Decomposing Interactive Systems

Philip Tchernavskij
LRI, Univ. Paris-Sud, CNRS,
Inria, Université Paris-Saclay
F-91400 Orsay, France
philip.tchernavskij@lri.fr

ABSTRACT
I argue that systems-oriented HCI should explore software engineering principles and architectures that emphasize user interaction over designer control. Many researchers have argued that user-empowering interaction should decouple tools from the objects they act on. Implementing this decoupling requires actively subverting the traditional architectures of interactive systems, including the encapsulation of interactive systems into closed applications, and the overly coupled event-driven programming model. I present a sketch of an architecture where *interaction instruments* are a first-class object to address these issues.

ACM Classification Keywords
H.5.2. Information Interfaces and Presentation (e.g. HCI): User-centered design

Author Keywords
Toolkits; Interaction paradigms; Software architecture;

INTRODUCTION

Interactive systems, which nowadays are primarily desktop, mobile, and web applications, are notoriously inflexible: they encapsulate a fixed user interface to manipulate a predefined type of data, with little user control over the configuration and capabilities of the software. Beaudouin-Lafon argues that *"the only way to significantly improve user interfaces is to shift the research focus from designing interfaces to designing interaction."* [2] He outlines several challenges for moving towards novel interactive systems in HCI research, among them developing novel *interaction architectures*, which support interaction at the tool and middleware level: *"Interactive systems are by definition open: they interact with the user (or users) and often with other programs. They must therefore adapt to various contexts of use, both on the users side and on the computer side. [. . .] I believe it is critical that we define interaction architectures that give more control to end users, that are more resistant to changes in the environment, and that scale well. I call these three properties reinterpretability, resilience and scalability."* [2]

Consider a user writing a document, who decides she wants to add a figure. She may have several applications with sophisticated illustration tools, but none of them allows her to just draw the figure directly on the "paper" of the document. If she is writing a math report, she can write formulae in her word processor, but she cannot ask it to evaluate them. By contrast, when interacting with the physical world, people spontaneously extend their capability to manipulate particular objects by adding tools, and use tools and objects in ways that they were not necessarily designed for. Can we achieve such flexibility in software systems? Can we decompose interactive systems into components such that users can compose them in ways that correspond to their idiosyncratic needs?

Allowing users to actively de-compose and re-compose systems is a way of letting them do more with less. Indeed, this would let users: replace basic tools that exist in many variations across a system with the one they prefer, e.g., they can choose their preferred way of picking and applying colors rather than the one imposed by each application they use; select and combine parts of different systems coming from different vendors to support their particular workflow; and adapt tools to contexts they were not designed for, e.g., use a statistical graphing tool to create drawings.

In software architecture, flexibility is the quality of being able to change a system by adding, rather than modifying parts [6, p. 35]. Gjerlufsen et al. distinguish between flexibility at *design-time* and *runtime* [10]. Whereas design-time flexibility is advantageous to engineers who will need to reuse and extend a system architecture, runtime flexibility can allow users to extend the capabilities of a system in use. Therefore, to create reinterpretable systems we should develop toolkits that shift flexibility towards users rather than towards designers and developers.

In this paper, I critique asepcts of common interaction architectures, and sketch a critical alternative. I argue that the application model of software and event-driven programming create static systems where user-facing flexibility is exceptional. I describe an architecture based on *interaction instruments* that users can freely appropriate and combine according to their needs.

CRITIQUING INTERACTIVE SOFTWARE

Today, most of our interactions with the digital world are mediated by *applications* (*apps* for short). Apps make for static, closed systems, where there is typically one user, one device, a prescribed set of tools, and one or more digital artifacts, such as a document. Apps are isolated from the environment in which they are used. Their internals are encapsulated by a strict interface for input and output. To work on a document stored in a file, an app has to load the file and create an internal representation that it can change. This encapsulation strictly limits how apps can be combined. Apps can be sequenced, i.e. a file output by one app can be loaded by another (if the formats are compatible), but they cannot concurrently work with the same file. On the output side, apps each have their own window, so their content cannot be mixed or exchanged, except through copy-paste — which duplicates, rather than shares, content. Some apps share content through a remote database, but then bear the burden of maintaining consistency between the database and their internal state. This is more akin to a distributed app than an open environment.

As apps couple what you can do (commands) with what you can do it to (content), they implement *procedures* rather than tools and materials. Procedures are idealized descriptions of how work is done. In real life, the boundaries between different types of work are porous, and people constantly add tools, materials, and collaborators to expand their capabilities. Apps are too rigid and monolithic for flexible interactive systems.

Gat argues that apps accumulate complexity *because* they are closed systems [9]. Since each app defines all the available tools in its particular domain, vendors end up competing on having the most features. According to Gat, the app model inevitably leads to large systems that function poorly. In practice, since it is impossible to meet the needs of every member of some particular community of practice, apps ends up being designed from a one-size-fits-all approach.

At the programming language level, the most common model for defining interactions is *event-driven programming*. Event-driven programming chains together statement of the form "When this input event happens on this graphical object, do that". In other words, event-driven interactions are programmed by creating design-time bindings between concrete user actions and concrete commands. This programming model creates strong coupling between tools and their targets by binding objects and input methods to particular interactions in program code. It also creates interactions that are opaque to users, because they have no knowledge of which bindings are in effect at any point in time.

Several programming models have been developed as alternatives to event-driven programming, such as functional reactive programming [8, 7] and hierarchical state machines [4]. These models attempt to improve on the limitations of event-driven programming for maintainable and flexible code, but do not address the user-facing flexibility of tools.

The app model is the result of common architectures and software engineering principles that have good properties for engineers and developers, *but not for users*. From an engineering perspective, the app model is very reasonable: Encapsulation means that each application can be developed with the assumption that it exists in a vacuum. It creates less opportunities for users to cause errors, and allows designers to keep a lot of control over how their software is used.

The limitations of the app model cannot be addressed at the level of individual systems. Rather, we should investigate and demonstrate alternative programming models and architectures that embed qualities such as reinterpretability, resilience, and scalability.

EXAMPLES OF RELEVANT SYSTEMS AND TOOLKITS

There are many cases of research advocating for and investigating user-facing flexibility. For example, Meyrowitz [14] critiqued the monolithic aspect of hypertext systems at the time, and Gat [9] argued for abolishing the division between users and programmers. The early Butttons system [13] demonstrated how to create and exchange small interactive components; Wulf et al. [17] created an architecture supporting the notion of "casual programmer"; Newman et al. [15] describe a system that encourages opportunistic uses of resources discovered in a ubicomp environment; Shared Substance [10] provides a flexible environment for multi-surface interaction.

I emphasize the next two examples as they are particularly relevant to the approach descibed in the next section: Olsen gives an analysis of how the Unix operating system model, where "everything is a file", allows users to compose and extend (terminal-based) tools flexibly: *"In the UNIX environment all commands are expected to read ASCII text from standard input and write ASCII to standard output. By unifying everything around ASCII text it was possible to build a wide range of pluggable tools that would store, extract, search, edit, and transform text. Because programs output readable text, users could readily see how some other program could manipulate such output for purposes not considered by the creators. This recognition of potential new uses in information, coupled with standard tools for text manipulation, is very powerful."* [16] This principle of designing systems from small composable parts is echoed in both functional and object-oriented programming, but is rarely present in graphical user interfaces.

Webstrates [12] is an example of an interaction architecture that shifts flexibility to users. It is a web server that supports real-time sharing of a large class of HTML documents: any change made to the Document Object Model (DOM) of a page loaded in a web browser is sent to the server, stored, and broadcast to the other browsers that have loaded that page. *Webstrates* turns the web into a medium where sharing and transclusion, the ability to include one document within another, are basic properties of the document model. Klokmose et al. show that this shareable medium can serve as a building block to create multi-user, multi-device systems that are extensible and reconfigurable at run-time.

Webstrates demonstrates that a toolkit can use existing infrastructure to create a novel model of interactive software. In the same vein, I argue that an interaction architecture where reinterpretable tools are first-class objects can be created economically by relying on existing platforms, e.g., the web.

A SKETCH OF REINTERPRETABLE TOOLS

The *instrumental interaction* model describes *instruments* as the primary means of interacting with the digital world: *"An interaction instrument is a mediator or two-way transducer between the user and domain objects. The user acts on the instrument, which transforms the user's actions into commands affecting relevant target domain objects. Instruments have reactions enabling users to control their actions on the instrument, and provide feedback as the command is carried out on target objects"* [1].

Interaction instruments are a good starting point for reinterpretable tools, because they are explicit objects, conceptually independent from apps, as opposed to the rules used by event-driven programming to define the behavior of each domain object when clicked on, dragged, etc. The physical tool metaphor also has the advantage that it is clear to users which instruments are available and active at any given moment.

Instrumental interaction is a descriptive and generative model that has been applied to design systems with novel interfaces, such as a bimanual colored Petri-nets editor [3] or digital curation on a tabletop [5]. In a *toolkit* for interaction instruments, several questions for the entities and processes around instruments occur:

- How is an instrument described?
- How are instruments decoupled from both the devices used to manipulate them and the target objects they operate on?
- What are the user actions and commands that instruments transduce?

Describing instruments

Instrumental interaction covers both physical and logical devices. The latter consist of a graphical representation to show their state and represent feedback, input channels to receive user actions, and a logic for mapping actions on the instruments to operations on the instrument's target. There are multiple abstractions that could implement this logical component, such hierarchical state machines (HSM's) [4, 11]. HSM's can be described as simple, isolated systems, which can be composed to create more complex behaviors. Importantly, instrument are defined independently of their concrete input mechanisms and output targets.

The instrument chain

Instruments can be chained, e.g., a pen instrument can be operated with a mouse or with a stylus. An action performed through a chain of instruments might look like this: "Alice clicks and drags the mouse to move the cursor instrument, which operates the paintbrush instrument, which leaves a red trail on the canvas."

The instrument chain, from physical action to final result, is continuously established, broken and re-created through use. In real life, we grasp tools and assemble them, e.g., combine a pen and a ruler to draw a straight line. In software, we implement the grasping metaphor with simple actions, e.g., clicking to select. A toolkit for instrumental interaction should have a richer set of elementary gestures and simple rules to combine instruments into a chain. In particular, a low-level collision-detection routine would determine when objects overlap to establish (and break) the instrument chain: The dragging instrument would activate the paintbrush when clicked on top of it, and the paintbrush would determine that it is over the canvas and lay ink on it.

User actions as signals

Once the chain of instruments is established, they can exchange actions and reactions. In event-driven programming, events travel from manipulated objects to observers. This is inappropriate for instruments, because instruments should not limit what type of event can be applied to them. Functional reactive programming extends the notion of events to signals — time-varying streams of values. This seems more appropriate for instruments: interaction is represented as a signal travelling through the instrument chain.

If an instrument cannot distinguish between operating another instrument or manipulating a domain object, this implies that instrument input and output are isomorphic. At the end of the instrument chain, the result of an action is some combination of reading and writing to the target object's state. We therefore model instrument operation in terms of mutating state: Instruments send each other *operations*, which can be stated as sequences of insertions, deletions, and updates. Returning to the previous example, the dragging instrument changes the position of the paintbrush instrument, and the paintbrush adds brushstrokes to the canvas. Operations do not need to be interpreted, as opposed to events. This means that the set of possible operations is open, and can be extended by new instruments.

With support for instruments at the operating system level, interaction scenarios such as those described in the introduction become feasible. Under this architecture, applications could be replaced by packages of instruments that fit a coherent domain, e.g., word processing or illustration. Users are empowered to reuse, adapt, and combine instruments as they see fit: One user may reuse a text cursor that supports his most used editing commands for writing e-mails, filling out forms, and taking notes; another may adapt a pen instrument from a drawing suite to handwrite annotations when reviewing homework; and a third may combine a word processing suite and a math evaluation instrument to write math reports. The ability to chain instruments further supports combining and extending instrument behaviors, e.g., a pen instrument can be used for freehand drawing, but chaining the same instrument with a dragging instrument constrained by a geometric shape turns it into a pen for drawing shapes.

CONCLUSION

Architectures are not neutral in implementing interaction. The app model of software couples tools and the objects they manipulate at design-time. At a lower level, this coupling is reflected in the coupling between input and output in, e.g., event-driven programming. This interaction architecture translates into static systems that limit the ability of users to de-compose and re-compose interactive systems. Shifting flexibility towards users is an outstanding challenge for systems-oriented HCI.

Interactive systems should leverage our natural ability to reuse, adapt, and combine tools. The tool metaphor of instrumental interaction is a good starting point for treating interactions as first-class objects and making interactive systems more flexible. By providing architectural support for interaction instruments, I hope to demonstrate the power of this approach. The ability to compose instruments should not only benefit end-users, but would also motivate software producers to create systems that are small and composable, rather than monolithic and isolated [9].

ACKNOWLEDGMENTS

Thanks to Michel Beaudouin-Lafon for editing this paper. This work was partially funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme grant n° 695464 ONE: Unified Principles of Interaction.

REFERENCES

1. Michel Beaudouin-Lafon. 2000. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. (2000), 446–453. DOI: http://dx.doi.org/10.1145/332040.332473
2. Michel Beaudouin-Lafon. 2004. Designing Interaction, Not Interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '04)*. ACM, New York, NY, USA, 15–22. DOI: http://dx.doi.org/10.1145/989863.989865
3. Michel Beaudouin-Lafon and Henry Michael Lassen. 2000. The Architecture and Implementation of CPN2000, a post-WIMP Graphical Application. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST '00)*. ACM, New York, NY, USA, 181–190. DOI: http://dx.doi.org/10.1145/354401.354761
4. Renaud Blanch and Michel Beaudouin-Lafon. 2006. Programming Rich Interactions Using the Hierarchical State Machine Toolkit. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '06)*. ACM, New York, NY, USA, 51–58. DOI: http://dx.doi.org/10.1145/1133265.1133275
5. Frederik Brudy, Steven Houben, Nicolai Marquardt, and Yvonne Rogers. 2016. CurationSpace: Cross-Device Content Curation Using Instrumental Interaction. In *Proceedings of the 2016 ACM on Interactive Surfaces and Spaces (ISS '16)*. ACM, New York, NY, USA, 159–168. DOI:http://dx.doi.org/10.1145/2992154.2992175
6. Henrik Bærbak Christensen. 2010. *Flexible, reliable software: Using patterns and agile development*. Taylor and Francis(Chapman and Hall/CRC).
7. Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 411–422. DOI: http://dx.doi.org/10.1145/2491956.2462161
8. Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 263–273. DOI: http://dx.doi.org/10.1145/258948.258973
9. Erann Gat. 2001. Programming Considered Harmful. *OOPSLA 2001 Feyerabend Workshop* (2001).
10. Tony Gjerlufsen, Clemens N. Klokmose, James Eagan, Clément Pillias, and Michel Beaudouin-Lafon. 2011. Shared Substance: Developing Flexible Multi-Surface Applications. (2011), 3383–3392. DOI: http://dx.doi.org/10.1145/1978942.1979446
11. Clemens N. Klokmose and Michel Beaudouin-Lafon. 2009. VIGO: Instrumental Interaction in Multi-Surface Environments. Association for Computing Machinery (ACM). DOI:http://dx.doi.org/10.1145/1518701.1518833
12. Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy E. Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. (2015), 280–290. DOI:http://dx.doi.org/10.1145/2807442.2807446
13. Allan MacLean, Kathleen Carter, Lennart Lövstrand, and Thomas Moran. 1990. User-tailorable Systems: Pressing the Issues with Buttons. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*. ACM, New York, NY, USA, 175–182. DOI: http://dx.doi.org/10.1145/97243.97271
14. Norman K. Meyrowitz. 1989. The Missing Link: Why We're All Doing Hypertext Wrong. In *The society of text: hypertext, hypermedia, and the social construction of information*. MIT Press, 107–114.
15. Mark W. Newman, Jana Z. Sedivy, Christine M. Neuwirth, W. Keith Edwards, Jason I. Hong, Shahram Izadi, Karen Marcelo, and Trevor F. Smith. 2002. Designing for Serendipity: Supporting End-user Configuration of Ubiquitous Computing Environments. In *Proceedings of the 4th Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques (DIS '02)*. ACM, New York, NY, USA, 147–156. DOI:http://dx.doi.org/10.1145/778712.778736
16. Dan R. Olsen, Jr. 1999. Interacting in Chaos. *interactions* 6, 5 (Sept. 1999), 42–54. DOI: http://dx.doi.org/10.1145/312683.312720
17. Volker Wulf, Volkmar Pipek, and Markus Won. 2008. Component-based tailorability: Enabling highly flexible software applications. *International Journal of Human-Computer Studies* 66, 1 (2008), 1–22.