Software Design Issues for Experimentation in Ubiquitous Computing

Thomas Reicher
Technische Universität München
reicher@cs.tum.edu

Timo Kosch
BMW AG
Timo.Kosch@bmw.de

Abstract

Designing smart user interfaces is a key challenge in making a new appliance useful. This is especially difficult to achieve when input devices such as keyboard and mouse which allow a very flexible way of interaction with the system in a static environment are not handy in a mobile environment. Good user interface design is not the application of magic but an iterative process of engineering. One technique of interface engineers is experimentation with users. Most experimentation is done by watching the user and the system from outside and treating it as a black box. We see the need for what we call experimentation-ready software that supports researchers in data gathering and data processing. We show that software techniques to realize this are already available. They are currently used for other purposes. We show that design patterns and other techniques can be applied and that the overhead for enabling experimentation will be small. We present the experimental setup of an environment for ubiquitous computing in automobiles.

1 Introduction

Designing smart user interfaces is a key challenge in making a new appliance useful. The need for this is even greater in mobile settings where the designers of the user interface are confronted with a dilemma. Input and output devices from desktop computers such as keyboard, mouse, and high resolution display allow a very flexible way of interaction between the user and the system. They are flexible because the user input can be semantically rich as with the keyboard and a shell and the user can select the interaction context as with the mouse and a windows based user interface. However, for a mobile user these devices are usually too bulky. Every additional pound and every additional square centimeter of a device can be too much. For mobile users the equipment must be as unobtrusive as possible. Since the adaptation of humans to technical devices is limited due to the physique of the human body, designers of user interaction devices cannot simply shrink the input devices as far as technically possible. Consequently they have to find a compromise in the number and size of input components such as keys on a keyboard, the expressiveness of the output devices such as a LCD, and the functionality of the device. Moreover, broadth and depth of the menu structure are limited.

To provide short paths through a menu to the desired functionality the device should be able to find out the user's intention. It needs to find out the user's context and apply rules about what functionality is needed in a certain context. If the designer of the user interface did a good job, the system behaves intelligently from the user's point of view. But how can we find out what will be considered as intelligent or annoying? How can we see whether we gathered the right data in our context? To figure this out the interface has to be developed iteratively and in cooperation with the user.

Experimentation to gather user feedback is usually costly and takes time, especially if prototypes are developed only to gather user feedback and thrown away afterwards. However, if the development process is component based and iterative with growing functionality of the components, experimentation and user tests can be based on preliminary versions of the final system. It is possible to set up the overall architecture of the system. Missing functionality of the components can be substituted by manual input of the person who carries out the experiment. That way, user tests can start early. Software components should be built for reuse. The effort to make them experimentation-ready should be done only once while the component will be used for more than one experiment. Even if components are not reused in other systems they will be reused most probably in new generations of the software they are part of.

The remainder of this paper is organized as follows: in section 2 we describe an iterative user interface design process based on experimentation. We present the general setup of experimentation environments in section 3. In section 4 we present our model for component based user interfaces that allows such experimentation. We describe fields in software engineering that also rely on data gathered from within the software's components in section 5. We show that techniques developed for these purposes can be reused. In section 6 we give an example of an experimental setup for research in mobile systems. We close with a conclusion in section 7.

2 An Iterative User Interface Design Process

The complexity of systems is increasing while the comprehensiveness of the user is not. User interfaces have to be

somewhat intelligent, so that only the actually needed functions are visible to the user. The system has to recognize the actual context and present the functions the user needs. There are two problems: first the interface designer does not know what the user needs in every situation. He cannot model it into the system so that it seems intelligent. The second problem is that the user doesn't know it either.
This is a well-known problem in software engineering when specifying the needed functionality of a software system. An approach to this problem is to iteratively develop and evaluate prototypes that implement more and more functionality.
In this approach, the software is developed in an evolutionary way starting with requirements elicitation and analysis and the design of first nonfunctional mock-ups. These mock-ups are reviewed together with the user, then refined, and functionality is added step by step. With this iterative approach the software engineer and the user are improving their shared understanding of the system. The engineer gets to know better what the user wants, and the user can more accurately specify what he really needs. There are several software development processes that describe such an iterative approach. One well known process is the spiral model [Boehm, 1988].
The same approach can be applied to the field of interface design, as described in [Maybury, 1999] or [Arndt, 1999]. Figure 1 from [Arndt, 1999] illustrates this process.

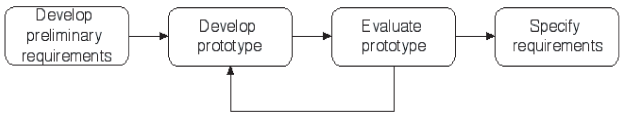

Figure 1: The Iterative Process

For the first draft the designer, often a psychologist, makes user studies about the actual situation and uses his or her knowledge to come up with a hypothesis for the desired interface. The term interface in this context does not only refer to the look and feel but also to the user interaction with the system. With this first design user studies can be made. Now user feedback is gathered iteratively and influences refinements of the interface design. At some time the non-functional prototype is becoming a functional one and the user can play with the device and the experimentors get new user feedback. And even when the system is delivered there is still the wish to get user feedback for the development of new generations of the system.

3 Experimental Setup for User Interface Studies

In the natural sciences as well as the social sciences researchers use experiments to find out new facts that help them develop new insights and to proove new theories. An experiment is the setup of a controlled environment where the studied procedures are running in a controlled manner that help the researcher to gather reliable and unbiased data. With the same setup and the same conditions other scientists must be able to redo the same experiment and to gather the same data.

Figure 2: Wizard of Oz Experiment Setup

User interface designers are also using experiments to find out wether a new design is useful or not. One example of such an experiment is the so called Wizard of Oz experiment[Dahlbäck *et al.*, 1993]. Figure 2 shows the setup of a Wizard of Oz experiment. On the left hand side we see the Wizard of Oz, a researcher, that watches the scenario but is invisible for the user. On other side we see the user. Note that not all of the functionality of the prototype needs to be implemented since the Wizard of Oz on the left has full access to the client system and can simulate system response (note the wireless connection between the client's system and the wizard's control terminal). There are some minor problems such as a slower responsiveness to user input by the researcher than by a fully functional system. The most obvious advantage is that not all of the system's functions need to be implemented before it can be figured out together with the user what is really needed.
For a good evaluation of user studies, interface designers must set up an experimentation environment that allows them to gather reliable data. There are different possibilities to set measuring points to gather data from. The data can be processed by the experimentor or by certain tools such as data mining tools. For data gathering we see a data source that is often neglected in user interface experiments: the data that is inside the software components. Usually the system is observed from outside, for example by watching the user interacting with the system. This reminds to black box testing where only the interfaces and specifications of the tested system are known. White box testing allows to see inside the component. Applied to the problem of interface design this would mean that the investigator could see the state of the software components inside the system at a given time.
There are many ways to observe an experiment but it is more difficult to influence it. For data acquisition during the experiment the scientist has several possibilities: he can take notes while watching, use a video camera, redirect the user's display and record it, let the user fill out questionnaires, interview the user, and many more.

4 User Interfaces seen as Communicating Components

With the term user interface we refer to the combination of all input and output devices that can react on user input or that can catch the user's attention. We indicated this in figure 2 with the circles that are on or around the user and that can communicate with each other and with the user. An example is the field of wearable computing. Another good example is mobile computing in combination with ubiquitous computing where the user moves through a very communicative environment where new components try to connect to the user in an adhoc manner and change the user's interface.

These devices either provide user interfaces on their own or delegate this task to other devices which provide I/O containers for them, for example a display. Today many devices cannot yet communicate with each other but we expect this to change.

A multimodal user interface comprises several distinguishable components that together provide the means for input and output of data from and to the user. These components want to send and receive user data. In figure 2 we indicate the bunch of devices that are on the user's body or in his environment with circles. They can communicate with each other, but in any case they also send their data to the wizard in the background that controls the experiment.

5 Experimentation-ready Software Architecture

For the development of software and for the monitoring and control during runtime there is the need to enhance software components with information that is not needed for the functionality. The techniques used in these cases can be also be valuable for our purpose. Namely, we present techniques for augmenting the code, for switching components, for observing the data channels, for observing the components by reflection, and for accessing the platforms the components run on.

Augmenting the code

A simple example for enhancing code is the usage of compiler switches for debugging. Compilers can integrate additional information into the code that can be used by debuggers for tracking the progress of the program execution or setting trap points. The debugging interfaces can be accessed by special tools. Another possiblity to access runtime information is the use of profilers.

A more advanced solution is the usage of aspects. Aspect oriented programming is a new paradigm that allows to weave cross concern aspects into the program. This could be debugging information but it could as well be any other code that is temporarily needed for certain purposes. Tools such as AspectJ weave code for an aspect into the existing code and thus clearly separate the functionality of the component from aspects such as debugging or tracking. If an aspect is not used any longer in the code, the aspect can be cleanly removed automatically. For more information on aspect oriented programming see [Parc, 2001; Highley *et al.*, 1999].

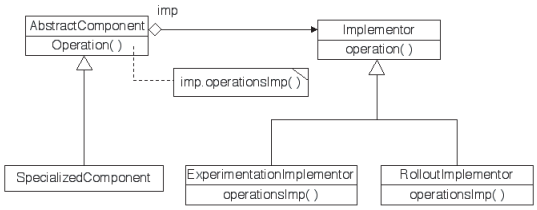

Figure 3: Bridge Pattern

Switching components

[Gamma *et al.*, 1995] provide several design patterns that can be used in our context. Examples are the Factory, Abstract Factory, Adapter, the Bridge, Decorator, Proxy, Observer, Strategy, Chain of Responsibility, and Template Method. Factory and Abstract Factory are construction patterns. They allow to decide at runtime what instance should be created. Adapter, Bridge, Decorator, and Proxy are structural patterns. They allow to change the structure of components without changing their interfaces. Observer, Strategy, Template Method, and Chain of Responsibility are behaviour patterns that allow to change the behaviour of the system. As a prominent pattern we use the Bridge pattern to explain our ideas. Figure 3 shows the structure of the pattern. This pattern is often used to replace components that are not yet implemented with dummy implementations that don't provide any functionality at all or only provide reduced functionality. This can also be used for simulating a device that is not yet available. So during development, the structure of the system is clear and does not need to be changed if the bridged component is ready. In figure 3 we use this pattern to change the component that implements the interfaces for experimentation with the component for the roll out. The overall structure remains the same, only one component is exchanged, even at runtime.

Observing the data channels

In distributed systems there is another source for getting information: the communication middleware. We can differentiate between asynchronous, event-based systems and synchronous, method-based systems. In event-based systems a dedicated service, the event service is the broker that transports events from event publishers to event subscribers. [Bruegge *et al.*, 1993] present a framework for dynamic program analyzers of distributed programs. Basically, the program analyzer connects to the event service as a subscriber for events he is interested in. But it can also connect as an event publisher and mimic other components by simply sending events instead of them. For example, in the Wizard of Oz experiment the Wizard can observe all event channels and publish events on behalf of a component that is not yet implemented. This can be done with all event based systems such as OWL, a framework for intelligent buildings[Bruegge *et al.*, 1999]. There is no runtime overhead if the observer is running in a separate process.

Reflection

The complexity of distributed systems is significantly higher than that of stand-alone systems. So it will become more and more important to be able to monitor the components and their state and behaviour at runtime or for post-mortem analysis. The ability for monitoring needs to be built into the system from the very beginning. Some information is integrated into the standard APIs of languages such as Java with the Reflection API or the Runtime Type Information (RTTI) in C++. For distributed systems this information is not enough. Systems such as Carp@[Breitling *et al.*, 1999] enhance the built in reflection mechanisms by adding services that gather runtime data about the active components in the system. This data can be used for system management. Also experimentation tools may access them.

Accessing the platform

Modern distributed systems such as CORBA 3[OMG, 2001] or Enterprise Java Beans[Javasoft, 2001] are platforms that provide services for the components. Among these services are persistency, startup, stop, load balancing and more. These platform services can also be accessed by tools that need to gather data about the components.

6 Example Setup for Mobile Devices

6.1 In-Car Computing Scenarios

In this section we present an experimental setup for the design of mobile systems. We describe the application of the presented component-based user interface design approach to user studies in experimental car settings. In the near future cars will be equiped with ever more intelligent driver information systems. In addition to information about the current traffic situation and the state of the car, the driver will have the possibility to interactively ask for information from Internet resources. One of the main challenges in this scenario will be the design of a context-aware and easy-to-use car interface that prevents the driver from information overload.

The usefulness of in-car multimedia and information system architectures depends largely on their ability to meet the customer's demands and on their effect on traffic safety. We understand the Internet-enabled car as the idea of ubiquitous computing applied to the automobile, supporting the driver with the information he needs when he needs it. We believe this is a much more convincing approach than simply to install a Web browser in the car. The concept of the perceptual interface [Pentland, 1999], adaptive both to the overall situation and to the individual user, is a promising concept to accomplish the goal of the smart car.

While for context-awareness in ubiquitous computing many mobile systems are additionally equipped with sensors to enable them to adapt to situation and user, the automobile already contains many sensors and is now equipped with powerful computing resources and innovative user interaction devices. A European research project focused on the development of an in-car mobile service platform using Java technology [Inform, 2001]. Based on this software platform car passengers can use mobile services while on the road. The services can be dynamically downloaded onto the car platform on demand using wireless communication networks. Among the services that can be offered are location-based services like information about nearby gas stations, hotels or restaurants, traffic information services such as local danger warnings and dynamic parking information, remote car diagnosis, emergency calls or multimedia services such as games or audio-on-demand.

The developments towards Internet-enabled in-car information systems require in-depth user studies to find new metaphors and design appropriate dialogues. The increasing complexity requires the support of methods and tools to effectively and efficiently build new services and applications and test new user interface components.

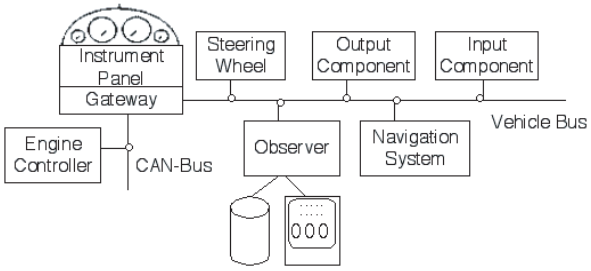

Figure 4: Car Network Architecture

6.2 System Design for Driver Interaction

System Architecture

In a car environment the user computer interaction has to be designed very carefully. The possibilities for input and output devices such as screen size etc. are better compared to small handheld computers. Nevertheless, the desktop paradigm doesn't work in a mobile driving situation. Special technology has been developed for user interaction within the car. Besides the classic knobs and dials for car control such as the steering wheel and gas pedal, other devices find their way into the cockpit allowing driver and passengers to enjoy multimedia features and receive useful information while traveling.

The fundamental car network architecture is depicted in Fig. 4. Different bus technologies are used in modern cars. The figure shows a part of the vehicle bus system that connects the user interaction devices. A gateway enables this bus to communicate with the Controller Area Network (CAN) to which internal electronic control units (ECUs) such as the motor controller are connected. Input and output components like the instrument panel or buttons which are integrated in the steering wheel are directly connected to the bus system just like the radio or navigation system. Every component can generate a message which is broadcasted on the bus. The other components listen for messages on the bus and filter out the information they're interested in. To this architecture we add an observer component which is connected to the bus, records the events from the distributed components and writes the information into a database.

Driving safety
It has been shown that the use of mobile phones while driving increases the risk of accidents [Redelmeier and Tibshirani, 1997]. Few studies have been conducted that examine the influence of the use of other information and communication services [Tijerina, 2000]. The experience of the mobile phone, however, makes us aware of the danger that the usage of such services during driving bears. On the other hand, danger warning and traffic information services may result in improved traffic safety. The effect may be quite different in changing driving situations. Different system designs have to be examined in appropriate test setups. Context-awareness is a key issue since it is crucial to support the driver with the right information in the right situation and at precisely the right time, i.e. not when he is concentrating on a lane change. While interaction can be rich when the car is parking, during driving the interaction must be kept to a minimum in order not to distract the driver from his task. In order to best support the driver, information needs to be given to him according to the situation. Context-aware, yet natural input and output components and system behaviour selection are crucial to a successful interface design. In order to optimize the use of the functionality modern cars offer to their owners, the user interface in a car environment needs special consideration.

Vehicle I/O devices and technologies for user interaction
New input and output devices are included in user design experimentation. The ErgoCommander is a special input device which can be turned clockwise and counterclockwise, moved horizontally and vertically like a joystick and pushed like a button. The Head-Up Display [Hooey and Gore, 1998] allows the projection of information onto the front window, e.g. for displaying warning information such as "accident ahead" or the allowed maximum speed in case the driver is too fast. Speech recognition technologies up to date allow for the input of a limited set of distinct voice commands. The noisy surrounding in the car keeps current systems from acceptable recognition rates for random voice input. Text to speech engines, though having made great progress, are not yet competitive to natural human voices. Using this technology means that the driver has to spend too much attention to the speech information which distracts him from the driving task. In the following section we view the different I/O devices as components. The interaction with the user is handled by software components which are controlling the device.

6.3 Experimental Setup for Researching Driver Car System Interaction

Conventional User Study Setup and Dialogue Recording
In current experimental setups, the data is usually collected by writing it directly to files with commands spread all over the actual application code. This involves a lot of work in adding the data collecting functionality to the code and later remove it. It also makes it hard to later extract relevant data from the generated files, process the data and produce charts. The necessary software support is usually added to the systems in an ad hoc manner. A consistent approach that supports experimentation, data collection and evaluation is missing in today's setups.

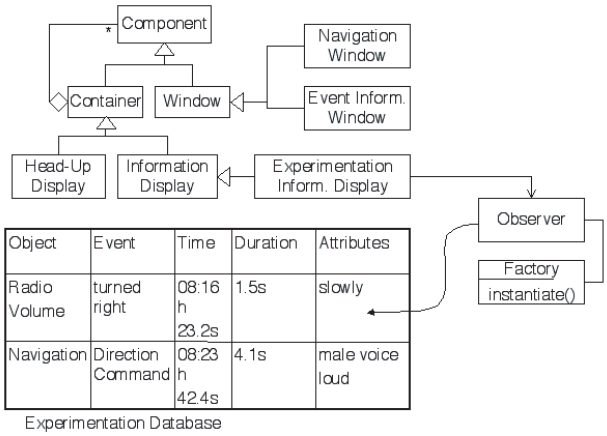

Figure 5: Component Architecture

Deploying a Component Based User Study Setup in a Car Environment
In this paragraph we show the use of the component based approach in the car environment. In the experimentation setup the driver is watched by a camera and a person on the rear observes the behaviour of the test person. User feedback is also captured through speech recording and videotaping gestures. The movement of the drivers eyes can be tracked as well as his behaviour towards different text-to-speech engines with different voices and volume. Interaction data like the choice of input channel or the reaction to different output modalities are recorded. Such data about user interaction can be gained from the software components. We gather and save this data. The interaction between the driver and the system is multimodal. While turning the ErgoCommander to choose a different MP3 song or another radio station the driver might specify a navigation destination to the navigation system or initialize a mobile information retrieval agent by voice. The software component architecture allows the a posteriori analyzation of concurrent or overlapping actions taken by the user.
We explore the component based design of user interaction with the navigation system using new user interface components, particularly the use of a head-up display. We take this example for two reasons: Route navigation is a familiar application, already in use especially in middle and upper class vehicles, and it is an application which is used while the car is moving. The basic structure of the component architecture is given in Fig. 5 in UML style.
The Head-Up Display is modeled as a container component. It can contain information display components which are themselves logical containers. Navigation information can be presented in a window which is placed into an information display component. This information display component can then be projected on the head-up display. For testing, special experimentation components inherit from the actual system components and are extended for test purposes. These components are controlled by the observer, e.g. instantiated using the component factory. The configuration of this test setup can easily be changed during experimentation. Tools with

graphical user interfaces for managing the setup and controlling the experiment can be implemented. The relevant information that the components generate is collected and stored by the observer component. Standard evaluation procedures can be run on the structured test data tables in the database.

7 Conclusion

In this paper we have shown an iterative design process for user interfaces. By using Wizard of Oz setups user experiments can start in an early development stage. We have presented a model of user interfaces that is based on a combination of interface components. Based on this we have shown that the enabling techniques based on well known principles are already available. Testing the user interface can start at an early stage without much more cost as already raise for other requirements such as management. A setup for user interface experimentation shows the application of these principles.

Acknowledgements

This work was supported by the Bayerische Forschungsverbund Softwaretechnik (Forsoft) and the BMW Group.

References

[Arndt, 1999] Timothy Arndt. The Evolving Role of Software Engineering in the Production of Multimedia Applications. In *Proceedings of ICMCS'99*. IEEE Computer Society, 1999.

[Boehm, 1988] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5): 61–72, May 1988.

[Breitling *et al.*, 1999] Max Breitling, Michael Fahrmair, Chris Salzmann, and Maurice Schoenmakers. Carp - Managing Dynamic Distributed Jini Systems. In *OOPSLA'99 Workshop on Reflection and Software Engineering*, pages 173–184, 1999.

[Bruegge *et al.*, 1993] Bernd Bruegge, Tim Gottschalk, and Bin Lou. A framework for dynamic program analyzers. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 65–82, September 1993.

[Bruegge *et al.*, 1999] Bernd Bruegge, Thomas Reicher, and Ralf Pfleghar. OWL: An Object-Oriented Framework for Intelligent Buildings. In *Proceedings of the 2nd Interational Workshop on Cooperative Buildings*, 1999.

[Dahlbäck *et al.*, 1993] N. Dahlbäck, A. Jönsson, and L. Ahrenberg. Wizard of Oz Studies - Why and How. In *Proceedings of the ACM International Workshop on Intelligent User Interfaces*, 1993.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Highley *et al.*, 1999] T. J. Highley, Michael Lack, and Perry Myers. Aspect Oriented Programming - A Critical Analysis of a new Programming Paradigm. Technical Report CS-99-29, University of Virginia, 1999.

[Hooey and Gore, 1998] B. L. Hooey and B. F. Gore. Development of human factors guidelines for advanced traveler information systems and commercial vehicle operations: Head-up displays and driver attention for navigation information. Washington, DC: Federal Highway Administration (FHWA-RD-96-153), 1998.

[Inform, 2001] Inform. Inform: Information for the Millions, http://www.inform-eu.org, 2001.

[Javasoft, 2001] Javasoft. Java(TM) 2 Platform, Enterprise Edition, http://www.javasoft.com/ejb, 2001.

[Maybury, 1999] Mark T. Maybury. Putting Usable Intelligence into Multimedia Applications. In *Proceedings of ICMCS'99*. IEEE Computer Society, 1999.

[OMG, 2001] OMG. Welcome to the OMG's CORBA Website, http://www.corba.org, 2001.

[Parc, 2001] Xerox Parc. Aspect-Oriented Programming Home Page. http://www.parc.xerox.com/aop, 2001.

[Pentland, 1999] Alex Pentland. Perceptual Intelligence. In *Hans-Werner Gellersen (Ed.): Handheld and Ubiquitous Computing, First International Symposium, HUC'99, Karlsruhe, Germany, September 27-29, 1999. Lecture Notes in Computer Science, Vol. 1707, Springer*, pages 74–88, 1999.

[Redelmeier and Tibshirani, 1997] Donald A. Redelmeier and Robert J. Tibshirani. Association between Cellular-Telephone Calls and Motor Vehicle Collisions. *The New England Journal of Medicine*, 336(7), 13 February 1997.

[Tijerina, 2000] L. Tijerina. Issues in the Evaluation of Driver Distraction Associated with In-Vehicle Information and Telecommunications Systems. Internet Forum on Driver Distraction. Hosted by the US DOT NHTSA. http://www.driverdistraction.org, 18 May 2000.