A Case for Human-Driven Software Development

Emilie Balland, Charles Consel, Bernard N'Kaoua, Hélène Sauzeon

A Case for Human-Driven Software Development

Emilie Balland, Charles Consel, Bernard N'Kaoua, Hélène Sauzéon
Inria/University of Bordeaux, France

***Abstract*—Human-Computer Interaction (HCI) plays a critical role in software systems, especially when targeting vulnerable individuals (*e.g.,* assistive technologies). However, there exists a gap between well-tooled software development methodologies and HCI techniques, which are generally isolated from the development toolchain and require specific expertise.**

In this paper, we propose a human-driven software development methodology making User Interface (UI) a full-fledged dimension of software design. To make this methodology useful in practice, a UI design language and a user modeling language are integrated into a tool suite that guides the stakeholders during the development process, while ensuring the conformance between the UI design and its implementation.

I. Introduction

Human-Computer Interaction (HCI) defines a range of principles and methodologies to design User Interfaces (UIs), aiming (1) to improve the interaction between users and computers, (2) to address how interfaces are implemented, leveraging techniques such as program generation and component architectures, and (3) to propose methods to evaluate and compare interfaces.

Despite the many successes of HCI, when it comes to software development, this domain expertise often does not go beyond guidelines (*e.g.,* addressing the needs of the elderly and users with disabilities [1]). Sometimes, guidelines are mapped into UI design artifacts. However, for a lack of tools, these artifacts remain contemplative. As a consequence, there exists a gap between UI design and software development. This gap is not typical of the HCI domain. Yet, its consequences are dramatically increasing in importance as software systems intertwine with our daily activities, both professional and domestic. Nowadays, a host of systems are playing a critical role for users in terms of safety, privacy, *etc*. Let us examine manifestations of the gap between UI design and software development.

UI design conformance. A UI design can be flawless but incorrectly implemented by the programmer. Indeed, UI design is poorly integrated in the software development process, leading to inconsistencies [2]. Some aspects of the UI design may be misinterpreted or overlooked by the programmer. For example, the programmer may omit to put the user in the loop when sharing sensitive data, leading to a privacy breach.

Under/over specification of UI design. Even if the UI design is rigorously followed, it can still be under/over-specified regarding the user preferences and capabilities. On the one hand, under specification can lead to implementation mistakes. For example, the programmer can implement an interaction using a least favorite modality, or imposing an inadequate cognitive burden on the user. This situation is illustrated by a visually impaired user prompted with a textual message or, the operator of the X-ray machine required to memorize configuration parameters. On the other hand, over specification can lead to an implementation that is completely tailored for specific users and unable to accommodate others.

Out-of-sync UI design. Like any design artifacts, the UI design can be out of sync with the software implementation, after a few evolutions. These discrepancies can create undesirable or unexpected features. For example, assistive technologies have to adapt to the evolution of the user's capabilities (*e.g.,* an increasing hearing impairment).

The above-mentioned problems are mainly due to the contemplative nature of the design artifact. As such, a UI design is an informal specification, which can be partial and ambiguous. As well, the concepts and notations used to specify UI design artifacts may not compose well with the design concerns of the overall software system. In an attempt to address these problems, practitioners may resort to manually ensuring some level of conformance between the design and the software being developed or evolved. This process is error prone and human intensive. It is known not scale in size and over time [2].

To bridge the gap between UI design and software development, our approach consists in making UI design a full-fledged dimension of software design. We introduce a language dedicated to designing UIs in a high-level manner, while capturing the key requirements of user interaction. We go beyond a contemplative approach and process a UI design artifact to produce a dedicated programming framework that supports the implementation of all the dimensions expressed in a design artifact. This programming framework guides the stakeholders during the development process, while ensuring the conformance between the UI design and its implementation over time.

II. Working Example

We illustrate our development approach with an example from the domain of assisted living; it is one of numerous scenarios that came out of our collaboration with a province-wide organization for elderly care. Our proposed scenario addresses the vital need for the elderly to use new communication mediums, such as email and social networks, towards preventing their social isolation, especially with respect to the younger generations. We thus started designing an application that would allow the elderly to check their email without requiring them to invest mental resources to operate new

technologies [3]. To do so, we strove to use familiar devices such as a TV set and its remote control, whenever possible.

The overall scenario of our application is as follows. At selected times during the day, the user is notified of new email messages. The user controls which messages to display and reply to, interacting with the available features of familiar devices, such as the screen, speakers and remote control of a TV set, matching his/her preferences, and sensory and cognitive capabilities.

III. Overall Approach

Our approach integrates a model of the user and the UI design of an application. It provides languages and tools to support the stages ranging from user modeling to software development.

At user-modeling time. Given a target population, user requirements are specified in a model, making explicit their preferences, and sensory and cognitive resources. This user model plays a pivotal role to ensure that a given application matches the specifics of a target population.

At design time. Given an application to be developed, its UI design addresses the *interaction* and *presentation* aspects, as is conventional in HCI [4]. Interaction design defines when and what user interactions are required by the application, whereas presentation design defines the static requirements of the UI, such as the modality and the UI components (*e.g.,* size and color). Accordingly, our UI declarations are decomposed into two layers: (1) interaction declarations and (2) presentation declarations, including the sensory and cognitive requirements. Besides documenting the design, UI declarations permit various verifications, such as matching an application against a user model.

At programming time. Because our approach compiles a design artifact into a customized programming framework, programming support can be generated and the host programming language can be leveraged to enforce the design both in terms of programming obligations and restrictions. For example, an interaction design is compiled into a set of Java types that are used in the abstract classes of the programming framework. By extending these abstract classes, the developer is forced to meet the interaction design. This generative approach ensures the conformance between the design and the implementation stages, while offering high-level programming support to the developers.

At deployment time. When the software system is deployed, the user needs to anticipate what interactions are enabled and required. This is critical information when the application is intertwined with the individual's activities. Our approach can guarantee that an application correctly announces its user interactions. This guarantee relies on the fact that UI declarations are part of the application design and are used to generate the underlying programming framework. Furthermore, when an application is deployed, the user model can be checked against the application interactions to detect a mismatch.

At run time. When constraints cannot be checked at compile time, this process is performed at run time. For example, getting a user input prior to sending sensitive information. Run time checks are included in the generated programming framework. Default policies are invoked to deal with constraint violations.

To make our approach useful in practice, it is being integrated into DiaSuite, a tool-based development methodology [5], [6]. DiaSuite is dedicated to a specific design paradigm, namely Sense/Compute/Control (SCC) [7]. This design paradigm applies to any application that interacts with an external environment. Such applications are pervasive in domains such as telecommunications, home automation, and avionics. The DiaSuite compiler generates a Java programming framework dedicated to an application design; it guides the programmer while ensuring the conformance between the application design and the implementation.

Figure 1 depicts the development lifecycle supported by DiaSuite, extended with a user modeling language and a UI design language. This paper focuses on user modeling and UI design, both presented in the following section.

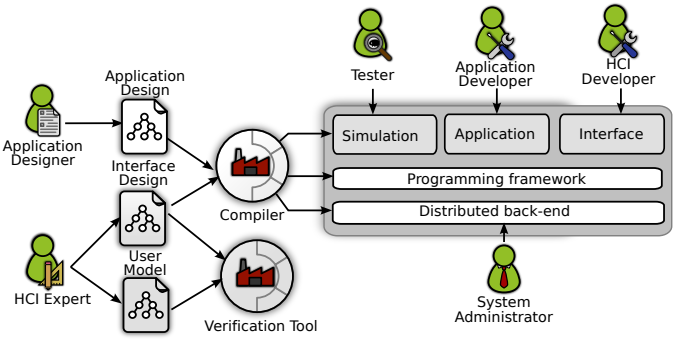

Figure 1. Our tool-based development process

IV. Design

The design aspects of our approach rely on three languages: (1) a *user modeling language* to specify the preferences, and sensory and cognitive capabilities of the target user; (2) a *UI design language* to define both the presentation and interaction design; and (3) the *application design language* with interwoven interaction design aspects, which abstract over presentation details. These languages allow to separate design concerns, making the same application adaptable to different target users with different preferences and capabilities.

A. User Modeling Language

For a lack of space, we focus this presentation on the cognitive dimension of the user, omitting user preferences. A specific area of Cognitive Science is devoted to the definition of theoretical models of the human mental activity. One approach consists in abstracting the human mental activity as a simplified computing process. HCI researchers have rapidly applied the resulting computational models to reason about user interaction during the design stage of a system [8]. For example, the

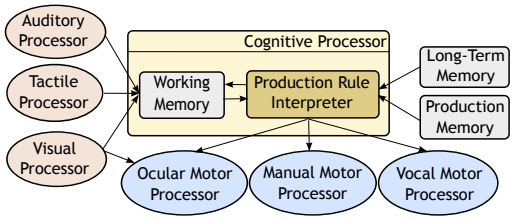

Figure 2. EPIC computational model (see [9])

Executive-Process Interactive Control (EPIC) architecture has been used to evaluate the effects of ageing [9]. As shown in Figure 2, the EPIC model represents the cognitive capabilities of the user as the simplified components of a computing machine (*e.g.,* visual processor, cognitive processor, working memory). We leverage this computational model to develop our modeling language of the user.

Both the user modeling language and user-interaction declarations are defined with respect to the EPIC model. Specifically, a user is viewed as providing various resources, in terms of EPIC components, to perform an interaction. A resource may be *void* (absent), *low*, *medium* or *high* (ordinary). Symmetrically, an interaction is defined as requiring a resource load from each EPIC component involved.

Let us examine an example of a user model. Consider an average elderly individual whose model is based on a study of the common effects of ageing using EPIC [9].

```
user_model average_elderly {
  cognitive_processor = medium;
  working_memory = medium;
  long_term_memory = medium;
  short_term_memory = high;
  visual_processor = medium;
  auditory_processor = low;
  manual_motor_processor = low;
  vocal_motor_processor = high;
}
```

Let us illustrate how this user model translates in terms of user interaction. An average elderly user has diminished visual capability, advising against interactions via abundant textual messages. In contrast, vocal capability is fully functional, allowing interactions via vocal commands. Because the working memory is reduced, interactions should be decomposed in manageable steps.

Our approach is user-centric in that it starts by defining a model of a user, or a group of users. This user model then plays two roles with respect to software design: (1) it drives the design of the application UI and (2) it is checked against a design for conformance.

B. UI Design Language – Interaction Layer

To define an interaction design, we introduce a set of *abstract interactors* that denotes the information exchange between the application and the user. Three types of abstract interactors are needed.

- **output**: information flows from the application to the user (*e.g.,* a notification of a new email message),
- **input**: information flows from the user to the application (*e.g.,* a user reply to an email message),
- **prompt**: information flows round-trip between the application and the user (*e.g.,* a dialog with the user to ask whether an email message is to be replied to).

In our working example, the interaction design is defined by the following abstract interactors:

```
prompt Ask_readMail as MailInfo -> Bool;
output Display_mail as Mail;
prompt Ask_reply as String -> Bool;
input Mail_reply as Mail
output Feedback_reply as MailInfo;
```

The abstract interactor declarations define the type of the input and/or output to be expected by the application logic. For example, the input interactor named Mail_reply expects a user input of type Email. These type declarations are leveraged by the compiler. Indeed, each abstract interactor is compiled into a Java interface that declares the input and output types.

Let us now show how these abstract interactors are interwoven with the application design language. In DiaSuite, an application design follows the SCC paradigm whereby sensing *entities* fuel *context operators*, which refine (aggregate and interpret) these data before passing them to *control operators*. These operators eventually trigger actions on entities.

In our approach, the DiaSuite design language leverages abstract interactors, making explicit the user interactions. Specifically, like an actuator, an **output** interactor is invoked by controller operators to provide some information to the user. Like a sensor, an **input** interactor fuels context operators with a user reply. A **prompt** interactor is both invoked by a control operator and sensed by a context operator. For example, consider a prompt interactor, instantiated with a TV and its remote control. When invoked, this interactor locks both devices from the time the prompt message is rendered by the TV, until the user inputs the reply with the remote control. In effect, a prompt defines a cycle in the SCC graph of an application. This semantics is more specific than sensors and actuators in that it guarantees the coupling between a prompt message and the reply. As a result, a prompt interactor must occur both in a controller and context declaration.

Let us express our working example in the SCC paradigm. To do so, we refine its functionalities: at an appropriate moment, based on a calendar of the user activities and a motion detector, if there are new email messages, the application prompts the user to determine whether (s)he wants to read them. If so, the email is displayed. As shown in Figure 3, the design can be represented by a data-flow graph of SCC components.

In this design, the Availability context determines whether it is an appropriate moment to read email messages based on the Calendar and MotionDetector sensors. This information is combined by the NewEmailToRead context with the output of the NewEmail context that checks whether new messages have arrived. If these conditions hold, the ReadMail controller is invoked to ask the user, via the Ask_readMail prompt, whether (s)he wants to read email messages. The ReadMailResponse context filters positive responses from the user, via the Ask_readMail prompt. Finally, the DisplayMail

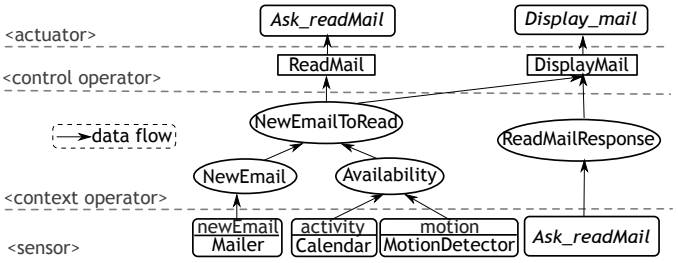

Figure 3. Extract of the design of the email application

controller is invoked to allow the user to read messages via the Display_mail output. As is required, the Ask_readMail prompt occurs both in the ReadMail controller and ReadMailResponse context declarations, ensuring that user interactions are consistent with the application design.

The programming framework generated by the DiaSuite compiler has been extended to ensure the conformance between the interaction design and the implementation. Indeed, each application component is compiled into an abstract class. When a component depends on an abstract interactor, the corresponding abstract class leverages the type generated from the abstract interactor declaration.

C. UI Design Language – Presentation Layer

The presentation layer of the UI design language allows to instantiate the abstract interactors with concrete devices (whether hardware or software). Because there are concrete, their required sensory and cognitive load must be specified. The burden of these interactions can be the result of the analysis of cognitive functions [10].

In our example, we can map the Ask_readMail interaction with two distinct devices: the TV display, to render the question, and the TV remote control, to acquire the reply:

```
prompt Ask_readMail {
 TVSet.display as output {
    visual_processor = medium;
    cognitive_processor = low;
 }
 TVRemoteControl.command as input {
    manual_motor_processor = low;
    cognitive_processor = low;
 }
}
```

The conformance with the user model can be automatically checked by performing a unitary matching between the interaction cost of each EPIC component and the load supported by the user.

In addition, the UI design can be leveraged to ensure general-purpose UI guidelines. For example, checking user feedback consists in verifying that an input interactor is followed by an output interactor in the UI design. In the email application, the Mail_reply input is followed by the Feedback_reply output.

V. Conclusion and Future Work

We have a prototype implementation of our approach, integrated in DiaSuite. We are actively revisiting assistive applications to validate our approach. As illustrated by the email application, we are finding a host of UI properties that can be automatically ensured by our approach.

We plan to make the UI design artifacts understandable to non-technical users, enabling the selection of assistive applications by end-users (*e.g.,* caregivers). Another direction is to leverage the UI design to provide specific instrumentation code, supporting the evaluation of assistive applications with real users.

Going beyond a unitary matching of an interaction against an element of a user model, we are developing strategies to compute an overall assessment of an application with respect to its possible sequences of interactions. This line of work also considers multimodal interactions.

Finally, we envision using our approach in the context of safety-critical systems. In particular, we have been developing various aircraft modules, such as an autopilot system, where UI declarations should greatly contribute to guaranteeing safety requirements involving the UI dimension.

Acknowledgment

We thank Martin Hachet, Fabien Lotte and Jean-Daniel Fekete for their helpful comments. We also thank the anonymous referees for their valuable feedback.

References

[1] "Ergonomics data and guidelines for application of ISO/IEC Guide 31 to products and services to address the needs of older persons and persons with disabilities," International Standards Organization (ISO), TR 22411, 2008.

[2] S. Viswanathan and J. C. Peters, "Automating UI guidelines verification by leveraging pattern based UI and model based development," in *Proceedings of the 28th International Conference on Human Factors in Computing Systems*, 2010, pp. 4733–4742.

[3] U. Lindenberger, M. Lövdén, M. Schellenbach, S. Li, and A. Krüger, "Psychological principles of successful aging technologies: A mini-review," *Gerontology*, vol. 54, no. 1, pp. 59–68, 2008.

[4] B. Shneiderman and C. Plaisant, *Designing the User Interface - Strategies for Effective Human-Computer Interaction (5. ed.)*. Addison-Wesley, 2010.

[5] D. Cassou, J. Bruneau, C. Consel, and E. Balland, "Towards a Tool-based Development Methodology for Pervasive Computing Applications," *IEEE TSE: Transactions on Software Engineering*, vol. 38, no. 6, pp. 1445–1463, 2012.

[6] D. Cassou, E. Balland, C. Consel, and J. Lawall, "Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE'11*. ACM, 2011, pp. 431–440.

[7] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[8] J. Olson and G. Olson, "The growth of cognitive modeling in human-computer interaction since GOMS," *Human-Computer Interaction*, vol. 5, no. 2, pp. 221–265, 1990.

[9] D. E. Meyer, J. M. Glass, S. T. Mueller, T. L. Seymour, and D. E. Kieras, "Executive-process interactive control: A unified computational theory for answering 20 questions (and more) about cognitive ageing," *European Journal of Cognitive Psychology*, vol. 13, pp. 123–164, 2001.

[10] G. A. Boy, "Cognitive function analysis for human-centered automation of safety-critical systems," in *Proceedings of the 16th International Conference on Human Factors in Computing Systems, CHI'98*. ACM, 1998, pp. 265–272.