What do Researchers Need when Implementing Novel Interaction Techniques?

Thibault Raffaillac, Stéphane Huot

What do Researchers Need when Implementing Novel Interaction Techniques?

THIBAULT RAFFAILLAC, École Centrale de Lyon - UMR 5205 LIRIS, France
STÉPHANE HUOT, Univ. Lille, Inria, CNRS & Centrale Lille - UMR 9189 CRIStAL, France

Interaction frameworks are the tools of choice for researchers and UI designers when prototyping new and original interaction techniques. But with little knowledge about actual needs, these frameworks provide incomplete support that restricts, slows down or even prevents the exploration of new ideas. In this context, researchers resort to hacking methods, creating code that lacks robustness beyond experiments, combining libraries of different levels and paradigms, and eventually limiting the dissemination and reproducibility of their work. To better understand this problem, we interviewed 9 HCI researchers and conducted an online survey. From the results we give an overview of the criteria for choosing frameworks, the problems often met with them, and the "tricks" used as solutions. Then we propose three design principles to better support prototyping for research in UI frameworks: (i) *duplicate* singular elements (e.g. mouse, caret) to foster opportunities for extensions, (ii) *accumulate* rather than replace to keep a history of changes, and (iii) *defer* the execution of predefined behaviors to enable their monitoring and replacement.

CCS Concepts: • **Human-centered computing** → **User interface toolkits**; **User interface programming**; *Empirical studies in HCI*; • **Software and its engineering** → *Software prototyping*.

Additional Key Words and Phrases: interaction frameworks, toolkits, prototyping, interaction techniques, hacking, interviews, online survey, design principles

ACM Reference Format:
Thibault Raffaillac and Stéphane Huot. 2022. What do Researchers Need when Implementing Novel Interaction Techniques?. *Proc. ACM Hum.-Comput. Interact.* 6, EICS, Article 159 (June 2022), 30 pages. https://doi.org/10.1145/3532209

1 INTRODUCTION

Programming user interfaces is generally bound to the use of interaction frameworks. These large software libraries provide *all-in-one* programming experiences, combining all the features a programmer would need – input events, layout managers, graphical rendering, template widgets, networking, logging, packaging, etc. Frameworks such as Qt, Android or JavaFX are so popular among developers, that they are also used for prototyping original interaction techniques, in the context of academic and industrial research.

While exploring and prototyping, researchers and UI designers consider new and unexpected scenarios for which frameworks were not necessarily designed. For example, to change the pointer behaviour on screen depending on user impairements [37], pointing context [3] or current task [40], one needs direct control over the cursor. This is usually feasible inside of a single application window, but extending it to the entire desktop and existing applications requires fiddling with private APIs at the operating system level, with is often exceedingly hard or straight impossible. Systems are prone to favor robustness over extensibility, and researchers often have to tweak and hack them to implement functional prototypes of their ideas. Given the active research on toolkits and architectures as alternatives to existing frameworks [24], the recurrent publications of methods

Authors' addresses: Thibault Raffaillac, thibault.raffaillac@ec-lyon.fr, École Centrale de Lyon - UMR 5205 LIRIS, Lyon, France; Stéphane Huot, stephane.huot@inria.fr, Univ. Lille, Inria, CNRS & Centrale Lille - UMR 9189 CRIStAL, Lille, France.

to circumvent frameworks and operating systems limitations [11, 13] and the occasional meetings of researchers to discuss usage and design of toolkits [33], we observe that a significant part of the HCI community is questioning available tools for implementing novel interaction techniques.

Existing research has focused mostly on understanding the activity of researchers when designing new interaction techniques, and providing software toolkits to accelerate research in specific domains. Practices such as opportunistic programming [5] and Web mashups [51] stressed the importance of documentation, reuse and debugging in supporting tools, and highlighted the enabling role of Web technologies in research. A number of studies targeted the collaboration between HCI researchers and designers [18, 28, 38], exposing an asymmetry of programming knowledge and showing the need for different levels of programming complexity. A few studies focused on the programming needs of researchers [2, 9, 27], and highlighted a great variety of needs that are best met by large software frameworks. However, few to our knowledge have specifically tried to uncover the problems experienced while using frameworks, in the context of research on new interaction techniques. As a result, despite a good understanding of the methods and frameworks used by researchers, framework designers have been given little insight in which design choices can best support research.

Our research question is: **What do researchers need when prototyping new interaction techniques, and how can we design or adapt existing frameworks and toolkits to support them?** We investigate this question by interviewing 9 researchers about their use of software tools to implement original interaction techniques. We analyze their choices for tools, their difficulties with them, and their strategies for bypassing limitations. From an online follow-up survey with 32 participants, we assess the relative prevalence of items in our classifications, and summarize our observations with five key insights. We then compare our observations with previous studies of researchers' needs. Finally, we rely on our studies and observations to propose three principles for the evolution of existing frameworks and the design of new tools.

2 INTERACTION FRAMEWORKS AND TECHNIQUES

What is an interaction framework? An interaction framework in general refers to a software library that facilitates the creation of interactive applications and user interfaces, by describing them as data structures (e.g. a scene tree) in addition to code. It provides reusable elements (e.g. widgets) that let one assemble new interfaces with minimal efforts. In comparison with other interaction libraries, a framework always "captures" the flow of execution with its own infinite loop. It executes application code through callback functions, a pattern called *Inversion of Control* [15, 20]. As a corollary to controlling the flow of execution, a framework can hardly be used in conjunction with other frameworks or versions of itself, and usually interacts with other libraries through extensions (e.g. with plugins). Finally, a framework contains *more than the sum of its parts*, binding many tools and functions with a common paradigm that facilitates the understanding of the whole application.

What is a toolkit? Our paper mostly discusses interaction frameworks because our study participants used them extensively. However, the HCI literature has been mostly concerned with *interaction toolkits*, i.e. software libraries that (i) have a smaller scope, (ii) have a smaller team or code base, and (iii) are developed by researchers for researchers. We rely on the definition by Ledo et al. [24] extending Greenberg's definition [16] as *generative platforms designed to create new interactive artifacts, provide easy access to complex algorithms, enable fast prototyping of software and hardware interfaces, and/or enable creative exploration of design spaces*. Note that with the above definitions, a toolkit may also be a framework (e.g. djnn [30], Amulet [39]). In this paper we use the term *interaction library* to refer to both frameworks and toolkits.

What is an interaction technique? Tucker [48] defines an interaction technique as: *the fusion of input and output, consisting of all software and hardware elements, that provides a way for the*

user to accomplish a task. The most common and studied examples of interaction techniques are those relying on a mouse, a keyboard and a computer screen. For example, clicking in a text field and typing some text is an interaction technique, as is right-clicking on an element and selecting a command in a menu. However, interaction techniques are not limited to this usual and standard context, and include e.g. gestural and body-based input, haptic feedback, and electrical sensors/lights.

What is the problem? Nowadays, user interfaces seem to have reached a point of stability. All major desktop (Windows, macOS, Linux) and mobile (Android, iOS) environments share interaction mechanisms inherited from the WIMP model, that make it easy for users to switch between systems. These interfaces have evolved slowly over time, often refining a mainstream user experience that is well known and accepted. However, as new input/output technologies emerged, the ecosystem of tools has made it hard to explore unconventional interfaces and to fully support them. Most touch and gaze-based inputs, for example, are still mapped to pointers like a mouse, often ignoring characteristics such as finger area/pressure, or eye saccades. As for outputs, VR environments are still dominated by flat 2D interfaces inside 3D environments, where the hand or motion controller acts as a pointer. Interaction frameworks are very popular development choices for researchers, yet seem inadequate for the purposes of unconventional interaction methods. And although researchers have proposed toolkits to facilitate their exploration, they have been seldom used in practice [7]. It is thus important to explain why frameworks and toolkits are insufficient, and how to design them to better support the implementation and dissemination of original interaction methods.

3 INTERVIEWS OF RESEARCHERS

Our research method to address these questions was to start by inquiring researchers about how they implemented new interaction techniques. We conducted *in situ* interviews, to examine past projects and identify the pain points in their workflows. During our studies we considered only the *implementation* of techniques, excluding their ideation and design that have received more attention in the literature (see the Related work section).

3.1 Study design

Due to the exploratory nature of the study, we conducted semi-structured interviews — a predefined interview plan and freedom to deviate on any interesting point. Each interview would cover a few past projects of a participant, focusing on the overall lifetime of each project. We relied on the Critical Incident Technique to help participants recall the context of their work, and remember the most striking problems [46]. The interview plan itself was designed to cover a full typical development cycle (refer to the appendix for the full list of questions):

- initial choice of development libraries
- nature and number of code (re)writes
- time until the first prototype, and its features
- initial ambitions versus actual result
- perception of what is “hacking” and “low-level”
- perception of code cleanliness
- resources for learning the software libraries
- sharing the resulting code to a community

3.2 Selection of participants

We selected 9 participants in the HCI domain, who were developing or had developed interaction techniques in a research context (i.e. with a goal of academic contribution). As a selection criterion

we asked potential participants if they had felt limited by their programming tools. This criterion was not strictly mandatory, but eased the choice among candidates as we expected them to express needs. We recruited participants locally from our research group, with priority given to those with the most experience. This locality may have created a "proximity bias", where participants would exagerate certain issues to satisfy us. However, the majority of the participants being senior researchers in HCI, they are familiar with this problem, and we consider that they remained objective so as not to bias the study. Of all participants, 6 were researchers, one engineer, one PhD student and one Master student, with an average programming experience of 14 years.

3.3 Setting

Each interview was conducted with a single interviewer and a single participant. They took place in the participant's office whether unoccupied, or a quiet room in the faculty otherwise. During each interview, we asked the participant to use their office computer (laptop or desktop), to show tools and code when appropriate. We chose to review past projects from participants, to avoid biasing the study towards the early stages of projects. They were selected by asking for projects where interviewees felt they had been "hacking", or were limited by their tools. Problems often spanned multiple projects and would sometimes be mentioned from projects outside the focus of the interview. We would move to another project when participants started repeating the same problems, and covered 2 to 4 projects during each interview.

3.4 Data collection

All interviews were recorded, using a smartphone microphone application. Each interview lasted more than one hour on average, for a total of 9.6 hours of audio files. They were full transcribed, to facilitate analysis and to allow tracing back the origin of each observation. Each transcription had to be done manually due to the lack of adequate transcription tools. Indeed, the recorded voices lacked clarity and were affected by background noises, while some participants spoke quickly and chopped their sentences. Notes taken on paper during each interview helped us remove ambiguities when the audio was difficult to interpret.

Our method for data analysis is based on Thematic Analysis [36], and synthesized in figure 1. We focused on the use of programming tools to prototype new interaction techniques. In particular, we sought to identify the kinds of difficulties participants had experienced, and why they had problems.

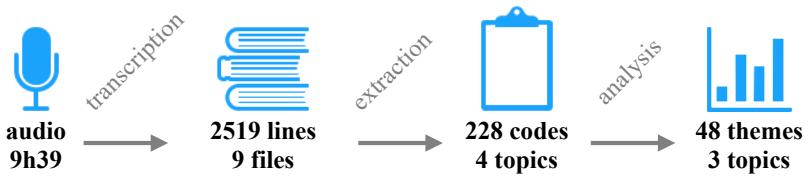


Fig. 1. Our four-staged pipeline to analyze interviews

3.5 Extraction

During the extraction phase we parsed the transcriptions and extracted *codes* on different topics, each stored in a separate text file. A code here is an observation taken verbatim from the transcription, either a direct quote if it can convey a statement by itself, or a summary if too long. Codes repeated many times by a participant for one project were always gathered as a single one. The

pilot participant 1 (P1 in short) is not counted in our study, the audio file was not recorded properly and was subsequently lost.

Our first topic was **problems**, which we detected by looking for perceived limitations, or what participants described they could not do. For example, P10 described: "*The interaction mode works, except that I did not have control over the scrolling speed*". We also extracted **needs** by looking at explicit requests, and to see if they would differ from problems. For example, P4 suggested: "*there would be a thing where the system knows for each method its percentage of use by developers in general*". While we kept record of the interaction libraries used by our participants, they would sometimes explicitly mention tools that had been key to their work, so we extracted **utilities** too. We noted libraries like P2 "*accessibility API to intercept command activations*", but also design patterns like P6 "*And then we also used decorator [pattern], it wasn't there, I added it.*"

Later during the extraction phase, we realized participants had often shared how they solved their problems. With each problem, they would describe how they came up with inventive ways to bypass a limitation, to unlock a hidden feature, or to make the library work the way they intended. While all participants were users (not developers) of interaction libraries, their use of software tools was often pragmatic and not necessarily dictated by recommended practice. They found "shortcuts" to solve problems, that would require the least time and efforts to obtain proper results. As we noted similar kinds of solutions between participants, we figured they could be classified into **strategies**: generic approaches to tackle certain types of problems. We looked for actions that were contrary to recommended and documented practices, particularly in response to a problem. For example, P2 mentioned: "*Hijacking an application by injecting code to instrument certain function calls*".

3.6 Analysis

After the extraction phase, we went from 2519 lines of transcriptions in 9 files to 228 codes in 4 files. At this level, the observations were factual and needed to be clustered so that we could reflect on them. We therefore looked for recurrent patterns between codes.

To analyze problems, we devised a first level of categories from six domains of work when engineering a software library: **choice of features**, **design of the API**, **documentation**, **extension and maintenance**, **bug fixing**, and **internal architecture**. Within these categories, we gathered the codes into themes, forming the types of problems encountered by our participants (see Table 1).

While analyzing needs, it appeared that they were very often redundant with problems. For instance, for P2, the need "*Displaying tooltip-styled text next to many buttons simultaneously*" would correspond to problem "*The tooltip is a static instance shared by all buttons, and cannot be used multiple times at once*", although they were expressed separately during the interview. We chose to keep only problems, since we extracted more of them thanks to the Critical Incident Technique.

Utilities were analyzed with the same categories as problems in mind. Given we had few observations here, this classification is not intended to be exhaustive but is kept to complement the understanding of strategies (see Table 2).

To analyze strategies, we started by looking for recurrent action verbs (e.g. intercept, replicate, divert, combine). Then we refined them and grouped them into three categories: **obtaining data**, **interacting with other libraries/tools**, and **opportunistic use** see Table 3).

In the tables, the number of participants who mentioned a code matching each theme is indicated, as well as the total number of codes that correspond to it (the sum of matching codes from each participant). Numbers that are above the average for each column are emphasized in bold.

Lastly, in the list of libraries mentioned, we noted that none of the participants had used toolkits designed specifically for research, apart from their own toolkits. This is problematic since a great deal of research is invested in toolkits that enable new kinds of interfactive artefacts [24]. This observation, along with the lists of problems and strategies, reflected the experience of a local

Problems	Examples	participants	occurrences
Features			
Insufficiently deterministic/specified behavior	*P3: Sometimes mouse press/release events are received without their opposite*	**4**	**6**
Lack of useful functionality for prototyping	*P7: diff does not support displacements*	**4**	**5**
Wrong engineering choice hindering the work	*P3: Arduino has a software clock, making it sensitive to user code slowdowns*	**4**	**5**
Inadequacy between features and needs	*P6: Apple trackpads give relative rather than absolute displacement*	**3**	3
API			
Functions and data hidden/forbidden from the programmer	*P2: Hot corner events (macOS) are not exposed to applications*	**5**	**6**
Insufficiently controllable API	*P10: Impossibility to control scroll speed when dragging*	**3**	**6**
Inconsistent/unintuitive API	*P7: A long keypress should result in an event with duration rather than a sequence of events*	**3**	**4**
API too complex to use	*P4: Swing's GridBagLayout is too complex to be used*	1	3
Documentation			
Lack of context and examples	*P5: Lack of clear documentation to create a custom event for Qt*	**3**	**6**
Search requiring too much investment	*P5: Extending signals/slots to custom events requires a lot of work*	**3**	**5**
Significant behavior not documented	*P9: Vicon does not document all of its data formats in use*	**3**	**4**
Insufficient/passive documentation tools	*P4: From the name of a method, "ok, how do I do it?"*	2	3
Fragmented/partial documentation	*P4: Information is "fragmented", gathered over time from different sources*	2	2
Limits and inconsistencies			
Problem scaling up	*P2: The tooltip is a static instance shared by all buttons, and cannot be used multiple times at once*	**4**	**4**
Inconsistent behavior over time (versions)	*P8: Apple's drop of backward compatibility with the Carbon API*	2	**4**
Artificial limitation (enforced) by a library	*P2: The drawing of a button is clamped to its bounds*	2	2
Inconsistent functionality between systems	*P7: The use of scrolling in Java is difficult between Windows and Mac*	2	2
Bugs and slowdowns			
Unnecessarity/abnormally slow behavior	*P5: Using a dedicated thread to read a serial port slows Qt down*	**4**	**5**
Documented function giving a wrong result	*P4: Under Windows, the pixel density given by the System API is wrong*	2	2
Non-deterministic error/crash	*P5: The callbacks of Qt's network sockets do not work every time*	1	1
Instrumentation too invasive	*P5: For heavy interactive applications, gdb and Valgrind may slow down the program*	1	1
Internal details			
Unanticipated interference from an internal behavior	*P3: Arduino's preconfigured timers interfere with user timers*	2	3
Lack of storage of execution traces	*P2: A command activation does not know the series of activations that led to it*	1	1
Internal complexity hindering introspection	*P2: Two widgets may refer to the same command but with indirections*	1	1

Table 1. Different types of problems observed during the interviews.

group of researchers. In order to confront them with a broader sample, we conducted a second study in the form of an online survey.

Utilities	Examples	participants	occurrences
Extensible class or protocol from a framework	*P9: The OSC protocol, to add custom events to an existing stream*	**6**	**8**
Design/engineering pattern	*P8: Naming conventions, to save time to converge on one's own rules*	**5**	**6**
Active or well-designed documentation	*P4: Completion of method names to help discover an API*	3	4
Tool to access private data	*P2: Accessibility API to intercept command activations*	3	3
Useful implicit behavior	*P2: Automatic animation between states in Cocoa's CALayer*	3	3
Debugging/visualization tool	*P3: printf to debug a text-based custom communication protocol*	1	1

Table 2. Different types of useful artifacts noted during the interviews.

Strategies	Examples	participants	occurrences
Obtaining data			
Reconstructing data from prior/raw states	*P4: Decoding the HID descriptor manually*	**5**	**7**
Introspecting an existing object/application	*P2: Injecting code to instrument command execution and determine their triggerers*	**5**	**5**
Accessing private or unexposed data/functions	*P3: Writing assembler code to access alternate timer modes on Arduino*	**5**	**5**
Retrieving information from different sources	*P4: Getting low-level data from different redundant System APIs*	1	1
Interacting with other libraries/tools			
Reimplementing an existing widget/mechanism	*P5: Drawing one's own widgets to be able to make "weird" shapes*	**7**	**9**
Reverse engineering (extracting knowledge) of a "black box"	*P8: Estimation of the network MTU by adaptive reduction*	**5**	**8**
Augmenting/modifying an existing mechanism	*P7: Modifying the progress bar widget to refresh more often*	**4**	4
Modifying the environment of a tool to alter it	*P5: Rooting a tablet to activate the debugging mode for Android applications*	**4**	4
Inhibiting/replacing an existing behavior	*P6: Filtering System events to block and move the cursor manually*	3	3
Overlaying on an existing application	*P2: Using a Canvas overlay on the home screen of an Android tablet to intercept touches*	3	3
Extending using the dedicated tools of a framework	*P5: Creation of custom events for Qt, with new metadata*	1	1
Opportunistic use			
Misusing a tool/parameter out of its intended scope	*P2: Adding spaces in menu titles to unroll all of them without overlapping*	**4**	**5**
Complementing with a tool/data from a lower level	*P4: Retrieving raw screen information from the Windows Registry*	**4**	**5**
Introducing a new API paradigm atop a different one	*P3: Development of a generic debugging protocol for serial communication*	3	3
Replicating a fake version of an existing application	*P5: Reproduction of a dummy Google Maps application*	2	3
Duplication to bypass an upper bound	*P9: Networking 4 computers to a single event bus to manage 4 cursors in a Swing application*	2	3
Combining tools to perform a more complex task	*P9: Creating a 3D pointer by sticking Vicon markers on a mouse*	2	2

Table 3. Different types of strategies observed during the interviews.

4 ONLINE SURVEY WITH RESEARCHERS

Our second study was designed to be semi-quantitative, in order to estimate the relative importance of our themes for HCI researchers, and possibly discover points that we had missed in the interviews. We framed the survey around three research questions targeting the prototyping of interactive applications in the context of HCI research:

What are the most important criteria for choosing interaction libraries? (R1)
What are the most limiting implementation problems for researchers? (R2)
Which strategies are most used to circumvent and overcome these problems? (R3)

4.1 Study design

The survey was structured into four sections, one for preliminary questions and one per research question. The first section collected the professional situations of participants (profession, type of institution, number of collaborators), their expertise in programming interactive applications, and the proportion of their time dedicated to programming. These questions were intended to compare our participants to our target population. The other sections consisted in rating the relevance of different items for each of the research questions: importance of each criteria for choosing interaction libraries (R1), severity of each problem if ever encountered (R2), and frequency of usage of each kind of programming strategy (R3).

For R1 we chose the initial items based on our own experience in the field, then refined from the answers of pilot questionnaires with 4 participants. For R2 and R3 we selected the more frequent themes from tables 1 and 3 (shared by more than 1 participant), merged similar ones, and refined them after the pilot questionnaires. All items were evaluated on scales with 5 levels, the "neutral" option being always level 1. Problems (R2) had an additional sixth option to indicate a problem had never been experienced, to differentiate from a problem already encountered but with no severity. Free form text fields were present below each item in R1 and R3 to let participants eventually detail their answers. We included them as opportunities for missing observations, and they also allowed us to correct ambiguities after the pilot questionnaires. Finally, with R1, we asked participants to fill a text field with frameworks or toolkits they used the most.

One of our hypotheses was that participants would use general purpose frameworks more often than specialized toolkits. Hence we took care to adopt neutral formulations (e.g. "interaction libraries", "frameworks/toolkits") so as to let participants consider every type of software library that allowed them to program interactive applications. Another concern was the inability of our survey tool to shuffle items when complemented with text fields. For R1 and R3, the order of criteria and strategies presented to participants was always the same. For these cases we chose a meaningful transition order between items to reduce the memory effort of going through the questionnaire. The complete list of questions is provided in appendix.

4.2 Selection of participants

The target of our study was people who had been developing novel interactive artefacts in a research setting. While it is difficult to define what is a sufficient experience here, we recruited participants exclusively from the HCI community. The survey was first sent through the *chi-announcements@acm.org* mailing list, then to our colleagues' former teams.

The introduction to the questionnaire made a clear point to target researchers: " *The goal of this survey is to better understand the process of programming new interactive artefacts for research and innovation purposes, and the software libraries used to support this process. We are interested in projects where you reached the limits of libraries (e.g. undocumented needs, accessing private functionalities, interfacing with existing applications, combining with other libraries), for the prototyping*

and implementation of innovative interactive artefacts (e.g. non-standard interaction techniques, data visualisations, UI toolkits, interactive applications) ".

A total of 32 responses were collected over a 2-month period. 25 of the participants indicated researcher as their main activity, and 27 worked for a university, school or public institution. 2/3 of the participants considered themselves to be advanced or expert at writing interactive applications, and again 2/3 of the participants dedicated less than 40% of their time to programming. The profiles of our participants are therefore very relevant to the context of our study. Nevertheless, we are not able to state whether they are representative of the general proportions of users who prototype interactive applications in a research context.

4.3 Data analysis and results

The goal of the survey was to understand why researchers choose any library over another and where they struggle in practice, so that we could suggest improvements to future libraries. Thus we focused our analysis on the aspects of libraries which matter the most to researchers, as well as those which matter less. In the rest of this section we go through the data and extract a set of key observations.

4.3.1 R1 - Choices of interaction libraries. In the second section of the survey, we asked the participants which frameworks or toolkits they used the most. They could give several answers, separated with commas. The results are presented in figure 2a. Libraries are sorted by the number of participants who cited them, those with equal numbers of occurrences being sorted alphabetically. We excluded two answers which were clearly not interaction libraries (*Atom* and *Eclipse*).

Observation 1: *Researchers prioritize well established interaction frameworks over research toolkits.*

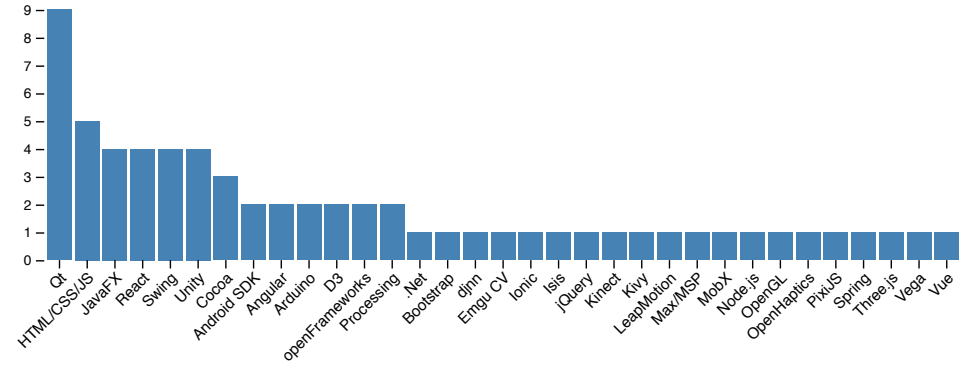
This is an obvious observation but has to be reminded. Out of all the citations, the vast majority mention frameworks (72.3%, counting the libraries that enforce Inversion of Control). Additionally, the average age of the libraries cited is 15 years, with a standard deviation of 7.6 years. This suggests that our participants tended to favor mature and extensive libraries.

As for research toolkits, we note that only 4 of the 33 libraries cited originate from academic research (D3 [4], djnn [30], Max/MSP [45] and Vega [47]), for a total of 7.6% of all citations. Moreover, only D3 is included in the list of 68 examples of HCI toolkits from Ledo et. al [24]. This observation should be tempered by the fact that D3 and Max/MSP are widely used in specific communities (Visualization and Computer music) which we did not target specifically. Nonetheless, the limited use of research toolkits is a problem if we want to support researchers' needs with the right tools. Considering the widespread use of "old" frameworks, it seems important to influence them so that they later support research — hence the importance of our analysis to devise and discuss appropriate design principles.

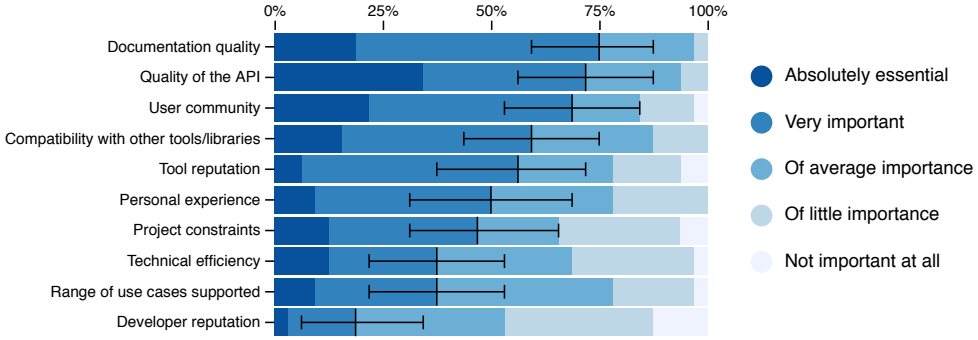
Observation 2: *The choice of a library is mostly based on its ease of use, and is directly controlled by its authors.*

The answers for rating the importance of different criteria are synthesized in figure 2b. We ranked them according to the number of participants who responded *Absolutely essential* or *Very important*, to identify those more important than the average. For each one, we computed a 95% confidence interval with the bootstrapping method [14]. It corresponds to the probability that a resampling with replacement from the same data gives a number of votes in the interval. When the lower bound of this interval is above 50% of the responses, we consider it *very likely* that more than half of the HCI researchers consider the criterion at least *Very important*.

Considering R1, this ranking suggests that *Documentation quality*, *API quality* and *User community* are likely to be among the most important criteria when choosing an interaction library for a



(a) List of software libraries used by our participants to program interactive artefacts in a research context.



(b) Evaluation of different criteria for choosing libraries, ranked by the number of *Absolutely essential* or *Very important* responses, with 95% confidence intervals.

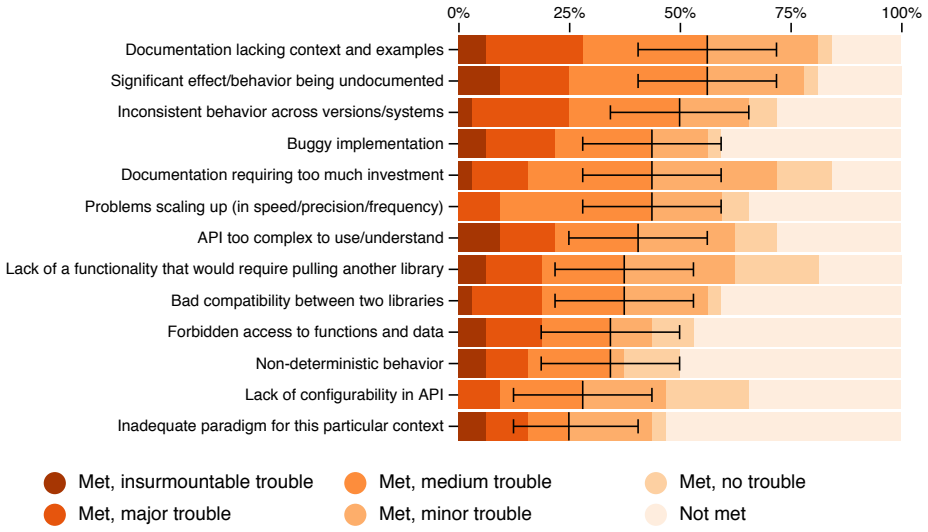
Fig. 2. Software libraries used by our participants and different criteria for choosing them.

research project. These criteria will help narrow down the range of acceptable tools initially, and then weight on the choices between alternatives. Library designers may influence these criteria with varying degrees of control, which we divided in three groups:

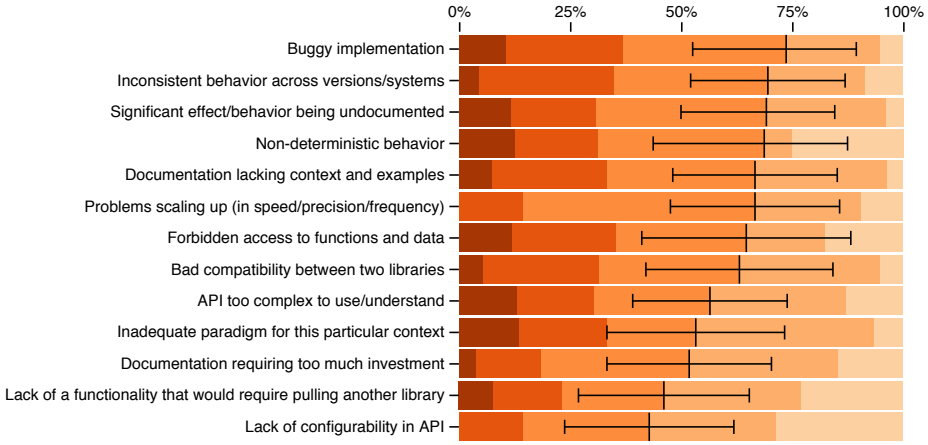
- direct control (*Documentation quality*, *API quality*, *Compatibility with other tools/libraries*, *Technical efficiency*, *Range of use cases supported*)
- indirect control (*User community*, *Tool reputation*, *Developer reputation*)
- no control (*Personal experience*, *Project constraints*)

For example, a framework or toolkit developer may not decide how big a *User community* is, but they may host events and animate forums to help it grow. Fortunately, the top criteria in our ranking have a high degree of control, so developers may effectively influence the dissemination of their work — albeit with a lot of work. To that end, our results suggest that most attention should be given to:

- writing documentation that is comprehensive and relevant to the context of HCI research (e.g. providing code samples for advanced interaction techniques, describing internal functions)
- designing an API that is tested and refined with examples of advanced interaction techniques (e.g. using multiple mice, patching into an existing application)



(a) Evaluation of different problems with interaction libraries, ranked by the number of responses at least *Met, medium trouble*, with 95% confidence intervals.



(b) Evaluation of the criticality of problems by the participants who already encountered them, ranked by the number of responses at least *Met, medium trouble*, with 95% confidence intervals.

Fig. 3. Assessment of problems encountered with interaction libraries, and their criticality.

4.3.2 R2 - Problems with interaction libraries. The answers for rating the criticality of different problems are synthesized in figure 3a. They are ranked according to the number of voters who answered at least *Met, medium trouble*, the threshold at which we consider the problems to have engaged a sufficient share of participants' time and energy. 95% confidence intervals are also represented on the graph.

Among the most important problems, those related to the documentation occupy the 1st, 2nd and 5th positions. This is consistent with the importance of documentation when choosing a library for a new project. The problems related to API occupy the 7th, 10th and 12th positions. This may

indicate that these types of problems were slightly overestimated during the interviews. In contrast, we notice that the 3rd, 4th and 6th positions (*Inconsistent behavior across versions/systems*, *Buggy implementation*, *Problems scaling up*) are problems which could not be foreseen before choosing a library, thus uncorrelated with R1.

In the evaluation of problems, we observed that many participants answered *Not met* (33% of all answers). Hence we plotted the same results in figure 3b by discarding *Not met* answers. This is arguably a more reliable estimate of problem criticality for HCI researchers. Indeed, if a group of participants from the same team responded to our questionnaire, they may have biased the results in figure 3a, in favor of the problems they encountered as a team. By estimating a relative criticality among people who actually met each problem, we limit this bias, at the cost of a lower number of responses for each problem. Note that the 95% confidence intervals are also recomputed by discarding *Not met* answers, hence their widths did not grow proportionally with the bars.

Observation 3: *Unpredictability is the most critical problem experienced by researchers with interaction libraries.*

Compared with the previous figure, the 1st, 2nd, 3rd and 4th problems now refer to buggy and unexpected behaviors. They point clearly at unpredictability being the most important source of problems in interaction libraries. That is, a framework or toolkit should always *document what it does and do what it documents.*

The most critical problems are those which take the biggest fraction of researchers' time and energy. They slow down researchers the most when programming, at the risk of compromising the final result versus the expected one. They are also the most frustrating and may deter choosing the same library for other projects. Reducing friction is therefore important to achieve more advanced research prototypes. To that end, our results suggest that most attention should be given to:

- writing documentation that covers edge cases brought by the context of HCI research
- using a tool for automated and fuzz testing, with research-inspired test scenarios and including internal API methods

4.3.3 R3 - Strategies with interaction libraries. In the last part of the questionnaire, we asked participants to rate the prevalence of different "coding techniques" (which we call strategies here) in their work. The results are presented in figure 4. They are ranked by the number of votes at least *Sometimes*, the threshold at which we consider that participants would be interested if a tool supported them.

Observation 4: *Strategies for gathering and processing interaction data are among the most frequent for our participants.*

These strategies correspond to positions 2, 3 and 4, namely: *Using accessible raw data to reconstruct/reinterpret a state that you do not have access to*, *Using an external mechanism to obtain and process data that is not exposed by an application* and *Aggregating multiple sources of interaction data (input, sensors, events), and fusing them into a single source.*

Interaction data may be input events fired by different devices, output buffers for writing audiovisual data, or even the internal state of certain widgets. This data is often hard to find and process, for various intentional or unintentional causes:

- it is internal and private to the interaction library
- it is exposed but cannot be modified (and writes may corrupt or even crash the application)
- it is ill-documented, explained in fragments of different pages or even undocumented
- it is provided in a specific format that should be converted before actual use

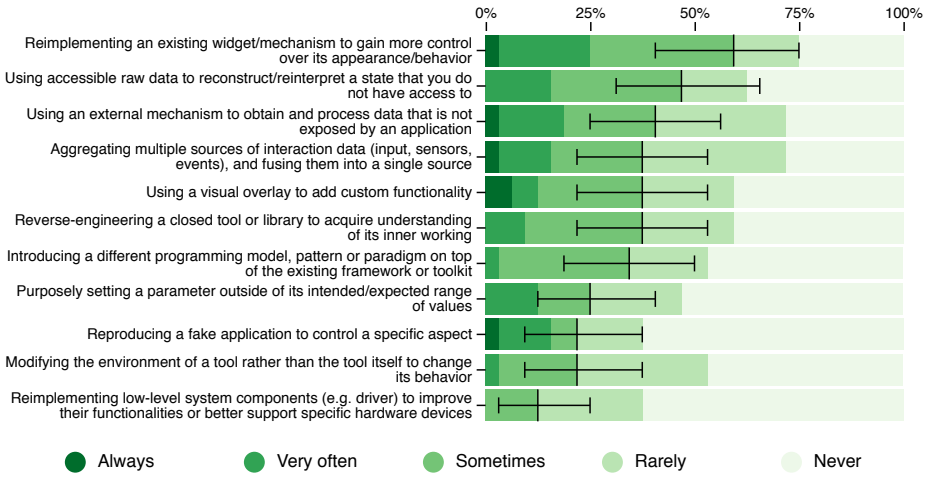


Fig. 4. Evaluation of usage frequency for different strategies, ranked by the number of responses at least *Sometimes*, with 95% confidence intervals.

Designers of frameworks and toolkits may find it challenging to disclose data that was previously protected, or change its flow or format inside the application. To that end, we suggest the use of software buses (e.g. Ivy [6]) and textual data stored in interoperable formats (e.g. JSON, XML, URL) to facilitate the gathering and processing of interaction data by researchers. We discuss these points further in the design principles at the end of this paper.

Observation 5: *Researchers will often implement new features from scratch rather than patch existing applications or widgets.*

This observation is based on strategies at positions 1, 5 and 9 in figure 4 (*Reimplementing an existing widget/mechanism to gain more control over its appearance/behavior*, *Using a visual overlay to add custom functionality* and *Reproducing a fake application to control a specific aspect*). In each of these cases, we consider that developers have the possibility to patch into an existing application or component. For example, to create an interaction technique for navigating maps, one may extend an existing map application like Google Maps or OpenStreetMap. However, these may not allow e.g. drawing custom shapes on top of the map, filtering the input events, or even distorting the view. Creating a fake application with a static fake map is arguably simpler, and will give developers a greater control over the effects they apply on inputs and outputs.

This observation is to oppose with strategies such as *Purposely setting a parameter outside of its intended/expected range of values* and *Modifying the environment of a tool rather than the tool itself to change its behavior* (positions 8 and 10), where new features are implemented by altering existing applications. They suggest that researchers will rather replace than reuse, not as a dogma but because in many cases it is simpler.

Unfortunately, the lack or absence of reuse means that the resulting research artefacts would not benefit from the extensive use and testing that made existing applications robust. It results in a gap from research prototypes to production-ready interaction techniques, which is not a desirable situation if we want to foster the dissemination of research. To that end, we suggest that library designers provide reuse mechanisms at finer levels. For example, instead of composing an application by assembling widgets, one should be able to assemble widgets from smaller parts, very much like composing elements with CSS properties.

In summary, we have provided observations and recommendations for three key aspects of interaction libraries: (i) disseminating them to new audiences, (ii) reducing their pain points on usage, and (iii) supporting exploration for research purposes. In the following section, we recall past articles that studied the needs of HCI researchers and what they suggested. Then we will present an original set of design principles, that complements previous recommendations and supports some of the most frequent strategies that we observed.

5 RELATED WORK

HCI research has been active for decades in exploring new ways of expressing interfaces and interactions (e.g., state machines, data flows, regular expressions), in order to promote innovative interaction techniques. However, mainstream frameworks have remained relatively impervious to these paradigms, as none of them emerged clearly. In this section, we review past works that studied the problems and needs of people who program interactive artifacts for research or innovation. We seek answers to the question **What are the recommended practices to design interaction frameworks and toolkits?**

5.1 Studies with designers

Among the different categories of developers, interaction and interface designers have been at the heart of many requirements studies. They are generally considered less proficient programmers although they need the same expressive power, which makes them extreme users for interaction libraries. Some works have thus focused on designers doing programming (alone or with developers), with the assumption that the contributions they make might be generalized to other contexts.

In a 2008 study of 259 designers, Myers et al. observed that 86% considered behaviors to be more difficult to prototype than appearance and that they needed more complex behaviors than predefined ones, forcing them to depend on developers 78% of the time [38]. They suggested the development of tools to explore more complex behaviors than predefined ones, to test multiple versions simultaneously, and to annotate behaviors with labels and descriptions.

In 2009, Grigoreanu et al. observed that designers' needs were not well known and their difficulties were poorly understood, with tools such as Adobe Dreamweaver, Adobe Flash, and Microsoft Expression Blend [18]. From a comprehensive study including a survey with 6 designers, the harvesting of mailing lists and forums, 10 formative interviews, 2 post-study interviews with design managers and 2 focus group sessions with 9 designers, they formed a set of 20 needs sorted by popularity. Their main observations were the needs to create, refine and communicate (i) the structure of the application and the *flow* of data inside it, and (ii) the interaction or *feel* of users within the application. From these they imagined a number of promising tools, and suggested using the 20 needs as a basis to propose more.

In a 2010 survey with 87 respondents, Carter and Hundhausen inquired researchers and professionals about the technologies used in HCI [7]. They observed a large variance among the weaknesses cited for tools, suggesting there was a likewise broad range of uses that should be acknowledged. Notably, they also mentioned that "*Only one respondent indicated using custom prototyping software developed by the HCI community*".

In the context of designer-developer collaborations, Ozenc et al. conducted 2 workshops in 2010 to observe how designers would conceive and communicate novel GUI controls [44]. They observed that static and annotated screens were insufficient to communicate a design concept, and recommended researching tools that would act like architectural blueprints, or objects at the boundary between designers and developers. We interpret this direction of research as promoting 2 levels of programming complexity, with designers being exposed to the "simpler" programming tools.

In the context of an industrial project of maritime surveillance, Letondal et al. studied the needs of designers related to the use of model-driven architectures [28]. They were interested in adapting and improving the models they used, and their own application engineering process in general. Using situational interviews and participatory design, they concluded that models were useful as a medium for communication between designers and engineers, although not used eventually to generate interfaces. Based on this reflection, they suggested a better sharing of representations, in order to improve collaboration between teams.

In 2019, Leiva et al. conducted 3 studies to gather the most common and critical problems arising during designer-developer collaborations [26]. An observation related to designers and tools was "*designers struggle to represent interactions and dynamic behaviors with current tools and use multiple design documents to communicate different aspects of their design*", and they also noted that developer involvement was crucial early to succeed at designing custom interaction techniques. They proceeded with 4 design principles for collaborative prototyping tools, one of which promotes an easier access to programming for designers (*support design by enaction*).

In summary, all articles acknowledged that programming is difficult but necessary for designers, and that the entry ticket should be lowered for them. However, most of the articles have focused on the design of novel GUI controls, giving a skewed sense of the programming of *interaction techniques*, and without proof that the recommendations could extrapolate to other kinds. While we observed that the range of problems and strategies is confusingly diverse, future work should address less stereotypical scenarios and perhaps suggest different recommendations for each.

5.2 Contributions from toolkits

In the history of toolkits designed for research, a few papers discussed their design choices or initial requirements. Here we review a handful of them, highlighting the diversity of forms that these requirements may take. We looked for explicit rationales or lists of requirements in papers, and tried to extract those that may apply to other libraries.

In XDStudio [41], Nebeling et al. discussed their requirements in the context of cross-device user interfaces: detecting devices connected at design time, designing for devices not available at the time, supporting different views for different user roles, switching seamlessly between design and testing, adapting widgets to device types, adapting the interaction on devices to tasks, reusing existing interfaces and their distribution to devices, detecting devices and users at run time, and adapting the design-time distribution of interfaces depending on the devices available.

In C4 [22], Kirton et al. detailed their lists of requirements for the development of a creative coding toolkit with mobile and multitouch applications in scope. The 6 design requirements were: turning media elements into first class objects of the language, facilitating the composition of elements of different natures (e.g. graphical shapes on video), extending direct interaction on media objects, using declarative animations, enabling rapid mobile development (mostly through templates), and being computationally efficient with lazy rendering. The 5 software requirements were: providing a simple architecture by reducing the number of classes, exposing properties rather than methods to control the states of elements, unifying methods that differ in input parameters, giving direct access to media objects rather than extending their features or reinventing them, and abiding by Objective-C's conventions and styles.

In D3 [4], Bostock et al. presented three key design rationales that are specific to visualization toolkits but may extrapolate to other contexts: (i) when a scene is generated from data, specify explicit transformations rather than letting the scene be generated implicitly; (ii) the update of a property depending on another is immediate rather than deferred to facilitate live inspection and debugging; (iii) intermediate representations rely on existing native formats to leverage existing user knowledge and helper tools.

In Intuino [50], Wakita et al. presented two design requirements for a tool aimed at designers when prototyping interactive behaviors with a goal of creativity: " *Behavior of inputs and outputs is designed with only visual operations and the tool is intuitive enough so that designers can concentrate on essence of interaction design* " and " *The operations are easy enough so that users of average PC operation skills can handle the tool without programming* ".

In subArctic [19], Hudson et al. made a clear point to support research: " *It seeks [to] make it easy to create new, unusual, and highly customized interactions tailored to the needs of particular interfaces or task domains* ". Then they described the features supporting this goal: reusable components implementing standard input mechanisms, possibility to add custom mechanisms for behaviors that are usually inflexible, and possibility to inspect and replace any aspect of the input route at run time.

Lastly, many articles detail anecdotal features that are presented as *generative*, in that they were useful for a given context but are justified as a more general programming practice. In XDKinect [42], for example, Nebeling et al. presented a time-based API that automatically keeps 4.5s of tracking data to facilitate custom gesture detection and various user statistics, with methods to filter and query on the history. In the design of the Proximity toolkit [32], Marquardt et al. presented the ability to record and replay proxemic (i.e. interaction) sequences, as an explicit feature to support the prototyping process. In Weave [10], a framework for cross-device wearable interactions, Chi et al. wrote " *It is challenging for developers to cover various types of devices and their combinations that might occur at runtime. [...] Therefore, in addition to directly retrieving a device by its specific type, a framework should enable flexible selection of devices based on their high-level input and output capabilities so that a cross-device service can adapt to specific sets of devices that a user carries at runtime* ".

In summary, supporting research and innovation often boils down to a list of key requirements (*it should be possible to ...*) or features (*the toolkit should include ...*). Yet these lists are very contextual and often hard to extrapolate to the design of new toolkits. Furthermore, few articles discuss or justify how these lists influence the activity of research positively. While they demonstrate in the end the production of innovative artefacts, it is unclear which features were key to research. As a result it is difficult to abstract these requirements into general recommendations useful to many toolkits, although they may serve as initial inspirations.

5.3 Contributions from frameworks

Many general purpose interaction frameworks have documented their design philosophies and goals, and a few empirical studies have evaluated their usage in practice. Here we compare them with our observations, in order to highlight what makes a framework convenient for research, and the specificities of software development in HCI research.

In a presentation during Qt World Summit 2017 [23], the CTO of Qt Lars Knoll detailed a list of design principles : APIs that lead to readable and maintainable code, easy to learn and use but hard to misuse, performant, flexible, keeping it simple, API stability, and world class tools. While all of these points may seem useful in a research context, it is often unclear how they are implemented in practice. For example, with lack of precisions about how to make the framework *flexible*, it is difficult to check whether Qt is flexible enough in the context of HCI research. Moreover, "hard to misuse" could be interpreted as preventing access to private data, thus conflict with our observation on gathering data.

In a 2021 note from the W3C [49], members of the Technical Architecture Group detailed 69 principles covering the design of Web standards and APIs, based on their experience in steering the evolution of Web technologies. The document is notable for addressing many tensions on conflicting needs, with actual examples of good and bad prior designs : high level versus low level

APIs, API usability versus consistency with prior practice, or privacy and reliability versus openness and exploration. What makes the Web a popular platform for research is probably that it seeks a balance between needs : security restrictions can be lifted under certain conditions, native APIs are available on certain browsers, etc.

Meta's Web framework React documents a list of design principles to explain how they accept community contributions [35]: composition, factoring popular usage patterns, providing alternate paradigms (*escape hatches*) when the preferred one falls short, stability of public API, interoperability with other frameworks, scheduling dependencies with the "pull" approach, providing a good developer experience, offering a systematic procedure to track bugs, avoiding global runtime configuration, decoupling from the DOM, not striving for internal elegance, verbose names to facilitate search and maintenance, and prioritizing own use (*dogfooding*). The composition principle is a direct answer to our fifth observation on implementing from scratch, and the debugging procedure an answer to our most critical problem of unpredictability. However, the lack of internal elegance implies that it may be opaque to advanced users seeking internal comprehension, and the dogfooding principle implies that scenarios not experienced at Meta might be ignored, such as multimodal inputs or tangible interactions.

While we are not aware of prior empirical studies on the use of frameworks in a research context, a few articles have considered the consequences of specific design choices. Among them, McDonnell et al. studied the consequences of API evolutions on Android applications [34]. They observed that changes in APIs are rarely followed by users and result in buggier code, highlighting that developers should avoid unstable APIs in general. Nosáľ et al. measured the number of copy-pasted documentation fragments in 5 open source projects [43]. They found 6102 such fragments in the source code of Java 8, which we interpret as a lack of consideration to advanced users. Lastly, Li et al. studied the prevalence and use of hidden APIs in the Android ecosystem [29]. They observed that internal and hidden APIs are indeed used in applications found in the official Google Play store, a dangerous yet tolerated practice, showing again a lack of consideration for advanced users.

As a summary, the design principles vary a lot between frameworks, probably because they are formulated occasionally to resolve conflicts. Moreover, design choices are often presented as net wins without considering their consequences on unusual scenarios, yet few acknowledge the tradeoffs they make (ex. better performance may require more user code, cross-platform support may restrict emerging technologies, etc.). As a consequence we cannot reliably extract a list of design principles from the frameworks alone, but we may consider that design choices are made along a number of dimensions, such as flexibility versus code conciseness, cross-platform versus native support, or security versus exploration. In our review of existing design principles we found that frameworks provided more appropriate features for researchers when they acknowledged these dimensions and had to make tradeoffs. An important lesson here is thus to seek polarizing needs early in the design of frameworks, and in particular to include test scenarios from research.

5.4 Studies on researchers' needs

Articles studying the needs of HCI researchers are remarkably rare compared to the number of toolkits designed for them. Nevertheless, a few have tried to understand the needs of researchers and present them outside the scope of a given artifact.

Among these, in 2010 Letondal et al. focused on the usability needs around interaction-oriented programming tools [27]. They acknowledged that the many tools, languages, architectural patterns and models designed to support interaction had not been significantly influential. Therefore they developed a list of 12 requirements for such work, built from the study of about 50 previous works. These requirements are framed as high-level domains (e.g. Graphics, Interaction modalities,

Managing collective development) that portrait what an interaction framework should contain overall.

In an article reflecting on the past design of over 50 novel interactive systems, Lee et al. discussed a number of aspects common to their projects [25]. Among them and related to programming, they noted that applications exploring immature technology (e.g. a rough object-detection algorithm like SIFT) could be unreliable if the technology was itself unreliable. Their main recommendation aimed at programming was to include user correction mechanisms, to mitigate the consequences of unpredictable behaviors.

During the design of the djnn framework, Chatty and Conversy sought to characterize future languages for interactive system designers [9]. Based on three research themes in systems engineering, they formulated six research directions to guide the evolution of languages to suit interaction : eliciting which functionalities matter to interactive programs, unifying existing concepts, formalizing existing concepts, extending language concepts, extending language notations, and consolidating past results. While this work guided the development of the Smala language [31], its results may be contextual but have still found a practical application.

In a 2016 essay reflecting on current practice and literature, Bergström and Blackwell identified six practices of programming that developers may carry out at any time [2]: Established software engineering practice, Bricolage & tinkering, Sketching with code, Live coding, Hacking, and Codebending. In our context, the engineering of novel interaction techniques spans all of these practices (perhaps except Live coding). This classification may be very relevant for future work, to study how much the needs and problems of researchers differ depending on the activities in play.

As a summary, the literature has provided a general understanding of the fragmented nature of artefacts in HCI research, and which domains come into play. However, there is a lack of knowledge on the classes of artefacts that are labeled *interaction techniques*, and to which programming practices they link. Moreover, recommendations and design principles have often been used to support a specific artefact, rather than to stand on their own. At this point, little is known about how toolkits and frameworks should be designed to support research.

6 DESIGN PRINCIPLES FOR INTERACTION LIBRARIES

Following the lack of clear guidance to design toolkits and frameworks supporting research, we want to provide straightforward directions. These should be easy to interpret for different libraries, without too much ambiguity on how to apply them. We present here three design principles, addressing most of the strategies observed in our studies. For each one we provide examples of how to apply them in different contexts, and their limits.

Principle 1 (duplicate): *Allow the duplication of singular elements to foster opportunities for extensions.*

This principle applies in situations where there is commonly a unique instance of an element (e.g. mouse, menu bar, display monitor). The uniqueness is very often enforced by design choices in frameworks, not as intentional limits, but because no use cases were foreseen to deviate from it. With the mouse, for example, event structures will lack a `mouse_id` field that could discriminate several mice. A more subtle limit will also be the representation of `hover` as a boolean, pointing implicitly at a unique mouse. To alleviate some of these limits and foster the exploration of new interaction techniques, our first design principle consists in analyzing existing elements, functions and properties, and asking the following questions: *1) Is it expected to be unique? 2) Could it make sense to allow many?*

Here are a few examples:

- A slider cannot have multiple values (1), but it would make sense to specify e.g. min-max (or min-median-max) values on a scale (2).
- A keyboard cannot give focus to multiple widgets (1), but it would make sense to allow filling or erasing multiple fields at the same time, like the support for multiple cursors in Sublime Text (2).
- Only one tooltip may be displayed at any time (1), but it would make sense to display all available tooltips with a shortcut (2).
- The user interacting with the interface is expected to be unique (single cursor/keyboard/view) (1), but it would make sense to support many in contexts such at collaborative interfaces or public displays (2).
- A window cannot have multiple titles (1), but it would probably not be useful as a tabbed view or multiple windows could be used instead (2).

This principle is a generalization of the contributions from toolkits that unlocked the duplication of inputs [12, 21], devices [10, 41], or users [32]. It also promotes a finer analysis of individual elements (e.g. the slider), to take down barriers that might be useful to future interaction techniques (e.g. sliders with probability distributions [17]). In answering the above questions, one may create countless hypothetical situations that are infeasible to all implement. This design principle should not be used to actually implement new widgets, but to design more flexible reuse mechanisms that make them easier to create. In that regard it is a partial answer to our fifth observation. Nevertheless, this principle finds its limit in non-trivial changes that may impact the entire architecture of an interaction library. For example, allowing multiple users would imply supporting concurrent accesses to common widgets like text fields or scroll bars, thus requiring the introduction of negotiation mechanisms. In these cases we advise developers to stick to light changes, and keep structural changes to eventual major updates.

Principle 2 (accumulate): *Accumulate rather than replace to keep a history of changes.*

This principle applies in situations where processes are arranged in pipelines or layers transforming data between stages (e.g. processing mouse events, rendering widgets down to pixels). In these cases, we observed that researchers would access layers of different levels, to fetch data that would be lost or altered at the framework level. In doing so they mingled functions from different APIs, with more learning beforehand and possibly incompatible combinations. Moreover, when researchers interpreted raw events they could not be sure their results were the same as frameworks. Our solution and recommendation here is to keep a common data structure across layers, such that data from every layer can be fetched with a single API. For any property or function argument, we propose asking the following questions: *1) Is this data replaced by another? 2) Could it make sense to keep both at any time?*

Here are a few examples:

- The position of a window replaces its previous position (1), but keeping a history may allow implementing inertia, or gestures based on window position (2).
- The raw coordinates of a mouse sensor are replaced by pixel coordinates (1), but keeping both may allow unclipped movements like 3D camera, implementing a different dots-to-pixels transfer function (e.g. with libpointing [8]), or implementing motion prediction (2).
- The pressure or area of a touch on trackpad is lost when exposed as a pointer by an incompatible library (1), but keeping it along with standard pointing data would allow prototyping interfaces for trackpads not available at the present time (2).

- The new values of widgets replace the old ones (1), but keeping both (temporarily) may allow highlighting the widgets that changed and their direction (e.g. Phosphor [1]), or reverting to the old values with an undo stack.
- The position of a mouse replaces the previous position (1), but keeping a history may allow recording and replaying an interaction session, or implementing a trail behind the cursor (2).

This principle is a direct answer to our fourth observation, and also a generalization of the contributions from toolkits that enabled the visualization of timely changes [1], record-and-replay of interactions [32], or automated the accumulation of input events for gesture recognition [42]. However its purpose is also to support polymorphism, in that elements should keep their raw characteristics when abstracted into other elements (e.g. touch events to pointing events). For such cases, an alternative design principle might be to allow access to layers where missing data is found. However, we consider this practice less desirable as it increases the number of APIs to use and generally makes every requested data tedious to find.

New data structures may be introduced to accumulate changes over time. For immutable data (integer, tuple, event structure), an array may be used, along with timestamps for entries. For mutable data (array, dictionary, object), a list of commands may be used, each storing the field to update and the delta to apply. Since size of storage may be an issue, circular buffers should be used, with a predefined time window (e.g. keeping 10s of mouse positions). Moreover, for cases where researchers would want to add custom data to an existing storage, we encourage developers to use extensible data structures that can accept new types of data (i.e. dictionaries instead of classes).

An important limit of the *accumulate* principle is that it does not keep track of causal relationships between events, only the sequences of events themselves. For example, if a researcher is interested in measuring how often a given command is executed by a button or a keyboard shortcut (an actual case from the interviews), they would not need the history of command executions but who triggered each one. Indeed, events and commands do not normally point at who triggered them. Another example would be to track the widget responsible for painting a given pixel (i.e. picking). Pixels do not remember which primitive or widget painted them. In these cases, accumulation is insufficient to keep a history of changes, and more complex mechanisms would be needed to record more complex changes.

Principle 3 (defer): *Defer the execution of predefined behaviors to enable their monitoring and replacement.*

This principle applies when prototyping new input/output interaction techniques into existing application frameworks (e.g. changing the mouse transfer function, or allowing any interface to be zoomed), and changing the behavior of an existing application (e.g. replacing the graphical rendering backend). In these situations, researchers often stumble on hardcoded behaviors that are not meant to be replaced, and would require hacking a framework with unadvisable practices. Here, our principle is to turn *direct* commands into *indirect* ones, to allow researchers to inspect them afterwards. For example, to draw a line on screen one would turn a call to `draw_line(...)` into putting a line primitive in a display list, then rendering the list. Deferral thus consists in splitting the call into (i) placing an order and (ii) executing the order (without delay in between). We propose analyzing functions and methods and asking the following questions: *1) Can this action be intercepted? (i.e. canceled, altered or repeated) 2) If not, could it be useful at runtime or compile time?*

Here are a few examples:

- When the mouse clicks inside a text widget it gets the keyboard focus (1), intercepting this behavior would allow inhibiting it (e.g. if the mouse was moving while it clicked), modifying

it (e.g. giving focus by simply hovering), or exploring more complex scenarios like multiple mice and keyboards (2).
- Deferring the drawing of all widgets with a display list (1) would allow highlighting all primitives on screen (i.e. wireframe), or prototyping a zoomable interface by enlarging geometric primitives instead of pixels (2).
- Deferring the interpretation of pointing inputs (1) would allow swapping the mouse transfer function, changing the speed of the double-click, or swapping the built-in recognizer for swipes and gestures (2).
- Intercepting the execution of any command (1) would allow replacing it at runtime, measuring its actual use, or recording sequences of commands into reusable macros (2).
- Intercepting drags on objects that cannot normally be dragged (1) would allow implementing the reordering of items in tabs and menus, or turning regular interfaces into WYSIWYG editors (2).

Interaction frameworks and toolkits are known to be very flexible in general, and some of the above examples are already familiar. However, they often rely on many indirection mechanisms like hooks and callbacks, that make overall architectures too complex to introspect. In line with our principle, we call for mutualizing the data structures used as indirections, to make them easier to browse. In particular, this principle has a strong relation with intermediate representations, as *deferring* implies storing a command before interpreting it. We advocate the use of documented intermediate representations, and the implementation of behaviors as *passes* on them. For example, one may use a generic software bus for commands [6] and input/output events, or a generic mechanism like dataflow that provides the ability to reroute any signal at runtime.

This principle finds its limits in privacy-sensitive applications, which need to conceal information and forbid unwanted interception. There is also a risk of information overload, in addition to the principle of accumulation, if commands and events were recorded without any sorting or hierarchy. Moreover, developpers might eventually add interaction techniques in the form of "hacks" taking advantage of indirect structures. As a consequence, any data structure used for deferral should provide bookkeeping to keep track of original versus patched data.

7 CONCLUSION

In this article we studied the work of HCI researchers when designing and implementing novel interaction techniques. We interviewed and surveyed them to uncover (i) their reasons for choosing software libraries for prototyping, (ii) their problems while using them and (iii) their strategies for overcoming difficulties. Then we proposed and detailed three design principles to better support research in interaction frameworks and toolkits: duplicate, accumulate, defer. Our observations and principles are meant to be used by designers of interaction libraries, to help them adapt their software to the programming practices in HCI.

Over the course of our studies we noticed that the range of interaction techniques reported by researchers was a lot more varied than the examples presented in the literature on interaction programming. The techniques observed in the wild may target new input/output technologies, different computing devices, users with varying expertise or impairments, and challenging environments. While we observed a few such cases during our interviews, we noticed they had their own specific problems that sometimes would not fit with our design principles.

In order to understand the programming of interaction techniques in details, we need to take a closer look at the different kinds of techniques that exist nowadays. Without a more precise understanding of how researchers program, most recommendations and design principles (including the present article) will remain at a high level of abstraction, making them unlikely to cover every

situation and specific problems. To that end, we encourage future work to classify interaction techniques into how they are programmed, and perhaps provide more focused design principles for each kind.

REFERENCES

[1] Patrick Baudisch, Desney Tan, Maxime Collomb, Dan Robbins, Ken Hinckley, Maneesh Agrawala, Shengdong Zhao, and Gonzalo Ramos. 2006. Phosphor: Explaining Transitions in the User Interface Using Afterglow Effects. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*. Association for Computing Machinery, Montreux, Switzerland, 169–178. https://doi.org/10.1145/1166253.1166280

[2] Ilias Bergström and Alan F. Blackwell. 2016. The Practices of Programming. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 190–198. https://doi.org/10.1109/VLHCC.2016.7739684

[3] Renaud Blanch, Yves Guiard, and Michel Beaudouin-Lafon. 2004. Semantic Pointing: Improving Target Acquisition with Control-display Ratio Adaptation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. ACM, New York, NY, USA, 519–526. https://doi.org/10.1145/985692.985758 event-place: Vienna, Austria.

[4] M. Bostock, V. Ogievetsky, and J. Heer. 2011. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec. 2011), 2301–2309. https://doi.org/10.1109/TVCG.2011.185

[5] Joel Brandt, Philip J. Guo, Joel Lewenstein, and Scott R. Klemmer. 2008. Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice. In *Proceedings of the 4th International Workshop on End-user Software Engineering (WEUSE '08)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1370847.1370848

[6] Marcellin Buisson, Alexandre Bustico, Stéphane Chatty, Francois-Régis Colin, Yannick Jestin, Sébastien Maury, Christophe Mertz, and Philippe Truillet. 2002. Ivy: Un Bus Logiciel Au Service Du Développement De Prototypes De Systèmes Interactifs. In *Proceedings of the 14th Conference on L'Interaction Homme-Machine (IHM '02)*. ACM, New York, NY, USA, 223–226. https://doi.org/10.1145/777005.777040

[7] Adam S. Carter and Christopher D. Hundhausen. 2010. How Is User Interface Prototyping Really Done in Practice? A Survey of User Interface Designers. In *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*. 207–211. https://doi.org/10.1109/VLHCC.2010.36

[8] Géry Casiez and Nicolas Roussel. 2011. No More Bricolage!: Methods and Tools to Characterize, Replicate and Compare Pointing Transfer Functions. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 603–614. https://doi.org/10.1145/2047196.2047276

[9] Stéphane Chatty and Stéphane Conversy. 2014. What Programming Languages for Interactive Systems Designers?. In *EICS 2014, Engineering Interactive Computing Systems Workshop* -. pp 47–51.

[10] Pei-Yu (Peggy) Chi and Yang Li. 2015. Weave: Scripting Cross-Device Wearable Interaction. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 3923–3932. https://doi.org/10.1145/2702123.2702451

[11] Morgan Dixon and James Fogarty. 2010. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. Association for Computing Machinery, Atlanta, Georgia, USA, 1525–1534. https://doi.org/10.1145/1753326.1753554

[12] Pierre Dragicevic and Jean-Daniel Fekete. 2004. Support for Input Adaptability in the ICON Toolkit. In *Proceedings of the 6th International Conference on Multimodal Interfaces (ICMI '04)*. ACM, New York, NY, USA, 212–219. https://doi.org/10.1145/1027933.1027969

[13] James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2011. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 225–234. https://doi.org/10.1145/2047196.2047226

[14] B. Efron. 1979. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics* 7, 1 (Jan. 1979), 1–26. https://doi.org/10.1214/aos/1176344552

[15] Martin Fowler. 2005. InversionOfControl. https://martinfowler.com/bliki/InversionOfControl.html.

[16] Saul Greenberg. 2007. Toolkits and Interface Creativity. *Multimedia Tools and Applications* 32, 2 (Feb. 2007), 139–159. https://doi.org/10.1007/s11042-006-0062-y

[17] Miriam Greis, Hendrik Schuff, Marius Kleiner, Niels Henze, and Albrecht Schmidt. 2017. Input Controls for Entering Uncertain Data: Probability Distribution Sliders. *Proceedings of the ACM on Human-Computer Interaction* 1, EICS (June 2017), 3:1–3:17. https://doi.org/10.1145/3095805

[18] V. Grigoreanu, R. Fernandez, K. Inkpen, and G. Robertson. 2009. What designers want: Needs of interactive application designers. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 139–146. https://doi.org/10.1109/VLHCC.2009.5295277

[19] Scott E. Hudson, Jennifer Mankoff, and Ian Smith. 2005. Extensible Input Handling in the subArctic Toolkit. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '05)*. ACM, New York, NY, USA,

381–390. https://doi.org/10.1145/1054972.1055025
[20] Ralph Johnson and Brian Foote. 1988. Designing Reusable Classes. *Journal of Object-Oriented Programming* 1, 2 (June 1988), 22–35.
[21] M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza. 2005. TUIO A Protocol for Table-Top Tangible User Interfaces. In *Proc. of the The 6th Int'l Workshop on Gesture in Human-Computer Interaction and Simulation*. 1–5.
[22] Travis Kirton, Sebastien Boring, Dominikus Baur, Lindsay MacDonald, and Sheelagh Carpendale. 2013. C4: A Creative-Coding API for Media, Interaction and Animation. In *Proceedings of the 7th International Conference on Tangible, Embedded and Embodied Interaction (TEI '13)*. Association for Computing Machinery, Barcelona, Spain, 279–286. https://doi.org/10.1145/2460625.2460672
[23] Lars Knoll. 2017. Qt Design Principles and Roadmap.
[24] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. 2018. Evaluation Strategies for HCI Toolkit Research. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*. ACM Press, Montreal QC, Canada, 1–17. https://doi.org/10.1145/3173574.3173610
[25] Hyowon Lee, Nazlena Mohamad Ali, and Lynda Hardman. 2013. Designing Interactive Applications to Support Novel Activities. *Advances in Human-Computer Interaction* 2013 (June 2013), e180192. https://doi.org/10.1155/2013/180192
[26] Germán Leiva, Nolwenn Maudet, Wendy Mackay, and Michel Beaudouin-Lafon. 2019. Enact: Reducing Designer–Developer Breakdowns When Prototyping Custom Interactions. *ACM Transactions on Computer-Human Interaction* 26, 3 (May 2019), 19:1–19:48. https://doi.org/10.1145/3310276
[27] Catherine Letondal, Stéphane Chatty, Greg Philips, Fabien André, and Stéphane Conversy. 2010. Usability Requirements for Interaction-Oriented Development Tools. In *Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group*. 12–26.
[28] Catherine Letondal, Pierre-Yves Pillain, Emile Verdurand, Daniel Prun, and Olivier Grisvard. 2014. Of Models, Rationales and Prototypes: Studying Designer Needs in an Airborne Maritime Surveillance Drawing Tool to Support Audio Communication. In *Proceedings of the 28th International BCS Human Computer Interaction Conference*. 72–81. https://doi.org/10.14236/ewic/hci2014.8
[29] Li Li, Tegawendé F. Bissyandé, Yves Le Traon, and Jacques Klein. 2016. Accessing Inaccessible Android APIs: An Empirical Study. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 411–422. https://doi.org/10.1109/ICSME.2016.35
[30] Mathieu Magnaudet, Stéphane Chatty, Stéphane Conversy, Sébastien Leriche, Celia Picard, and Daniel Prun. 2018. Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming. *Proc. ACM Hum.-Comput. Interact.* 2, EICS (June 2018), 12:1–12:27. https://doi.org/10.1145/3229094
[31] Mathieu Magnaudet, Stéphane Chatty, Stéphane Conversy, Sébastien Leriche, Celia Picard, and Daniel Prun. 2018. Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming. *Proc. ACM Hum.-Comput. Interact.* 2, EICS (June 2018), 12:1–12:27. https://doi.org/10.1145/3229094
[32] Nicolai Marquardt, Robert Diaz-Marino, Sebastian Boring, and Saul Greenberg. 2011. The Proximity Toolkit: Prototyping Proxemic Interactions in Ubiquitous Computing Ecologies. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 315–326. https://doi.org/10.1145/2047196.2047238
[33] Nicolai Marquardt, Steven Houben, Michel Beaudouin-Lafon, and Andrew D. Wilson. 2017. HCITools: Strategies and Best Practices for Designing, Evaluating and Sharing Technical HCI Toolkits. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '17)*. ACM, New York, NY, USA, 624–627. https://doi.org/10.1145/3027063.3027073
[34] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *2013 IEEE International Conference on Software Maintenance*. 70–79. https://doi.org/10.1109/ICSM.2013.18
[35] Meta. 2022. Design Principles – React. https://reactjs.org/docs/design-principles.html.
[36] Ditte Hvas Mortensen. 2021. How to Do a Thematic Analysis of User Interviews. https://www.interaction-design.org/literature/article/how-to-do-a-thematic-analysis-of-user-interviews.
[37] Martez E. Mott and Jacob O. Wobbrock. 2019. Cluster Touch: Improving Touch Accuracy on Smartphones for People with Motor and Situational Impairments. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*. ACM Press, Glasgow, Scotland Uk, 1–14. https://doi.org/10.1145/3290605.3300257
[38] B. Myers, S. Y. Park, Y. Nakano, G. Mueller, and A. Ko. 2008. How designers design and program interactive behaviors. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. 177–184. https://doi.org/10.1109/VLHCC.2008.4639081
[39] B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. 1997. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering* 23, 6 (June 1997), 347–365. https://doi.org/10.1109/32.601073

[40] Mathieu Nancel, Stanislav Aranovskiy, Rosane Ushirobira, Denis Efimov, Sebastien Poulmane, Nicolas Roussel, and Géry Casiez. 2018. Next-Point Prediction for Direct Touch Using Finite-Time Derivative Estimation. In *The 31st Annual ACM Symposium on User Interface Software and Technology - UIST '18*. ACM Press, Berlin, Germany, 793–807. https://doi.org/10.1145/3242587.3242646

[41] Michael Nebeling, Theano Mintsi, Maria Husmann, and Moira Norrie. 2014. Interactive Development of Cross-Device User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. Association for Computing Machinery, Toronto, Ontario, Canada, 2793–2802. https://doi.org/10.1145/2556288.2556980

[42] Michael Nebeling, Elena Teunissen, Maria Husmann, and Moira C. Norrie. 2014. XDKinect: Development Framework for Cross-Device Interaction Using Kinect. In *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '14)*. Association for Computing Machinery, Rome, Italy, 65–74. https://doi.org/10.1145/2607023.2607024

[43] Milan Nosál' and Jaroslav Porubän. 2016. Preliminary Report on Empirical Study of Repeated Fragments in Internal Documentation. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 1573–1576.

[44] Fatih Kursat Ozenc, Miso Kim, John Zimmerman, Stephen Oney, and Brad Myers. 2010. How to Support Designers in Getting Hold of the Immaterial Material of Software. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. Association for Computing Machinery, Atlanta, Georgia, USA, 2513–2522. https://doi.org/10.1145/1753326.1753707

[45] Miller Puckette. 1991. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal* 15, 3 (1991), 68–77. https://doi.org/10.2307/3680767 Publisher: The MIT Press.

[46] Maria Rosala. 2020. The Critical Incident Technique in UX. https://www.nngroup.com/articles/critical-incident-technique/.

[47] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. 2014. Declarative Interaction Design for Data Visualization. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 669–678. https://doi.org/10.1145/2642918.2647360

[48] Allen B. Tucker. 2004. *Computer Science Handbook, Second Edition*. Chapman & Hall/CRC.

[49] W3C Technical Architecture Group. 2021. Web Platform Design Principles. https://www.w3.org/TR/design-principles/.

[50] Akira Wakita and Yuki Anezaki. 2010. Intuino: An Authoring Tool for Supporting the Prototyping of Organic Interfaces. In *Proceedings of the 8th ACM Conference on Designing Interactive Systems (DIS '10)*. Association for Computing Machinery, Aarhus, Denmark, 179–188. https://doi.org/10.1145/1858171.1858204

[51] J. Yu, B. Benatallah, F. Casati, and F. Daniel. 2008. Understanding Mashup Development. *IEEE Internet Computing* 12, 5 (Sept. 2008), 44–52. https://doi.org/10.1109/MIC.2008.114

A INTERVIEW PLAN

Introduction

This interview is part of my PhD, where I look at the limits of GUI libraries and frameworks for prototyping and building interactive applications, or advanced interaction techniques (Qt, Cocoa, Swing, SDL, …), and in particular how they are hacked around in actual projects, to get things done. I would like to backtrack with you a few of your past works, where the library could not do everything you intended, so you had to hack your way in. I selected some on the Internet already, but we can review another one if it is more relevant to you.

Questions

1. Age? Years of experience? In main language? Frequency of programming? Languages/IDEs/frameworks of choice?
2. Which platform/language/IDE/framework(s) did you use, and why these choices? Approximately how many lines of code is the project? How long did it take to code it? How many versions did you do, w.r.t. refactoring?
3. At which point in the design/prototyping process did you get a working software prototype? What did it implement already? What was left to implement?
4. What were your ambitions at the start of the project? Is there some of your ideas that you could not implement and test because of technological issues/limitations?
5. What was the most difficult thing you had to implement? Would you consider it *hacking*? What would you consider *low-level* there?
6. On a Likert scale (from 1 very dirty to 5 very clean), how “dirty” is it now? If you had the opportunity to recode it, how different would it be?
7. How did you learn the framework(s)? (official doc, book, tutorials, copy/paste examples) How much time did you dedicate to it? Did you have to learn some additional API over the course of the project?
8. With hindsight, what would have helped you best to complete the project? (excluding any library done after) A better framework? A better tutorial?
9. Now if you were to add this code to one of the libraries you used, which one would it be? (higher/lower level, new library) Why?
10. How do you think the framework(s) should have been designed to best suit your need? (may answer weeks later)
11. [Do you have any expectations about my work? :]

Final words

Thank you for your time!
I can send you the results of this study later if you want. Also, it would be nice if we can schedule a short meeting in about two weeks, in case you have some more feedback for this study.

B SURVEY QUESTIONS

The goal of this survey is to better understand the process of programming new interactive artefacts for research and innovation purposes, and the software libraries used to support this process. We are interested in projects where you reached the limits of libraries (e.g. undocumented needs, accessing private functionalities, interfacing with existing applications, combining with other libraries), for the prototyping and implementation of innovative interactive artefacts (e.g. non-standard interaction techniques, data visualisations, UI toolkits, interactive applications).

This research is conducted by Thibault Raffaillac (thibault.raffaillac@inria.fr) and Stéphane Huot (stephane.huot@inria.fr), members of the Inria's team Loki (http://loki.lille.inria.fr/).

This survey takes about 20 minutes to complete. Please read this consent form carefully before proceeding.

Basis to take part in this survey

You must be over 18 years old.
You must have prior experience in developing interactive applications, preferably in a context of research and/or innovation.

Voluntary participation in the project

Your participation to this survey is entirely voluntary, and without any constraints or outside pressure. If you are lacking information needed to make your decision, or have any questions about the project or your rights as a participant, do not hesitate to ask for additional information from the contact person (thibault.raffaillac@inria.fr).

Withdrawal from the project at any time

You are free to terminate your participation to this survey at any time, without any justification, by closing this web page. In this event, the answers you already have responded will not be logged and their will be no trace of your participation in our data.

Anonymity and confidentiality

All the raw information collected from your participation is anonymous and only the members of the research project may have access to it. The data we are collecting is only your answers to the questions (checkboxes and text). We are not collecting any personal data (e.g. email, IP) and the survey system we are using does not allow to retrieve your identity while you are answering the survey or once you have submitted your answers.
Collected data will be used to illustrate the results of our work in scientific communications (articles, posters, oral presentations) or scientific mediation after it has been processed and analyzed. Thus, there will be no way to identify you as a participant. In any case, while processing the data, we will take care to anonymize any collected data that could be used to refer back to you as an individual in your answers.

Informed consent *

☐ I acknowledge that I have read and understood this consent form, and voluntarily consent to participate in this research project

Page suivante >

Début | Preliminary | Choice of library | Issues with libraries | Addressing them | Terminé

Main professional activity*

- Researcher
- Interaction designer
- Project manager
- Engineer
- Software developer
- Other

Other main professional activity

Which kind of institution/company do you work for? *

- University/School
- Public institution
- Startup
- Self-employed
- Small enterprise (< 50 employees)
- Medium enterprise (< 250 employees)
- Big enterprise (≥ 250 employees)

How many people on average (excl. yourself) are working with you on a single project?*

- 0
- 1
- 2
- 3
- 4
- 5+

How do you evaluate your own expertise in writing interactive applications?*

- Basic knowledge
- Novice
- Intermediate
- Advanced
- Expert

How much of your time (professional or leisure) do you devote to programming?*

- 0%~20%
- 20%~40%
- 40%~60%
- 60%~80%
- 80%~100%

< Page précédente | Page suivante >

Début — Preliminary — Choice of library — Issues with libraries — Addressing them — Terminé

In this section we ask you to evaluate the importance of various criteria when choosing a library for a project. Here we consider general-purpose frameworks (e.g. Qt, Cocoa, JavaFX, ReactJS) as well as research toolkits for specific needs (e.g. remote collaboration, wall-sized displays, low-power devices). Below each criterion you may give examples to illustrate your answer.

Which frameworks/toolkits do you use the most? (comma-separated)

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Tool reputation (timely and frequent bug fixes, online comparisons, recommendation from peers, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Developer reputation (number of developers, brand, other tools from the same developer(s)/company, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
User community (activity, size, companies using it, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Documentation quality (abundance, clarity, time to learn, examples, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Range of use cases supported (number, variety, relation to your context, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Quality of the API (coherency, simplicity, adequate paradigm for your needs, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Compatibility with other tools/libraries (bindings already available, extensibility, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Project constraints (already in use, expertise from colleagues, licensing terms, ...) *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Personal experience/familiarity with the library *	○	○	○	○	○

Example criteria

	Not important at all	Of little importance	Of average importance	Very important	Absolutely essential
Technical efficiency (performance, latency, memory use, ...) *	○	○	○	○	○

Example criteria

Are there other important criteria we did not mention here? (comma-separated)

< Page précédente | Page suivante >

Début — Preliminary — Choice of library — Issues with libraries — Addressing them — Terminé

In this section we ask you to rate the severity of selected issues, that you might have encountered while using interaction libraries. In the table below you may select "Not met" if you never encountered a given issue. Otherwise, indicate how difficult it was to overcome.

Did you encounter any of these issues, and if so how much did they hinder your work at worst? *

	Not met	Met, no trouble	Met, minor trouble	Met, medium trouble	Met, major trouble	Met, insurmountable trouble
Inadequate paradigm for this particular context *	○	○	○	○	○	○
Problems scaling up (in speed/precision/frequency) *	○	○	○	○	○	○
API too complex to use/understand *	○	○	○	○	○	○
Documentation lacking context and examples *	○	○	○	○	○	○
Significant effect/behaviour being undocumented *	○	○	○	○	○	○
Bad compatibility between two libraries *	○	○	○	○	○	○
Forbidden access to functions and data *	○	○	○	○	○	○
Inconsistent behavior across versions/systems *	○	○	○	○	○	○
Documentation requiring too much investment *	○	○	○	○	○	○
Non-deterministic behavior *	○	○	○	○	○	○
Lack of a functionality that would require pulling another library *	○	○	○	○	○	○
Buggy implementation *	○	○	○	○	○	○
Lack of configurability in API *	○	○	○	○	○	○

< Page précédente Page suivante >

Début — Preliminary — Choice of library — Issues with libraries — Addressing them — Terminé

In this section we consider how you addressed the various issues met in the previous section. We selected a number of coding techniques from previous interviews, and ask you to rate their prevalence in your work. For each technique there is an optional text entry, where you may give examples of the coding techniques you used.

	Never	Rarely	Sometimes	Very often	Always
Using an external mechanism to obtain and process data that is not exposed by an application *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Using accessible raw data to reconstruct/reinterpret a state that you do not have access to *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Aggregating multiple sources of interaction data (input, sensors, events), and fusing them into a single source *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Reimplementing an existing widget/mechanism to gain more control over its appearance/behavior *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Reimplementing low-level system components (e.g. driver) to improve their functionalities or better support specific hardware devices *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Introducing a different programming model, pattern or paradigm on top of the existing framework or toolkit *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Using a visual overlay to add custom functionality *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Reverse-engineering a closed tool or library to acquire understanding of its inner working*	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Modifying the environment of a tool rather than the tool itself to change its behavior *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Purposely setting a parameter outside of its intended/expected range of values *	○	○	○	○	○

Example techniques

	Never	Rarely	Sometimes	Very often	Always
Reproducing a fake application to control a specific aspect *	○	○	○	○	○

Example techniques

< Page précédente Soumettre

Received July 2021; revised October 2021; accepted November 2021