Support for Context Monitoring and Control

Author(s) and institution(s) removed for blind submission

Abstract. Monitoring and control are important user interactions in context-aware applications that significantly affect user frustration. Because they directly involve users, these interactions should be designed when an application's users and uses are known, so they can best meet user needs. Supporting interface designers in building monitoring and control interfaces for context-aware applications requires application logic to be exposed in some structured fashion. As context-aware infrastructures do not provide generalized support for this, we extended one such infrastructure with enactors, components that enable monitoring and control interfaces while placing minimal burden on an application developer and facilitating designer access to application state and behavior. We developed support for interface designers in Visual Basic and Flash. We demonstrate the usefulness of this support through the augmentation of four common context-aware applications, and through an informal study of experienced Flash designers using this support.

1 Introduction

Context-aware applications utilize context – information regarding the state of entities that is relevant to interaction with users [9]. In this paper we focus on two classes of end-user interaction with context-aware applications, *monitoring* and *control*. Context monitoring is any interaction where a user retrieves contextual information from a context-aware application (*e.g.* monitoring a friend's location). Context control is any user action that intentionally results in a change in context processing (*e.g.* changing from sending alerts when a friend is within 50 meters of your location, to within 10 meters). Context monitoring and control are necessary for supporting users in understanding what an application is doing and how to change it and this support will significantly impact adoption of context-aware applications. Because context monitoring and control involve application state, and context-aware applications often possess distributed state, these interactions can be challenging to implement and support. Infrastructure support may be of help. However, we propose that any such support should *extend to interface designers*, who are more aware of an application's users and their tasks, *in addition to developers* who produce reusable context components. In this paper we present a solution that exposes the internals of context-aware applications and facilitates the design of monitoring and control interfaces from multiple interface development platforms. By enabling designers to customize previously inaccessible applications, we broaden the domain of people able to develop context-aware applications beyond the systems programmer and enable more usable applications.

The general structure of a typical context-aware application is displayed in Fig. 1a. Context input from either sensors or users is made available to applications. Application logic is programmed to acquire and analyze input, and issue or execute context

output when appropriate. This includes controlling actuators, modifying data or notifying user displays. The acquisition of context input and the execution of context output is often implicit, performed without direct user interaction [21]. Context monitoring and control, shown in Fig. 1b, are, by definition, explicit user interactions. They rely on application logic being exposed in some way, in order to access not only the state of context input and output in an environment, but also any analysis or state resident in the application that is processing inputs and outputs.

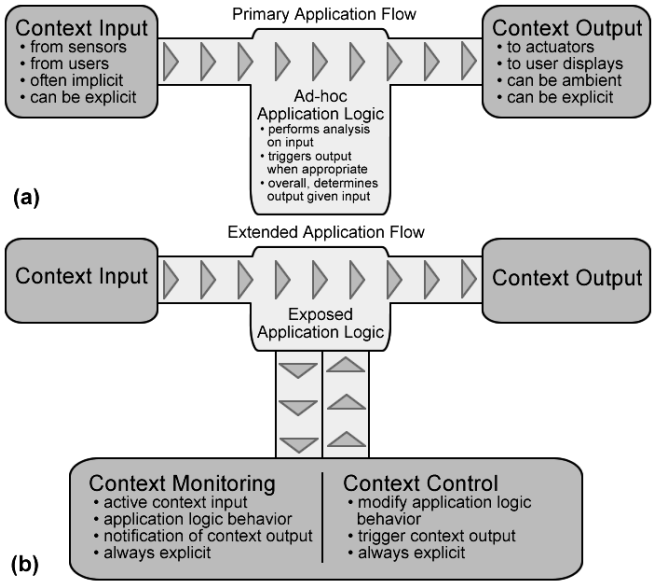

Fig. 1. Context-aware applications structure (a) without and (b) with monitoring and control.

Bellotti and Edwards state that two key design principles for context-aware systems are informing the user of the system's understandings (*monitoring*) and providing *control* to the user [3]. A recent study of context-aware systems showed that users become very frustrated when they do not understand why a system has performed an action, or have the ability to fix it [2]. Monitoring and control interactions are a significant part of context-aware applications and have a large impact on adoption. There will always be situations where users want to actively retrieve or modify application state. One such situation is remote monitoring and control under unexpected conditions. Consider a home with a context-aware security system, where a delivery person has arrived with a package. The homeowner is away at the office and notified of the delivery by the home system. She remotely allows the deliverer into the home foyer while locking access to other rooms, and confirms through system video feeds that the delivery person exits the home. The homeowner might desire such explicit monitoring and control even if the system could handle such a protocol on its own. Another example is user response to perceived application error. Consider a home lighting application that turns on lights in a home for occupants, at the same time trying to save energy costs (*e.g.* [17]). During normal operation this action is completely implicit,

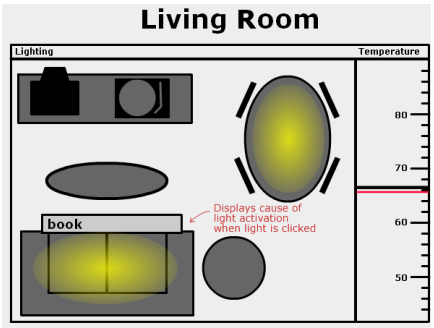

Fig. 2. Unified Room Control interface.

turning lights on and off based largely on user movements and object locations but not according to user commands (Fig. 2 shows a monitoring and control interface for such an application). However, if the system performs unexpectedly or erroneously, *e.g.* turning off a light in a room where a user is reading, that user will likely shift into a set of explicit interactions with the application, perhaps trying to figure out why the system turned the lights off and almost certainly trying to turn the lights back on. While an extreme case, evidence from the MavHome shows that the lack of a monitoring and control interface can result in a very frustrating user experience [23]. The MavHome learned lighting behaviors over time with occupants who did not have visitors late at night. When an occupant moved in that had guests over late at night, the lights remained off, not having had time to learn the new occupant's patterns. Apparently the occupant chose to literally "remain in the dark" because there were no mechanisms for him to control the home directly. These examples show that although context-aware applications may operate implicitly, they will inevitably involve explicit monitoring and control interactions from users. Explicit interaction demands that applications have interface(s) with which users can readily interact, to avoid user annoyance and, ultimately, rejection of the applications.

In our work, we focus on supporting designers in building monitoring and control interfaces. Effective design of these interfaces is at heart an interaction design problem – it requires specific knowledge of the application users and their tasks or activities. Interface designers are skilled at gathering this knowledge and in performing interface design. It is often difficult to know what monitoring and control support is needed when the application is being built. But even when this is known, as the use and/or the users of the application change over time, the type of support needed will invariably change and leverage designers' skills even more.

There are three main challenges in performing this work. First, interface designers have limited general programming ability. Rather than simply augmenting a context-aware infrastructure to support monitoring and control, any solution must entail extending that solution to designers in programming environments that they are familiar with. Second, because it is valuable to separate the building of the application from the design of the monitoring and control interface, there must be support for building these interfaces after the application has already been designed and deployed. Designers may wish to customize the presentation of information to meet particular user needs, or compose multiple context-aware applications through a coherent interface to improve usability. Our third challenge is that the ability to perform this sort of interaction design can actually be compromised by context-aware toolkits and infrastructures that promote component reuse [3]. Reuse implies that the design of components such as sensor abstractions occurs early in a design process, before any users or tasks are definitively known. Effort must be employed to ensure that an infrastructure supports the construction of reusable components but also supports access to those components

in a way that encourages flexible interaction design at the application level. This point has been argued for collaborative systems in general [10], and for feedback and control of context-aware systems in particular [3]. At the same time, such support should enable interface design while placing minimal extra burden on application developers.

Our goal is to expose internals of context-aware infrastructures so that interaction designers (as well as programmers) have the ability and freedom to develop user interfaces to support the explicit interactions of context monitoring and control. To this end we have implemented enhancements to an existing context-aware infrastructure, the Context Toolkit (CTK) [9], that provide rich access to context-aware components and application logic. The enhanced API, a component called an *enactor*, was architected while considering the needs of monitoring and control interaction design. Enactor clients are available for designer use in a variety of platforms, including Java, Visual Basic, and Flash. They enable designers to build monitoring and control interfaces both during and after application deployment. Designers can build new and more usable interfaces for existing context-aware applications and combine multiple applications together into a single, more usable interface. In doing so, users of these applications will be less likely to reject these applications out of frustration.

In the remainder of this paper we will show that structuring context-aware application logic through enactors provides designers with support to help users monitor and control context-aware applications. After surveying related research, we describe in detail the enactor and platform extensions we have implemented. Then, we demonstrate the usefulness of these extensions through a number of applications. Finally, we describe an informal designer evaluation of the Flash extensions that illustrates how designers can effectively access the internal logic of context-aware applications to support end users.

2 Related Work

Context-aware applications utilize context in their environments, and will often take action on that context without explicit input from users. Schmidt, for instance, considers this phenomenon "implicit human-computer interaction" [21]. Bellotti and Edwards argue that for such systems to be usable they must make provisions for explicit interaction [3]. Our work falls squarely within the issues framed by these researchers. We are trying to provide better support for a particular class of explicit interactions, monitoring and control, by extending a toolkit that already supports the sort of implicit interaction Schmidt describes.

Several infrastructures exist that support context-aware computing applications. In general they address the challenges involved in using and reusing a distributed set of computing resources in a variety of applications, providing services such as asynchronous message communication, resource discovery, event subscription, and platform independent identification and communication protocols. Examples include JCAF [1], EasyLiving [4], Cooltown [5], Solar [6], iQL [8], and the CTK [9]. All of these implement mechanisms to access context data, and some provide mechanisms to invoke services. None, however, offer higher-level abstractions that describe the actions an application takes and the context data involved in those actions as an accessible unit.

Enactors complement these infrastructures in providing an accounting of input and how that input maps to output in the form of actions and services, as shown in Fig. 1b.

Our proposed enactors are a componentization of application logic to support inspection and manipulation via an established API. They organize organize applications in a similar fashion to application component architectures such as JavaBeans™, Open Agent Architecture and XWeb [16,19]. Like the context-aware architectures described earlier, however, these architectures do not provide explicit support for the development of monitoring and control interfaces.

Some research has addressed aspects of monitoring and control. The Jigsaw Editor and iCAP both facilitate end-user control by allowing them to build their own context-aware applications [12,22], but do not support monitoring and control interfaces that users need for understanding and using context-aware applications. Cooltown, Speakeasy, and iCrafter address the need for ubicomp user interfaces (UIs) to be highly dynamic and adaptable, by delivering interfaces to users through template-based or automatic UI generation [5,18,20]. However, as Dourish maintains, enabling design customization in interfaces is crucial [10]. These 2 approaches toward effective user interaction with context-aware applications are complementary to our approach.

Our work attempts to support interface designers in developing interfaces for applications that were built at an earlier point using context-aware infrastructures. Context-aware systems are likely to be long-lasting and evolving. Dourish argues that such systems should support reflection and offer customizability during a continual design cycle [10]. He focuses on end-user customization, but acknowledges the importance of systems that allow customizability by individuals with varying degrees of expertise, as in the Buttons system [15]. Similarly, Mobile Bristol and Topiary demonstrate the value of supporting designers in building interfaces for location-aware systems [11,14], however none of these systems focus on support for monitoring and control.

In summary, current context-aware infrastructures greatly assist the creation of context-aware applications but do not provide support for context monitoring and control. Supporting designers in building monitoring and control interfaces has great value but little direct support exists for this. We address both of these issues in our work.

3 Architecture

In this section, we describe how enactors support monitoring and control of context-aware applications. Then we describe how we support the designers in creating monitoring and control interfaces for end-users through VB and Flash libraries.

3.1 Enactors

Although the application logic (or context-aware rules) that integrates data from context components benefits from regularity and reusability of those components, it is itself completely *ad hoc* and does not expose operational details. There is little way to support monitoring and control interactions for users except to either implement it when the entire application is built, or to access internals in some custom way at a

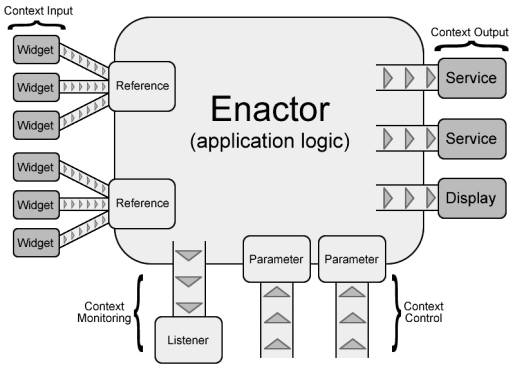

Fig. 3. Block diagram of the enactor receiving context data from inputs (references), monitoring changes in data and actions (listeners), exposing parameters for control, and using services and displays for output.

later time. Both strategies are unrealistic. We address this problem by componentizing the application logic already present in applications.

To accomplish this, we introduce *enactors* (see Fig. 3). To illustrate their value, take the home lighting system described in the introduction (Fig. 2). In most context-aware applications, the application logic would consist of finding a discoverer, querying it for relevant people inputs, and subscribing individually and maintaining connections to each of those inputs. When the application receives data from each input, it must maintain internal state information keeping track of each person's location. When someone's location matched a location of interest, an output or service would be called to change that location's lighting appropriately. With enactors, this situation is simplified. Now, the application logic consists of creating an enactor with a description of the information it is interested in (locations of specific people) and what the enactor should do when (set the lighting level based on the occupancy). The enactor handles the remaining logic: dealing with the discoverer, individual inputs and data, determining when input is relevant and executing appropriate output services.

Enactors are designed to allow developers to easily encapsulate this logic in a component. They have three subcomponents: *references*, *parameters*, and *listeners*. An enactor acquires distributed context input (*e.g.* location, light level) through sets of references. It processes information internally, exposing any relevant properties as parameters (*e.g.* bedroom light is *on* and *Joe* is in the room). Listeners are notified of occurrences within the enactor, such as any actions that are invoked (*e.g.* turn the lights down) and context data received. Unlike context inputs, enactors do not produce context input themselves, and operate only as consumers of context data. Enactor listener notifications occur in parallel to normal context dataflow.

References: Enactors may require data from a variety of context components. The context of interest is passed from the enactor to reference objects, which query the discoverer and subscribe to components that match this context. References notify enactors whenever components newly satisfy or fail to satisfy a match (*e.g.* when a new person input is discovered or an existing one fails), and whenever a matched component provides new context data and this information is passed onto listeners. These events signify that a context input change of interest has occurred. For instance, an enactor that manipulated lighting for a home in response to the presence of people would have a reference that matched person inputs, receive matches whenever new people entered the home, and receive evaluations whenever a person changed rooms.

Parameters: Application developers can expose parameters as part of their specification of application logic in the enactor and are key in supporting monitoring and

control. Parameters are analogous to JavaBean™ properties and parameters in other component frameworks. They can be read–only or read/write, and have a description advertising their type and what they do. Enactors inform listeners when a user changes a parameter value, signifying that the behavior of the enactor itself has changed. In our example lighting application, an enactor would offer parameters detailing the intensity to which a light should be set when a particular person enters a particular room.

Listeners: An enactor may execute an action at any time that results in context output. Output may include the execution of a service, or the delivery of data to some display. Enactors inform listeners whenever an action occurs or whenever context input is received. One particular enactor listener of interest that has been implemented is an *enactor server*. The enactor server translates listener method calls into XML and sends that XML to any connected clients (including Flash and Visual Basic clients). It also listens for XML sent to it, and modifies enactor parameters in response.

Enactor application design: In our work, we implemented enactors on top of the open-source Context Toolkit, however, we could have used one of the other frameworks described above that support discovery, context inputs and context outputs or services. In the Context Toolkit, discovery is a core feature, context input in handled by components called widgets, and context output by services. A context-aware application will typically contain one or more enactors, each encapsulating some unit of processing on context input. Conceivably every application could contain just one enactor that retrieved all context input needed and performed all processing by itself. However, grouping an application into sets of enactors maintains modularity and can encourage reuse. Moreover, the implementation of enactor references efficiently multiplexes context input subscriptions between enactors, so there is no additional operational cost on the context-aware system at large by applications with many enactors.

Once a developer has decided upon the number and function of enactors in the system, application development is similar to that of traditional context-aware applications. Acquisition of context input is actually made much easier because of the fully declarative mechanism provided by enactor references. Service execution occurs much like before. Enactors provide notification of events like context acquisition and execution through listeners with little additional effort for the developer.

The main difference between using enactors and traditional context-aware architectures is the parameter. Developers must determine which values in their application logic to expose and permit manipulation of. Indeed, this is the task we want to strongly encourage application developers to undertake, and enactors make the task easier than in an *ad hoc* scenario. All developers need do is declare parameters, since enactors provide mechanisms for others to inspect and manipulate these parameters in a standard way with no additional developer management. The enactor does not drastically impact context-aware application development processes, and where it does, the impact is largely positive, imposing little burden on the developer. It also provides tremendous benefit to designers and end users as we will describe in the next section.

3.2 Client Extensions

While monitoring and control interfaces could be built in Java using the CTK augmented with enactors, we added Visual Basic.NET and Macromedia Flash support

because they are the most commonly used design environment for graphical user interfaces. This opens up the possibilities for a large development community to build monitoring and control interfaces. We created extension libraries in VB and Flash that utilize the XML-based *enactor server* mentioned above. Enactor servers publish their locations on a public web page. With this information, a designer can build an interface. At design time, to discover the structure of a particular enactor, a designer can visit that server on a web browser. She will see a description of the enactor, names and types of its exposed parameters, reference queries, and descriptions of any currently matched components. The designer decides what information to extract and use.

Whereas access to the application logic of a traditional context-aware application would need to be implemented in a custom manner, enactors allow such access in a standard fashion. Our extensions allow arbitrary applications to monitor and control context-aware applications through enactors. Control interfaces can be designed and implemented in Flash or VB, independent from the main context-aware application.

Macromedia Flash: Macromedia Flash is an interface development tool that deploys programs that execute within the Macromedia Flash Player. The Flash interface is largely visual, using a timeline metaphor. It has XML support and allows scripting through ActionScript, which supports the use of objects, and provides object representations of most visual Flash elements. Flash is a similar medium to HTML in that it is typically delivered via HTTP, and the player is widely available. However, it departs from HTML in that it is intrinsically stateful and has been very consistently implemented on a variety of systems and devices. Monitoring and control interfaces written in Flash can execute on any computer or environment with network access to the extended CTK infrastructure. We implemented an enactor connection object in Flash that provides a set of custom high level events to Flash designers. It essentially extends the enactor listener interface into Flash and the high level events it provides map directly onto listener methods. Designers can attach custom handlers to these events using precisely the same semantics as event handling for all other Flash objects. For instance, our library contains `onComponentAdded` event handlers (for new reference matches) that are used in Flash in exactly the same way as the common `onPress` event used by buttons. Component description data is converted a native ActionScript data type (associative arrays). Flash interfaces can set enactor parameters via the connection object; parameter arguments are sent to the enactor via the enactor server.

Visual Basic: Visual Basic.NET is an extremely popular and more full-fledged object-oriented development environment. Typically applications written in VB are standalone executables. Similarly to our Flash work, we implemented an enactor connection object in VB that parses XML and creates an object representation of enactor notifications. This representation is closely modeled after our Java object implementation. Custom application code can be attached to enactor events using .NET delegates.

4 Demonstration Applications

For users to interact effectively with context-aware applications, they should be able to monitor and control them. Our implementation of enactors makes it easier for application developers to build their applications, while, at the same time, exposing

application logic. The client extensions provide designers with access to this logic and enable them to produce monitoring and control interfaces without having to implement the entire context-aware applications. Designers can produce interfaces targeted to the needs of specific users, independent of the original development process.

In this section we describe four applications implemented using enactors and the Flash communication library that demonstrate how users may benefit from increased designer support for the construction of context monitoring and control interfaces. Each interface provides monitoring and control capabilities to a CTK application using enactors. Because we are primarily interested in the interface design possibilities, we did not actually instrument spaces for these applications. Widgets provide context input abstraction, and we were able to feed widgets simulated data while shielding the rest of the application from the fact that context was being simulated.

Each of our applications is intended to explore particular aspects of the relationship between enactors and user interfaces for monitoring and control. Our first application is a location-based file sharing system with a monitoring interface that provides an explanation of what the application is doing. Second, we describe a unified home controller interface that controls temperature and lighting. This interface demonstrates how designers can take existing applications and integrate them into novel monitoring and control interfaces. Third, we present a museum exhibit interface for museum administrators that monitors and controls visitors' context-aware tour guides. Here, we show how a designer might design a useful monitoring and control interface for a different set of users than are targeted by the actual context-aware application. Our final application is an activity monitoring system with a privacy control added, demonstrates how enactors enable the enhancement of existing context-aware applications.

4.1 File Sharing Control

Fig. 4. File Sharing interface.

Description The Wi-File Sharing system allows users to explicitly share files with others based on physical proximity. The interface (Fig. 4) provides a radar-like view of people that are using the application, with the user in the center. The further people are from the center, the longer the distance to the user. The application only allows file transfers when both parties are in the innermost circle, where the wireless signal is strong enough to support transfers. The user is notified when others move. If the user wants to transfer a file, she can hover another user's icon and is told whether she can initiate the file transfer, and if not, what she must do before she can. When others initiate a transfer with the user, the user receives the sender's identity sender, name and size of the file, and is given the option to accept the file transfer.

Implementation This application uses a single enactor with a single reference that monitors the location of each person in the interface. The enactor exposes no parameters but simply allows the Flash interface to view the location of others. The CTK application contains about 160 lines of code for the enactor. The Flash interface con-

tains 115 lines of ActionScript code; about 15 lines are dedicated to CTK enactor communication and the rest are primarily display logic.

Discussion This application demonstrates the implementation of a useful context-aware application, where files can be transferred based on physical proximity. The interface shows how monitoring information can be added to an application to make it more usable by helping users understand what an application is doing and why.

4.2 Unified Room Control

Description A designer might want to customize an interface to present an efficient means of monitoring and controlling a set of context-aware applications in an environment. We developed a wall-mounted interface (Fig. 2) that composed two applications, temperature and lighting, into one interface for a living room. The temperature portion of the interface on the right is similar to the temperature control application described above, with color and orientation modifications. It monitors the current temperature and allows the user to control the target temperature. The lighting application turns on local light sources when certain items enter the proximity of those light sources, and is similar to those found in other research projects [4,7]. The interface indicates which lights are active, and by clicking on the light the user can see what item is responsible for the application behavior. For instance, in Fig. 2 there is a book on the sofa; the application provides increased illumination to aid in reading. The application is simplified for the purposes of demonstration, not taking into account the time of day and per-user preferences.

The interface displays only a subset of the information exposed in the enactor, assuming that the users of the interface are actually resident in the room it represents. So, it does not display the location of people, who are assumed to be easily visible to all room occupants. Also, it does not show the status of the climate control device, *i.e.* heating or cooling, instead only showing numerical information. The interface displays lighting changes in the interface, along with the responsible item, since occupants may not be able to figure out exactly why a light turned on by itself. The designer of the interface chose to display only the information an occupant of the space requires to understand the application behavior.

Implementation The context-aware application here comprises two enactors, one for the temperature application and one for the lighting application. The temperature enactor is exactly the same as utilized in the previous section, with no modifications. The lighting enactor has three references that retrieve widgets representing light source intensity and the locations of people and items. The enactor tracks regions with local light sources for the presence of people and items. If an item has an eligible type (e.g. book), the enactor activates the light source. If the person leaves the region but the item remains, the enactor leaves the light on in case the person might return. If both the person and item leave the region, the enactor shuts off the light source. The Flash monitoring and control interface connects to the two enactors and monitors context input as it changes over time. It tracks items as they move into and out of regions of the room, and caches the data for display when a user clicks on a light source. Whenever the interface receives context input that a light is activated it displays that light source on the floor plan. The lighting enactor contains about 260 lines

of code with about 10 lines required to expose the necessary parameters. The Flash interface utilizes about 220 lines of ActionScript, with 35 lines managing enactor communication and the remaining lines implementing display logic.

Discussion The room control display unifies two previously and independently developed context-aware applications into one monitoring and control interface. It demonstrates how a designer might choose relevant facets of an application exposed by multiple enactors and expose those facets to the user. Moreover, it shows how designers may use knowledge of how and where users will interact with an interface to selectively present useful information.

4.3 Museum Exhibit Control

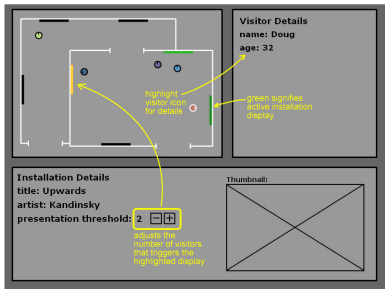

Fig. 5. Museum Exhibit Control interface.

Description Interactive tour guides are *the* canonical context-aware application. They are often considered in a museum setting, where visitors can retrieve extra information about exhibits as they roam the museum. Simple audio tour guides are commonly used today; the plaques that describe information about a particular installation display a numerical code that visitors can enter into a portable audio device and receive more information than is available on the plaque itself.

We considered a possible extension of these future tour guides and built a prototype context-aware application. In our museum guide, users carry location-sensitive PDAs that can provide audiovisual commentary about exhibits. The installation plaques are dynamic displays, enabling the presentation of more content than can fit on one static plaque or a small PDA. The context-aware application uses knowledge of users' proximity to installations to periodically initiate short presentations on plaques that entice users to explore topics in greater depth on their own displays.

This application could provide great value to the experience of museum visitors. However, the reality of any particular museum setting may be impossible for application developers to anticipate. The solution is to expose relevant controls and support museum administrators in tuning the application. To this end, we have implemented a control interface (Fig. 5), for an exhibit that utilizes the application described above. The interface displays a floor plan noting all installation locations. Visitor movements are tracked and displayed on the floor plan. Installation and visitor icons can be highlighted to provide detailed information in areas to the left of and below the floor plan, respectively. Administrators can view the status of visitor displays and installation plaque displays as either inactive or in presentation. Moreover, they can set the visitor proximity threshold of any installation plaque display to begin presentation playback.

Implementation This context-aware application utilizes two enactors; one is for the application logic that monitors the location of PDAs and delivers appropriate content to dynamic plaques when instructed by visitors. It has one reference to widgets representing the PDAs and exposes no parameters. The second enactor implements logic that invokes installation plaque displays based on visitor proximity. This enactor

has two references, one to PDA widgets and one to installation plaque display widgets. It exposes one parameter, the number of visitors should be near an inactive display before it should begin a presentation. It monitors visitor locations and initiates a presentation when the appropriate number of visitors is near a display.

The Flash interface allows the administrator to adjust the threshold parameter for an individual installation, sending a parameter change request, and waiting for a parameter change event to arrive from the enactor server before changing its display value. The interface caches the latest details about visitors and installations as it receives them so that when a user highlights an icon, the display can immediately display the requested information. The CTK application contains about 250 lines of code for the two enactors (about 20 lines for exposing parameters) and the Flash interface contains 170 lines of ActionScript code (about 30 lines for enactor communication).

Discussion The museum control interface is the kind of custom interface, implemented after a typical tour guide system is developed, that can improve the experience of end-users by allowing administrators to monitor and control that experience. Even if the general concept of administrator control is accounted for in the original application, the ability of designers to customize a monitoring and control display for a particular installation is valuable. By fitting all relevant information on a single, dynamic display, this application is suitable for the particular needs of museum employees who may be stationed right outside the exhibit. Designers can customize without needing to get their hands dirty in the internal application logic of the application.

4.4 OfficeView Activity Monitoring

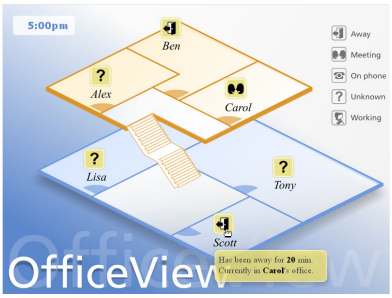

Fig. 6. Activity Monitoring interface.

Description Our activity monitoring application follows in a long line of awareness systems that have been shown to increase work productivity and efficiency [13]. It provides no privacy controls, delivering exactly what information is sensed to interested parties. The OfficeView monitoring and control interface (Fig. 6) provides a floorplan view of a workplace on which a user can view information about activities in other offices. In addition, it augments the existing application with privacy measures, allowing users to specify whether their activity information can be revealed to others or not. Similar to the lighting interface, users can also request additional information about what sensed information led to another user's activity setting, by hovering over that person's icon. If a person is not in his office, information about his location is also provided, assuming the user allowed this.

Implementation This application uses an enactor with a single reference that monitors the activity of each person in the interface. The enactor exposes a single parameter for a user's activity setting. The Flash interface provides a visualization of activity information acquired from the enactor and allows users to change their activity setting to anything, including "Unknown" (*e.g.* private). When multiple people run this interface, they all connect to the same enactor, allowing value changes made by one person

to be propagated to the other instances. The CTK application contains about 180 lines of code for the enactor (about 15 lines for parameter exposure) and the Flash interface contains 130 lines of ActionScript (about 20 lines for enactor communication).

Discussion This application and interface demonstrates the implementation of a useful context-aware application, allowing users to view each other's activities as well as specify what information is released to others. In addition, the interface shows how a monitoring and control application can be built on top of an existing application, by adding privacy control to the activity monitoring system.

5 Evaluation

We presented 4 applications and described how they can be built with enactors with little overhead to their developers. Additionally, they highlight a designer's ability to create interfaces that work and add value to a single application, combine multiple previously built and deployed applications, and support different user groups or needs after an application has been implemented and deployed. To further establish that enactors can be used effectively by interface designers to develop monitoring and control interfaces to context-aware applications, we conducted an informal evaluation of the extended CTK with Flash designers. Study participants unfamiliar with the CTK were asked to implement a home temperature monitoring and control system.

5.1 Study Design

Ten Flash interface designers participated in a 2-hour long study. They averaged 3.4 years practicing interface design and 2.0 years working with Flash. They were given a scenario that described the intended users of the temperature control application, a three person family with particular needs who moved into a new home. The home was outfitted with a CTK-enabled temperature control system in 3 rooms, and the participants were to design and implement an interface for controlling and monitoring temperature in each room. Temperature control is a common operation and is a canonical, if basic, context-aware application. While the required interface may be basic, the task required designers to perform all the necessary steps for building more complex interfaces: interacting with enactors, acquiring and displaying information about context changes and component availability changes, allowing users to change parameters and updating those in the enactor. Participants were given technical documentation for the CTK connection object described above including object member details, and basic usage examples and used this to implement their interface.

5.2 Study Results

All participants accomplished the task we set out for them, with examples shown in Fig. 7. They each answered exit questionnaires containing a number of demographic questions and Likert scale questions (1 = strongly disagree to 5 = strongly agree). All felt that they understood how to use the Flash library (avg=4.3, SD=0.48), and could

accomplish similar interface development tasks in the future using the CTK connection object (avg=4.7, SD=0.48). The interfaces contained an average of 51 lines of ActionScript code. The participants' positive task performance and impressions suggest that Flash designers and developers with similar expertise can indeed fulfill a role similar to that advocated in this paper, customizing interaction and presentation of context-aware applications to task-specific user requirements.

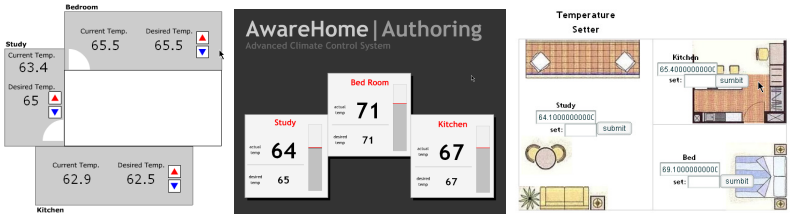

Fig. 7. Three temperature monitoring and control interfaces implemented by study participants.

To review, all participants were able to implement a monitoring and control interface for our temperature system, lending credence to the participants' own impression that they would be able to accomplish similar tasks with the CTK connection object in the future. We expect that we would see similar results with a study of VB designers. This study suggests that interface designers can use their existing skills to implement interfaces to context-aware applications as well, when offered a library like the CTK connection object and the infrastructure support of enactors.

6 Conclusions and Future Work

We have shown that monitoring and control are essential interactions in context-aware applications. To support these interactions, designers must have the ability to design interfaces that account for user needs, with the freedom to customize and compose the presentation of information. To this end we presented a system that exposes context-aware application state, and supports access from interface design environments. Enactors allow application developers to encapsulate application logic and expose relevant facets in a standard way with minimal burden to application developers. They allow designers to create interfaces without requiring them to have implementation knowledge of application state and behavior. We demonstrated this through augmentation of four context-aware applications. Our informal study indicates that Flash designers can successfully use the system to construct monitoring and control interfaces.

Our primary goal in this research is to improve the end-user experience in context-aware applications by supporting monitoring and control. Toolkit support is a necessary first step. However, this first step should be followed by in-depth exploration and evaluation of particular monitoring and control interaction techniques (including debugging) that are shown to benefit users. The work described here should assist such research by enabling the rapid prototyping of monitoring and control displays.

There are two important, interesting directions in which further research may extend enactors. First, enactor components standardize the means by which applications implement their core logic and expose relevant input, output, and parameters. This standardization improves the ability to easily make displays that show what context-

aware applications are doing. It is also critical, however, to expose some notion of what applications *will do* (*i.e.* feedforward) [3]. For instance, you might like to know what context outputs will be triggered after a particular enactor is changed to some value. We are exploring this issue through a combination of enactor extensions and simulation support. Second, although exposing application logic directly to designers, who then can expose it in turn to end users, is extremely useful and enables usable context-aware applications, designers cannot change or enhance the application logic implemented by developers. We are interested in implementing a subclass of enactors that support declarative, rule-based definitions of application logic. Rather than just supporting the exposure of a finite number of parameters, the entire application logic can itself be represented as a modifiable construct.

References

1. J. Bardram. The Java Context-Awareness Framework (JCAF) – A service infrastructure and programming framework for context-aware applications. *Pervasive 2004*, 98–115.
2. L. Barkhuus and A.K. Dey. Is context-aware computing taking control away from the user? Three levels of interactivity examined. *Ubicomp 2003*, 149–156.
3. V. Bellotti and K. Edwards. Intelligibiliety and accountability: Human considerations in context-aware systems. *Human-Computer Interaction*, 16(2–4):193–212, 2001.
4. B. Brumitt *et al*. EasyLiving: Technologies for intelligent environments. *HUC '00*, 12-29.
5. D. Caswell and P. Debaty. Creating Web representations for places. *HUC '00*, 114–126.
6. G. Chen and D. Kotz. Context aggregation and dissemination in ubiquitous computing Systems. *WMCSA '02*, 105–114.
7. M. Coen. Design principles for intelligent environments. *AAAI Spring Symposium'98*, 547–554.
8. N.H. Cohen *et al*. Composing pervasive data Using iQL. *WMCSA '02*, 94–104.
9. A.K. Dey *et al*. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2–4):97–166, 2001.
10. P. Dourish. Developing a reflective model of collaborative systems. *ACM Trans. CHI*, 2 (1):40–63, March 1995.
11. R. Hull *et al*. Rapid authoring of mediascapes. *Ubicomp 2004*, 125-142.
12. J. Humble *et al*. Playing with the bits – User-configuration of ubiquitous domestic Environments. *Ubicomp 2003*, 256–263.
13. E. Isaacs *et al*. Information communication reexamined: New functions for video in supporting opportunistic encounters. *Video-Mediated Communication*, 459–485. Lawrence-Erlbaum, 1994.
14. Y. Li *et al*. Topiary: A tool for prototyping location-enhanced applications. *UIST 2004*, 217-226.
15. A. MacLean *et al*. User-tailorable systems: Pressing the issues with Buttons. *CHI '90*, 175–182.
16. D. Martin *et al*. The Open Agent Architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1–2):91–128, 1999.
17. M.C. Mozer. The Neural Network House: An environment that adapts to its inhabitants. *Intelligent Environments '98*, 110–114.
18. M. W. Newman *et al*. User interfaces when and where they are needed: An infrastructure for recombinant computing. *UIST '02*, 171–180.
19. D.R. Olsen et al. Cross-modal interaction using XWeb. *UIST '00*, 191–200.
20. S.R. Ponnenkanti *et al*. User interfaces for network services: What, from where, and how. *WMCSA '02*, 138–147.
21. A. Schmidt. Implicit human computer interaction through context. *Personal Technologies*, 4(2&3):191–199, 2000.
22. T. Sohn and A.K. Dey. iCAP: An informal tool for interactive prototyping of context-aware applications. *CHI'03 Extended Abstracts*, 974–975.
23. M. Youngblood *et al*. A learning architecture for automating the intelligent environment. *Innovative Applications of Artificial Intelligence 2005*, 1576–1583.