

# Designer Support for Context Monitoring and Control

Alan Newberger, and Anind Dey

IRB-TR-03-017

June, 2003

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

# Designer Support for Context Monitoring and Control

Alan Newberger<sup>1</sup>  
alann@cs.berkeley.edu  
<sup>1</sup>EECS Department  
UC Berkeley  
Berkeley, CA USA

Anind Dey<sup>1,2</sup>  
anind@intel-research.net  
<sup>2</sup>Intel Research, Berkeley  
Intel Corporation  
Berkeley, CA USA

## ABSTRACT

The distributed and often implicit nature of context-aware applications can make it difficult for users to interact with them effectively. Users should be able to monitor and control the state and behavior of such applications, but current context-aware infrastructures do not effectively address such interactions. Our contribution is to more fully expose application state in an accessible way, enabling designers to produce interfaces that allow users to monitor and control context-aware applications. We accomplished this by extending a context-aware infrastructure, the Context Toolkit, and an interface builder, Macromedia Flash. These two enhancements place minimal burden on an application developers while facilitating designer access to application state and behavior. We demonstrate these through the augmentation of three common context-aware applications: a tour guide and two home environment control applications.

**KEYWORDS:** Context-aware computing, ubiquitous computing, context, design, toolkits, monitor, control

## INTRODUCTION

Context-aware applications utilize context – information regarding the state of entities that is relevant to interaction with users [11]. The awareness of context is critical as applications leave the desktop and involve mobile devices and smart environments. Such applications need to handle input from a variety of data sources and change behavior dynamically as features of that data change over time. Also, they may have multiple outputs that support a rich interaction with users above and beyond traditional desktop displays. Context-aware infrastructures provide value by supporting the acquisition of input and the generation of output in a standard fashion. However, they typically provide no reusable for constructing application logic that ties together context input and output; such logic is usually implemented for each application in an *ad hoc* fashion. The lack of systematic access to the internal state of a context-aware application makes it challenging to provide any of that state to users.

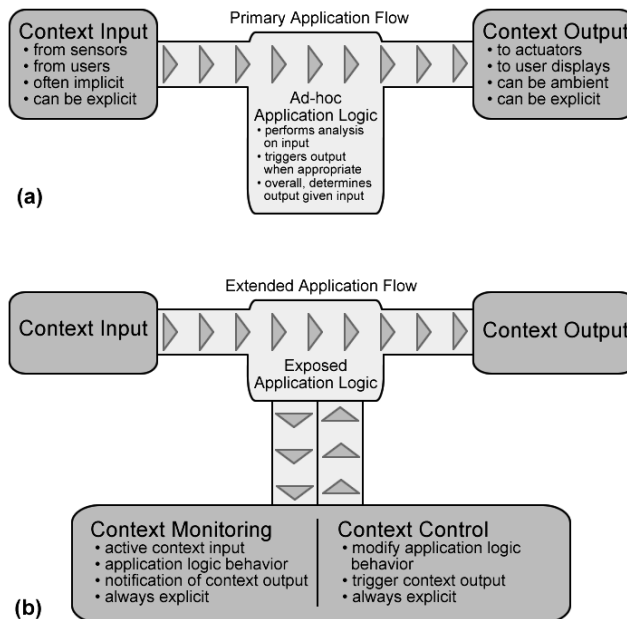


Figure 1: Context-aware application structure (a) without and (b) with monitoring and control.

Consider the “not-so-smart” home of the future, where your entrance into a room causes the lights to turn off and you to stub your toe on the bedpost. There should be some way to inspect why the home lighting application is behaving contrary to expectations, and hopefully some way to modify the behavior to align with users’ desires and goals. If context-aware applications are to be usable, they must deliver this sort of information to users and allow them to act upon it.

That is, context-aware applications need to offer users the ability to *monitor* the state of a context-aware application, and *control* that state. Context-aware applications perform actions on behalf of users, often without explicit user interaction, making their behavior hard to understand and predict and this significantly impacts usability. Monitoring and control is not of primary concern to the system: the main interaction of the context-aware lighting application is the implicit adjustment of lighting levels in response to changes in room occupancy. When this primary interaction behaves in a way unexpected to the user, however, monitoring and control in-

teractions become a critical part of the application.

### **Example Application: LITE**

It is useful to describe a hypothetical home lighting application, called LITE for brevity, as we will refer to it as an example throughout this paper. The overall purpose of LITE is to provide lighting to occupants of a home, maximizing both occupant satisfaction and energy efficiency (see [16] for an actual implementation of a system with similar goals). The general structure of a context-aware application such as LITE is illustrated in Figure . An application contains application logic that processes context input and performs analysis on that input. Application logic triggers context output when current context input satisfies some criteria. LITE receives context input from two types of sources: light sensors that provide data regarding the current light intensity in a room, and presence sensors that provide data regarding the identity and location of occupants and objects in rooms. It has one kind of context output: actuation of light levels in a room to a certain intensity. That is, the lights themselves are the context output. LITE performs analyses that may cause it to adjust light levels. One example is when a person enters a dark room, its light is initially raised to 75% intensity, and dark adjoining rooms are raised to 10% intensity; if the person remains in the room for a certain time period, lights in the adjoining rooms fade slowly back to darkness. In another example, LITE turns on a local light source if an occupant brings certain objects to designated spaces and remains there for a certain time threshold. For example, it turns on the lamp when someone sits in an easy chair with a book or magazine, or turns on the light over the kitchen table when someone brings a plate to it.

LITE also offers a separate monitoring and control interface to its users. As shown in Figure 1, context monitoring and control are classes of operations that allow user inspection and manipulation of application logic, respectively . Context monitoring includes interactions that detail the specific input processed by application logic, including any parameters that logic may be using in its analysis, and notification of any output that occurs. LITE provides context monitoring through a per-room control interface that indicates the intensity level of each light source in that room, as well as a listing of the context inputs responsible for each active intensity. For instance, when someone eats at the kitchen table the overhead table light would display as being active, with both the person eating and his plate listed as the cause.

Context control entails the modification of application logic behavior, often through specific parameters, and the ability to trigger context output. LITE provides basic context control in the form of traditional light switches throughout the house that allow users to manually control light levels. It also provides various controls through the aforementioned per-room control interface, allowing users to set all light intensity values LITE uses, time thresholds for action, *etc.* If the intensity of the kitchen table light needs to be reduced for comfortable eating, or a new set of plate IDs needs to be added so as to be

recognized appropriately, users can use the control interface and accomplish these tasks.

### **Motivation for Infrastructure-Level Support**

Context monitoring and control benefits users, but context-aware applications can be difficult to implement and supporting these classes of interactions certainly doesn't make implementation any easier. Several toolkits and infrastructures have been developed to support the general task of constructing and executing context-aware applications [4, 6, 11]. They assist the "primary application flow" as illustrated in Figure by providing frameworks of components that make it easier to build context data providers and consumers and reuse them across applications. This type of toolkit certainly resolves some major issues facing context-aware application development. For example, without some kind of infrastructural support every application would have to undertake the expensive task of instrumenting environments with sensors and actuators from scratch and connecting them. However, by enabling the reuse of context input and output, context-awareness toolkits may also raise certain challenges with respect to monitoring and control. In their paper on human considerations in context-aware applications, Bellotti and Edwards argue that

[s]eparating applications from the basic sensing, inferring, and service components ... risks making the human task of controlling a system's action or responding to a suggestion for action much more difficult. Designers must not be lulled into a false sense of security that these aspects of their design are "already taken care of" and do not require scrutiny and possible refinement. [2]

They make an excellent point. Because context-aware infrastructures afford reuse, significant portions of a context-aware application that do not face end users may be developed before needs of such end users are accounted for. It is plausible, for instance, that the LITE control interface was designed separately from the construction of the rest of the application. It could be designed to accommodate special needs such as motor impairments that were not known at the time of application development.

Although the nature of reuse in context-awareness toolkits accentuates the challenges facing monitoring and control interfaces, we believe that the right kind of toolkit support can also mitigate them. Effective monitoring and control relies upon access and manipulation of application logic. Our solution is to provide external access to application logic at the toolkit level through a standard API, and to facilitate such access through interface design tools. We have extended a particular context-aware infrastructure, the Context Toolkit (CTK) [11], to include a new component, the *enactor*. Enactors allow context-aware application developers to implement application specific logic (as they did before) and expose the design-time and run-time characteristics of that logic. Enactors tie together context input and application

output in a way that inherently supports external access to internal application logic.

Interface designers may wish to integrate monitoring or control of application logic in particular ways suited to the needs of their particular users. To this end we have implemented support in a popular interface authoring tool, Macromedia Flash, to communicate with CTK applications through enactor components over an XML protocol. Flash is an appealing choice as an interface builder and design tool to integrate with CTK applications because it is in use by over one million designers [14], the corresponding player is widely available, and it is implemented fairly consistently over a number of platforms and devices. Moreover, Flash interfaces are inherently stateful with bidirectional communication support (unlike HTML).

One of the contributions of context-aware infrastructures is the separation of concerns between context acquisition and context use. Environments can be instrumented with sensors and actuators up front, and developers can access them at a later date for use in a variety of context-aware applications. Our contribution is a further separation of concerns between context-aware application logic and user-facing context displays. With little added burden, *developers* skilled in a particular domain may articulate detailed enactors governing the behavior of a context-aware application and expose particular facets controlling that logic. *Designers* may wish to access those facets after the application has already been implemented and deployed and present them to users in novel ways, perhaps composing multiple applications into one display or implementing displays for users with special needs. Rather than attempt to eliminate the need for such “scrutiny and possible refinement” over user monitoring and control of context-aware applications, we intend to use the advantages of a common context-aware framework to support that very process [2].

### Paper Overview

In the remainder of this paper we will show that structuring context-aware application logic through enactors provides designers with support to help users to monitor and control context-aware applications. After reviewing related research, we review the CTK architecture and describe in detail the enactor and Flash extensions we have implemented. Then, we demonstrate the usefulness of these extensions through a number of applications.

### RELATED WORK

Context-aware applications utilize context in their environments, and often will take action on that context without explicit input from users. This is to some degree the whole point of context-aware computing, and context-awareness researchers are trying to enable these kinds of applications. Schmidt, for instance, considers this phenomenon “implicit human-computer interaction” [21]. Bellotti and Edwards argue that for systems that support implicit interaction to be usable they must make provisions for explicit interaction un-

der certain design principles [2]. 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, extending a toolkit that already attempts to support the sort of implicit interaction Schmidt describes. In this section, we will highlight existing context-aware architectures and the lack of support for context monitoring and control. We will also discuss component frameworks that offer inspiration for our enactor model, and existing support for interface designers, as they both address important problems relevant to monitoring and control.

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 EasyLiving [4], Cooltown [6, 8], and the CTK [11]. All of these implement mechanisms to access context data and 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.

Cooltown [6, 8] in particular is relevant as it explicitly addresses user interfaces for context-aware applications by allowing the development of “web presence” interfaces to context entities (people, places and things). Interface designers can create custom HTML views of context entity information by providing dynamic templates that specify where context data should be inserted into arbitrary HTML. This work is similar to our own in that the infrastructure gives designers a high degree of control over the look of particular interfaces to context data requiring only knowledge of existing interface tools. However, HTML as a medium does not intrinsically support stateful user interfaces, and the mechanism for updating information on a page is quite limited (*e.g.*, a polling approach that forces an update of an entire HTML page or frame). Moreover, Cooltown appears to support HTML interfaces at the granularity of context entities. This could be appropriate for context monitoring, although it imposes a web-page-per-context-entity limitation upon any interface design. This kind of granularity may pose significant challenges to application control, for control parameters or functions of an application will not always apply to individual context entities or their relationships.

The enactor extensions to the CTK that we propose are a componentization of application logic to support inspection and manipulation via an established API. Just as CTK widget components are analogous in function to GUI toolkit widgets, enactors organize applications in a similar fashion to GUI application component architectures such as JavaBeans [22]. Such component architectures were first implemented in the Andrew system [17]. Some recent architectures such as the Open Agent Architecture [15] and XWeb [20] also support inspection and manipulation of application compo-

nents in a distributed environment, and address issues like the execution of remote services and interface presentation of remote application logic. However, their relative generality makes them less suited to context-aware application development than context-aware infrastructures, particularly when considering the acquisition of context input.

Our research seeks to empower designers and developers to build context-aware monitoring and control interfaces, by providing access to CTK applications from within the Macromedia Flash authoring environment, a popular interface builder. In doing so we benefit from previous work in interface builders and end-user programming that shows how individuals without expert programming experience can build functional applications. In general, interface builders, including Flash, allow the visual arrangement and modification of interface components through direct manipulation, and allow properties of these components to be set in on-screen editors. Early examples of such tools include Trilium [7] and MenuLay[5]. Nardi argues that hybrid visual and programming systems can empower end users to program effectively, where the programming in particular is conducted through task-specific textual languages, and utilizes case studies of spreadsheet and CAD system users [19]. Flash is arguably an example of such a system.

Recent work on SLICE shares our theme of extending the capabilities of popular designer tools to enable the creation of applications with enhanced functionality [13]. SLICE allows the specification of behavior and meaning to interfaces in Adobe Photoshop, and asserts a strong preference toward keeping the system as visual as possible. Our work differs in that we are enhancing a designer tool that already includes a textual programming component, and we can benefit from the expressiveness that language provides.

As we have shown, current context-aware infrastructures greatly assist the creation of context-aware applications but do not provide support for context monitoring and control. Tools such as component architectures and interface builders address important problems that are relevant to context-aware systems with respect to monitoring and control. In the next section we will apply some of these ideas to context-aware infrastructures to enable monitoring and control on context-aware applications, and support the design task of creating context monitoring and control interfaces for end users.

## ARCHITECTURE

We will first provide a brief overview of the CTK framework. Then, we will detail CTK extensions that support context monitoring and control. Finally, we will describe the extensions to Macromedia Flash that allow Flash interfaces to communicate with the CTK.

### CTK Overview

The Context Toolkit [11] is a Java-based component framework and distributed infrastructure that supports the creation and execution of context-aware applications. Relevant components in the CTK include *widgets*, *services*, and *discoverers*.

All share a common subscription mechanism that allows applications and other components to receive updates through callbacks. Widgets serve as the basic abstraction for context input. They encapsulate context information in such a way as to hide the complexity of the actual input. For instance, a location widget may provide location information that comes from varying mechanisms such as GPS, an internal RF location system, *etc.* Applications need not be concerned with the details of accessing these different sorts of systems. Widgets are context input building blocks that remove the need to interface with particular sensor infrastructures during context-aware application construction. Services abstract actions that may be taken on behalf of the application. Often services correspond to actuators in the environment, *e.g.* a light switch. They perform an analogous function to widgets, providing access to actions without necessarily exposing the details of implementation. Finally, discoverers maintain registries of available CTK components. Applications subscribe to discoverers to dynamically locate CTK components as they enter and exit an environment.

The CTK supports the creation of context-aware applications as follows. In a CTK-enabled environment, instantiations of the above components will exist independently on the network, generating data and offering capabilities that can be used in applications. A particular application locates and subscribes to a discoverer in order to receive notifications of components in the environment that it is looking for. It can then subscribe to widgets as necessary. For instance, LITE as a traditional CTK application would subscribe to lighting widgets that generated status data regarding light source intensity, and presence widgets that would generate location and identity information for people and objects. Even if people were tracked using vision techniques and objects using proximity RF, widgets would abstract those details from the LITE application. The application also can execute services at its discretion. LITE would adjust light intensities using services when it deemed such action necessary using strategies as discussed above. This general application model has been validated through the use of several context-aware applications [3, 9, 10, 18]. There is a significant drawback to this model, however, when considering it with respect to context monitoring and control. Although the code that performs the integration of CTK components, the application logic, benefits from the regularity and reusability of those components, it is itself completely *ad hoc* and not inherently exposed at all. In order to facilitate external inspection, crucial to monitoring and control, application logic in CTK applications must itself be componentized.

### CTK Extensions

If application logic is developed in an *ad hoc* manner, 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 some later time. Both of these strategies are unrealistic in general. We solve this problem by exposing application

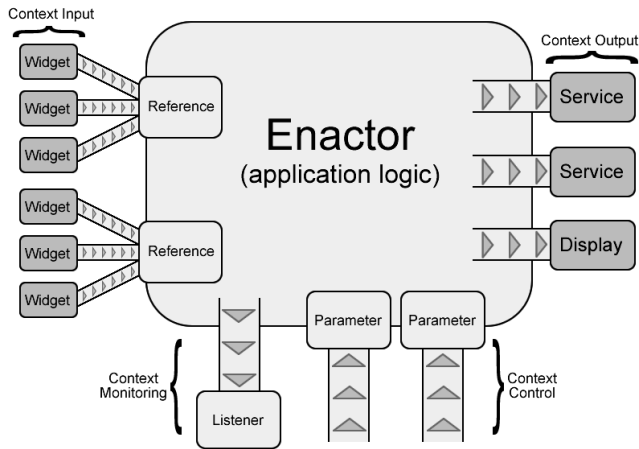


Figure 2: An enactor has *references* that acquire context data from widgets, *listeners* that monitor all changes, and *parameters* that allow control. Enactors can utilize services or other targets, such as displays, as output.

logic already present in CTK applications via a standard API. To accomplish this, we added another class of primary component, *enactors* (see Figure 2). Enactors are designed to allow developers to easily encapsulate application logic in a component. They have three subcomponents: *references*, *parameters*, and *listeners*. An enactor acquires context input through sets of references. It processes information internally, exposing any relevant properties as parameters. Listeners are notified of occurrences within the enactor, such as any actions that are invoked and context data received.

**References** Enactors may require data from a variety of CTK components, representing data from a variety of entities. A traditional CTK application would query a discoverer, and then manually subscribe to all matching widgets. References support a declarative specification of interest in a set of CTK components through a general query package. Queries may be arbitrary boolean queries that can condition upon all facets of a component. No explicit subscription is required; references automatically process queries with discoverers and subscribe to any components that match. References notify enactors whenever components newly satisfy or fail to satisfy a match (a *reference match*), and whenever a matched component provides the reference with new context data (a *reference evaluation*). Enactors notify listeners whenever reference matches or evaluations occur. These events signify that a context input change of interest has occurred.

**Parameters** A set of parameters are exposed by enactors that comprise the public view of the component in the distributed infrastructure. Parameters are analogous to JavaBean properties and parameters in other component frameworks [22]. Parameters can be read-only or read/write, and have a description advertising their type and what they do. Enactors inform listeners when a parameter value changes. Such an event signifies that the behavior of the enactor itself

has changed.

**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 clients). It also listens for any XML sent to it, and applies modifications to enactor parameters in response.

**Enactor application design** 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 internal implementation of enactor references efficiently multiplexes widget subscriptions between enactors, so that 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. Execution of services and other context output occurs much like before. Enactors provide notification of events like context acquisition and execution through listeners with little additional effort for the developer.

The facet of enactors that has no ready analogue with traditional development is the parameter. Developers must determine which values of their application logic to expose, and which to permit manipulation of. Indeed, this is the task we want to strongly encourage context-aware 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 the designer and end user as we will describe in the next section.

### Flash Extensions

Macromedia Flash is an interface development tool that deploys programs that execute within the Macromedia Flash Player. The Flash interface is for the most part visually-based using a timeline metaphor. It also allows scripting through Actionscript, a close variant of ECMAScript [12]. Actionscript supports the use of objects, and provides object representations of most visual Flash elements. Recent versions

of Flash also offer XML support in the form of a parser, a DOM, and an XMLSocket object. XMLSockets offer bidirectional communication to a server, and events that trigger whenever XML is transmitted to the program.

We implemented a CTK connection object that fully wraps an XMLSocket and provides a set of custom high level events to Flash designers. This object essentially extends the CTK enactor listener interface into Flash, and the high level events it provides map directly onto listener methods. These events can be attached with custom handling code of a designer's choosing 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 to sets of associative arrays, a native Actionscript data type. Flash interfaces can also set enactor parameters via the CTK connection object; the object sends parameter arguments up to the enactor server which then sets the parameter on the enactor.

If a designer knows the location of a CTK enactor server, they can design an application to connect to it. 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 data dictionary, describing the enactor, the names and types of its exposed parameters, any reference queries, and descriptions of any currently matched components. Using this information the designer can decide what information to extract and utilize in a Flash interface.

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 Flash extensions in turn allow arbitrary Flash applications to monitor and control context-aware applications through enactor servers that function as enactor listeners. The LITE control interface we described earlier can be designed and implemented in Flash, completely independent from the main LITE context-aware application.

#### **Building an application and interface with enactors**

Our hypothetical LITE application can be implemented straightforwardly so as to expose its application logic via enactors. Two enactors are required for the functionality we have discussed; one handles the activation of room lighting upon room entry by an occupant, the other handles the activation of local lighting as in our kitchen table example. The first enactor declares two references, one describing lighting widgets and one describing presence widgets. The enactor receives reference evaluation updates whenever a light intensity changed or a person changed location, and runs application-specific code that executes lighting services as necessary. It also exposes a number of parameters, including an integer specifying the light intensity of a light in a newly entered room, a boolean specifying whether or not to turn on the light in adjoining rooms upon entry, the time after which

an exited, unoccupied room should go dark, *etc.* The second enactor subscribes to local lighting widgets by declaring a query that matches lighting widgets of a particular type, and all presence widgets, for people and things. It executes local lighting services as necessary. Exposed parameters include an integer specifying the light intensity of an activated light source, a list specifying the types of presence widgets to regard as things needing extra light, *etc.*

The LITE Flash control interface invokes a CTK connection object in Java and connects to the enactors described above. It then receive events for all context input received by enactor references, all values of parameters, and all context output generated. The interface turns these into visual elements signifying light intensity and location. All objects that influence the activation of light sources, such as people or things, are context input for the enactors, so the interface would naturally receive information about them that it could then display to users. The modification of enactor parameters, such as thresholds for light source activation, sets the parameters through the CTK connection object and propagates them back to the enactors.

#### **DEMONSTRATION APPLICATIONS**

For users to interact effectively with context-aware applications, they should be able to monitor and control them. Our implementation of enactors and the Flash enactor communication library address this need by enabling designers to produce monitoring and control interfaces without being concerned with the implementation of entire context-aware applications. Designers can produce specific interfaces targeted to the needs of specific users, independent of the original context-aware application development process.

Throughout this paper we referred to a hypothetical application, LITE. In this section we describe three implemented applications 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 implemented with 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 input 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. First, we describe a simple temperature controller, the goal of which is to demonstrate end-to-end operation of all the architectural components described above. It is an application that establishes baseline capabilities. Next, we present a unified home controller interface that controls temperature and lighting. This Flash interface monitors the same context-aware temperature application from our first example, and it demonstrates how designers can take existing context-aware applications and

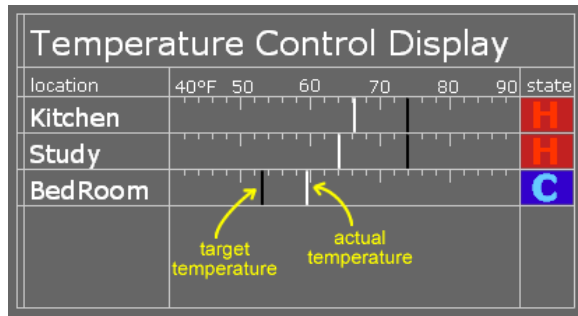


Figure 3: Temperature Display Interface

integrate them into novel monitoring and control interfaces. Finally, 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. This last application is a museum exhibit interface for museum administrators that monitors and controls a prototypical context-aware tour guide application for museum visitors.

### Temperature Control

*Description* Temperature control is a common operation supported in residences and is a canonical, if basic, context-aware application. The goal in modern temperature control is to regulate temperature based on several context inputs including the season, the time of day, ambient external temperature, and the presence of inhabitants. The simplest mechanism to regulate temperature in a space is manual adjustment of a heating and/or cooling source, turning it on and off when necessary to achieve the desired room temperature. The thermostat makes this task much easier: users simply set a target temperature and it takes care of adjusting the state of heating and/or cooling to bring room temperature within some threshold. A disadvantage of current thermostats is that it can be difficult to actually achieve desired temperatures across a large space with different temperature profiles and only one point of control. Modern temperature systems will have increasing number of thermostats to monitor and control.

If the number increases to essentially one per room, adjusting the temperature by using a fixed control device in each room can also be a laborious task. To address this concern, we designed a basic temperature monitoring and control display that unifies all of the thermostats in a house. This interface, shown in Figure 3, displays in a tabular format the temperature of each location available for temperature control. Current temperature per location is shown by a white bar that moves over time, and target temperature by a black bar that can be changed by the user with a drag. The activity of the temperature system in that location is shown as either heating, cooling, or inactive.

*Implementation* The actual context-aware application can operate independently of any interface, and often regulates temperature autonomously. It utilizes one enactor that monitors room temperatures and adjusts the activity of the climate

control system to achieve target temperatures for each room. It exposes one parameter that sets the target temperature for each room, and one reference that declares interest in temperature widgets. Whenever the enactor receives new data from a temperature widget, it decides whether or not to change the climate control system status. If the status is either heating or cooling and the temperature is close to the target, the enactor executes a service changing the system to inactive. If the temperature is above or below the target, the enactor changes the system status to cooling or heating, respectively, if that is not already the system status.

The Flash monitoring and control interface connects to this enactor via an enactor server when it initializes. It receives all new temperature data that the enactor itself receives as reference evaluations, including temperature changes and status changes. It also receives notification whenever a target parameter for a particular room changes. This is all the information needed to generate the display described above. The interface also instructs the enactor to change the target temperature for a particular room whenever a user drags a target bar to a new value.

The context-aware application was written using the CTK library, in Java. The enactor contains about 160 lines of Java code, where only 10 lines were needed to actually expose the temperature parameters. The remaining lines of code are roughly what would be needed to implement the application logic without the enactor object, illustrating the low added burden of exposing parameters. The Flash interface contained about 140 lines of Actionscript code. Only 20 lines are dedicated to CTK enactor communication and handling. The remaining lines manipulate display logic, *e.g.* controlling the position and visibility of on-screen items. This illustrates the ease with which designers can access application logic and build monitoring and control interfaces for users.

*Discussion* This temperature control display demonstrates a very basic system using one widget type, one service, one enactor, and one display interface. The simplicity of implementation of the display interface is the primary accomplishment. Presumably, a real-world context-aware temperature control application would encapsulate more complex logic in its enactor. For example, it could reference location and occupancy context information, and expose multiple parameters detailing temperature targets and occupancy thresholds. Then, the application could dynamically maintain different targets depending upon whether rooms were occupied or not, improving efficiency. This additional complexity would be entirely encapsulated in the enactor; the use of parameters in the monitoring and control interface would be very similar to this example application. As the complexity of the context-aware application increases, the means of accessing enactor information stays the same.

### Unified Room Control

*Description* A designer might want to customize an interface to present an efficient means of monitoring and control-

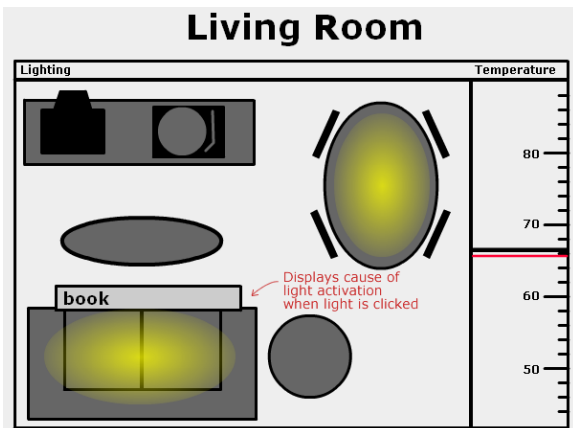


Figure 4: Room Display Interface. On the left of the interface is the floor plan of a living room. On the right is a temperature readout for this room.

ling a set of context-aware applications in an environment. We developed an interface, shown in Figure 4, that composed two applications, temperature and lighting, into one interface for a particular living room. The temperature portion of the interface on the right is similar to an individual element of 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 is essentially a subset of LITE, and turns on local light sources when certain items enter the proximity of those light sources. 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 Figure 4 there is a book on the sofa; the application provides increased illumination to aid in reading.

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 presumed easily visible to all occupants of the room. 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 themselves. The designer of the interface displayed 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. The first retrieves widgets representing light source intensity. The second and third retrieve widgets representing the identity and location of people and items, respectively. 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 stores the data locally 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 interface receives temperature changes and modifies the location of the meter line. It sends notification to the enactor to change the target temperature when a user adjusts the visual target on the display.

The lighting enactor was written using the CTK library, in Java, and 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 code. About 35 of those lines manage enactor communication and handling, and the remaining lines implement display logic.

**Discussion** The room control display unifies two previously and independently developed context-aware application into one monitoring and control interface. It demonstrates how a designer might choose relevant facets of an application exposed by enactors and expose those facets to the user. Moreover, although the context-aware applications are built in a general way to be used in any properly instrumented environment, this interface is designed for a specific room, with a sofa, a table with four chairs, a television, *etc.* This division of general development and specific, situated design is beneficial only if the various costs of context-aware application development are significantly higher than interface design for context monitoring and control. We believe enabling such design with a tool like Macromedia Flash is the right direction for lowering those costs, as evidenced by the large community of Flash designers [14].

### Museum Exhibit Control

**Description** Interactive tour guides are *the* canonical context-aware application [1]. Such tour guides 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 (including artist name and piece material, creation date, and author) also 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 that represented our conception. In our museum tour guide, users carry portable displays that are location sensitive and can provide visual and auditory commentary about installations. These displays are location-aware, and no longer require

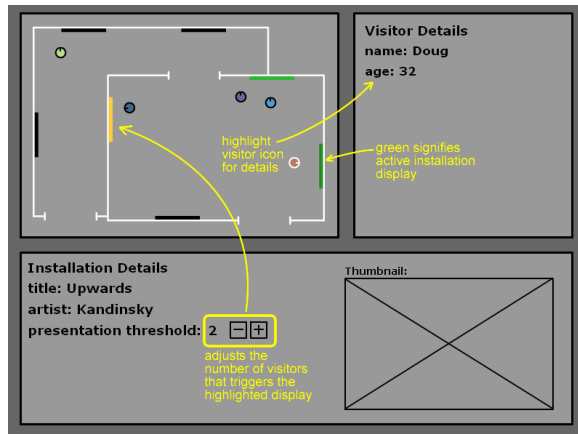


Figure 5: Museum Exhibit Display Interface

numbers from installation plaques to determine what content to present. The installation plaques are themselves dynamic displays, enabling the presentation of more content than can fit on one static plaque or a small PDA. The context-aware application utilizes knowledge of people's proximity to installations to periodically initiate short presentations on installation plaques that entice users to explore topics in greater depth on their own displays.

This application could conceivably provide great value to the experience of museum visitors. As mentioned, portable tour guide displays for visitors is a common context-aware application. However, without significant monitoring and control capabilities this application will quickly become less and less useful. The criteria that the application might use to trigger plaque presentations may vary widely over time; when the exhibit is very crowded the presentations may need to run continuously or be deactivated depending on their content, whereas when the exhibit is nearly empty, proximity by one visitor should be enough to initiate the presentation. 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 function appropriately.

To this end, we have implemented a control interface, illustrated in Figure 5, for a particular exhibit that utilizes the context-aware application described above. The interface is designed especially for this exhibit, and displays a floor plan noting the location of all installations. Visitors are displayed on the floor plan as icons that track their actual movements. Installation and visitor icons can be highlighted to provide detailed information in areas to the left of and below 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. The first implements the application logic that monitors the location of visitor displays and delivers appropriate content to the dynamic plaques when instructed by visitors. It has one reference to widgets representing the visitor displays. It exposes no parameters; in other words, this enactor enables monitoring but not control. When users invoke an interactive presentation on their display, the enactor is notified of this through a change in widget status, and executes a service that delivers content to the display. The second enactor implements the logic that invokes installation plaque displays based on visitor proximity. This enactor has two references, one to visitor display widgets and one to installation plaque display widgets. It exposes one parameter, the per-installation proximity threshold that determines how many visitors should be near an inactive display before it should begin a presentation. The enactor monitors the location of visitors and initiates presentation playback when the appropriate number of visitors are near a display.

The Flash monitoring and control interface communicates continuously with the two enactors via enactor servers. It receives reference match events upon initialization, and then receives a stream of reference evaluations detailing visitors' change in location, or the initiation of a dynamic presentation by an installation. When an administrator adjusts the threshold for an individual installation, the interface sends a parameter change request, and waits for a parameter change event to arrive from the enactor server before changing its display value. Although this technique provides sound confirmation to the user that the value actually did change in the application, interface responsiveness could be an issue. The designer could just as easily taken an optimistic approach and changed the value immediately, reconciling possible problems if they arose. 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 information that is sought.

The CTK application, written in Java, contains about 250 lines of code for the two enactors, with about 20 lines handling the exposure of parameters. The Flash interface contains 170 lines of Actionscript code; about 30 lines are dedicated to CTK enactor communication and the rest are primarily display logic.

*Discussion* The museum control interface presented here 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 tailor 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 at a standing desk right outside the exhibit. Designers can customize without needing to get their hands

dirty in the internal application logic of the application.

## Discussion

In this section, we have described three typical context-aware applications built with the enactor-extended CTK and their corresponding Flash interfaces for monitoring and control. The examples illustrate how common applications can be built with enactors requiring little overhead on the part of application developers. The examples also illustrate how Flash designers can easily build monitoring and control interfaces. Our examples highlight a designer's ability to create interfaces that work with a single application, combine multiple applications and support different user groups or needs after an application has been implemented and deployed. These examples demonstrate the ease of use and benefits provided by the enactor-extended CTK and Flash extensions.

## CONCLUSIONS & FUTURE WORK

We have described a system that fully exposes application state in an accessible way, enabling designers to produce interfaces that allow users to monitor and control context-aware applications. The CTK enactor component allows context-aware application developers to encapsulate application logic and expose relevant facets in a standard way while placing minimal burden on an application developer. This allows interface designers to create displays without requiring them to have implementation knowledge of the application state and behavior. We demonstrated these through the augmentation of three common context-aware applications: a tour guide and two home environment control applications.

Our primary goal in this research is to improve end user experience in context-aware applications by improving monitoring and control. Toolkit support for monitoring and control 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 that are shown to benefit users. The work described here should assist such research by enabling the rapid prototyping of monitoring and control displays for a variety of devices.

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) [2]. For instance, you might like to know what context outputs will be triggered after a particular enactor is changed to some value. We are currently 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 are still constrained by the application logic implemented by developers. That is, designers can-

not change or enhance the actual application logic. 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. G. D. Abowd et al. Cyberguide: A mobile context-aware tour guide. *ACM Wireless Networks*, 3(5):421–433, 1997.
2. V. Bellotti and K. Edwards. Intelligibility and Accountability: Human Considerations in Context-Aware Systems. *Human-Computer Interaction*, 16(2–4):193–212, 2001.
3. J. Brotherton et al. Supporting Capture and Access Interfaces for Informal and Opportunistic Meeting. GVVU Technical Report GIT-GVVU-99-06, Georgia Institute of Technology, 1999.
4. B. Brumitt et al. EasyLiving: Technologies for Intelligent Environments. In *Proc. HUC '00*, pages 12–29, 2000.
5. W. Buxton et al. Towards a Comprehensive User Interface Management System. In *Proc. SIGGRAPH '83*, pages 35–42, 1983.
6. D. Caswell and P. Debatty. Creating Web Representations for Places. In *Proc. HUC '00*, pages 114–126, 2000.
7. J. D. Austin Henderson. The Trillium User Interface Design Environment. In *Proc. CHI '86*, pages 221–227, 1986.
8. P. Debatty and D. Caswell. Uniform Web Presence Architecture for People, Places, and Things. Tech. Report HPL-2000-67, Hewlett Packard Laboratories, 2000.
9. A. K. Dey and G. D. Abowd. CybreMinder: A Context-Aware System for Supporting Reminders. In *Proc. HUC '00*, pages 25–27, 2000.
10. A. K. Dey et al. The Conference Assistant: Combining Context-Awareness with Wearable Computing. In *Proc. Wearable Computers '99*, pages 21–28, 1999.
11. 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.
12. ECMA. ECMAScript Language Specification, 1999. URL <http://www.ecma-international.org/publications/standards/ECMA-262.HTM>.
13. J. Fogarty et al. Specifying behavior and semantic meaning in an unmodified layered drawing package. In *Proc. UIST '02*, pages 61–70, 2002.
14. Macromedia, Inc. Flash MX Feature Tour, 2003. URL <http://www.macromedia.com/software/flash/productinfo/features/>.
15. D. Martin et al. The Open Agent Architecture: A Framework for Building Distributed Software Systems. *Applied Artificial Intelligence*, 13(1–2):91–128, 1999.
16. M. C. Mozer. The Neural Network House: An Environment that Adapts to its Inhabitants. In *Proc. AAAI Intelligent Environments*, pages 110–114, 1998.
17. B. Myers et al. Past, Present, and Future of User Interface Software Tools. *ACM Trans. CHI*, 7(1):3–28, 2000.
18. K. Nagel et al. The Family Intercom: Developing a Context-Aware Audio Communication System. In *Proc. of Ubicomp '01*, pages 176–183, 2001.
19. B. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, 1993.
20. D. R. Olsen et al. Cross-modal Interaction using XWeb. In *Proc. UIST '00*, pages 191–200, 2000.
21. A. Schmidt. Implicit Human Computer Interaction Through Context. *Personal Technologies*, 4(2&3):191–199, 2000.
22. Sun Microsystems, Inc. JavaBeans, 2003. URL <http://java.sun.com/products/javabeans/>.