control any device in the environment.

Iterative Development of Multiple Applications

The first browser was designed to use a full-blown graphical interface, and was also a “peer” of the other devices in the Speakeasy environment. In other words, rather than simply providing controls for discovered devices, it could itself participate in sharing data with any discovered devices.

The second browser was a web-based tool, designed for use from a handheld computer with a wireless network card. This application—since it was built using web technology—had a more minimalist interface, and acted essentially as an extensible remote control.

These applications were built in parallel with not only the infrastructure, but also each other. Each stressed different aspects of the infrastructure, and so provided feedback on different infrastructure features.

Lessons from the Design Gap

Rapid Iteration Benefits Middleware, not Just Applications

The benefits of rapid iterative design-develop-evaluate cycles are well-known as a means of interface and application design. But we believe that similar cycles can also be useful for middleware design.

In particular, the Speakeasy browser applications were constructed in parallel with the infrastructure using such a rapid design-develop-evaluate cycle. These cycles were useful in determining several key features in the infrastructure. In particular, we quickly noticed that users would easily become confused in settings with large numbers of devices, or when those devices didn’t behave in ways the users would have predicted. These problems were especially apparent in the “narrow bandwidth” web-based tool. This feedback drove the addition of new technical features in the infrastructure, including the ability to query a device to see what it’s doing (to determine if it’s busy, for instance), and to retrieve more information about the set of devices in an environment.

Lessons 1-5, learned from Placeless, stood us in good stead here, in that we only added infrastructure features critical to the applications, and our early test-applications were so lightweight that they could be thrown away easily.

Experiences of the Middleware Evaluation Gap

Traditional Application Evaluation Techniques can Also Benefit Middleware

Since the Speakeasy browsers afforded completely new ways of interacting with networked devices and services, we believed it was essential to test the actual usability of the browsers themselves. Even though our experiences, as discussed in the design section, indicated that users would prefer not to use a generic browser exclusively, it was clear that they would use such a tool if none other were available. If the features provided by the infrastructure could not be used to yield a usable browser, then this would provide important guidance for the infrastructure.

To this end, both browsers were evaluated, although through different degrees of formality. The Speakeasy developers used the laptop-based tool regularly as a way to test both the infrastructure and the devices meant to be accessible through Speakeasy.

We conducted more formal evaluations of the web-based browser tool on a handheld computer, with actual users, through three different approaches. First, we used a task-oriented walkthrough of a paper prototype of the UI design, where users were instructed to use the UI to complete a number of tasks, for example, finding and selecting a projector to use for a presentation.

Second, we built an online prototype based on feedback from the paper prototype. Again, users were given a set of tasks and we collected data on their experiences.

Finally, we performed trials of the actual, running browser during real events. For example, users would give a presentation using the browser, and would have to interact with real devices in their surroundings.

As mentioned earlier, this progression from extremely lightweight prototypes to actual code gave the infrastructure developers the opportunity to evaluate the features provided to the browser, and modify the infrastructure accordingly.

Lessons from the Evaluation Gap

Beware the Misleading Scenario

We believed that the chief value of Speakeasy was in its ability to allow users to create ad hoc assemblages of devices and services to accomplish some function, even in the absence of any particular domain-specific application for that function. This was the reason for our focus on generic browser-style applications in many of our scenarios: the browser was to be the tool that provided access to unforeseen functionality, and allowed users to interconnect devices and services in unplanned ways.

To a degree, the browser applications we built were successful, and certainly avoided the problems encountered in the other case studies by leveraging the system's power without demanding the implementation of extraneous, heavyweight application features.

However, we quickly learned that users often did not want to go through the machinations of using a generic tool to access the system. For example, users told us that, to access specific functions such as displaying a PowerPoint file on a projector, they'd prefer to have some specific "wireless PowerPoint" application, if such a thing existed. Users asked for the creation of specific applications to accomplish specific configurations of devices. This experience—while validating the ability of Speakeasy to support the ad hoc composition of a function such as "wireless PowerPoint"—indicates that a highly generic, browser-style applications is not a preferred embodiment of the technology. The browser is only good insofar as it does allow access, for user evaluation purposes, to functionality for which no specific application yet exists. If a specific application *does* exist, users will likely prefer it to a generic browser.

In this case, the scenarios used to drive the design were selected by a desire to maximally leverage the underlying technology. These scenarios were supported by rapid iterative design that led to new infrastructure features. Only once in actual use, however, did the negative user feedback appear.

Such feedback can be an important tool for informing design, of course. More directly, such feedback can go beyond simply guiding the design of an individual application, and affect the design of the middleware itself. In this case, the feedback led to two important new directions in the infrastructure. First, we are using the feedback to design new infrastructure features to ease the transition between specific applications (which users prefer) and generic browsers (which we believe will always have a place). Second, since browsing is more palatable if it is embodied in the tools users are already comfortable with, we are working to build browser-like functionality into existing applications, as well as new applications created to work in the "Speakeasy style."

Lesson 9—Understand that the scenarios you use for evaluation may not reflect how the technology will ultimately be used. *Evaluation scenarios are just that—tools for evaluating the middleware, not for determining its market niche. While evaluation scenarios may provide hints about future uses, the process of determining the most appropriate embodiment of the technology comes from further scenarios, market analysis, and studies of the intended setting of the technology.*

Application "Coverage" is Essential

While the rapid iterative design-develop-evaluate cycle provided useful feedback on infrastructure features, it only

provided feedback on the features exercised by the particular application set we created.

As mentioned above, our scenarios led us to the creation of highly generic browser tools, along with the necessary infrastructure refinements to better accommodate such tools. Any evaluation of the middleware based on this limited set of applications is limited, however, in that it does not stress infrastructure features that might be needed to build more domain-specific applications.

In other words, the “coverage” of application test cases is limited. This experience points to the need to develop a wider range of applications atop the infrastructure. The lightweight Placeless applications, for example, were largely successful because of their range of coverage of infrastructure uses. So despite the fact that we benefited from, and responded to the lessons learned in Placeless in this project, yet another lesson remains to be learned here:

Lesson 10—Anticipate the consequences of the trade-off between building useful/usable applications versus applications that test the core features of the middleware: *It may be that the best applications to start with are not the best applications from the users’ point of view. Nonetheless, one can expect to discover—and thus have to build later on—important useful extensions to the middleware as a result of getting users to try out a range of simple core-feature-testing applications early on.*

DISCUSSION AND CONCLUSIONS

So, what have we learned from our three examples about the challenges of bridging the user-centered design and evaluation gaps for middleware. What we find is a list of apparently commonsense lessons. 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 Middleware Features.
- Lesson 2—First, build prototypes with high fidelity for expressing the main objectives of the middleware.
- Lesson 3—Any test-application built to demonstrate the middleware 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 middleware.
- Lesson 6—Do not confuse the design and testing of experimental middleware with the provision of an infrastructure for other experimental application developers.
- Lesson 7—Be sure to define a limited scope for test-applications and permissible uses of the middleware.
- Lesson 8—There is no point in faking components and data if you intend to test for user experience benefits.
- Lesson 9—Understand that the scenarios you use for evaluation may not reflect how the technology will ultimately be used.
- Lesson 10—Anticipate the consequences of the trade-off between building useful/usable applications versus applications that test the core features of the middleware.

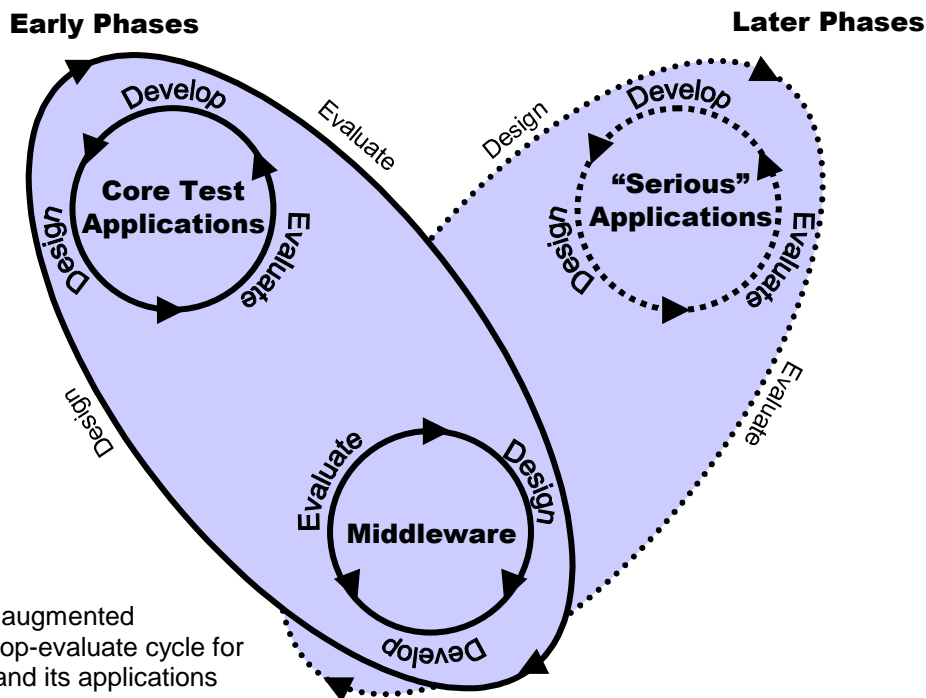


Figure 2: An augmented design-develop-evaluate cycle for middleware and its applications

How to Bridge the Middleware Design and Evaluation Gaps

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

Based on our experience, user-centered middleware design does indeed require application design to demonstrate the power of the middleware. But the focus at first must be on applications with simple proof-of-concept goals only (core test applications). These applications are selected based on their fidelity to the features of the middleware (the degree to which they express the power of the middleware), 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 middleware). In the cycle shown in the left of the diagram, the design-develop-evaluate cycle for the test applications is a very lightweight process, and the design and evaluation feedback to the middleware concerns core features only.

Only in later cycles should serious applications be built as these will have demanding engineering and usability requirements of their own that do not serve to prove the success of the middleware. These later cycles are shown to the right in Figure 2. There, the design-develop-evaluate cycles for the so-called "serious" applications is a much more stringent process, since these are intended to be applications that will be deployed for real use. By this point, the design and evaluation cycles for the middleware 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 focusing on the parts of the figure that are represented by dashed lines. They became overly concerned with supporting, engineering and evaluating serious applications rather than with a user-centered evaluation of the middleware via simple, core feature-oriented test applications. Speakeasy benefited from many of the lessons learned in Placeless, but still suffered from a number of problems, which in turn led to the realization that core test applications are not, in fact, the same as the serious applications that users really want. Despite this, such test applications are still essential, demonstrating as they do the power of the middleware and pointing to needs for the ultimate applications of the technology.

ACKNOWLEDGEMENTS

Thanks to the members of the various projects used as case studies in this paper.

REFERENCES

1. Apple Computer, Inc. *Macintosh: Macintosh Toolbox Essentials*. Apple Technical Library, 1993.

2. Cousins, S.B., Paepcke, A., Winograd, T., Bier, E.A., Pier, K. "The Digital Library Automated Task Environment (DLITE)," *Proceedings of ACM International Conference on Digital Libraries*, 1997.
3. Dey, A.K., Futakawa, M., Salber, D., Abowd, G.D. "The Conference Assistant: Combining Context-Awareness with Wearable Computing," *Proceedings of the International Symposium on Wearable Computers (ISWC)*, 1999.
4. Dey, A.K., Abowd, G.D. "CybreMinder: A Context-Aware System for Supporting Reminders," *Proceedings of the Symposium on Handheld and Ubiquitous Computing (HUC)*, Springer-Verlag, 2000.
5. Dey, A.K., Salber, D., Abowd, G.D. "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications," *Human-Computer Interaction (HCI) Journal*, Lawrence Erlbaum Associates, Vol. 16 (2-4), December 2001.
6. Dourish, P., Edwards, W.K., LaMarca, A., Lamping, J., Petersen, K., Salisbury, M., Terry, D.B., and Thornton, J. "Extending Document Management Systems with User-Specific Active Properties." *ACM Transactions on Information Systems*, 18(2), 2000.
7. Dourish, P., Lamping, J., and Rodden, T. "Building Bridges: Customisation and Mutual Intelligibility in Shared Category Management." *Proceedings of the ACM Conference on Supporting Group Work (GROUP)*, 1999.
8. Edwards, W.K., Newman, M.W., and Sedivy, J.Z. "The Case for Recombinant Computing," Xerox PARC Technical Report CSL-01-1, April, 2001.
9. Espinoza, F., Persson, P., Sandin, A., Nystrom, H., Cacciatore, E., Bylund, M. "GeoNotes: Social and Navigational Aspects of Location-Based Information Systems," *Proceedings of UBIComp*, Springer-Verlag, 2001.
10. Hudson, S.E., Stasko, J.T. "Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions." *Proceedings of ACM Symposium on User Interface Software and Technology (UIST)*, 1993.
11. LaMarca, A., Edwards, W.K., Dourish, P., Lamping, J., Smith, I., and Thornton, J. "Taking the Work out of Workflow: Mechanisms for Document-Centered Collaboration." *Proceedings of the European Conference on Computer-Supported Cooperative Work (ECSCW)*, Copenhagen, Denmark, 1999.
12. Nagel, K., Kidd, C., O'Connell, T., Dey, A.K., Abowd, G.D. "The Family Intercom: Developing a Context-Aware Communication System," *Proceedings of UBIComp*, Springer-Verlag, 2001.
13. Roseman, M., Greenberg, S. "GROUPKIT: A Groupware Toolkit for Building Real-Time Conferencing Applications," *Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW)*, 1992.
14. Salber, D., Dey, A.K., Abowd, G.D. "The Context Toolkit: Aiding the Development of Context-Enabled Applications," *Proceedings of the 1999 Conference on Human Factors in Computing Systems (CHI)*, 1999.