What programming languages for interactive systems designers?

Stéphane Chatty
Université de Toulouse - ENAC
7 avenue E. Belin, 31055 Toulouse, France
chatty@enac.fr

Stéphane Conversy
Université de Toulouse - ENAC
7 avenue E. Belin, 31055 Toulouse, France
conversy@enac.fr

ABSTRACT
We highlight the role of programming in the engineering of interactive systems, in the long term perspective of creating general theories of interaction to support engineers. We outline a research roadmap aimed at both providing designers with appropriate programming languages and understanding the nature of interactive programs.

Author Keywords
interactive software, programming languages, notations, engineering, design, theory

ACM Classification Keywords
H.5.2 Information Interfaces and presentation: User Interfaces; D.3.3 Programming Languages: Language Constructs and Features

INTRODUCTION

For the computing industry and the world in general, this has been the decade of interactive systems. Long announced by researchers, natural user interfaces have reached the industrial stage and found their way to our pockets and our bedside tables. This has changed the computing industry by giving a more central role to individual programmers and to design professionals. This also has bolstered the public awareness about software programming, with the popularization of sentences such as "programming is the new literacy" or "program or be programmed" [26].

However, there is a paradox: what makes computers so interesting is their interactivity, and still the programming techniques that are proposed to the eager masses are the old techniques, invented for designing algorithms and not interaction! This might create some disillusionment with computer science, and we indeed are observing its first signs in engineering students. We contend not only that addressing this paradox is our responsibility, but that creating programming languages for interactivity would bring benefits to engineers.

In this article we discuss three statements and their consequences for research on engineering interactive systems:

- *engineers need general theories of interaction*
- *designing interactive systems is programming*
- *programming languages are user interfaces.*

We elaborate on these statements and why they should be important to our research community, then we outline possible areas of research aimed at exploring their consequences.

WE NEED GENERAL THEORIES OF INTERACTION

Engineering has been defined by the Engineers Council for Professional Development, in the United States, as [1, emphasis ours]:

> The creative application of *scientific principles* to *design or develop* structures, machines, apparatus, or manufacturing processes [...] *or to forecast their behaviour* under specific operating conditions [...].

Let us first note the central role of scientific principles in this definition: engineers need scientific theories, and progress in engineering is often triggered by theoretical progress. What is less intuitive is the combination of roles assigned to engineers: to design (with an emphasis on creativity) or to forecast. We derive two lessons from this:

- designing is a major component of engineering
- scientific theories are used both to design and to forecast

Indeed, examples abound of theories that are used to both design and forecast. Mechanical engineers use the same concepts of forces and pressures to forecast the behavior of bedrock and to design a bridge built on it. The same holds for chemical engineering, for bio-engineering, and even for traditional software engineering. Theories of computation allow to describe computation systems (even natural ones) and to predict their behavior. Through programming languages derived from them, they also allow to design computation software and to forecast its behavior. What allows these theories to support both design and forecast activities is their generality: they encompass all the relevant aspects of the system being designed and its environment. We contend that this should be an explicit goal in the field of interactive systems too.

Engineering interactive systems involves, before designing and developing a software system, the analysis of the human and physical environment in which it will operate. This requires the ability to model software, human cognition and perception, activities and tasks, application domains, and even physical interaction. Currently, there are specialized theories for each of these, and we spend considerable efforts developing empirical methods that help engineers combine

these theories. But this focus on methods should only last as long as we feel compelled to teach separate theories in our engineering courses. The ultimate goal should remain the elicitation of general theories that encompass all these aspects, as attempted for instance by Palanque et al. when using Petri nets to model the behavior of both the user and the software [23].

DESIGNING INTERACTIVE SYSTEMS IS PROGRAMMING

Programs are a central concept in theories of computation, and in the whole field of software engineering. At the opposite, in the field of interactive software engineering they are often relegated to the status of mere by-products. But programs are nevertheless a major part of interactive systems, and any general theory of interaction will need to clarify their status and the role of programming. This section aims at demonstrating that the activity of designing interactive systems can be considered as programming.

Generalized programming

In the following, we consider programs as descriptions of the behavior of an entity when an execution device runs them, and programming as an act performed by a human to design the description. It is possible to picture the activity of programming as carried out by professionals trained in computer science and software engineering, who write algorithms in C, Java or C#. We contend that this picture is harmful to our ability to support engineers who design or need to forecast the behavior of interactive systems. Both the design of systems, that is the definition of their structure and their behavior, and the analysis of their environment are closer to programming than it seems.

To start with, programming does not only consist in writing. It also involves reading, understanding, checking, designing, forecasting the behavior of code. It is an engineering activity, and cannot be distinguished from the engineering of interactive systems on purely methodological grounds.

Then programming has ceased to be a task reserved to computer scientists and software engineers. More and more graphical designers, web designers and communication professionals learn programming languages for producing parts of interactive systems [8, 21]. For them, programming is part of their activity just like drawing sketches. Languages and environments such as Processing [25] have been created explicitly for designers.

Programming should also not be restricted to the engineering of algorithms. When designing an interactive system, a number of entities that must be analyzed or designed can be modelled as programs. The most obvious is the interactive behavior of visual components: it is now commonplace that interaction designers program them, using whatever notation is available to them. The same holds for animation, and even for graphics themselves: sometimes, graphical designers want to produce effects that are best described as programs [8]. Operational procedures, often present in safety critical systems, are programs. Even user tasks are similar to programs, as illustrated by the conceptual similitude between CTT [20] operators and parallel programming languages control structures.

Programming algorithms, programming interaction

Overall, designing interactive systems and programming are much closer than usually advertised. We suggest that the traditional view of programming is biased. Turing and the generations that came after him have created such a consistent body of theories and programming languages that the theory of computation is used ubiquitously for analyzing systems, for designing algorithms, and even as a natural science [3]. This success sometimes obscures the existence (even the prevalence!) of other kinds of programs.

In this context, two courses of actions are possible for researchers. The first option is to design languages that help user interface designers assimilate concepts from the theory of computation. The second option, that we suggest is more promising, would be to acknowledge the difference and design adequate languages and theories to support user interface designers. Then, over the years, reaching the ability to consider interactive programs, human procedures and tasks as manifestations of the same theoretical principles could lead to a more balanced situation, with two kinds of programs and two bodies of knowledge: algorithms, and interaction.

In the rest of this paper we focus on the second option: how can we design programming languages and theories for all actors of user interface design? We also abandon any distinction between "user interface programming" and "programming in general", because in today's computing industry most programmers are confronted to user interfaces. Therefore, we choose to tackle the following question: *how can we design theories, notations, languages and tools that support future programmers, those who will have various backgrounds and will all build interactive software?*

PROGRAMMING LANGUAGES ARE USER INTERFACES

Software programming is an act performed by a user through a machine. As such, it is like any other computer-supported activity and requires usable tools, i.e. tools that enable their users to accomplish a task with a minimum amount of resources and in a delightful way. A review and classification of the usability requirements expected of interactive development tools has been made in [16].

When thinking about supporting software programmers, integrated development environments (IDE), user interface management systems (UIMS) and user interface builders are the first tools that come to mind. But programming languages play a more central role because they ultimately condition how programmers think about programs. To analyze the role of programming languages, and more generally, of any notation, we use Norman's theory of action [22] and study three aspects of interaction with them: evaluation, conceptual model and execution.

Evaluation

Programming languages allow programmers to express programs through a notation. Whether textual or so-called "visual", notations employ various graphical "features": texts,

shapes, alignments, colors, arrows, etc, to encode information. The representation of a program is called "the code".

An implicit but important aspect of programming languages is that they must support the production of readable code, for oneself and for others [24]: "Programs must be written for people to read, and only incidentally for machines to execute [2]". In this regard, the readbility of code is like the readbility of any visual representation: the first step toward forming a mental model and acting.

The graphical appearance of the code has been shown to have an impact on understanding. For example, indentation length has been experimentally shown to have an impact on the comprehension of code: 2- and 4-space indentation makes readers better at understanding the code than 6-space indentation, for both novice and expert readers [18]. More surprisingly, Green et al. found that textual representations outperformed LabView's graphical representations for each and every subject [12].

Conceptual model

Interacting with a tool is more than just its look and feel. One of the essential aspects is the underlying conceptual model, that is an explanation, usually highly simplified, of how a system works [22]. For example, the conceptual model of a file system relies on the concepts of File and Directory and the related operations. It can be represented either as icons, or as lists of names in a command line user interface. Conceptual models are considered essential to usability by HCI specialists: a badly designed conceptual model is often at the root of poor usability.

Programming languages are no exception to this rule: they can be analyzed as a visual representation that reflects an underlying conceptual model. For example, LISP code with its parentheses is one possible representation of the underlying hierarchy of expressions. Another representation of the same program would be a graphical tree that shows the hierarchy in 2D.

A conceptual model shapes the way their users think about their problem at hand and the ways to solve it. In the case of programming languages, it must be usable enough to help programmers think about, design, write and read programs. The conceptual model of a programming language is often derived from a general theory. Consequently, it does not only support the production of code, but the analysis of programs and their environment, to the extent of what the theory can describe. For example, the "functional" conceptual model[15] is well-adapted to the description of computation, while the "reactive" conceptual model is well-adapted to interactive behaviors. This brings us back to the aforementioned general theory of interaction: to produce usable programming languages for interactive systems designers, theories that encompass the appropriate concerns must be available.

Execution

When programming, execution consists in writing code or modifying it. This involves actions such as creating entities and referring to existing entities. IDEs are often considered as instrumentation of these tasks, designed to make programming with a given language more usable. For example, refactoring tools in current IDEs such as Eclipse enable programmers using functional or object-oriented languages to efficiently modify the names of functions or object methods. But, prior to IDEs, the evolution of languages can also be considered as a process to offer better support for these actions. For example, method inheritance is a way to factor common code in a single place, thus facilitating the evolution of behavior in multiple parts of the code ("mass updating" [13]). Similarly, aspects offer programmers the possibility to express cross-cutting concerns in a single place.

Moreover, some of the properties associated to "good software" can be related to usability concerns. For example, the goal of modularity is related to action and interaction: it is supposed to facilitate the maintenance of code since with well-modularized software a modification of a component performed by a programmer requires minimal adaptation and rewriting on other components.

RESEARCH DIRECTIONS

Our three statements and their discussion can be translated in a long term goal: provide interactive systems engineers with usable languages and notations for designing, developing and analyzing systems, grounded in theories that they can apply to forecast their behavior. How can this be turned into practicable research directions? The state of the art in the extended field of user interface engineering, as well as the history of traditional computer science, provide many possibilities. We list a few here:

Eliciting functionality

Programming language designers have spent decades to identify the functions of programming languages that best support traditional programmers, alone or in groups. Which of these functions are relevant for interactive systems designers, what requirements are not covered and how can they be addressed? Some authors have started to address this question [8, 21, 16] but this is only a start.

Conceptual unification

A number of concepts have been proposed to describe the behavior of interactive systems. The design of programming languages, as much as the design of theories, traditionally requires that the relationships between concepts are defined. This usually involves the definition of primitive concepts from which other concepts are derived. Such unification has been attempted in the past [5, 14] and should be pursued, even if this implies reaching out to disciplines that are currently alien to our field, including philosophy. The work presented in [17] is an example in this direction.

Formal definition of concepts

Little effort is devoted in our domain to establish consensual definitions of concepts such as task, activity, event, component, interface, animation, etc. Actually, it would be difficult to negotiate any solid consensus without more formal

candidate definitions: it is easy to agree with different interpretations in mind. Progress toward more general theories could include collective work on formal definitions, using standard community tools such as workshops, incremental or controversial publications, and consensus statements. For instance, the word "specification" has grown a particular meaning in software engineering, sometimes very different from its meanings in other engineering fields. Must we adhere to this idiomatic meaning and why? An article written in French by one of the authors [6] has sparked verbal debate about this, and it might be useful to have more structured debates, anchored in activity analysis of engineering.

Designing language concepts

If we consider that interactive systems programming is not well supported enough by existing programming languages, we should design new ones: programming languages for professional programmers and for interaction designers, notations for analysis, etc. There already are many research efforts in this direction. However, the goal of producing general theories that encompass all engineering activities requires that the concepts used by all languages be compatible, even if the notations are different. Our own research suggests that the conceptual models of traditional programming languages are not sufficient to fully describe interactions and that new, more comprehensive and unifying models of execution should be used instead. More research should be conducted to assess this aspect of the usability of programming languages, in order to identify the relevant properties and to design appropriate evaluation methods.

Designing language notations

We have already started to study the design of language notations. We notably have analyzed and modeled the process of perceiving a program using a framework based on the Semiotics of Graphics [9]. This work shows that code representation is not about aesthetics but performance, and should not be an art [11] but a science following principles from visual perception. It also suggests that there may be no substantial difference in terms of graphical perception between textual and visual languages. The Physics of Notations framework focuses on the properties of notations [19]. It adresses numerous aspects of graphical properties and defines several principles for the design of notations e.g., semiotic clarity, perceptual discriminability, semantic transparency, visual expressiveness, etc. In addition, designing a programming language should use a "programmer-centered design" approach: it should emphasize the act of designing representations targeted at tasks meaningful for end-programmers, and not designing the representation in isolation. Such work should be of interest for software engineers who often use various UML diagrams (another notation) to document their software.

Consolidating available results

A number of available results in user interface engineering have limited impact or are only used as general guidelines because they cannot be used directly by programmers and designers: architecture patterns, language constructs implemented in toolkits, dedicated algorithms, etc. A requirement for the design of new languages should be that these results can be checked against the proposed languages, so that they can be reused more directly. Reciprocally, effort should be spent on identifying available results and assessing against the proposed designs. For instance we have carried out an assessment of software adaptation against reactive programming [17] and it would now be useful to determine how architecture patterns proposed for plasticity translate in this framework. As another example, we are working on how the MDPC pattern [10] fits in an interaction-oriented language. Interestingly, some of the available results currently are implemented in operating systems [4, 7] and this is reminiscent of the relationships that existed in the past between new languages and new operating systems.

CONCLUSION

In this article, we have highlighted the role of programming in the engineering of interactive systems, in the long term perpective of creating general theories of interaction to support engineers. We have outlined a research roadmap aimed at both providing designers with appropriate programming languages and understanding the nature of interactive programs.

As researchers on engineering, one of our roles is to provide engineers with better theoretical tools, including languages and notations. As researchers on human-computer interaction, we have tools and methods that no other scientific community has for designing new theories, languages and notations. What about eating our own dog food?

REFERENCES

1. *Encyclopaedia Britannica*, vol. 8. 1972.
2. Abelson, H., and Sussman, G. J. *Structure and Interpretation of Computer Programs*, 2nd ed. MIT Press, Cambridge, MA, USA, 1996.
3. Arrighi, P., and Dowek, G. The physical Church-Turing thesis and the principles of quantum theory. *International Journal of Foundations of Computer Science 23*, 5 (2012).
4. Chapuis, O., and Roussel, N. Metisse is not a 3d desktop! In *Proceedings of the ACM UIST'05 conference* (2005), 13–22.
5. Chatty, S. Extending a graphical toolkit for two-handed interaction. In *Proceedings of the ACM UIST'94 conference* (Nov. 1994), 195–204.
6. Chatty, S. Réconcilier conception d'interfaces et conception logicielle : vers la conception orientée-systèmes. In *Proceedings of Ergo-IHM 2012* (2012).
7. Chatty, S., Boulabiar, M.-I., and Tissoires, B. L'évolution de linux vers les nouvelles formes d'ordinateurs personnels. In *Proceedings of the 6th International Conference on the Sciences of Electronics, Technologies of Information and Telecommunications (SETIT 2012)* (2012).
8. Chatty, S., Sire, S., Vinot, J., Lecoanet, P., Mertz, C., and Lemort, A. Revisiting visual interface

programming: Creating GUI tools for designers and programmers. In *Proceedings of UIST'04*, Addison-Wesley (Oct. 2004), 267–276.

9. Conversy, S. Unifying textual and visual: a theoretical account of the visual perception of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Splash Onward! '14, ACM Press (New York, NY, USA, apr 2014), (to be published).
10. Conversy, S., Barboni, E., Navarre, D., and Palanque, P. Improving modularity of interactive software with the MDPC architecture. In *Proceedings of EIS (Engineering Interactive Systems) conference 2007, joint HCSE 2007, EHCI 2007 and DSVIS 2007 conferences, Lecture Notes in Computer Science*, Springer Verlag (March 2007), 321–338.
11. Green, R., and Ledgard, H. Coding guidelines: Finding the art in the science. *Commun. ACM 54*, 12 (Dec. 2011), 57–63.
12. Green, T., and Petre, M. When visual programs are harder to read than textual programs. In *Proc. of the 6th European Conference on Cognitive Ergonomics (ECCE 6)* (1992), 167–180.
13. Green, T. R. G., Borning, A., O'Shea, T., Minoughan, M., and Smith, R. B. The Stripetalk papers: Understandability as a language design issue in object-oriented programming systems. In *Prototype-Based Programming: Concepts, Languages and Applications*. Springer, 1999, 47–62.
14. Jacob, R., Deligiannidis, L., and Morrison, S. A software model and specification language for non-WIMP user interfaces. *ACM Transactions on Computer-Human Interaction 6*, 1 (1999), 1–46.
15. Landin, P. J. The next 700 programming languages. *Commun. ACM 9*, 3 (Mar. 1966), 157–166.
16. Letondal, C., Chatty, S., Phillips, G., André, F., and Conversy, S. Usability requirements for interaction-oriented development tools. In *Proceedings of the PPIG 2010 Workshop on the Psychology of Programming* (2010), 12–26.
17. Magnaudet, M., and Chatty, S. What should adaptation mean to interactive software programmers? In *Proceedings of the ACM EICS 2014 conference* (2014).
18. Miara, R. J., Musselman, J. A., Navarro, J. A., and Shneiderman, B. Program indentation and comprehensibility. *Commun. ACM 26*, 11 (Nov. 1983), 861–867.
19. Moody, D. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Softw. Eng. 35*, 6 (Nov. 2009), 756–779.
20. Mori, G., Paternò, F., and Santoro, C. CTTE: Support for developing and analysing task models for interactive system design. *IEEE Transactions on Software Engineering 28*, 2 (2002), 797–813.
21. Myers, B., Park, S. Y., Nakano, Y., Mueller, G., and Ko, A. How designers design and program interactive behaviors. In *Proceedings of IEEE VLHCC'08*, IEEE Computer Society (2008), 177–184.
22. Norman, D. A. *The Design of Everyday Things*, revised and expanded edition ed. Basic Books, New York, 2013.
23. Palanque, P., Bastide, R., and Paternò, F. Formal specification as a tool for objective assessment of safety critical interactive systems. In *Proceedings of the Interact'97 conference* (1997).
24. Raymond, D. R. Reading source code. In *Proceedings of the 1991 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '91, IBM Press (1991), 3–16.
25. Reas, C., and Fry, B. *Processing - A Programming Handbook for Visual Designers and Artists*. MIT Press, 2007.
26. Rushkoff, D. *Program or Be Programmed: Ten Commands for a Digital Age*. Soft Skull Press, 2011.