Paul Holleis

Programming Interactive Physical Prototypes

June 19, 2007

Abstract Most application and device developers agree that prototyping makes sense to get an early idea of the usage of smart objects. Numerous advantages can result from getting feedback from users at an early stage in the development process. However, creating such prototypes is often a difficult and time consuming task. This especially applies to physical, tangible objects. Often, quick methods like paper prototyping cannot create enough information or convey the intended feeling of the finished application. This paper elaborates on an integrated approach to quickly develop applications using combinations of software and hardware components necessary for the envisioned system and shares some of the experiences gathered during this development.

Keywords Prototyping · Development · Infrastructure

1 Motivation

It is commonly agreed that prototypes can play an important role in developing applications. Early user feedback is especially important when physical, tangible devices are involved since the iteration cycle normally is quite long and costly. Unfortunately, exactly these types of prototypes are often especially hard to create. Several research projects have introduced some tools and frameworks that support the quick development of prototypes involving sets of pre-configured smart objects. Most of those target a specific group of users.

Many try to equip those who hardly have any knowledge of programming with means to build simple programs. This mainly includes some kind of visual natural language based programming and is supposed to enable high-level designers, children, or end-users to combine inputs with outputs, possibly creating simple rules to fire certain events. The main problem of such approaches is that the applications that can be developed in this fashion have a relatively low ceiling, i.e. the possible complexity of the prototypes is limited.

Other systems provide interfaces, drivers, or libraries in different languages to support specific sets of hardware components. However, these often assume deep knowledge in communication protocols, device discovery, or component internals and require developers to read large amounts of documentation or datasheets.

In this paper, we talk about our own approach to that problem and what experiences we could draw from that. One of the main points we found and upon which we built our approach is that in cases where a community of people (i.e. different kinds of developers) should be supported, it is questionable to develop completely new toolsets instead of enhancing or augmenting the ones they already use. The development of the majority of those environments in use has been started several years ago and they have been undergoing several iterations. It is very hard, if not impossible, to catch up with such established tools and the amount of time necessary to get to know new environments does often not pay off. Adding additional features and support to these tools, however, largely decreases the threshold to create physical prototypes. We present an approach which fosters the interactive development of prototypical applications using software and hardware components.

After a description of available tools to create prototypes in general and their application to physical and tangible objects, we introduce the EIToolkit. It is used as underlying framework and supports the simple integration and control of software and hardware components. The third part of the paper concentrates on the integration of the entire process into the Eclipse IDE. After an outline of the use and potential applications of the system, a summary and possible future work is given.

Paul Holleis
b-it, University of Bonn
Dahlmannstr. 2, 53113 Bonn
E-mail: paul@hcilab.org

2 A Glimpse at Existing Prototyping Tools

Prototyping has a long tradition as a means to avoid errors or obstacles in expensive systems. Since prototypes need to be created quickly and cheaply, frameworks and tools help developers build prototypical versions of their envisioned applications or devices. There are numerous such projects and we only treat a range of those that, in out opinion, are of greater influence, scope, or use in literature, research or industry. For software projects, programming languages and prototyping environments have been introduced to enable a quick assembly of non- or semi-functional applications. This includes graphical user interface builders such as Borland JBuilder as well as authoring tools like Macromedia Flash, primarily designed to support the rapid use of graphical widgets. These lessen the effort of creating graphical user interfaces and offer a WYSIWYG editor to quickly change their appearance according to results of user studies, taste, or guidelines. In addition to such specialized tools, many other technologies can be used if the goal is to create visual interfaces only. Such graphical interfaces can often be created using presentation software like Microsoft Powerpoint, image processing software like Adobe Photoshop, or simple HTML editors. Another quick way of sketching graphical user interfaces is simply drawing them. Paper prototypes can, however, not only be used to show design of a graphical application or the appearance of a web page, but also to simulate the sequence of actions for complex interactions like with mobile phones.

High-level languages and tools have been invented to enable people with few or no programming skills to create applications. The Scratch system [11] from MIT Media Lab is one of many examples. It is targeted at children who drag and drop action elements (play a sound, move a picture, etc.) on a stage. By changing the properties of these elements (e.g. set the duration of a sound) and adding simple control structures like loops, children can quickly develop an interactive story or game. Although this system is designed to allow a range of applications to be built, the design space is still narrowly limited and the controls dependent on the set of built-in functions. Hardware devices are still even harder to develop than pure software. This is mainly due to the myriad of different hardware platforms that exist, each with a different communication type and protocol. Even most simple sensors like light or acceleration sensors as well as actuators like displays use different low-level protocols. The fact that the hardware device itself has to be built (i.e. design the case) and that software must be written implementing the semantics add to that problem.

As a start, toolkits have been developed to provide basic components for hardware. Phidgets (Greenberg and Fitchett, [6]) provide physical building blocks to create devices that can be controlled from a PC through USB using a multitude of systems including C, .NET, JAVA and Flash. Stanford's iStuff toolkit (Ballagas et al. [2]) also provides some basic hardware components like buttons or card readers. In addition, they offer a software framework to dynamically map devices to applications. Similar to the EIToolkit framework described later in this paper, there exist small pieces of software for each hardware or software component supported by the framework responsible for the communication between components. Several such systems that especially focus on enabling the use of hardware devices in software applications with only little programming knowledge necessary have been developed in the last years. One example is the EQUATOR Component Toolkit (ECT, Egglestone et al. [5]) that provides an event-based mechanism between components. Developers define the way software and hardware items connected to networked computers communicate. A graph editor allows connections between components to be drawn such that it is easy to, e.g., display the value of a (hardware) slider in a text field. Another recently introduced tool that follows a state-based paradigm is d.tools (Hartmann et al. [7]), implemented as a plug-in of Eclipse. A blueprint of the device to be prototyped can be drawn and widgets representing hardware buttons, sliders, displays, etc. can be placed on the drawing. In another editor, a state graph can be created that specifies into which state the device should be transferred on a specific action (like a button press).

3 Underlying Architecture - EIToolkit

As it is so often the case, none of these toolkits exactly met our requirements. In addtion, we wanted to get more experience in building such systems. We developed the EIToolkit framework[1]), concentrating on giving a central point of access in software to support a variety of different software and hardware applications and objects. It is a component-based architecture prescribing a small set of interfaces for supported components. Resources, internal or external, are represented by proxies, (stubs). These can pass and receive messages along defined paths. This ensures that components are separated from each other and thus easily exchangeable. Applications are independent from the implementation of message passing. Components communicate using a simple own format as well as OSC (Open Sound Control [2]) and RTP (Real-time Transport Protocol [3]). This enables the use of many hardware platforms like, e.g., Wiring and Arduino[4] microcontrollers. The Particle Computer system [4], one of the platforms used in our labs, is directly supported.

[1] EIToolkit: http://www.eitoolkit.de

[2] Open Sound Control, OSC: http://www.cnmat.berkeley.edu/OpenSoundControl/

[3] Real-time Transport Protocol, RTP: http://www.cs.columbia.edu/ hgs/rtp/

[4] Wiring, I/O Board: http://wiring.org.co, Arduino, Physical Platform: http://www.arduino.cc

Since many smart objects are connected to the serial port of a PC (either directly or indirectly, e.g., using Bluetooth), serial support easily connects, e.g., PIC microcontrollers to the toolkit. Messages can be passed over the widely used internet protocols UDP and TCP and are thus not restricted to one computer.

The stubs take on the role of drivers or proxies providing the interface of specific hardware and software components to the toolkit. This implies that the only step to integrate an additional component into the framework is to write a stub that converts messages from the EIToolkit into messages that the component understands and vice versa. As examples, stubs have been implemented for software programs like the media player Winamp , for MIDI output on computers running Microsoft Windows, and for devices based on Particle Computers (described in some detail in [8]).

The EIToolkit is versatile with respect to platform and programming language because the only requirement is the support of UDP or TCP since all information is passed this way. Most components of the EIToolkit are implemented natively in C++, C#, and Java. Since the communication can be passed over UDP or TCP, however, following one of the simple protocols supported by the toolkit, the system can be easily used with other programming languages or in conjunction with other toolkits. Some of the features we concentrated are:

Interface Description Using a set of especially tagged control messages, each component can voluntarily or by request provide its interface to the framework.

Device Exchange For each device, an interface is generated. If two or more devices have the same or a very similar interface, it is very easy to extract the common methods into a common interface (e.g. using the refactoring methods built into Eclipse).

Device Simulation Small component stubs can provide information about the interface that component offer. These either do not need to have any semantics at all, or, more importantly, can already portray the behavior of the corresponding component.

Run-time Development Support Additional important applications in such a system of heterogeneous components are logging, replay and debugging.

Documentation, Interface and Component Description To ease the development process, devices can also carry details about the documentation of their interface and mode of use.

Context Provision and Sensor Fusion As has been shown in multiple projects (e.g., [3]), information fusion methods are often employed for applications that rely on external data. A set of components can be used to aggregate context and sensor data into more abstract information. This can happen in different layers providing different abstractions. Applications can choose which abstraction layer they use.

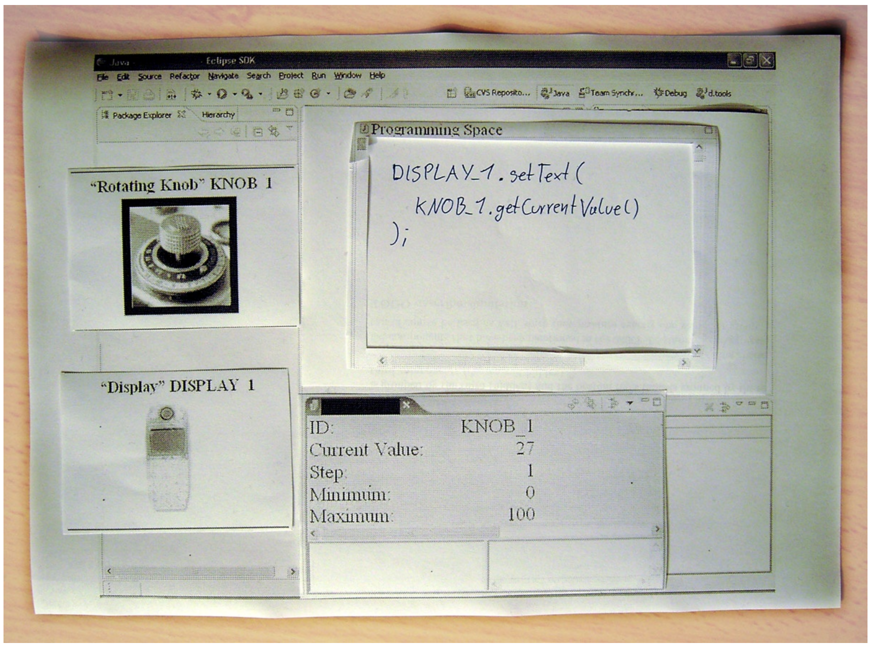
4 The Eclipse Development Environment and Proposed Extensions

One of the important lessons we learnt is that people are very much used to the tools they use. To be able to create prototypes, developers want to stick to these tools. One widely used integrated development environment is Eclipse (http://www.eclipse.org). It consists of the mainframe which provides the core technology and semantics to control a set of plug-ins. Eclipse itself is not specialized to a certain domain of use. However, a specific set of plug-ins turns it into an specific environment. The reason for which it gained reputation is mainly its powerful support for programming with the Java programming language. However, support for languages like C++ and entirely different tasks is continuously added.

Java developers especially appreciate the ability of eclipse to compile code in the background and display syntax errors while typing. Other major advantages include the auto-completion feature, i.e. the system can make well grounded guesses what the user wants to type in next (such as names of variables, methods, lists of parameters, exceptions to be caught), based on code syntax, variable scope, class interfaces, etc. Refactoring, formatting, and integrated debugging and inspection facilities during design and run-time add to its usability as a platform to develop code. The modularity of Eclipse makes it a perfect environment to add new features and functionality.

The EIToolkit, which has by now been used in several of our projects, is used to manage the communication between Eclipse plug-ins and (external) components. The integration into Eclipse is done without altering the way developers work with it. The new plug-in manages all of the work in the background and only one additional view is presented to the user. It shows the IDs and a description of all devices currently connected to the framework and shows all communication traffic routed through the EIToolkit to and from the corresponding devices. It is also possible to directly input and send messages which vastly eases logging and debugging tasks.

The next step of adaptation highlights that prototyping applications are not excempt from the need of early prototyping and user input. In the view, certain parameters can be specified. This is the only point where we deviated from the standard Eclipse philosophy. Users judged a separate listing of parameters in the preferences menu to be too complicated and liked the compact presentation of all aspects belonging to the plug-in at one dedicated space. This was also seen as we tested one of the first designs (paper prototype in Figure 1). In addition, we found that icons or images representing the found components are barely necessary unless they convey highly substantive documentation or description.


Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
CVS Reposito... Java Team Synchr... Debug d.tools
Package Explorer
Hierarchy
Programming Space
"Rotating Knob" KNOB 1
"Display" DISPLAY 1

```
DISPLAY_1.setText(
   KNOB_1.getCurrentValue()
);
```

```
ID:              KNOB_1
Current Value:       27
Step:                 1
Minimum:              0
Maximum:            100
```

Fig. 1 Initial paper prototype of a design of the Eclipse plug-in presented in this paper. Whenever a device or component comes in range, it is displayed and can be used as simple as possible. Prototyping systems also profit from prototyping.

5 Application Development

The work of a developer using Eclipse as development environment does not change in any way. The only actions necessary are to include an additional external Java library that covers methods for communication and to start the plug-in which is - after the very first time - done automatically by Eclipse. Whenever a supported device is switched on, classes are generated exporting the interface of this component. We now present the workflow and some examples of how applications can be quickly generated with a set of available components. This includes existing sample applications as well as a smart environment setting to show the breadth of the design space.

When eclipse is running and the plug-in has been started, only 3 steps are necessary to begin writing an application by combining any available components. Switch on or start any devices or software components you want to use. Classes are generated that represent their interface. Or download the interface when the component itself is not available. Make your project reference the project containing the set of available stubs. Create instances of the components you want to use. At this point, you can work with the components as if they were integrated into the Java language as is the case with any other library. All features of Eclipse can be used to explore the interface of the components including auto-completion (e.g. pressing CTRL-SPACE after the name of the instance and a '.' brings up a list of names of possible methods or fields, see Figure 4) and different class viewers and explorers.

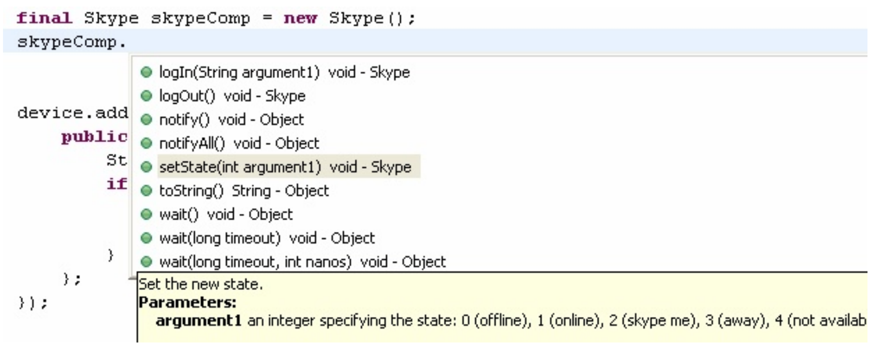
6 Sample Applications

This section presents example applications that can be quickly assembled using the approach presented in this paper.

6.1 SkypeTUI

The SkypeTUI is a tangible device with 6 distinct states. These states are mapped to possible states of a user of the phone and instant messaging application Skype (like "online", "offline", "away", etc.). The physical device used is a small wireless cube (about 6x6x6cm). Accelerometers inside the cube can sense which side of the cube is faces up. Each face of the cube thus is mapped to a state in Skype. Whenever the orientation of the cube changes, an event is generated informing all listeners about the change. The code that passes the state that the cube broadcasts on to the Skype component is simply a matter of registering a listener to the component representing the cube and calling the "setState" method with appropriate parameters. We first presented this idea in [10]. We slightly added to the functionality of the input device and used a display packaged into a cube-like housing. In this way, there is an additional output and feedback possibility. The display can be controlled, e.g., by sending a text to it ("setText" method).

The code snippet in Figure 4 shows all statements necessary to get an instance of the component representing the cube and the one responsible for controlling the Skype application. An event listener is registered that



```
final Skype skypeComp = new Skype();
skypeComp.

device.add
    public
        St
        if


        }
    };
});
```

```
logIn(String argument1)  void - Skype
logOut()  void - Skype
notify()  void - Object
notifyAll()  void - Object
setState(int argument1)  void - Skype
toString()  String - Object
wait()  void - Object
wait(long timeout)  void - Object
wait(long timeout, int nanos)  void - Object
```

Set the new state.
Parameters:
argument1 an integer specifying the state: 0 (offline), 1 (online), 2 (skype me), 3 (away), 4 (not availab

Fig. 2 The auto-completion feature of Eclipse shows possible methods that can be called on an instance of class "Skype". Beside the standard methods, the "logIn", "logOut", and "setState" methods can be seen (the last with its javadoc documentation.).

updates the state of Skype and presents some feedback to the user through the display on the cube whenever the state (i.e. orientation) of the input device changes.

It should be noted again that it is extremely easy to switch from one input device to another as long as they have similar functionality. Likewise, controlling another application than Skype just means instantiating another component and calling the appropriate method.

6.2 Multimedia Control

The sample components delivered with the EIToolkit include support for the basic controls of the Winamp5 multimedia player. The interface provides methods for the most important controls like start, stop, and pause playing a song.

Having this component, it becomes possible to control a running instance of Winamp reacting to any possible events. As in the previous application, we plugged a cube equipped with an orientation sensor into the system. Thus, a tangible object can be used to have an interface to a music player which is completely separate from the standard computer interfaces. One simple scenario in everyday work environment is that listening to music through the computer can be controlled with a simple twist of the hand without needing to switch between applications. In initial studies we saw that such a control can even be used quite unconsciously without interrupting the main task at hand [9]. A more elaborate example is the MusicCube described in [1]. It features a larger range of feedback possibilities with color LEDs possibly illuminating each side and an additional button that can be pressed and rotated. Having such a device connected to the proposed framework, it is easy to use it instead of our own display cube.

6.3 Smart Environments

Smart rooms and environments offer and use a lot of different components. These most often also support high-bandwidth communication channels and they are commonly connected to some kind of backend server or PC. This can be used to tunnel all kinds of communication and as the computer on which the toolkit and Eclipse development platform is running.

From a developer's point of view, prototyping application using complex devices like smart boards or steerable projectors is not much different than implementing some semantics against some available interfaces. Interesting applications can be built when having a bunch of different types of sensors that can be dispersed in the environment. A developer who wants to try out several variants of algorithms for context acquiring or activity recognition, need only use these sensors in the same way as they would use inputs from other resources like input streams. As stated above, sensors that are not yet available or prove unreliable in a certain context can be simulated using specific types of stubs.

7 Summary and Discussion

We presented our experiences with the building of a component-based architecture and a method to facilitate the development process of interactive physical prototypes by tools integrated into a common development environment. The focus is on the simple and quick development using a multitude of hardware and software components.

A potential drawback of the framework and development method presented here is that, for several types of applications, there is a considerable gap between prototyping and deployable application in that all communication has to be passed through a single system (currently a PC) on which the EIToolkit is running. That means some overhead is created when, e.g. two devices are used that could theoretically talk directly to each other. To solve this issue, however, it would be necessary to move the semantics to one or the entire device themselves. This requires complete support of programming of those devices. We learnt that this is has become a substantial part of the development process since the

```
final TDisplay tdisplay = new TDisplay();
final Skype skypeComp = new Skype();

tdisplay.addEventListener(new EIEventListener() {
   public void processEvent(EIEventObject event) {
      String source = event.getSource().toString();
      if (source.equals("state")) {
         skypeComp.setState(
            Integer.parseInt(event.getEventDesc().toString()));
         tdisplay.setText("Switching to state #" + newstate);
      }
   };
});
```

Fig. 3 This code snipped shows how simple an input device (TDisplay) can be connected to a software component (Skype).

community has emerged from the state where smart objects could be used by specialists only. We are currently investigating how such features could be incorporated in some generic way.

A project that we are working on in parallel looks at further possibilities that such a component based approach offers. Especially the chance to seamlessly add observing and simulating components seems to have a big potential for profiling, benchmarking, and advanced logging. Using the abilities of each component to describe its structure and setting, it also seems possible to include components incorporating usability metrics that can be plugged on top of the created applications.

References

1. Alonso, M. B. and Keyson, D. V. MusicCube: Making Digital Music Tangible. In CHI '05 Extended Abstracts. ACM Press. 1176-1179. 2005.
2. Ballagas, R., Ringel, M., Stone, M., Borchers, J. iStuff: a Physical User Interface Toolkit for Ubiquitous Computing Environments. In Proc. CHI'03. 537-544 ACM Press. 2003.
3. Chen, D., Schmidt, A., Gellesen, H.-W. An Architecture for Multi-Sensor Fusion in Mobile Environments. In Proc. Fusion'99. Volume II, 861-868. 1999.
4. Decker, C., Krohn, A., Beigl, M., Zimmer, T.: The Particle Computer System. In Proc. IPSN'05. 443-448. 2005.
5. Egglestone, R. S., Boucher, A., Greenhalgh, C., Humble, J., Law, A., Pennington, S., Rodden, T. Supporting Collaboration in the Deployment of Ubicomp Experiences. In Proc. UbiSys'06. 2006.
6. Greenberg, S. and Fitchett, C. Phidgets: Easy Development of Physical Interfaces Through Physical Widgets. In Proc. UIST'01. 209-218. ACM Press. 2001.
7. Hartmann, B., Klemmer, S.R., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher, A., Gee, J. Reflective Physical Prototyping through Integrated Design, Test, and Analysis. In Proc. UIST'06. 2006.
8. Holleis, P., Rukzio, E., Kraus, T., Schmidt, A.: Environment Based Messaging. In Advances in Pervasive Computing 2006, Pervasive'06. 2006.
9. Kranz, M., Freund, S., Holleis, P., Schmidt, A., Arndt, H. Developing Gestural Input. In Proc. IWSAWC'06. 63-68. 2006.
10. Kranz, M., Holleis, P., Schmidt, A.: Ubiquitous presence systems. In SAC '06. ACM Press. 1902-1909, 2006.
11. Peppler, K., Kafai, Y. Creative Coding: The Role of Art and Programming in the K-12 Educational Context. Unpublished Report. Source: http://llk.media.mit.edu/projects/scratch/papers/. Visited June 2007. 2005.