Enact: Reducing Designer-Developer Breakdowns when Prototyping Custom Interactions

Germán Leiva, Nolwenn Maudet, Wendy Mackay, Michel Beaudouin-Lafon

Enact: Reducing Designer-Developer Breakdowns when Prototyping Custom Interactions

GERMÁN LEIVA, Université Paris-Sud, CNRS, Inria, Université Paris-Saclay
NOLWENN MAUDET and WENDY MACKAY, Inria, Université Paris-Sud, CNRS, Université Paris-Saclay
MICHEL BEAUDOUIN-LAFON, Université Paris-Sud, CNRS, Inria, Université Paris-Saclay

Professional designers and developers often struggle when transitioning between the design and implementation of an interactive system. We conducted three studies that focused on the design of custom interactions to understand the mismatches between their processes, tools and representations. We found that current practices induce unnecessary rework and cause discrepancies between design and implementation. We identified three recurring types of breakdowns: omitting critical details, ignoring edge cases, and disregarding technical limitations.

We propose four design principles to create tools that mitigate these problems: *Provide multiple viewpoints*, *maintain a single source of truth*, *reveal the invisible* and *support design by enaction*. We applied these principles to create Enact, a live environment for prototyping touch-based interactions. We conducted two studies to assess Enact and to compare designer-developer collaboration with Enact versus current tools. Results suggest that Enact helps participants detect more edge cases, increases designers' participation and provides new opportunities for co-creation.

CCS Concepts: • **Human-centered computing → Interface design prototyping**; **Empirical studies in collaborative and social computing**; *User interface programming*; *Empirical studies in HCI*;

Additional Key Words and Phrases: Designer-developer collaboration; prototyping; touch-based interaction

ACM Reference Format:
Germán Leiva, Nolwenn Maudet, Wendy Mackay, and Michel Beaudouin-Lafon. 2019. Enact: Reducing Designer-Developer Breakdowns when Prototyping Custom Interactions. *ACM Trans. Comput.-Hum. Interact.* 26, 3, Article 19 (May 2019), 47 pages. https://doi.org/10.1145/3310276

1 INTRODUCTION

Interactive software development in the 1970s and 1980s involved traditional software engineering processes, with limited interdisciplinary collaboration between developers and graphic designers. By the 1990s, the advent of full color, high-resolution displays enabled high-quality interactive graphics, with a corresponding need for professional designers. Today, graphic designers, interaction designers and user experience specialists are routinely part of the teams creating interactive software, together with software developers. However, integrating designers' and developers' work practices has proven difficult, often leading to friction between them [19] and negatively affecting both the process and the final interactive system [44].

Designers and developers of interactive systems have different backgrounds and skills [12] and focus on different aspects of the design process [40]. Despite these differences, they need to collaborate in order to create interactive systems. However, while individuals often work closely together, their tools and artifacts do not.

Authors' addresses: Germán Leiva, leiva@lri.fr, Université Paris-Sud, CNRS, Inria, Université Paris-Saclay; Nolwenn Maudet, nolwenn@iis.u-tokyo.ac.jp; Wendy Mackay, mackay@lri.fr, Inria, Université Paris-Sud, CNRS, Université Paris-Saclay; Michel Beaudouin-Lafon, mbl@lri.fr, Université Paris-Sud, CNRS, Inria, Université Paris-Saclay.

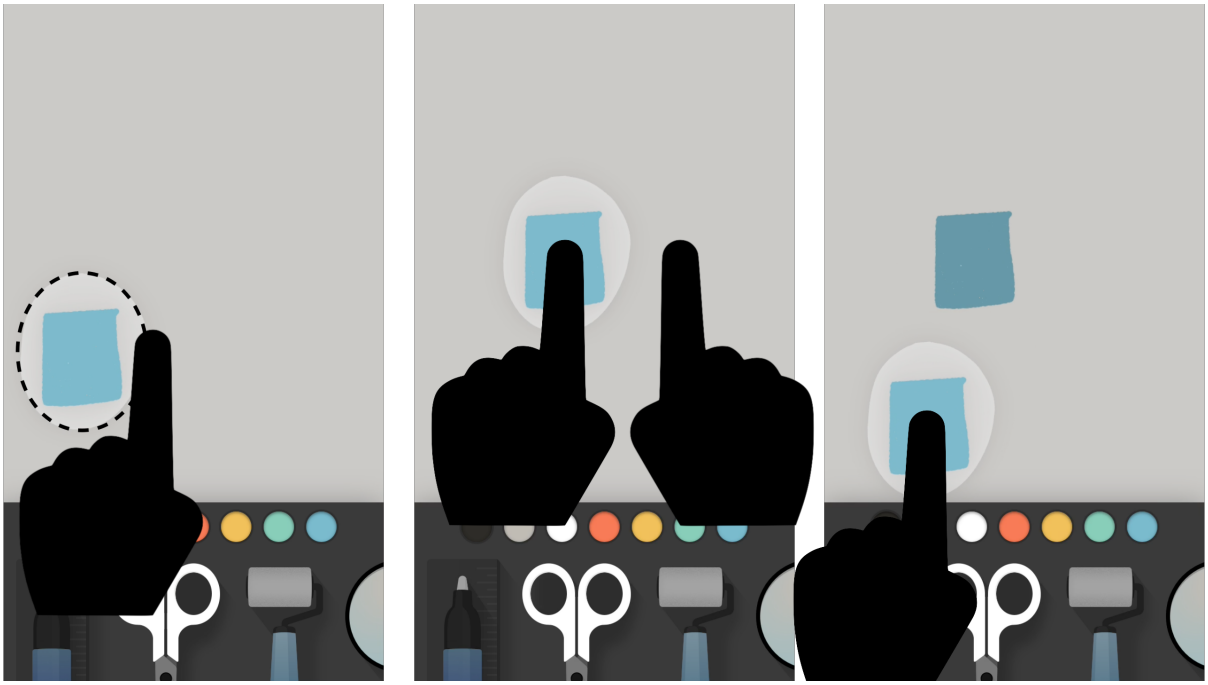

Fig. 1. An example of a custom interaction in the Paper mobile app from FiftyThree. In the first screen, the user has selected the scissor tool and draws a circular area with one finger. In the second screen, the user drags this area to move the content to a new position and taps outside the circular area with another finger to create a copy of the selected area. In the third screen, the user drags the selected area to reveal the copied shape.

This is particularly true when the design involves custom interactions and non-standard dynamic behaviors such as animations: Designers find it much easier to communicate the static visual appearance of the design than the dynamic aspects [46]. By custom we mean interactions that current designer and developer tools do not provide out-of-the-box (Figure 1). This includes completely new interface widget as well as interactions that rely on non-standard system output, such as a highly crafted animation, or a non-standard use of user inputs, such as using a pinch gesture to delete an object.

Designers are trained to communicate visually. They use graphical editors, e.g. Bohemian Sketch or Adobe Illustrator, to create static design artifacts such as wireframes and mockups [47]. However, common designer tools, such as vector and raster graphics, are not designed to handle interactions and animations. Designers usually prioritize visual design [15] over the rules and data structures that govern the software.

Complementary, developers need to create working interactions from the design specification. Developers are trained to work with abstractions, generally manipulated through textual representations. They use tools such as text editors and Integrated Development Environments (IDEs) to create functional systems. They prioritize the translation of design artifacts into implementable formats over the details of visual design and user interaction.

Addressing the gap between designers and developers, and more precisely the gap between the tools that they use to create interactive systems thus remains an open research question. In this work we identify problems faced by designers and developers as they collaborate, with a particular emphasis on the representation, communication and interpretation of custom interactions. Our goal is to inform the design of new collaborative prototyping tools

that reduce these problems, without forcing professionals to abandon their preferred representations. We want to facilitate the transition from design to implementation, as well as the transition from implementation back to the design when changes occur. More specifically, we address the following research questions:

- What are the most common and critical problems that impair designer-developer collaboration when creating interactive software that involves custom interactions?
- How can prototyping tools help mitigate these problems?

We start by describing the motivation and methodology of the project, and review relevant related work. We then describe three studies that we conducted to better understand designer-developer practices, leading to a set of design principles to reduce the breakdowns they encounter. We introduce Enact, a prototyping tool for touch-based mobile interaction based on these principles, and report on two studies that we conducted to assess Enact.

2 MOTIVATION AND METHODOLOGY

In this work, we define an *interaction* as the set of rules that coordinate the system outputs with the user inputs, i.e. when and how the system reacts to certain user inputs. For example, on a mobile phone, user inputs include finger touches captured by the capacitive touchscreen, user movements captured by the accelerometer, facial expressions captured by the camera, or voice commands captured by the microphone. The system outputs include both the *feedback* in the user interface, such as highlighting a button, and the corresponding *operations* to be executed by the underlying system, such as modifying a database or sending a request to a server. This paper focuses on visual outputs, but the system outputs can also include tactile or auditory feedback.

Designing, communicating and implementing interaction is complex. Fernaeus and Sundström [18] report that *"creating a fully working interactive system is not merely a matter of 'translating' a static sketch into its dynamic gestalt"*. Similarly, Ozenc et al. [51] argue that designers currently "*struggle to have a conversation with the material*". Park et al. [52] report on a laboratory study that highlights the different uses of text to represent interaction. Designer tools influenced *"their natural expression of behaviors"*, while developers used more verbose descriptions.

Designers and developers often coordinate their work through a so-called *handoff* phase. The handoff includes all the design artifacts that designers send to developers in order to construct the user interface. However, the name "handoff" is misleading, because these artifacts are not delivered and forgotten. Designers and developers refine these artifacts collaboratively and iteratively, requiring a back-and-forth work between them. This is especially true when the design includes custom interactions that lack a standard user interface vocabulary to describe them.

A number of collaborative prototyping tools have emerged to support the handoff phase[1]. For example, InVision[2] lets designers and developers communicate with text annotations and create click-through prototypes from static design artifacts. However, developers cannot manipulate this prototype with their existing tools and the available user inputs are limited to standard discrete interactions such as button clicks. This type of tool focuses more on the information architecture of the system and the communication among practitioners, rather than on the design of custom interactions and its communication with the immaterial domain of software [51]. Indeed, few commercial tools let designers prototype custom interactions, usually forcing them to work with developer representations, such as code[3] or visual programming[4]. Grigoreanu et al. [24] suggest that designers

[1]See, e.g., http://www.prototypingtools.co
[2]https://www.invisionapp.com/
[3]https://framer.com
[4]https://origami.design/

needs, such as "reusing code and designs" or "automating redundant steps", are not well supported by current software.

In order to address this gap between designers and developers, we need a better understanding of cross-disciplinary collaborative work. In their study of collaboration among different communities of practice, Star and Griesemer [58] found two main factors that contributed to a successful collaborative work: the use of *methods standardization* and the creation of *boundary objects*.

2.1 Methods standardization

Standardization can facilitate the designer-developer collaboration by providing a shared vocabulary to describe an interaction. All major software vendors provide their own guidelines and standards for interaction, such as Google's material design[5] and Apple's iOS Human Interface Guidelines[6]. Practitioners' prototyping tools such as InVision[7] also provide a limited and standardized interaction vocabulary to activate transitions between screens. These standards are widely used and they do facilitate the communication between designers and developers. However, they limit the vocabulary of interactions to a predefined set and therefore they cannot solve the representation issues that occur when creating custom interactions.

Another aspect of method standardization is to extend engineering representations to encompass interaction design. For example, UML[8] diagrams have been extended to describe interactive behaviors [16] and to serve as a communication tool between designers and developers. But as reported by Shipman and Marshall [55], such a level of formalism is not well received by the design community, as it does not match the types of representations used by designers.

Inspired by the popularity of software design patterns [22], Borchers [5] proposed the creation of catalogs of interaction design patterns, and Wiemann [64] proposed using them as the Lingua Franca [17] between designers and developers. Expanding the shared vocabulary among designers and developers can certainly facilitate collaboration. However, this language should not be limited to predefined interactions. Instead, it should enable the description of custom interactions and be understandable by both designers and developers.

2.2 Boundary objects

Boundary objects [56] are shared artifacts used by different communities of practice to satisfy the *information needs* of each of them. Star was motivated "by a desire to analyze the nature of cooperative work in the absence of consensus" [57]. Boundary objects are both flexible in collaborative use and structured in individual use. A key feature of boundary objects is *interpretive flexibility* [50], i.e. the fact that the same boundary object can have different meanings for different communities of practice. For example, the ambiguity and incompleteness of a sketch [12] provide flexibility when collaborating. The same sketch can be precise enough to describe a particular design choice when practitioners work alone.

During the design of an interaction, the most common boundary objects are *design artifacts*. For example, Newman and Landay [47] analyzed the specificities of intermediate artifacts such as sitemaps, storyboards, mockups and prototypes. This network of boundary objects establish what Bowker and Star [6] call a *boundary infrastructure*.

However, few studies have focused on designer-developer collaboration with respect to design artifacts. An exception is Brown et al. [9] who established twelve categories of artifacts used between designers and developers. They found that regardless of the form of organization, all the designer-developer collaborations were mediated

[5]https://material.io/design/
[6]https://developer.apple.com/design/human-interface-guidelines
[7]https://www.invisionapp.com/
[8]http://www.uml.org

around the use of artifacts. In their study, the four most commonly used artifacts were *design ideas*, *design questions*, *interface proxies* and *stories*. However, the interactivity of these artifacts is limited or non existent, thus requiring designers and developers to envision the interaction instead of experiencing it.

2.3 Our approach

We argue that the tools used to create interactive representations, such as visual and functional prototypes, are not yet equipped to support the collaborative practices of designers and developers, especially when they create custom interactions. Therefore we seek to create new boundary objects, in the form of shared, interactive representations of the interaction being worked on. These new representations should be useful both for the collaborative process and the individual work of designers and developers.

In order to ground our work in existing practices, we started by conducting three studies to better understand the collaboration challenges of professional designers and developers. In the first study, we collected stories and identified three common designer-developer collaboration issues. In the second study, we observed how a group of designers and developers worked around these recurring breakdowns. In the third study, we conducted a participatory design session to explore the limitations of current interaction representations.

Based on these findings, we identified four design principles to reduce designer-developer breakdowns and facilitate the collaborative construction of custom interactions. In order to demonstrate the validity of these principles, we created Enact, a prototyping tool for touch-based mobile interaction based on these principles. Enact features novel representations of interactions that cater to the respective needs of designers and developers within a single environment. We conducted two studies to assess Enact: a structured observation of designers and developers using Enact, and a study comparing the use of Enact with traditional tools in a collaborative setting.

3 RELATED WORK

We grouped the related work on designer-developer collaboration in two areas. First, we review descriptive research focused on the processes and artifacts currently used during the designer-developer collaboration. Second, we review research on novel prototyping tools for creating interactive systems.

3.1 Studies of designer-developer practices

Researchers have studied collaboration among *communities of practice* [63], such as designers and developers of interactive systems, from multiple perspectives. Some seek to understand the implications of combining software development processes, such as agile methodologies[9], with design methodologies, such as user-centered design.

Although user-centered design methods for interactive systems emerged in the 1980s [48], their integration with software engineering processes is challenging. By the mid-1990s, Poltrock and Grudin [53] found that large corporations' *"development practices blocked the successful application of accepted principles of interface design"*. Agile methodologies were created to address a number of software engineering issues, with an emphasis on delivering working software as quickly as possible. However, Ferreira et al. [19] showed how this approach is often at odds with the user-centered design tradition of iterating the design from the user's perspective, before development begins. Letting developers understand what they are expected to implement, as soon as possible, remains one of the most important challenges of integrating these methodologies [54].

In the context of agile methodologies, Brown et al. [9] analyzed two major aspects of the designer-developer collaboration process: collaboration events and artifacts. Their study of collaboration events shows that designers and developers constantly perform alignment work [10] and that the collaboration process is organized around the use of artifacts [8]. However, the transition from design to implementation is not only a translation from

[9]http://www.agilemanifesto.org

sketches to dynamic behaviors [18]. Most of the joint work of designers and developers is focused on managing the tensions between them. Brown et al. [10] introduce the term *alignment work* as the activity of aligning designers and developers in order to seek, expose and resolve tensions. These tensions can result in *breakdowns*, which are considered to be a normal part of the collaborative work. However, even if breakdowns are commonly found in designer-developer work, we are interested in whether the limitations of the artifacts themselves are responsible or not for some of these breakdowns. Our goal is to identify current designer-developers breakdowns and propose actionable design principles that will help mitigate them.

3.2 Prototyping tools

Researchers have explored a number of novel tools for prototyping interactions, but most are targeted at a single community rather than the collaborative work of designers and developers. An exception is Apparition [35], a tool that helps designers prototype web-based systems in real time by crowdsourcing the creation of behaviors to workers, such as the position of objects in a simple platform video game. Lee et al. [37] studied the communication problems between designers and workers when using Apparation and found that designers "may speak in a style that makes it difficult for crowdworkers to comprehend their requests". Also, this system relies on the availability of crowdsourcing workers with the right expertise to fulfill the task at hand, which could be a problem when designing custom interactions and more complex prototypes. Other approaches, such as programming by demonstration [45] or the use of state machines and inference engines, have not been studied in a collaborative context.

Outside research, practitioners have acknowledged the gap between design artifacts and their subsequent implementation, leading to a number of commercial tools. Within the past few years, over 40 commercial prototyping tools have appeared[10], and a 2015 survey of 4,000 designers found that 53% of them use such tools [61]. These commercial tools focus on supporting remote communication between communities, assisting the extraction of design information, and helping to prototype standard and recurrent interactions. However, they do not support co-located collaboration, ignore the back-and-forth interplay between activities, and disregard the creation of custom interactions.

3.2.1 Tools for interaction developers. Code-oriented artifacts can be enhanced with other representations such as notations, diagrams and test-cases. For example, InterState [49] combines constraints and state machines to facilitate reuse. InterState provides a live editor where developers can edit a program and visualize the states as they interact with the interface. Proton [33] and Proton++ [32] let developers use Regular Expressions to express multi-touch interactions. Juxtapose [26] lets developers create code alternatives and modify variables at run-time to facilitate the exploration of multiple alternatives. CodePilot [62] supports novice programmers by integrating coding, testing, bug reporting, and version control management into a collaborative system.

While these tools are heavily inspired by developer practices, we believe that they provide representations and enhancements that could also be suitable for designers. We are interested in how these mechanisms might be adapted to meet the needs of both audiences.

3.2.2 Tools for interaction designers. Another approach focuses on augmenting traditional design artifacts. For example, SILK [34] lets designers quickly create interactions using interactive sketches and envisioned *"a future in which the user interface code will be generated by user interface designers using tools"*. More than 20 years later, however, the most common approaches still require the collaboration with developers to create user interface code.

DEMAIS [2] provides an interactive multimedia storyboard also based on sketches. The designers' strokes and text annotations are used as an input design vocabulary to transform static sketches into working examples.

[10] http://www.cooper.com/prototyping-tools

Similarly, FrameWire [38] infers interaction flows from paper-prototype videos to detect hot spots and generate page-based prototypes. Forsyth and Martin [20] use tagged digital storyboards to infer behavioral information, such as states and actions. These representations and mechanisms are closer to current practices of designers, and need to be present during the collaboration. However, we also want to bridge designer-friendly representations with more abstract representations, such as code, without excluding the participation of either community of practice [63].

While these tools support standard discrete interactions such as button clicks and menu selection, a few let designers prototype continuous, custom interactions. Monet [39] lets designers prototype continuous widgets by demonstrating interactions on top of sketches. Designers explicitly define interaction states and the system infers the correct state through multiple examples. Using inference improves informal prototyping, but these interaction descriptions are opaque and are of limited use to the developer for the final implementation.

Other tools use intermediate representations to allow non-programmers to create interactions. EventHurdle [30] is a visual authoring tool for prototyping gestural interactions designed to facilitate designers' understanding and generate code automatically. While we want to provide similar mechanisms to facilitate the transition from design to development, we also want developers to be part of the prototyping process. Our work is closer to d.tools [25], which brings test-driven development benefits to physical prototypes. d.tools lets designers rapidly test their design and analyze results, for example to identify the most frequently used interaction. Testing is a great intersection for the design and implementation of a prototype, and it can work as a "hinge" between the two activities.

3.2.3 Tools for interactive illustrators. Recently, several authors have proposed integrating graphical and symbolic vocabularies to create dynamic graphics. Kitty [29] is a dynamic drawing tool supporting the creation of animated scenes through functional relationships between graphical entities. Kitty relies on direct manipulation of graphics but also supports indirect manipulation of input-output functions. Apparatus [11] is a graphics editor that combines direct manipulation with data-flow programming. This combination of representations, with direct and indirect manipulation, enables users to think both spatially and symbolically.

While these tools provide authoring capabilities based on various representations, they focus on the creation of dynamic drawings, illustrations and diagrams, not on prototyping interactions. For this reason, user input follows a fixed path, e.g., a constrained drag and drop, and does not take into account all the input capabilities of the target device. Victor [60] proposes a tool that lets artists interactively create dynamic drawings with behavior simulations. The tool provides designer-friendly direct manipulation of graphics but also relies heavily on developer-friendly concepts such as linear algebra, parameterization and recursion. We are building on this trend of tools that bridge the worlds of visual output and dynamic input, but with a particular focus on the creation of interactions instead of dynamic graphics.

3.3 Summary

Designer-developer collaboration has been studied from two main perspectives: processes and artifacts. With respect to processes, practitioners and researchers both apply iterative methodologies for creating interactive systems. One of the challenges of this iterative process is to manage tensions and breakdowns in the collaboration. Some researchers see these breakdowns as a normal part of the collaborative work. We argue that we need to better understand these collaborative tensions to determine which can be mitigated by new design artifacts.

With respect to design artifacts, most of the literature describes their use in a collaborative context, but does not provide guidelines for building new collaborative tools that reduce breakdowns. Researchers proposed myriad prototyping tools, but they have been studied with a single community, either designers or developers. Involving

[11] Toby Schachman. 2015. Apparatus: A Hybrid Graphics Editor / Programming Environment. https://www.youtube.com/watch?v=i3Xack9ufYk.

designers and developers in the construction of interaction requires tools to support both design and development activities [13]. Such tools do not exist yet, to the best of our knowledge, especially for the collaborative creation of custom interactions.

Our goal is to create better tools that support designer-developer collaboration during the prototyping of custom interaction. To do so, we must first better understand how designers currently represent interactive behaviors to developers, as well as which aspects of these representations hinder collaboration[12].

4 STUDY ONE: UNDERSTANDING DESIGNER-DEVELOPER COLLABORATION PRACTICES

The goal of this study is to examine the existing practices of professional designers and developers in diverse settings with a particularly interested in:

- How *designers* represent and communicate a design;
- How *developers* interpret the design; and
- How *designers and developers* identify and overcome breakdowns that occur during the process.

We thus focus on understanding the difficulties that designers face when trying to express interaction to developers and the difficulties that developers face when trying to actually implement the designers' specifications.

4.1 Method

We conducted critical object interviews [41] about recent design projects in order to obtain specific, detailed stories of their successes and failures targeting the collaborative aspect of the work. We focus on their most recent projects and use recent artifacts to prompt rich stories. We are particularly interested in the problems they encounter when representing and communicating interactions with each other.

4.1.1 Participants. We recruited 16 professional designers and developers (7 women, ages 24-46) from France (8), Sweden (3), Argentina (2), the USA (1), Canada (1) and China (1), who create web sites, mobile applications or interactive installations. Their work environments include: digital agency (6), design studio (4), start-up (2), freelance (2), and software factory (1).

Participants $P1_{ds}$–$P8_{ds}$ are designers (ds), self-described as UX Designer, Visual Designer, Interaction Designer, or Graphic Designer. Participants $P9_{dv}$–$P16_{dv}$ are developers (dv), self-described as Mobile Developer, Web Developer, Front-End Developer, or Creative Coder. Their experience in collaborating across disciplines, i.e. from design to development or from development to design, ranges from 1.5 to 20 years (mean 8). Half of them typically collaborate remotely and none of the participants have worked with each other. All participants reported that they were following agile methodologies during their projects.

4.1.2 Procedure. We interviewed participants in their studio or office for approximately 90 minutes. We asked designers to choose recent projects in which they collaborated with a developer, and developers to choose recent work in which they collaborated with a designer. For each project, we asked them to show us their tools and the specific artifacts they created, and to describe, step-by-step, the details of how they communicated the design or implementation. We probed for both successful and unsuccessful collaboration examples.

4.1.3 Data Collection. We collected 25 stories (one or two per participant) from different projects. During the interviews, we recorded audio and video of the participants manipulating the artifacts they created. We also photographed the final products and took notes.

[12]The following three studies are presented in more detail in [43]

4.2 Results and Discussion

We analyzed the 25 project stories using Thematic Analysis [7]. We first illustrated each project with a StoryPortrait, first introduced in [27], to represent the most interesting stories from the interviews (see Figure 2). A StoryPortrait includes a summarizing title, a photograph of the interactive system or a key artifact created during the project, and a top-to-bottom timeline illustrating the key steps in the collaboration process with drawings and quotes. A vertical line separates the actions of the designers from those of the developers. Arrows crossing the timeline signify a handoff between the two groups.

We studied the projects with a particular focus on breakdowns related to creating or interpreting the design documents. Then, we selected examples that formed natural categories, looking for higher-level concepts that emerged from the details of each projects. We iterated and mapped each story to one or more categories. We identified three main issues encountered by the participants during designer-developer collaboration: **reworking and redundancy**, **design breakdowns** and **late developer involvement**.

4.2.1 ***Reworking and Redundancy.*** The first and most salient finding is the pervasive presence of reworking in the process and redundancy in the artifacts, not only within, but also across these communities.

Designers produce multiple design documents. Designers communicate with developers using three primary types of artifacts: *design documents* represent the system to be implemented; *guidelines* communicate higher level information (e.g. color codes, measurements) and *corrections* describe the misalignments with the envisioned design.

All designers create multiple documents to communicate different aspects of their designs. Designers create additional design documents when the original design documents lack specificity or lead to confusion. Unfortunately, much of the information in these extra documents is redundant. Designers felt they spent too much time recreating the same information across separate documents. For example, $P2_{ds}$ created five documents to communicate the design of a small application: UxPin *"for sharing mockups"*; Pixate *"for detailed animations that cannot be expressed with words"*; InVision *"for interactive mockups with basic interactions and annotations for non-obvious features"*; Photoshop because *"these developers are used to work with .psd files"*; and Illustrator, *"the software we actually use to produce the screens."* She also used email to communicate additional design details to the development team. Unfortunately, even by using all these documents, this designer was unable to clearly communicate the design.

In addition to using *images* of *"screens"* to represent the visual design of the interface, designers also used video or computer-generated animations (24% of the projects) to communicate custom and dynamic effects. However, designers take more time to produce videos than static images, and developers have difficulties manipulating these dynamic representations, e.g. extracting relevant information from them. Finally, designers occasionally create *interactive mockups* (12% of the projects) using the built-in set of interactions in the tool of choice. These communicate how the interaction should feel, but only when the tool has the right set of pre-defined interactions.

Developers recreate design documents. The most common activities mentioned by the developers include interpreting the design documents and recreating them with developer tools. For example, $P9_{dv}$ received an informal text description of a custom animation, but had to ask for a visual representation in order to fully understand the design.

We were surprised by the amount of time that developers spent recreating design documents. Some developers came up with interesting strategies to increase their productivity. One challenge is copying with different types of media: the designer's mock-ups, the code representation, and the current visual view of the system. For example, $P11_{dv}$ created a setup with two monitors. To implement the visual design, he places the mockups on the smaller screen to assess them and he splits the other screen between a coding view and the current state of the implemented website.

Similarly, when developing a mobile radio application, P14$_{dv}$ inserted the provided image as the background of her corresponding view in the IDE's Interface Builder. She then positioned the UI components on top of the image provided by the designer, to recreate the expected layout. She could then *"figure out the [layout] constraints"* of the screen to make it responsive, such as determining that some elements were center aligned.

Developers used different strategies to recreate the interactions described on the design documents. For example, in an interactive installation project, P10$_{dv}$ struggled to implement the animations provided by the motion designer (Figure 2). He first unsuccessfully tried to use *Adobe After Effects* to extract the keyframes and curve types. Then, he asked the designer to create a text file with the needed information. This was time consuming and boring for the designer: *"We try to set up standards, but we haven't found the right one so far"*.

Developers misinterpret designs. Many design decisions are lost, as developers struggle to interpret and implement the designer's original intent. In fact, none of the initial implementations exactly matched the original design. P1$_{dv}$ felt that the developer *"used our design as an inspiration, then he made many design decisions that he did not have to make"*. Similarly, P3$_{ds}$ provided a video that showed the developer how to vary a text-box color according to the background picture. He later realized that the developer had only partially implemented his idea by sampling a single pixel, instead of generating an average color based on several pixels.

During the implementation phase, designers create *correction documents* to show the location of the mismatches and what should be modified. For example, to correct a vertical misalignment, P3$_{ds}$ created a video. He first traced a segmented line to highlight the misalignment and then animated the correct repositioning of the elements. In the context of a real estate website project, P6$_{ds}$ discovered several visual mismatches including wrong margins, colors and fonts. Even though he was a designer, he decided to modify the CSS code and correct the mistakes by himself, using the web browser Developer Tools[13]. Because these changes were local to P6$_{ds}$'s browser, he screencaptured the new website's look and added some annotations linking the modified CSS code to the visual result. From these images, the developer then recreated all of these steps with his own tools.

Strategies to avoid rework and redundancy. We found cases of rework and redundancy in all the interviews, but only two developers and one designer explicitly mentioned strategies to avoid them. P14$_{dv}$ had a simple solution: *"Designers should just create their interfaces directly in Xcode"*, referencing the Interface Builder within the IDE. P5$_{ds}$ designed a complex casino website with many similar UI components. To avoid recreating them each time, she *"was inspired by the developers' way of working"*: she created a modular styleguide that served as a shared visual library. She could then copy modules from the styleguide to create each new screen, gradually adding new modules or missing information such as the color of the hyperlinks, as requested by the developers. P12$_{dv}$ set up a different approach. He initially received mockups and specifications for a web-based interactive advertisement builder. He built the architecture of the interface in Flash and he *"wrote the code so that the designer could very easily touch it"*. The designer was able to fine tune the look (e.g. the particular images assets to be used) and feel (e.g. type of animation, time delays and duration) by directly modifying the code. This workflow helped him avoid unnecessary translation and reproduction of the designer's intentions, reducing the back and forths.

4.2.2 ***Design Breakdowns***. The second main finding is the identification of three recurrent types of *design breakdowns* related exclusively to the collaboration between designers and developers (Figure 3). We use the term "design breakdown" to describe an impediment that must be fixed before the design can be implemented. We observed many design breakdowns such as misinterpretations, file formatting issues, mistakes introduced after modifying the original design, disagreement among the stakeholders, etc. Based on the 25 projects we analyzed, we found that the most frequent breakdowns could be categorized in three types: *missing information*, *edge cases*, and *technical constraints*.

[13]Developer Tools are a set of tools to inspect and modify the web content of a browser window

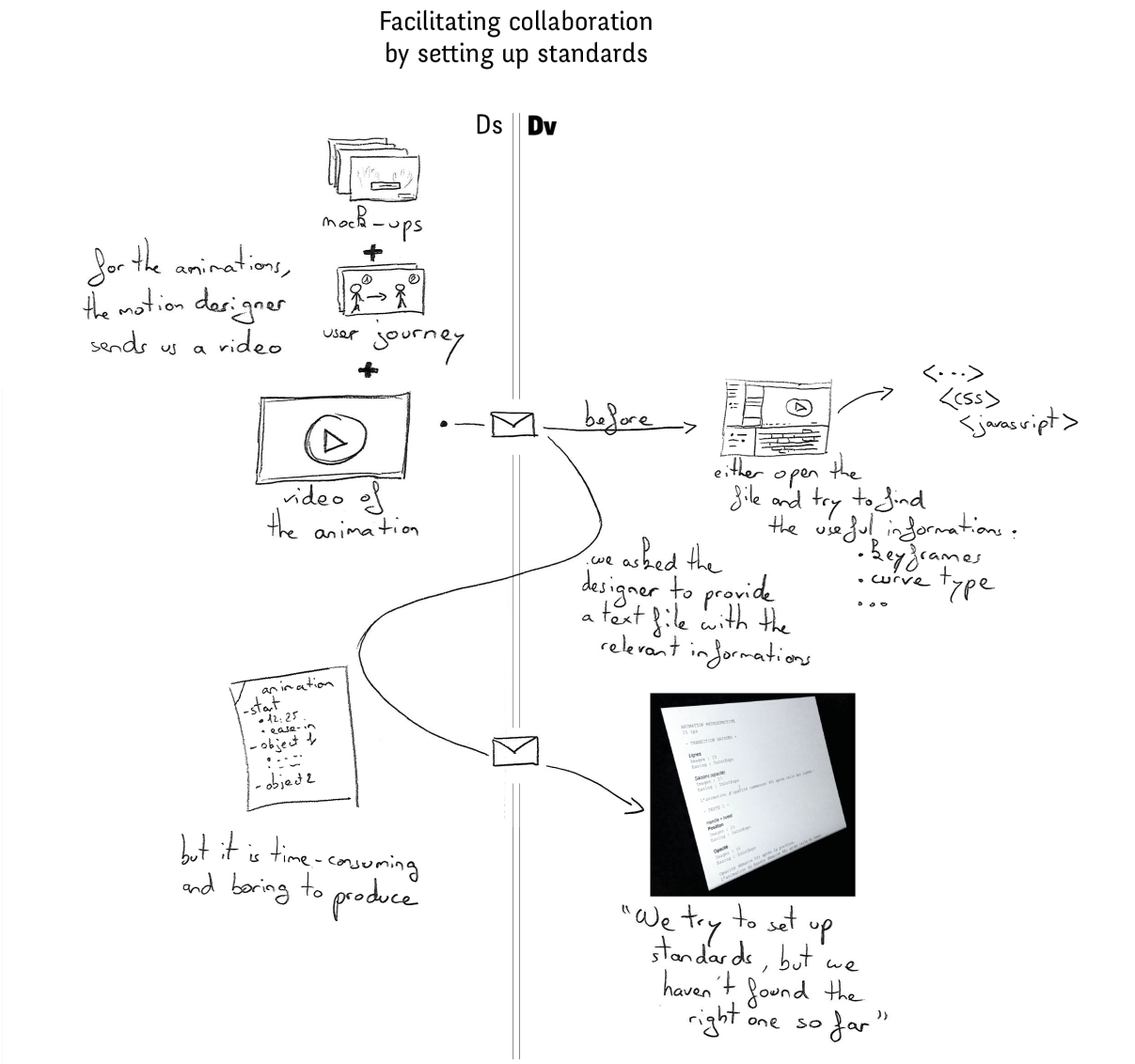

Fig. 2. A StoryPortrait has a summary title, a photograph of a key artifact, and a top-to-bottom timeline to show the successive steps of the collaboration between designers (on the left) and developers (on the right). Participant's quotes and drawings enrich the story. Here, P10$_{dv}$ started his story by saying that *"the motion designer sends us a video"*. The designer sent the mockup-up, user journey and video file via email. P10$_{dv}$ translated the video into code: *"[I] try to find the useful information"* in the provided artifacts. He failed to extract the relevant information about the animation, such as the keyframe timestamps and the interpolation curves. P10$_{dv}$ finally asked the designer for a text file with all the animation parameters. The designer sent by email a the text file created by hand with all the details of the animation. Creating this file was *"time-consuming and boring to produce"*. P10$_{dv}$ finished his story by saying: *"we try to set up standards, but we have not found the right one so far"*.

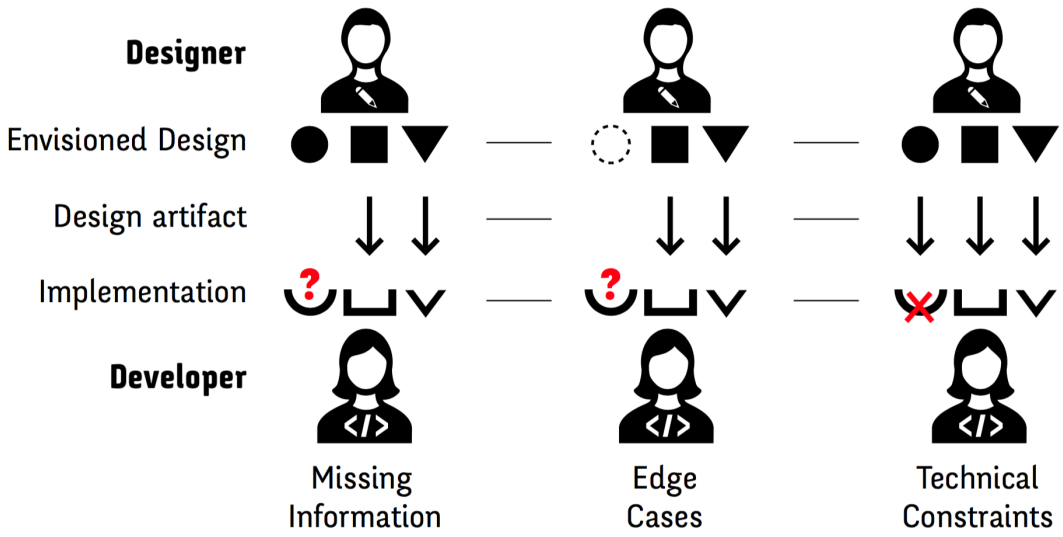

Fig. 3. Key design breakdowns between designers and developers: *Missing information*: Designers do not communicate necessary details. *Edge Cases*: Designers do not consider certain problematic situations. *Technical constraints*: Designers are not aware of technical limitations.

Missing Information. A *missing information* breakdown occurs when the designer makes a decision without communicating it to the developer. Two designers and four developers reported cases of *missing information*. For example, $P9_{dv}$ received an interactive mockup of a webpage. He could not determine whether the page's calendar widget was interactive or simply the output of another interaction. $P9_{dv}$ also lacked the design rationale: *"What did they create that calendar for?"* Similarly, $P12_{dv}$ received only static mockups for a sports application, and could not determine how to move from one screen to another.

Designers sometimes found it difficult to represent and communicate dynamic behavior to the developers. For example, $P8_{ds}$ wanted to create an animation of a blossoming flower but did not know how to represent her idea in After Effects. She ended up drawing a few sketches and then sat next to the developer as they worked out how to implement her idea directly in code.

Surprisingly, in two cases, the designers explicitly did not communicate the interaction at all, or only partially. They relied on the developer, especially because they wanted off-the-shelf interactions. This supports Myers et al. [46]'s argument that designers find interactions hard to represent. For example, $P6_{ds}$ provided static design documents without representing some transitions, even if they were simple: *"I let the developer pick the interaction between the screens, since they are very basic."*

Edge Cases. An *edge case* breakdown occurs when the designer only focuses on typical scenarios and does not consider extreme or problematic situations. Developers are trained to think about edge cases; designers are not. All developers reported that designers omit important edge cases from their design documents, and that they had to decide how to handle these situations themselves. $P13_{dv}$ received only mockups to develop a sports application. Because the designer had only specified the *"sunshine cases"*, $P13_{dv}$ had to make design decisions for each of the different edge cases. For example, the client asked him to include advertisements, so he modified

the original design to accommodate the ads. Similarly, $P16_{dv}$ prepared a responsive grid for a cruise company website. The original mockup only featured the desktop version of the website. $P16_{dv}$ did not know how to handle large elements that did not fit within the width of the screen of the mobile version: *"Should the rectangle be transformed into a square or should it take a full row?"* For $P16_{dv}$, designers usually *"don't take into account the dynamic nature of the data"*.

Some designers try to overcome these issues with design guidelines. For example, $P4_{ds}$ created a 16-page specification with annotated wireframes to explain the sign-up functionally of a website. She reported that *"specifications make me think of all the states and exceptions"*. She also used the guideline to capture and communicate the rationale for her design decisions. Sadly, the client's API did not support her design and she had to modify it. Similarly, $P3_{ds}$ created a spreadsheet to help him think and *"explain the rules of the game and the limits"* for each element in the website.

Technical Constraints. A *technical constraint* breakdown occurs when the designer lacks awareness of technical limitations, either in general or with respect to the developer's skills. Five designers and four developers reported breakdowns due to such misunderstandings, which created additional work for the developer. For example, $P13_{dv}$ received a design for an iPad application that called for horizontal scrolling when in portrait orientation. But $P13_{dv}$ *"could not recycle his code from the landscape version to create it"*. He had to reimplement it from scratch, since it had already been approved by the client. This type of misunderstanding leads developers to modify the design themselves.

Lack of awareness of technical constraints is also a problem for designers. For example, when working on a complex website, the developer first told $P6_{ds}$ that *"everything was possible"*. $P6_{ds}$ soon discovered that the developer was unable to implement many elements with his tools, even though they had already been validated with the client. $P6_{ds}$ said: *"He should have said it earlier, we would have adapted our design."* Instead, they were forced to redesign the project several times to accommodate the developer's limitations.

Collaboration is usually smoother when the designer is aware of the developer's constraints and possibilities. For example, $P5_{ds}$ worked on a project with two different developers. The first asked her to specify all the dimensions, such as the distances among all the elements on the screen, *"so we lost a lot of time"*. The second developer asked for a grid specification, which she created with 12 columns, a gutter size and a maximum size of 1200px. *"Now we have the same, each one in our own tool."* The grid allowed the developer to express dimensions in percentages, sparing $P5_{ds}$ the need to make additional annotations and saving a great deal of time.

4.2.3 ***Late developer involvement.*** The third finding is that late involvement of the developer has a negative impact on the collaboration, especially when creating custom interactions. Even though all participants claimed to use agile methodologies, only five of the 25 projects (two remote and three co-located) included face-to-face sessions between designers and developers, dedicated to co-design the initial interaction. For example, $P4_{ds}$ had an idea for a custom navigation rule so she drew a few ideas and invited all the designers and developers to help design it. The developer was able to implement the resulting navigation behavior without additional instructions or documents: *"Nothing was written down, we only had the screens."*

Other similar examples suggest that involving developers during the design phase makes it easier to create complex interactions (Figure 4). In such cases, developers gain an understanding of the desired interaction during the meeting and designers need not fully represent it in the design documents. Developers were most likely to be called in for the design phase when the project included custom interactions. In most of these cases (6/8), developers successfully implemented the desired custom interaction, as in the aforementioned example of $P8_{ds}$'s flower animation in 4.2.2.

However, when the project required a custom interaction and the developer was not involved at the design stage, most developers were not able to implement the proposed interaction (5/7). For example, $P7_{ds}$ reported that the developer *"just did not implement"* the custom transition he had proposed. One of the remaining cases

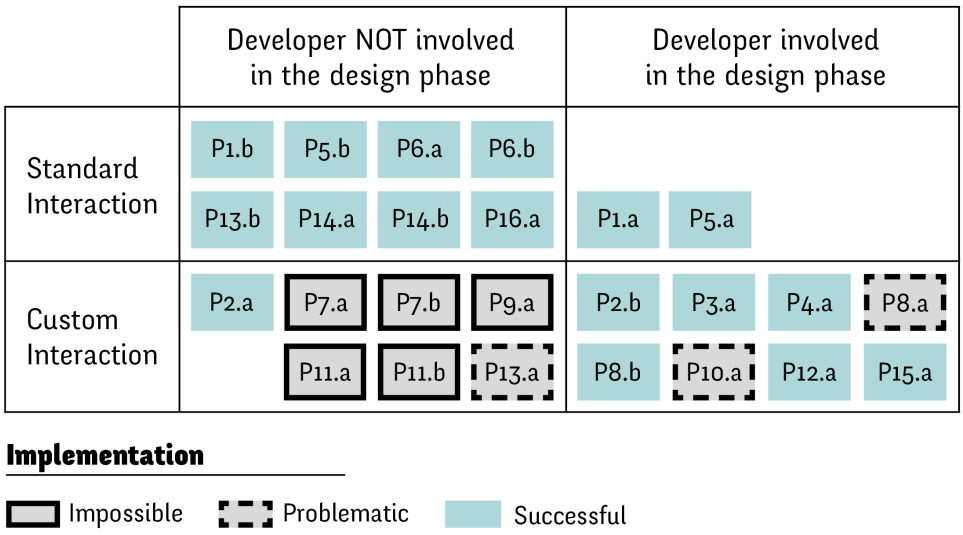

Fig. 4. Relationship between interaction type (standard vs. custom) and developer involvement. Lack of developer involvement in the early phase of custom interaction design is correlated with problematic or impossible implementation. Nomenclature: *P1.a* identifies the first story of participant one, while *P1.b* identifies the second story of participant one.

was still problematic: $P13_{dv}$ was frustrated with the proposed interaction: *"I could not recycle my code, but as the design had already been validated by the client I still had to implement it. I lost a lot of time."*

4.3 Summary

We identified three primary issues when designers and developers collaborate on the creation of interactive systems: reworking and redundancy, design breakdowns and late developer involvement. Both designers and developers spent too much time reworking, i.e. redoing previous work in the same or another representation, primarily due to redundancies within and across their artifacts. Designers struggle to represent interactions and dynamic behaviors with current tools and use multiple design documents to communicate different aspects of their design. Developers spend an excessive amount of time recreating the designer's documents and correcting their misinterpretations. During the implementation phase, developers face three types of design breakdowns — missing information, edge cases and technical constraints — that undermine the collaboration process. Our data also suggests that projects requiring custom interaction benefit greatly from the early involvement of the developer, during the design phase.

5 STUDY TWO: ANALYZING DESIGNER-DEVELOPER BREAKDOWNS

To further understand the breakdowns identified in the first study and how they are addressed, we conducted a case study of a team of designers and one developer. Unlike the first study, where interviews were based on the participants' recollection of recent projects, we observed a team during the entire duration of a one-month project. The goal of the project was to create a website for a crowd-sourced directory of companies. We were interested in whether design breakdowns still appear when a developer is involved early in the project, and, if so, which strategies are used to avoid or mitigate these breakdowns.

5.1 Method

5.1.1 Participants. We studied three designers and one developer (ages 24-25, one woman). This was the first time that this group of designers had collaborated with this developer. This grouping occurred naturally, the authors did not intervene in any of the details of the setup. One of the designers was $P1_{ds}$ from Study One.

5.1.2 Procedure. We observed the two face-to-face design meetings that involved all the designers and the developer. The first two-hour meeting (Figure 5) focused on the design of the website. The second meeting lasted an hour and focused on implementation. We also interviewed the designers separately, prior to the second meeting, to learn more about their design tools.

5.1.3 Data Collection. We video recorded both meetings and took notes. We took pictures of collaborative actions, i.e. exchanges between the designers and the developer, and their manipulation of artifacts such as drawings, notes and software.

5.2 Results and Discussion

We used Chronoviz [21] to annotate relevant, interesting events during the meetings. Two coders marked and analyzed the times when a participant asked a question, or when a designer sought confirmation from a developer or vice versa. We correlated these annotations with the classification of design breakdowns from Study One.

We focus our analysis on the two face-to-face meetings.

5.2.1 First Meeting - Accounting for design breakdowns. The main benefit of the early face-to-face meeting was to let participants seek validation from each other and to avoid potential problems. We identified examples of avoiding *missing information*, considering *edge cases*, and clarifying *technical constraints*.

In order to avoid *missing information* (12 occurrences), the developer often encouraged the designers to specify concrete details about their design ideas. The mere presence of the developer pushed the designers to be more explicit about certain design issues. The developer also pushed the designers to think about *edge cases* (5 occurrences).

Similarly, when the designers proposed adding a gesture for deselecting a category on the mobile version, the developer asked them to consider how this design decision would affect the desktop version of the website. Given the developer's warning, the designers decided to skip the feature. Designers often sought validation, confirmation or information about *technical constraints* (17 occurrences). This echoes the "considering implementability" category observed by Brown et al. [10].

In order to make informed decisions, they asked the developer about the complexity of implementing certain designs. When the designers proposed a search feature for companies, the developer asked them to specify exactly what should be searchable. The designers idea was to search within all company-related information, including their descriptions. The developer replied: *"Everything is possible... but if you really want to make a search inside the description, it will be a bit more complex."* He suggested only looking up names and tags, but with an autocompletion feature; the designers agreed.

5.2.2 Second Meeting - Fixing design breakdowns. Even though they were able to handle many design breakdowns during the first meeting, new ones appeared during the implementation process. The developer found new *edge cases* (4 occurrences). For example, he noticed that a company card with multiple subcategories would occlude the company's name. The designer responded: *"Maybe we can put three dots and display the extra ones only on [mouse] hover."* The developer also requested *missing information* that he could not infer (8 occurrences). For example, when reviewing the search feature, the designer asked for a clarification: *"In which order are the items shown when they are displayed as results?"* Designers also questioned some of the developer's decisions:

Fig. 5. First meeting. The developer interacts with an existing application to discuss possible interactions with two designers while another represents it on paper.

"Why do we need pagination?" The developer proposed an alternative: *"We should [be able to] put the maximum number of items on the page without loading problems."*

5.2.3 Vocabulary mismatch between designers and developers. In both meetings, differences in the vocabulary used by designers and developers led to miscommunication. Sometimes, designers and developers used different terms for the same concept. For example, during the second meeting the developer talked about a *"fixed"* element, referring to the CSS terminology. The designer, who tried to take the user's perspective, referred to the same object as a *"moving"* element, an element that scrolls with the page. It took some time for them to discover that they were talking about the same behavior.

We observed several strategies for overcoming these issues (5 occurrences). Developers and designers tried to bridge the vocabulary gap by adopting each other's terminology. For example, when discussing whether an item should appear in several categories, one designer started using mathematical concepts when communicating with the developer: *"Is it the union or the intersection of these two categories?"* The same designer gave a specific use-oriented story to explain their decision to the other designers: "imagine if you click here [on a category] ... and then if you click here [on another category], it deselects automatically that first one." The developer also reformulated the example in terms of UI widgets: *"It is either a radio button or a checkbox."* On several occasions, designers and developers looked up specific interaction techniques on a particular website or found examples from a mobile application on a smartphone to show the others. This "communication-by-example" helped them verify that they were talking about the same interaction technique.

5.3 Summary

Involving the developer at the beginning of the design process helped the team reduce the amount of *missing information*, handle *edge cases* and set clear *technical constraints* for the scope of the design. This echoes the recommendations of Salah et al. [54] about the benefits of developer early knowledge. However, new design breakdowns occurred during the implementation phase and had to be solved collaboratively. Vocabulary mismatches also created several collaboration issues, especially when discussing interactive behavior.

Most of the work of the designers and developer was alignment work [10], to manage tensions. The design breakdowns classification allowed us to analyze these tensions more finely. We found that both designers and developers actively try to mitigate design breakdowns when meeting face-to-face. However, it is still unclear which of these breakdowns are inherent to the designer-developer collaboration and which are a consequence of limitations in the design artifacts.

6 STUDY THREE: EXPLORING DESIGN SOLUTIONS

In order to find out whether some breakdowns could be alleviated by using appropriate representations of the design, we used a triangulation approach [42] and conducted a third, participatory design study. We were interested in whether design breakdowns are simply a natural result of the collaboration process, or if they are also by-products of the limitations in the representations used to describe interactive systems. Since these representations are traditionally the product of designers, we wanted to elicit new kinds of representations by asking designers and developers to create them together.

6.0.1 Participants. We recruited two designers and two developers (all men, ages 24-33). Two of them (one designer and one developer) participated in Study One and were invited due to their interest in this research. The other two professionals were recommended by participants from the previous studies. The developers had not previously worked with the designers. They had 1.5 to 10 years of experience collaborating across disciplines. Besides the four active participants, the authors of this paper attended the workshop: two as observers and two as participant-observers.

6.0.2 Procedure. The workshop lasted three hours and featured two activities designed to examine how designers represent and communicate *existing* custom interaction behaviors. We selected two unfamiliar interaction techniques from two mobile applications that rely heavily on continuous gestures. Participants were given the opportunity to explore these techniques for themselves on a mobile device we provided. The techniques were:

- *Pinch-to-create*: The `Clear` to-do list mobile app[14] uses a pinch out gesture to progressively split apart two items and create a new one between them; lifting the fingers creates the item (Figure 6).
- *Pan-and-stamp*: The `Paper` note-taking mobile application[15] uses a lasso technique to select an area of the canvas to be cut, which can then be moved with a panning gesture. While moving it, a tap with another finger copies it at that particular location.
- *Lasso-fill*: The `Paper` app also uses a lasso selection to specify an area to fill with a color previously selected. When the lasso crosses itself, the area is colored with the so-called even-odd winding rule, leading to unexpected results (Figure 8a shows a sketch of the interaction).

Activity 1 (1h): Designers and developers were divided into two pairs, grouped by roles. Designers received *pinch-to-create* and developers received *pan-and-stamp*. We made sure that none of the participants knew these interactions beforehand. We asked the designers to describe the interaction as they would ideally communicate it and asked the developers how they would like to receive a description from a designer. We asked them to give as complete a description as possible, and gave them access to all the tools and means they use in their daily work

[14]http://www.realmacsoftware.com/clear/
[15]http://www.fiftythree.com/paper

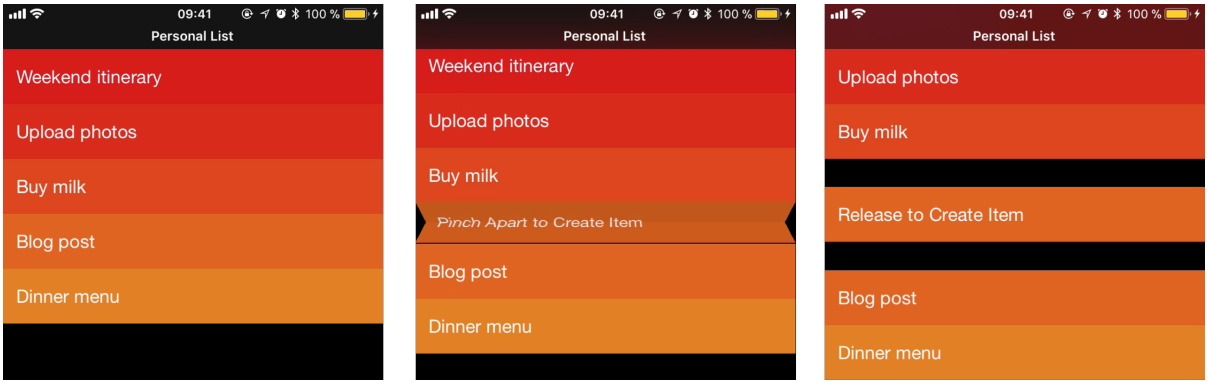

Fig. 6. The *pinch-to-create* interaction is based on the `Clear` to-do list mobile app. First the user puts down two fingers simultaneously. Then, by spreading them, the new item is revealed progressively. Finally, the new item is created when the user lifts her fingers off the screen.

practices. When participants were satisfied with their representation, they gave the resulting artifacts to the other pair. Each pair then tried to describe what they understood from the representation. We also asked them to try to find the interaction in the real application. Afterwards, participants discussed the issues they encountered when creating and interpreting the representations.

Activity 2 (2h): The two designer-developer pairs were shown the *lasso-fill* interaction. We asked each pair to come up with strategies or new representations that fully communicate the original interaction. We asked them to create representations that satisfy both members of the pair.

6.0.3 Data Collection. We collected all artifacts created by the participants: sketches, diagrams, text descriptions, paper prototypes and stop-motion videos. We took photographs and videos as they manipulated these artifacts, and took notes during the discussions.

6.1 Results and Discussion

The workshop provides clear evidence that current design artifacts are not adequate to provide complete descriptions and avoid design breakdowns. Designers and developers had to collaboratively combine different types of representations, such as sketches and diagrams, to create complete and general descriptions from concrete and specific examples.

6.1.1 Current representations do not encourage completeness. Participants took approximately 15 minutes each to create and be satisfied with their representations in the first activity. Even though they were given a fully functioning interaction and instructed to create complete descriptions, the four proposed representations were clearly incomplete. This suggests that some design breakdowns are a by-product of inadequate representations.

The designers relied primarily on visual representations based on drawings and annotations. Developers felt that these were effective in communicating the overall idea, but left too many unanswered questions for a correct implementation. For example, the designers did not communicate certain types of feedback, such as the gesture spread threshold or the animated transition that placed new items at the top of the list. During the discussion, one of the developers explained that a picture requires more translation steps than a text description for implementation: *"If I receive a picture, I first need to translate it into text and then I need to translate it into code."*

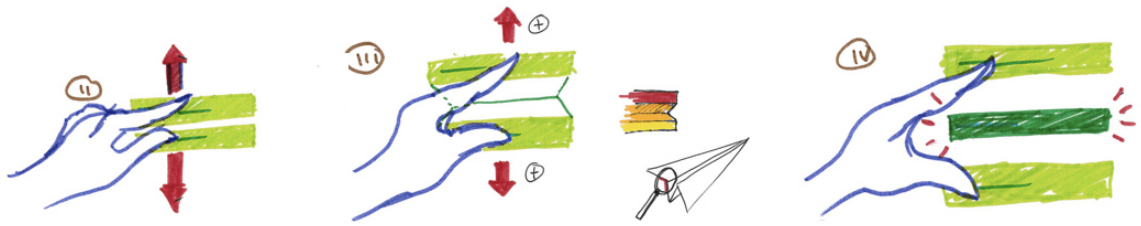

Fig. 7. An example of a designer's representation of *pinch-to-create*. The designer depicts a continuous interaction by discretizing key visual states and adding user input annotations.

Developers relied mainly on text, including programming vocabulary such as conditionals and loops, complemented by a few visual elements. Text descriptions provided specific information for the implementation but did not clearly convey the look and feel of the interaction to the designers. For example, when trying to represent the *pan-and-stamp* interaction, developers did not communicate the increased opacity outside the selection and the flash effect when pasting the copied area.

6.1.2 Strategies for creating complete representations. During activity 2, the two pairs explored seven different strategies to fully represent and communicate the interaction. The most promising strategies were:

(1) Pair 1 decomposed the *lasso-fill* interaction using examples from other applications: the lasso tool from Photoshop combined with the paint bucket from Illustrator. The designer from Pair 1 proposed recording videos to demonstrate the use of these tools and combine them. He argued that this strategy would avoid misunderstandings as well as provide a complete description of the interaction.
(2) The developer from Pair 1 proposed a shared "lexicon" describing the objects of the program, their characteristics and the tools that can interact with them. Pair 1 thought that a common vocabulary would facilitate the discussion about how to extract common components.

In order to reach a shared and complete representation, both pairs refined their representations through multiple iterations. They started with a visual example, and then added rules and annotations to produce a more complete description. For example, the designer from Pair 1 drew a snapshot of the interaction at four points in time: touch, move, release, and create closed shape (Figure 8a). Next, the developers and designers collaborated to gradually generalize the description of the interaction. The developer, inspired by the designer's representation and his knowledge of "flow programming", drew a diagram representing the different components of the interaction (Figure 8b): finger, shape, line and closed shape. Based on the developer's representation, the designer built a new representation that combined the strengths of both proposals (Figure 8c). He color-coded a visual example and mapped each graphical element to a detailed description of the expected behavior.

6.2 Summary

Even when provided with an existing and complete interaction, both designers' and developers' representations suffered from *missing information* and *edge cases*. This suggests that current representations are limited and may result in design breakdowns. To create complete representations, designers and developers gradually added rules in addition to concrete examples. This way of working helped designers and developers to mitigate breakdowns. We argue that computer-based prototyping tools supporting this workflow between designers and developers could also mitigate collaborative breakdowns.

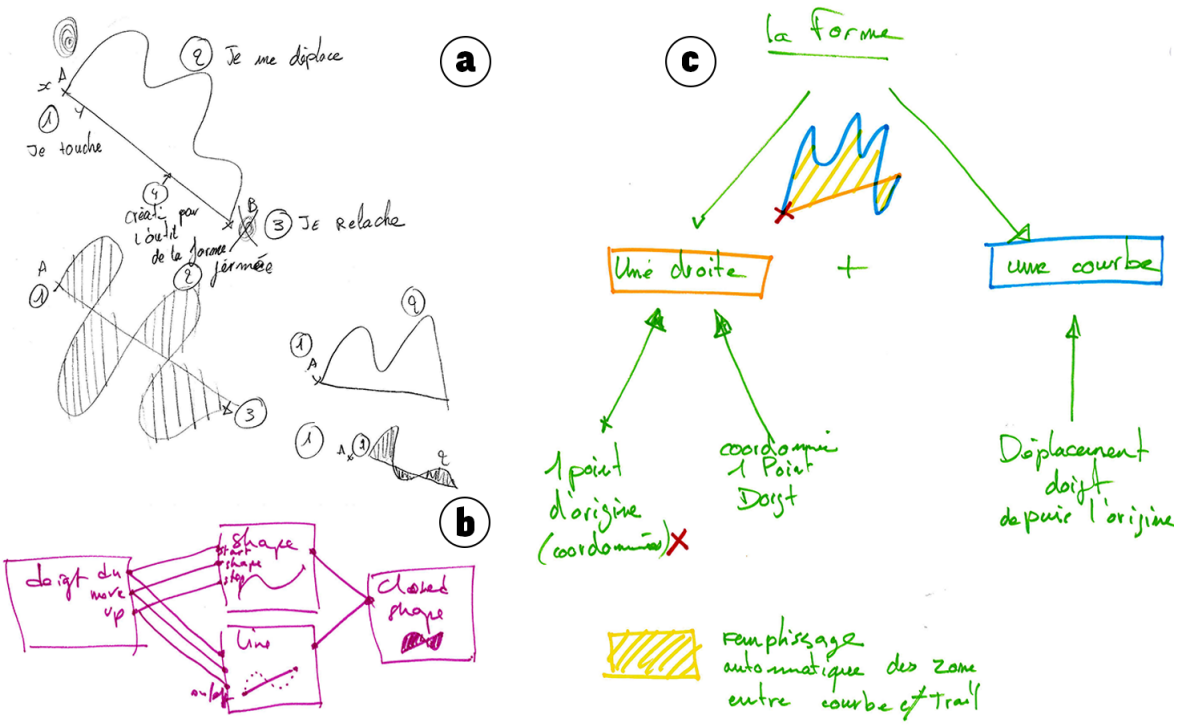

Fig. 8. (a) A designer drew a snapshot of the lasso-fill interaction at four points in time. (b) A developer created a diagram connecting primitive graphical elements and functions with user inputs. (c) The designer merged the two representations with an example.

7 DESIGN PRINCIPLES FOR DESIGNER-DEVELOPER COLLABORATIVE TOOLS

These three studies show that breakdowns occur routinely during designer-developer collaboration and can result in severe mismatches between the system originally envisioned by the designers and its final implementation. To avoid such consequences, we should help designers and developers identify these breakdowns and solve them as early as possible.

A key source of designer-developer breakdowns is the lack of an infrastructure supporting the *boundary objects* used to represent interaction. Designer tools generate isolated representations that are rarely updated when the system implementation changes. Designers are also responsible for manually keeping the design consistent across multiple design documents, often in different formats, e.g., images, diagrams and video. By contrast, developer tools work with precise and "executable" representations. These representations govern the actual system and developers must ensure that they are up-to-date with the expected design. In such a workflow, when changes emerge during iterations, the design specification is rarely materialized outside of the final implementation due to the high costs of updating multiple and unconnected representations. We believe that tools for the design and implementation of interactive systems must accommodate the skills, values and practices of both designers and developers, and support an integrated workflow.

Based on these findings, we propose four principles for the design of computer-based tools that support designer-developer collaborative prototyping and reduces breakdowns:

(1) Provide multiple viewpoints;
(2) Maintain a single source of truth;
(3) Reveal the invisible; and
(4) Support design by enaction.

These design principles address the following questions:

- How can we encourage early participation by developers?
- How can we reduce reworking and redundancy?
- How can we mitigate design breakdowns?
- How can we support a workflow from example-based to rule-based descriptions?

7.1 Principle One: Provide multiple viewpoints

A collaborative prototyping tool should provide multiple representations of the interaction in order to support the different viewpoints of each community of practice involved in the process.

In order to encourage *early developer involvement* in the design process, we need to provide a designer-developer "sandbox" that supports both designer- and developer-friendly representations for manipulating the system under construction. In other words, if we want designers and developers to, e.g., "sketch" an interaction together, we should embrace the needs and perspectives of both communities. In the following, we use the word *viewpoint* to characterize a representation of the design targeted to a particular community of practice or activity.

Today's design and development practices do not encourage the use of multiple viewpoints concurrently. Our studies show that in the early stages of a project, tools tend to focus on rapid prototyping, disregarding the developer's view; in later stages, tools tend to focus on development, disregarding the designer's view. From the start and throughout the entire design process, designers and developers should be able to manipulate the interactive system under construction with their own representations and tools. By providing multiple viewpoints, we can leverage the existing skills and knowledge of both designers and developers.

These viewpoints should provide rich interactions to manipulate the design, similar to those in the current individual tools. For constructing the user interface, IDEs such as Eclipse or Xcode feature a user interface builder as their "design view" and a code editor as their "code view". However, they provide limited authoring capabilities compared to existing vector and raster graphics software. UI builders focus on assembling predefined elements, dragged from a catalog of interface components, rather than creating or exploring custom interactions. These "design views" are used differently than graphical authoring tools; designers can only assemble canned interfaces within a constrained environment, while graphical authoring tools let them explore new ideas in an unrestricted environment.

During the workshop, designers and developers started with specific, concrete examples, and then, through several iterations, generalized these examples to create more abstract representations. This suggests that designers and developers would benefit from *viewpoints at different levels of abstraction*: some more concrete to represent examples and some more abstract to describe rules. We believe that the ability to transition from specific examples to higher-level abstractions would help designers create more complete descriptions.

7.2 Principle Two: Maintain a single source of truth

A collaborative prototyping tool should keep the multiple representations of the interaction synchronized, i.e. changes in one representation should be reflected in the others as quickly as possible.

In order to reduce *rework* and *redundancy*, prototyping tools should not only provide multiple viewpoints, but also keep these representations in sync. The previous studies showed the fluid and iterative nature of designer-developer work, and the need to modify the representations during the process ($P4_{ds}$, $P13_{dv}$, $P16_{dv}$). Some graphical tools support "symbols" or "smart objects" that are referenced across documents instead of being copied.

Designers can modify these smart objects and see the changes reflected wherever they are used, alleviating the need for manual synchronization. However, while this encourages modularity it also requires planning: "smart objects" must be created before being used. Such planning is at odds with the fluidity of the design and implementation process.

We argue that design tools should support objects that are consistent across representations, minimizing the need for planning. To foster collaboration and understanding across activities, changes in one representation should be instantly reflected in the others, similar to live programming [59]. One strategy is to use automatic transformations among representations. For example and when possible, changes to a designer-friendly representation would be made available to implementation representations and vice versa. This approach encourages reuse across activities and reduces redundancy by leveraging shared entities.

In practice however, some information might not be transformable among representations. For example, a representation focused on the *look* might not be able to represent the *behavior* of an interaction. Also, it is usually impossible to transform among representations working at different levels of abstractions. In such cases, users should be notified that changes have been made, even if they cannot be automatically transformed into the target representation. In addition, the system should help users regain a synchronized state.

7.3 Principle Three: Reveal the invisible

A collaborative prototyping tool should reveal information only available in one representation in the other representations; when not possible, intermediate representations should be provided.

In order to reduce cases of *missing information* it is not enough to maintain multiple viewpoints synchronized. Unlike in current workflows, where some elements are only visible and manipulable in certain representations, we need to augment or create new intermediate representations to expose information that is not available in existing design artifacts.

In our studies, developers often lacked access to important elements of the design such as measures ($P5_{ds}$), colors ($P6_{ds}$) or animation time values ($P7_{ds}$). Some of these elements were available in design documents, but developers did not have access to them and had to ask the designer to send them explicitly ($P5_{ds}$) or in a specific format ($P2_{ds}$). In some cases, the visual elements used to communicate extra information, such as annotations, measures or user inputs, were misinterpreted as actual visual elements that were part of the user interface. For example, in the second study the developer misinterpreted a cross in the corner of a widget for an annotation, when in fact it was a close button. Prototyping tools should reveal this design information to the developers without adding noise to the design being communicated.

Conversely, designers lacked access to information hidden in the code. For example, measures or equations used as parameters of an interaction exist in code but do not have a manipulable visual counterpart in the design representations, therefore designers cannot modify them by themselves. Even worse, designers needed to produce extra documents to communicate the appropriate request to the developer. Prototyping tools should reveal these interaction parameters to the designers in a form compatible with their viewpoints.

Ideally, all entities and concepts should be available in all representations. When this is not possible, hints or links should be made explicit to understand these invisible relationships among representations. For example, *missing information* could be more easily detected when a design decision is not available on the developer's view. Developer concepts such as parameters and conditions should be manipulable by the designer, especially if they are linked to concrete aspects of the design, such as distances, relative positions or extreme values.

7.4 Principle Four: Support design by enaction

A collaborative prototyping tool should support the design and implementation of an interaction through enactments of the interaction as a rapid, active and contextualized medium for design.

In order to find *edge cases*, the prototyping environment should let designers and developers experience the interaction being created in context as early as possible. In the same way as we informally communicate an interaction idea by gesturing with our hands, computer-based tools should take as inputs enactments of the interaction performed with our own body, including hand gestures, voice commands or gaze.

According to Bruner:

> "Any domain of knowledge [...] can be represented in three ways: by a set of actions appropriate for achieving a certain result (enactive representation); by a set of summary images or graphics that stand for a concept without defining it fully (iconic representation); and by a set of symbolic or logical propositions drawn from a symbolic system that is governed by rules or laws for forming and transforming propositions (symbolic representation)" —Bruner, 1966 [11].

Victor [60] calls these three modes of representation *interactive* (enactive), *visual* (iconic) and *symbolic*. In our studies, we observed that designers create visual representations of interactions and dynamic behaviors as screen flows with different levels of precision, such as sketches, wireframes or mockups. Designers typically describe how the system reacts to an imprecisely specified user input. Based on these designs, developers create symbolic representations of the interaction in the form of code that they run in the target environment, or in an emulator. Developers then assess the extent to which the implementation follows the specified design.

This process restricts the use of interactive devices for testing, rather than a full-fledge design medium. We argue that interactivity should be used during the entire design and development process, not only at the testing stage. For example, in the second study, the developer interacted with existing mobile applications to show different design possibilities to the designers. In the first study, $P4_{ds}$ organized a co-creation session so that she and the developers could "act" over some drawings to show how the interaction should behave.

User inputs, not just visual outputs, should be manipulable. They should be created in context on the target environment to inform the design and encourage "learning by doing". Also, the relationships created between user inputs and visual outputs should be "tweakable" and explorable. In all three studies, we observed that designers were less likely than developers to specify *edge cases*, even though their early identification can avoid significant problems later on. However, they paid more attention to *edge cases* during the workshop, when designers and developers were encouraged to work with the target device.

Current practices do not encourage enaction: designers typically work during the entire process with graphical editors and do not explore their design on the final device until an implementation starts working. *Supporting design by enaction* lets creators play an active role as the final user; finding *edge cases* becomes part of the construction itself.

7.5 Summary

The goal of these four principles is to minimize designer-developer breakdowns due to the limitations of their design artifacts:

- *Provide multiple viewpoints* of the designers and developers needs and practices, to help developers participate early in the design process;
- *Maintain a single source of truth* to keep all these viewpoints in sync and reduce *reworking* and *redundancies*;
- *Reveal the invisible* information, either hidden from the designers in the code or sheltered from the developers in the design artifacts, to avoid *missing information* in the design descriptions; and
- *Design by enaction* to find more *edge cases* by experiencing the design in context as soon as possible.

These design principles are also tightly interconnected. Each principle targets a particular breakdown but can also help mitigate other breakdowns. For example, *providing multiple viewpoints* encourages the early participation of developers, and should also reduce *missing information*. *Maintaining a single source of truth* reduces reworking and redundancy, which should also facilitate early developer participation. *Revealing the invisible* reduces *missing*

information, and can also make *edge case* more visible. Finally, *supporting design by enaction* can help find *edge cases*, but can also reveal *technical constraints* by reducing the time-to-interaction.

Taken together, these principles lead to the definition of a *network* of inter-related design artifacts for prototyping interactive systems, instead of the current situation where the design artifacts are loosely connected. However, we acknowledge that applying these principles is not sufficient to guarantee successful collaborative work. Indeed, some designer-developer breakdowns are not directly related to the artifacts that they use: Interpersonal relationships, different styles of communication and technical knowledge, just to mention a few, also have an important impact on designer-developer collaboration. Nevertheless, these design principles, which are grounded in our in-depth studies, can help researchers and practitioners create new prototyping tools that significantly reduce avoidable breakdowns, as we now illustrate.

8 ENACT: A TOOL FOR COLLABORATIVE PROTOTYPING OF TOUCH-BASED INTERACTIONS

In order to validate the above design principles, we need to use them in real collaborative scenarios. One option is to start with existing designer and developer tools, but this requires integrating proprietary software from different vendors and potentially incompatible data representations. Instead, we chose to create a new tool, called Enact, which gave us more freedom to explore the use of the design principles in an unrestricted environment.

8.1 Overview

Enact is a live environment for prototyping custom touch-based interactions that supports collaboration between designers and developers. Among the many interaction styles that can benefit from collaborative support, we decided to focus on multi-touch interaction for mobile devices. As we have observed in the participatory workshop, multi-touch interactions are familiar yet challenging for professionals to describe and prototype. Enact lets users work graphically and with concrete examples, but also exposes low-level touch events through an interactive state machine.

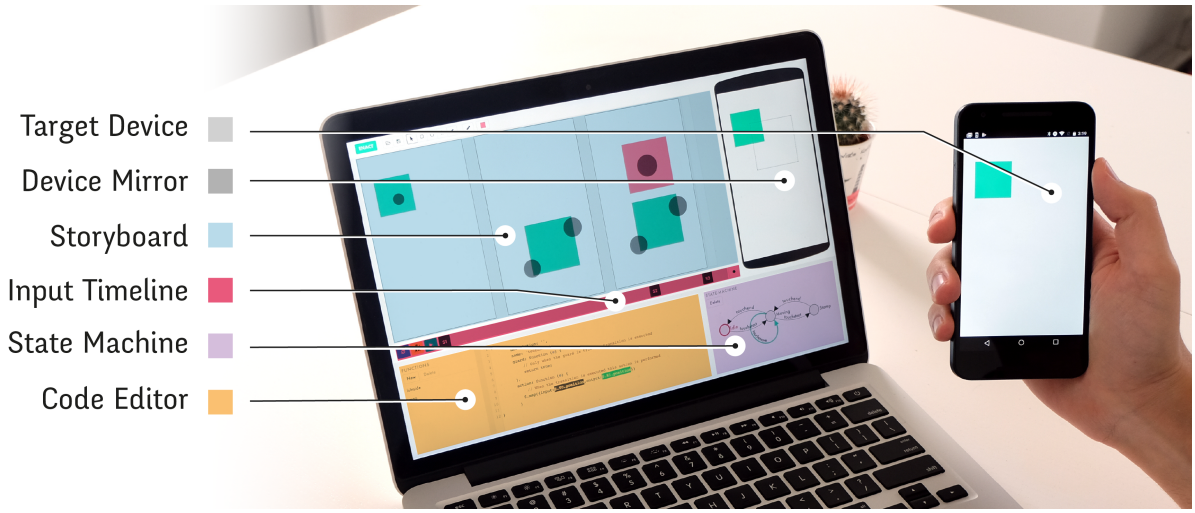

Fig. 9. Enact uses a target mobile device and a desktop interface with five areas: a storyboard with consecutive screens, an event timeline with a handle for each screen, a state machine, a code editor and a device mirror

There are many aspects of the construction of an interactive system that are outside the scope of Enact. Our focus is on supporting designer-developer collaboration when creating custom interactions rather than supporting the creation of a complete interactive system. Therefore Enact does not cover back-end integration for networking, storage, application distribution, etc. It also does not have means to specify the navigation among multiple screens. While Enact could be extended to support these features and allow the creation of full-fledged touch-based interfaces, the current version only supports the definition of a single interaction on a single screen.

Enact addresses the main collaboration issues identified in the first three studies by applying the four previously described design principles:

- The desktop interface is organized in five areas providing *multiple viewpoints* of the interaction (Figure 9): a storyboard, an input timeline, a state machine, a code editor and a device mirror;
- The *single source of truth* is the interaction under construction, maintained automatically across viewpoints or manually with the assisted testing;
- The device mirror *reveals invisible* information such as measures and touches; and
- The target device supports *design by enaction* by letting users perform gestures and test the interaction in real time.

We first illustrate Enact through a use scenario. Then, we describe the main features of Enact and provide their design rationale.

8.2 Use scenario

Anton, a designer, and Petra, a developer, want to prototype a custom interaction to create items in a to-do list application using a continuous spread gesture, or "pinch apart", i.e. the reverse of a pinch gesture (Figure 6).

8.2.1 Representing the visual design. To communicate the visual design to Petra, Anton first draws the *look* of the interface at different stages, similar to the sketches presented in the third study (Figure 7).

Anton starts by drawing the to-do items as rectangles in the first screen of the storyboard. When Anton adds a second screen, Enact duplicates all the elements from the previous screen, a feature called storyboard automatic propagation (Figure 10a). In the second screen, Anton adds a 6-vertex polygon as the new item and adjusts its size to fill the space between the two existing items. He then creates a third screen where he moves the items apart and increases the size of the new item in between by modifying the vertices. Finally, Anton decides to tweak the color and the size of the first two items. He only needs to modify these in the first screen because changes automatically propagate to the following screens. This propagation reduces redundancies and ensures consistency across the design.

8.2.2 Specifying user inputs. To communicate the interaction design to Petra, Anton records a spread gesture directly on the mobile device, mimicking how the user should interact with the system (Figure 11). The recorded input events are stored in the timeline which features a handle for each screen in the storyboard (labelled S1, S2 and S3 in Figure 10b). Anton can move the screen handles along the timeline to position the screen at the moment in time that corresponds to the right distance between the finger touches of the recorded gesture. He can also adjust the items in screen S2 to better match the gesture. Since these items are in a different position in screen S3, the automatic propagation does not apply. By interacting with the device and by seeing the user inputs on top of the screens, Anton can make informed decisions about the layout of the interface with respect to the position of the user touches.

8.2.3 Making the design interactive. Petra receives Anton's design and examines the screens. Enact automatically generates an animation combining the storyboard screens and the recorded user inputs. She watches the animation to better understand the behavior of the interface.

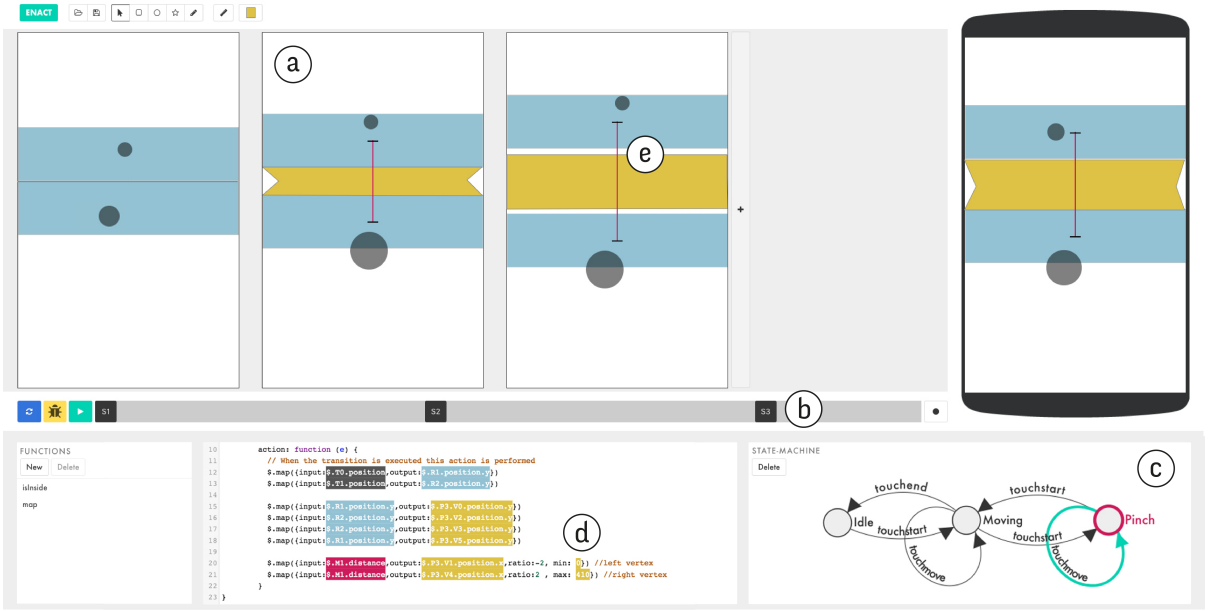

Fig. 10. Scenario. The first screen of the storyboard shows the initial look of the interaction. The two items created in the first screen (light blue) propagate to the second screen, where a third item (in yellow) is added (a). Each screen has a corresponding handle in the input timeline (b). The state machine shows the selected transition in green and the active state in red (c). State machine actions can be edited in the code editor (d), either by coding or by linking a storyboard elements to create *design references*. In the third screen of the storyboard, a measure `M1` is added between the top and bottom rectangle (e).

Enact uses a state machine [1, 49] to specify the interaction behavior. When creating a new interaction, a default state machine is provided. As Petra interacts with the target device, she can visualize the transitions among states in real time. She decides to add a new `Pinch` state to create the custom pinch apart gesture (Figure 10c). During the pinch, the user touches should control the position of the top and bottom items. To do so, Petra selects the `touchmove` transition of the `Pinch` state and programs a new action in the code editor (Figure 10d). She can write code but she can also drag touches, visual elements, and properties from the storyboard to automatically insert a reference to these objects, called a *design reference*. After adding this new action, Petra can directly test the interaction on the mobile device.

8.2.4 Revisiting the interaction. Petra notices that the new item, in the middle of the other two, is not being resized. To control the positions of the new item's vertices, she needs to know the distance between the top and bottom items. Instead of writing an equation in the code editor, Petra creates a *measure* (`M1`) in the storyboard: She drags a line between the two original items with the measurement tool (Figure 10e). Now, she can drag the new measure into the code to create two *mappings* between `M1`'s distance and the positions of the new item's vertices. She is satisfied with the result and sends the design back to Anton.

Anton runs the assisted test, i.e. an automatic comparison of the storyboard elements with the result obtained after running the code with the recorded inputs, and notices that the new to-do list item grows beyond the width of the previous items. Anton decides to fix the implementation himself. He adds a maximum value to the second mapping created by Petra. Anton drags the values from the third screen, and drops them as maximum values in

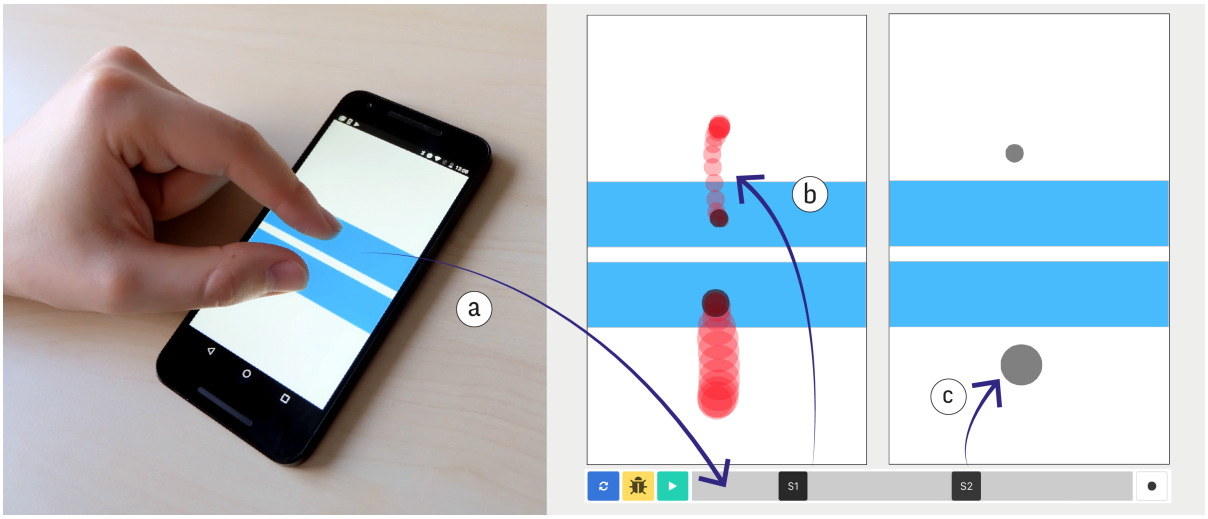

Fig. 11. Recording an input on the target device. The input events are saved in the timeline (a), the designer can navigate the recorded inputs by dragging them or by moving the associated screen handle in the timeline (b). Current touches are displayed as translucent grey ellipses on each screen, the radius of the ellipse represents the size of the touch (c).

the code (Figure 10d). Since these values are now bound to screen elements, Anton can tweak the size directly in the storyboard instead of modifying the code by hand.

8.3 Drawing the user interface

The initial studies showed that designers generally depict the different visual states of the interaction with sketches (Figure 7), diagrams, wireframes or mockups. In the first study, we found that designers needed to maintain consistency among screens manually, e.g. with copy-paste, therefore introducing redundancy in the artworks that later turned into mismatches or inconsistencies.

In Enact's storyboard, each screen is aware of its past and future screens (Figure 10a). Following the *maintain a single source of truth* principle and in contrast to "smart objects", Enact's propagation mechanism is active by default. Whenever a designer creates a new shape or changes a property of an object, that change propagates to all future states. Creating a new screen automatically imports the shapes and objects from the previous screen. Changes include transformations, such as moving, resizing, or changing colors. A change in, e.g., $Screen_2$ propagates to subsequent screens ($Screen_{3,4,...}$), but also stops propagation from $Screen_1$. To reactivate propagation for an object, the user simply needs to modify it in the screen where it was changed to match the same object in the previous screen. Propagation reduces the amount of redundant information across screens, without modifying current designers' practices.

8.4 Providing concrete input examples

The first study showed how designers generally describe user input as annotations on top of the visual design. Following the *support design by enaction* principle, we wanted to generate this information on the fly from actual interactions rather than using static annotations drawn by the designer. Unlike current design tools, Enact lets designers associate screens with actual user input events, recorded directly on the target device. This approach is similar to "programming with examples" [28]. First, the designer presses the record button at the right of the

input timeline (Figure 11). While the designer performs the desired input on the target device, real-time feedback appears on the device mirror. Once all touches have ended, the recorded input events are saved in the timeline and associated with the storyboard's screens.

Touch events are treated as first-class objects: They can be displayed and manipulated like other graphical elements. This builds on the existing design practices of annotating sketches with gestures, but also provides new capabilities. For example, touch inputs can be used to position other graphical objects or to compare relative sizes. This helps designers better understand the *technical constraints* of designing for mobile, e.g. to determine if an object is big enough to be touched reliably.

8.5 Generating an animation from the storyboard

The first study showed that designers rely on animations or videos to describe custom interactions. Animation is a simple medium for illustrating an interaction, but designers currently need specialized tools to create them. In order to *provide multiple viewpoints*, Enact automatically generates animation descriptions based on the storyboard and the recorded input example. Therefore, designers can create simple animations in Enact by reusing the storyboard instead of creating extra documents.

Since each screen is associated with an input event, Enact knows the time between screens and can therefore animate the visual changes by using the screens as keyframes. The current implementation uses linear interpolation, but other interpolation functions could be added.

After pressing the play button, the animation is executed on the target device and the mirror, displaying both the visual objects, the touch events and any created measure. These lightweight animations let designers check the timing between the recorded touch input and the screens in the context of a real device. While these animations are not interactive, they provide a stepping stone towards creating an actual interaction, which requires programming mappings between user inputs and screen outputs.

8.6 Programming interaction

The initial studies showed that the traditional dichotomy between "design view" and "code view" is not enough to articulate designer-developer collaborative work. Rather than forcing designers and developers to abandon their practices, we want to augment them. Developers frequently use diagrams to describe an interaction and think in terms of states. However, this is rarely reflected in the tools they use. Enact uses an interactive state machine to represent the code of an interaction. The state machine *reveals invisible* details that are usually buried in the source code. It also organizes the interaction code as a sequence of transitions that occur over time, as opposed to the traditional set of event handlers organized by event type.

Enact also leverages the visual representations in the storyboard to support programming. Developers can use *design references* to connect interface elements or specific values between the storyboard and the code. They can also create *dynamic measures* by direct manipulation instead of defining symbolic formulas in the code.

8.6.1 State Machine. The goal of Enact is to let designers and developers create custom touch-based interactions. Implementing such interactions typically requires processing low-level touch events, which is tedious and error-prone [31]. Instead, Enact organizes the interaction code around a state machine that exposes touch events as transitions (Figure 12).

Previous work has demonstrated the power of state machines to describe interactions [1, 49]. State machines also *provide multiple viewpoints*: the high-level logic of the state machine, described by its graph, and the low-level details of transitions and actions, programmed in code. These viewpoints provide representations that both designers and developers can manipulate. State machines also gather an entire interaction within a single object, providing a single *source of truth*. Finally, Enact highlights the active state and transition in real time when interacting with the mobile device, *revealing the invisible* inner working of the interaction.

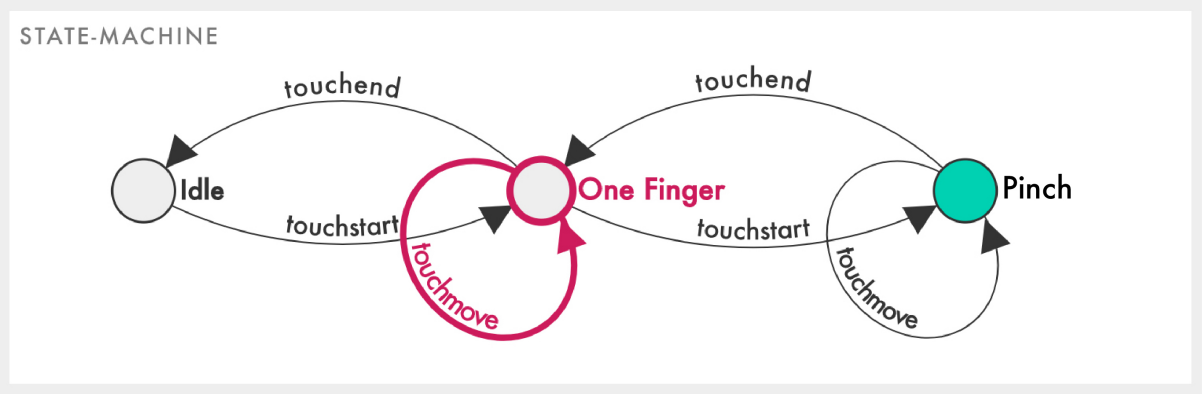

Fig. 12. A state machine for a pinch interaction. The state machine diagram is interactive: states and transitions can be added, removed or edited. The selected state, in green, can be edited in the code editor. The currently active state and transition are highlighted in red.

Enact provides a default state machine, with two states and three transitions, that supports continuous touches with one finger. To add multi-touch capabilities, users can create new states by double-clicking an empty space in the diagram and new transitions by control-dragging from the source state to the target state. When the user selects a state or transition, Enact shows its code in the editor as a plain JavaScript object (Figure 13). Transitions are named after their input event, they have a source and a target state (extracted from the diagram), an optional guard and an action. Enact only executes the transition's action when its input event is detected and the guard is satisfied. States can also execute actions when activated (on enter) or deactivated (on exit).

Enact is a live environment: guards and actions are written in JavaScript and interpreted right away. Because there is no waiting time, guards and actions can be edited live and the result is immediately available. As a result, users can *design by enaction* by immediately testing the current design in context.

8.6.2 Reusing design elements with design references. For developers, the storyboard not only describes the interaction, it also provides a repertoire of visual objects ready to be used in the code. Developers can drag elements from the storyboard to the code editor to create *design references* (Figure 10d), i.e. code-based representations of the visual elements manipulated by designers. To help match the code representations with its visual counterpart, they share the background color of their linked visual element (Figure 13). Developers can modify them directly with a double-click in the code editor.

When the developer hovers over a *design reference*, the storyboard highlights the corresponding element. As an alternative to dragging the visual element, users can type `$` followed by a dot to use autocompletion. They can access visual elements such as screens (`$.S`*n*), rectangles (`$.R`*n*), circles (`$.C`*n*), polygons (`$.P`*n*), touches (`$.T`*n*) and measures (`$.M`*n*). Position and size labels can also be dragged into the code. For example, dragging a rectangle's x-position label into the code generates `$.R1.position.x`. Since *design references* refer to their associated visual elements, they reduce reworking and ensure a unique *source of truth* that is always accessible by both designers and developers.

8.6.3 Global and local design references. *Design references* can have a global or a local scope. The global scope includes the whole storyboard, while the local scope only includes one particular screen of the storyboard. A *global design reference*, such as `$.R1.position.x`, refers to the x-position of `R1` in any screen of the storyboard,

```
1  {
2          description: '',
3          name: 'touchmove',
4          guard: function (e) {
5              // Only when the guard is true the transition is executed
6              return $.isInside({touch:$.T0,shape:$.R1});
7          },
8          action: function (e) {
9              // When the transition is executed this action is performed
10             $.map({input:$.T0.position, // position of a touch
11                    output:$.R1.position, // position of a rectangle
12                       max:237})          // y-position of a rectangle in the first screen ($.S1.R1.position.y)
13         }
14 }
```

Fig. 13. The editor shows the code of the selected state or transition. Here, a `touchmove` transition is selected. The guard ensures that the transition triggers only when touch `T0` is inside rectangle `R1`. In the action, the developer has defined a mapping between the positions of the touch and the rectangle. The editor creates *design references* around recognized design elements: `$.T0.position`, `$.R1.position` and `$.S1.R1.position.y`. The number 237 is a *local design reference* representing the y-position of rectangle `R1` in screen `S1`. If the storyboard changes, this value will also change.

i.e. it refers to that design element property at run time. However, developers also need to use particular values from the proposed design, such as an initial color or a particular position or size. This is achieved by using a *local design reference*, which targets a value from a specific screen of the storyboard. By shift-dragging rectangle `R1` y-position from screen **S1** to the editor, Enact generates the *local design reference* `$.S1.R1.position.y`. However, instead of displaying the underlying code as in the global case, a *local design reference* displays its current value, e.g. 237 in this case (Figure 13, line 12). However, this value is bound to the visual element in the corresponding storyboard screen. This lets the designer make changes in the storyboard that are automatically reflected in the code, maintaining a single *source of truth*.

Local design references support the story of $P12_{dv}$ in the first study, when he *"wrote the code so that the designer could very easily touch it"*. In Enact, designers, as well as developers, can influence the code by directly modifying elements in the storyboard. Other use cases for *local design references* include creating initial/final screens or thresholds such as minimum or maximum values, e.g. to return an element to its original position or to constrain an element's size to be always smaller than a certain value.

8.6.4 Creating input-output mappings. In order to program interactive behaviors, developers typically define the rules that connect user inputs with system outputs. To facilitate programming, Enact provides built-in functions, such as `$.isInside({touch:,shape:})` and `$.map({input:,output:})`. Users can also create their own functions to avoid code duplication and use them on any state machine action.

The `map` function connects changes in input properties, such as the position of a touch, to changes in output properties, such as the position of a graphical object. This alleviates the need for maintaining the initial offset between the touch input and the target shape. For example, dragging rectangle `R1` with finger `T0` is achieved by `$.map({input:$.T0.position, output:$.R1.position})` in the `touchmove` transition of the default state machine.

The `map` function takes optional additional parameters for further customization: `min` and `max` set the minimum and maximum values of the output property, and `ratio` controls the relationship between the changes in the input and the changes in the output. By default, there is no `min` and `max` value, and the `ratio` is one. More generally,

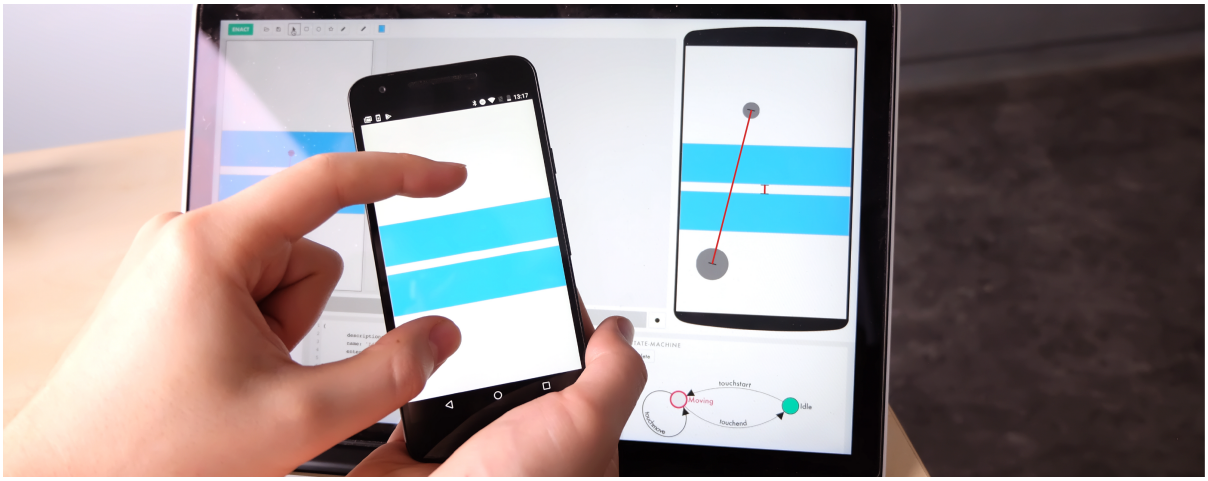

Fig. 14. The designer created two measures, one between the touches and the other between the two rectangles. These measures are invisible on the target device (left) but they are revealed in the device mirror (right) and updated in real time to facilitate debugging.

developers have access to the full power of JavaScript, including variable declarations, control structures and function definitions. Developers can also create and save functions, to facilitate reuse across projects.

8.6.5 Reifying distances with measures. Developers are used to working with expressions and formulas calculated from properties of visual elements, such as when the size of a rectangle controls the position of another object. Designers, on the other hand, are more familiar with direct manipulation and visual properties. Enact makes it possible to *reveal invisible* relationships by reifying [3] them as *measures*.

Measures are first-class visual objects that can be created between two points of interest of a shape, touch input or other measure (Figure 14). When a measure starts and ends on the same point, it represents a point of interest, e.g. the middle of a segment. Measures can be dragged into the code to create *design references*, like other visual elements. For example, a measure can be created between two touches to represent the spread of a pinch gesture, and used as a parameter of the `map` function to control the size of an object.

By making formulas explicit, measures also reduce code duplication and help create one *source of truth* for the design. Measures can also help identify *edge cases*: instead of having to figure out from the code why a formula evaluates to a given value, the user can to look at the device mirror where the measures are visualized in the context of the interface elements and the user inputs. For example, it is easy to see on the device mirror if the correct points are being measured or if the extremities of a segment cross during the interaction.

8.7 Target device and mirror

In the second and third study we observed the use of external devices as a medium to communicate ideas. Current tools involve the target device typically only to test an implementation. Enact instead encourages early exploration of interactions during prototyping by giving the target device a more active role as a design medium, supporting *design by enaction*.

By default, Enact shows the first screen on the target device. This contextualizes the design, letting designers and developers evaluate decisions on the device where they will be used. Designers can use the target device to

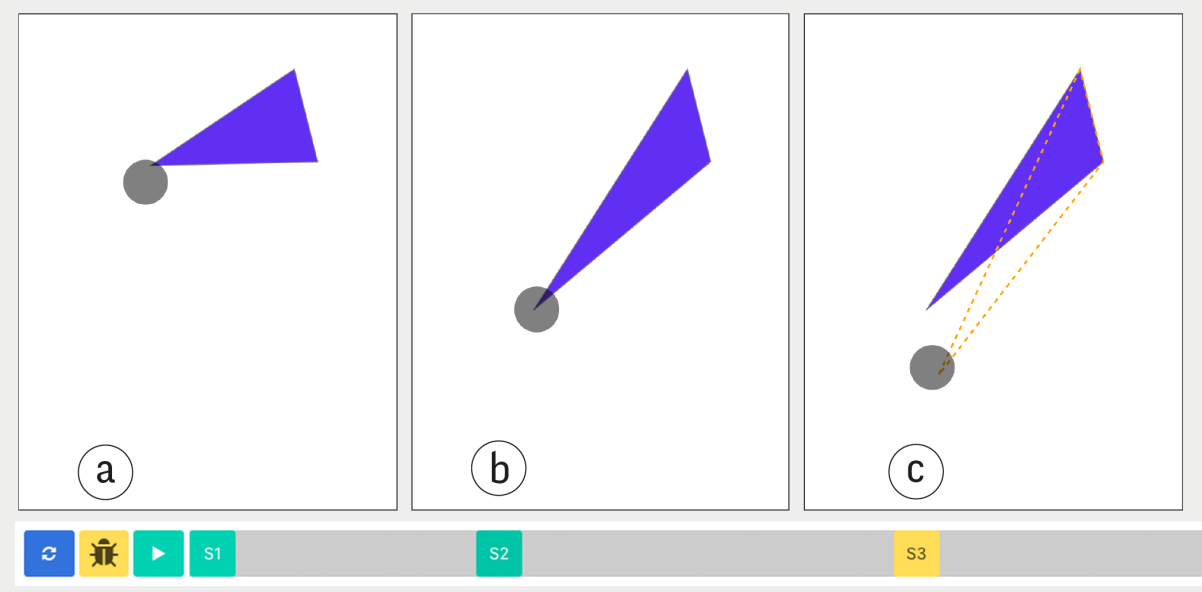

Fig. 15. Testing in Enact combines the code with the recorded user inputs. To run the test, the designer presses the Test button (a). Here, the first and second screen match the test result and their screen handles are shown in green (b). The third screen has a mismatch displayed as a dashed orange line and the screen handle is shown in yellow (c).

record touch inputs, to see the generated animation in the early stages of the design, and the actual implementation later on.

Enact features a device mirror that duplicates the visual state of the target device in real time and augments it with visual information such as the position and sizes of the touch points and the measures (Figure 14). The mirror therefore *reveals invisible* but important information to understand the current state of the interaction and for debugging purposes. Such information should not appear on the target device to avoid confusion with the actual interface and additional information. The target device and the mirror therefore also *provide multiple viewpoints*.

Designers and developers are encouraged to *design by enaction* by using the target device to try out the interaction as often as possible. In the process, users sometimes generate an interesting state on the target device, either because it is an *edge case*, or because it complements the storyboard. In such cases, Enact lets them drag a copy of the mirror and insert it as a new screen in the storyboard.

8.8 Assisted testing

Assisted testing helps maintain *a single source of truth* among *multiple viewpoints*. The first study showed that one key challenge in designer-developer collaboration is to keep the design and the code synchronized. In software engineering, checking the consistency between a design specification and its implementation involves testing. Enact supports assisted tests to help designers and developers maintain the code and the design representations synchronized when mismatches occur. Assisted testing executes the code with the recorded input and shows incongruences in the corresponding storyboard screen.

When the user presses the Test button, Enact sets the target device to match the first screen, then synthesizes input events from the recorded gesture[16] and executes them on the target device, triggering the same code as if they were actual user inputs. Enact compares each screen in the storyboard with the corresponding state of the target device during the test. If the output matches the screen, the corresponding screen handle turns green in the timeline (Figure 15a and Figure 15b). If there are differences, the screen handle turns yellow and the output is displayed on top of the original design, with the differences highlighted in orange (Figure 15c).

After the test, users can drag the screen handle along the timeline and replay the test, as if they were controlling a video of the interaction. This ability to navigate the test results helps designers and developers understand the behavior of the interaction and pinpoint the sources of the differences.

Mismatches between the screens and the test results reveal inconsistencies between the design and its implementation. These inconsistencies break the *single source of truth* principle, requiring manual changes to return the *multiple viewpoints* to a consistent state. Deciding what needs to be modified, either the storyboard, the code, or both, is up to the designer and developer. Enact helps match the storyboard with the code by snapping the visual objects to the test results when they are moved or resized close to them.

8.9 System Implementation

Enact is a client-server web application developed with Vue, Node.js, Socket.io, CodeMirror and D3[17]. We use reactive data bindings to provide liveness within the desktop interface. The target device is connected through an ad-hoc protocol on top of Socket.io messages.

Design references use CodeMirror's text markers to style the code with custom widgets that react to user changes in the storyboard. We extended the CodeMirror parser with regular expressions to support definitions of the form `$.{screen}.{object}.{property}.{sub-property}`. The `map` function only accepts properties, such as position and size, and sub-properties, such as x-position, y-position, width, and height.

To minimize the learning curve for developers, we wanted to handle properties and sub-properties, i.e. objects and primitive values, uniformly. In particular, these properties and sub-properties needed to keep a record of both their previous and current value. To overcome this challenge, we wrapped these objects behind a JavaScript proxy object that manages *delta values*, i.e. the difference between the current and previous value, and a function that transforms these deltas. Using deltas instead of absolute values simplifies the code by alleviating the need for offsets, such as the distance from the touch position to the shape's origin. The proxy object also adds an interface for the sub-properties so they behave like primitive values, e.g. numbers. Thanks to this approach, from the point of view of the user properties are regular objects and sub-properties are primitive numbers.

8.10 Limitations and Future Work

We chose to focus Enact on mobile multi-touch interactions because of the familiarity of the participants with this interaction style and the vast opportunities for customization that it provides. The goal was to illustrate how the design principles informed the design, with the hope that a similar approach can be used for other types of tools. While Enact supports the creation of advanced multi-touch interactions, it lacks proper support for temporal interactions, such as using timeouts to distinguish between short and long taps, or single and double taps. These could easily be added to the state machine. Also, we focus on continuous interaction, not symbolic gestures that use a recognizer. Such gestures could be implemented by invoking the recognizer at touch-up in the state machine. More generally, applying Enact to other types of interactions is left for future work.

Enact uses JavaScript, making it possible to integrate the code into existing development environments. However, manually exporting the code will generate the same problems of reworking and redundancies as with

[16] Synthetic events are similar to regular browser events but instead of being triggered by user actions, they are executed programmatically.
[17] https://vuejs.org, https://nodejs.org, https://socket.io, https://codemirror.net/, https://d3js.org/

traditional design artifacts. We argue that prototyping tools such as Enact should be part of an integrated design and development environment. This would support the creation of evolutionary prototypes, a type of prototype "intended to evolve into the final system" [4]. However, if this tight integration is not possible, other lightweight mechanisms such as live reloading or hot module replacements could maintain *a single source of truth*.

Enact should support more powerful graphical editing capabilities. Currently, Enact only provides simple graphical elements but designers are used to the power features of their current tools, such as gradients, compound shapes, text, and imported images. Supporting these features in the context of other *viewpoints* is an interesting area for future work.

Enact should also support the design of more than one interaction at a time. This will increase the likelihood of conflicts, which could be handled by supporting separate state machines associated with different objects.

Enact uses a form of programming-by-demonstration but without sophisticated inferencing so as to provide a simpler user interface for professional designers and developers [45]. However, inferencing could open new possibilities to *maintain a single source of truth*. Currently, when assisted testing detects a mismatch between the storyboard and the code, Enact highlights the difference. However, Enact could provide an inference engine that suggests the code modification needed to reach the desired visual state. This type of inferencing could also be bidirectional, i.e. when changes occur in the code, Enact could suggest the visual modifications needed on the storyboard to maintain consistency.

Finally, Enact is a single-user application that lets designers and developers work together when they are co-located, or asynchronously when they are remote. In order to facilitate asynchronous communication within the tool, Enact should support annotations and comments associated with design objects. Enact should also support versioning system to track changes and to let designers explore and compare alternatives more easily. Finally, designers and developers often want to work concurrently. Supporting real-time remote collaboration introduces extra challenges in order to maintain a consistent, executable design, and is another area for future work.

9 STUDY FOUR: ASSESSING ENACT

To better understand how designers and developers interact with Enact, we conducted a structured observation study [23] with an earlier version of the tool. In terms of the evaluation strategies of Ledo et al. [36], this is a type 2 usage study. We conducted this study wih an earlier version of the tool that featured graphical rules to program the interaction (Figure 16) instead of the state machine and code editor. The rules provided a visual template for the `map` function presented earlier. Users could fill out the template by dragging elements from the storyboard. The goal was to enable designers, not just developers, to "program" the interaction.

We observed how designers and developers interacted with Enact and the strategies they used to individually prototype several interactions.

9.1 Method

9.1.1 Participants. We recruited four participants (1 woman, ages 26-34): two professional developers ($P1_{dv}$ and $P2_{dv}$) and two professional designers ($P3_{ds}$ and $P4_{ds}$), who create web sites, mobile applications or interactive installations. Their experience collaborating across disciplines ranges from 3 to 8 years.

9.1.2 Apparatus. We run Enact in a Macbook Pro 13-inch with a 2,7 GHz Intel Core i5 processor and 16GB of memory. The target device connected to the Enact system was an LG Nexus 5X running Android 7.0.

9.1.3 Procedure. We gave a short demonstration of Enact's features and asked participants to create three different interactions of increasing difficulty: "simple drag", "pull down curtain" and "pinch-to-create". We prompted the first task orally and presented the last two in the form of rough sketches. After the three tasks, we

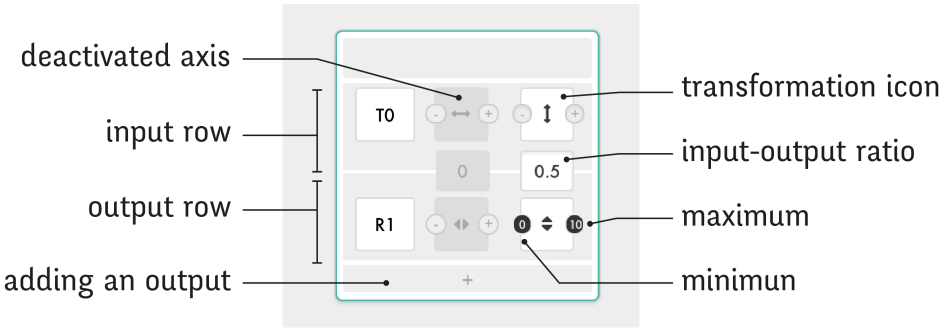

Fig. 16. Visual specification of rules in the earlier version of Enact used in Study Four. Each input-output rule connects one input element with one or more output elements. The first column shows the elements (T0 and R1), which can be dragged from the storyboard. The second and third columns specify the horizontal and vertical mappings. This rule maps the Y-axis translation (↕) of touch point T0 to the Y-axis scale (⇳) of rectangle R1 with a ratio of 0.5. The X-axis transformation is disabled. Each transformation can have a minimum and maximum value taken from the storyboard.

gave participants 15 minutes to experiment freely with Enact. The study used a think-aloud protocol and took approximately one hour, after which we asked a set of post-hoc questions.

9.1.4 Data Collection. We recorded audio and video of the participants' interactions with Enact, on the computer and on the target device. We also took notes during the interviews.

9.2 Results and Discussion

We performed a thematic analysis [7] of the collected data to extract common themes across participants, both during the tasks and the post-hoc interviews. We then revisited the data to specify the themes and to extract relevant quotes.

All participants were able to create the three proposed interactions. Participants were not asked to work quickly, and were encouraged to talk aloud while doing the tasks. Even so, all participants finished the first task in less than three minutes without prior training. All participants finished task 2 in less than five minutes and task 3 in less than 15 minutes.

9.2.1 An embodied perception of interaction. Without rules relating touch inputs and shapes, a prototype is not interactive. We were surprised to see all participants try to interact with the shapes on the mobile device before they create a rule. Both designers and developers wanted to immediately explore interactivity during prototyping. $P1_{dv}$ noted that Enact approaches interaction from *"a sensible point of view, just like the end user would experience it on the mobile"*.

All participants interacted with the target device as soon as they created rules. Thanks to the live nature of the system, $P1_{dv}$ realized that one of his rules was incomplete: *"Now I realize that it grows only downwards, I need to move it up."*

Participants also liked the assisted testing. For example, after the first interaction, $P2_{dv}$ decided to rely exclusively on the assisted testing to verify the rules. As $P4_{dv}$ could not understand why the interaction on the target device did not react as expected, he ran the assisted test and slowly navigated the history of the test results to find the problem.

9.2.2 A pedagogical tool. Both designers and developers saw the value of the tool to help them collaborate across disciplines. For example, $P3_{ds}$ explained: *"Today, I try to create simple things so that the developer does not have to get headaches while implementing them"*. $P3_{ds}$ added: *"What is interesting is the pedagogical aspect. I am forced to ask myself what I'm gonna ask the developer."*

On the developers' side, $P1_{dv}$ thought that this tool would help designers better understand custom interactions: *"There are not too many constraints at the beginning. It shows that it can be fun and quick to create things that are very hard to implement today. But you can also increase the complexity afterwards"*.

9.2.3 The need for more power. Participants were able to use Enact's features but, as they engaged with the tool, they wanted to manipulate finer details of the interaction. For example, $P1_{dv}$ wanted more control over the rules: *"I want to add this delta value. I want operators to correct rules."* $P2_{dv}$ also wanted to be able to link shapes and values in the rules, so that he could directly modify the latter by moving the corresponding shapes in the storyboard. These remarks led us to replace the visual rules with the `map` function and introduce *design references* in the current version of Enact.

10 STUDY FIVE: COMPARING ENACT WITH TRADITIONAL TOOLS

Based on the feedback from the previous study, we created the version of Enact described in section 8. We replaced the graphical rules with the interactive state machine diagram and added the augmented code editor to provide more control.

We then conducted a second study with this new version in order to better understand how Enact affects designer-developer collaboration. The goal was to observe and compare the strategies used by designer-developer pairs to represent, communicate and implement interactions with their own tools and with Enact. We focused on the issues that arise from the inability to successfully represent the interaction itself, rather than those that arise from generating design ideas. We therefore used the same strategy as in Study Three and provided interaction examples from existing applications that participants had to describe and implement. For ecological validity, we organized the study in three phases that reflect the common collaboration patterns we observed involving a design hand-off: communication of the initial design (designer only), initial implementation (developer only), and side-by-side collaboration (designer and developer, co-located).

10.1 Method

10.1.1 Participants. We recruited 12 participants (6 women and 6 men, ages 23-35): six professional developers ($P1_{dv}$ to $P6_{dv}$) paired with six professional designers ($P1_{ds}$ to $P6_{ds}$), who create web sites, mobile applications or interactive installations. Their experience in collaborating across disciplines ranges from 0 to 7 years. $P1_{dv}$ and $P6_{dv}$ reported no significant collaboration experience as they were just starting their front-end developer careers.

10.1.2 Apparatus. Participants used their own setup, i.e. laptop, mouse, and pre-installed software, to work with their Traditional tools. We provided two LG Nexus 5X running Android 8.0 and an Apple iPhone 6S running iOS 10 as the target mobile devices. We helped participants to connect these devices to their development environments before starting the study. We run Enact in a Macbook Pro 13-inch with a 2,7 GHz Intel Core i5 processor and 16GB of memory. The target device connected to the Enact system was the LG Nexus 5X. The other LG Nexus 5X was used to show examples of interactions to the designer.

10.1.3 Procedure. The study uses a think-aloud protocol and takes approximately two hours. After each condition we ask participants to fill out a short questionnaire. At the end of the session, we conduct a post-hoc interview.

Each pair first creates an interactive prototype of an existing interaction with their preferred tools (Traditional condition), then does the same for a different interaction with Enact (Enact condition). For the Traditional

<table>
<tr><td>Condition</td><td colspan="3">Traditional tools</td><td rowspan="2"></td><td colspan="3">Enact</td></tr>
<tr><td>Interaction</td><td colspan="2">Initial version</td><td>Final Version</td><td colspan="2">Initial version</td><td>Final Version</td></tr>
<tr><td>P_{ds}</td><td>10 min</td><td></td><td rowspan="2">15 min</td><td rowspan="2">Enact intro and training</td><td>10 min</td><td></td><td rowspan="2">15 min</td></tr>
<tr><td>P_{dv}</td><td></td><td>15 min</td><td></td><td>15 min</td></tr>
</table>

Fig. 17. Study Five. Each pair consist of one designer (P_{ds}) and one developer (P_{dv}). The designer has 10 minutes to create a design artifact for the initial version of the proposed interaction, then the developer has 15 minutes to implement it. Finally, both work together for 15 minutes to evaluate the current implementation and work on the final version of the interaction.

condition we do not impose any restriction on the tools they can use. We only let them know that they will work for a mobile platform supporting multi-touch interactions. To avoid any influence from Enact on the participants' usual workflow, they all perform the Traditional condition before Enact. Once the Traditional condition is over, we give a short presentation of Enact and let each participant individually practice for 10 minutes before performing the Enact condition.

For both conditions we follow the same protocol (Figure 17). First, we show the designer an initial version of an interaction preloaded on one of the mobile devices. Only the designer has access to this device throughout the study. The designer has 10 minutes to create a design that communicates all the details he deems relevant for creating a prototype with the same behavior. Then the designer sends the document to the developer, who has 15 minutes to create an interactive prototype based on the received design. During this time, we show a final version of the interaction to the designer. Finally, we give the pair another 15 minutes to review the initial implementation and work together to prototype the final version of the interaction.

For each condition we use a different multi-touch continuous interaction with two versions: initial and final. This lets us study the introduction of changes in a controlled environment: the initial version features the basic interactive functionality while the final version includes changes and adds complexity to the previous design.

We use two custom interactions already used in the participatory design workshop (Study Three, section 6): *pinch-to-create* and *pan-and-stamp*. We simplified certain aspects of the visual design such as gradients, texts and 3D effects to focus on interactivity. Both interactions feature at least seven *edge cases*. We expect these interactions to have a similar level of difficulty.

The first interaction, *pinch-to-create*, is inspired by the *Clear* to-do list mobile app[18], which uses a spread gesture to create a new item between two existing items (Figure 6). The initial version lets users simultaneously move two rectangles with a pinch gesture. The final version lets users manipulate the size of a third rectangle with the spread of their fingers. The *edge cases* include: The interaction should only work when the fingers are inside the rectangles; The third rectangle is always positioned in the middle of the other two; The upper rectangle cannot move lower than its starting position.

The second interaction, *pan-and-stamp*, is inspired by the *Paper* note-taking mobile application's[19] cut and paste feature, where one finger drags the object and a tap with another finger copies it. The initial version lets users pan a rectangle with one finger and copy the rectangle with another finger tap. The final version lets users

[18]https://www.realmacsoftware.com/clear/
[19]https://www.fiftythree.com/paper

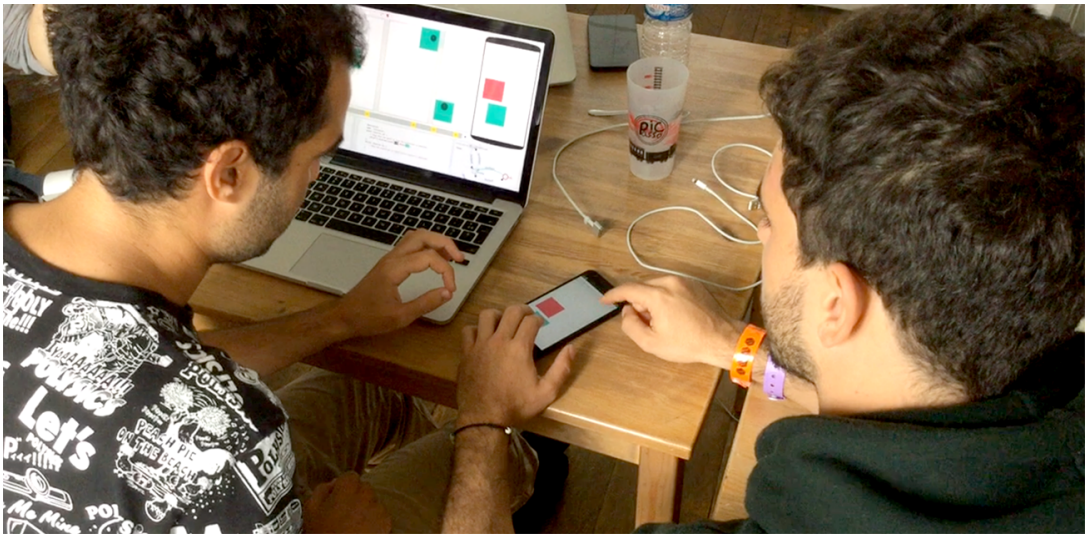

Fig. 18. $P4_{ds}$ and $P4_{dv}$ working side-by-side on the final version of *pan-and-stamp*. On the left, the developer performs an off-device mimicking gesture with his left hand to understand the proposed design. On the right, the designer performs an on-device mimicking gesture with both hands to communicate the design.

pinch to resize the rectangle and, with a third finger, tap to create new copies of it. The *edge cases* include: New rectangles are centered at the tap position; The first rectangle needs to be resized from the center, not the top-left corner; and should also be panned with two fingers. The two interactions are balanced across pairs: P1, P2 and P4 start with *pinch-to-create* while P3, P5 and P6 start with *pan-and-stamp*.

10.1.4 Data Collection. We recorded audio and video, both over the shoulder and with a screen recorder. We also took notes during the tasks and the post-hoc interviews.

10.2 Results and Discussion

We performed a thematic analysis [7] of the collected data to extract the prototyping strategies used by the participants. After looking for themes, we revisited the data to specify the classification and extract relevant quotes.

We also measured the number of mimicking gestures in the video data, i.e. when participants acted out the interaction with their hands, either to understand it or to communicate it, both away from the device and on the device. In our initial studies we observed designers and developers using hand gestures to communicate the interaction. We were therefore interested in whether Enact encourages participants to use more mimicking gestures when collaborating.

We focused our analysis on the side-by-side collaborative phase (not the individual phases) and on on-device mimicking gestures (Figure 18). We also measured the amount of edge cases found by each pair from the notes and the video data.

10.2.1 Tools of choice.

Traditional condition. The six designers chose vector and raster graphics software: Sketch (5/6) and Photoshop (1/6). The six developers chose mobile web (3/6) or mobile native (3/6). The developer web tools included browsers,

such as Google Chrome (2/3) and Mozilla Firefox (1/3), and code editors, such as Sublime Text (2/3) and Atom (1/3). The developer native tools were IDEs: Android Studio (2/3) and Apple Xcode (1/3). All developers used search engines to look for terms such as "JavaScript pinch" or "Android button onpress", complemented with inline help when available, such as Android Studio and Xcode documentation. Pair P1 never used the provided mobile device and only used the on-screen emulator. Pairs P2 and P3 used the mobile device but relied almost exclusively on the on-screen debugger of the web browser, even though it does not support multi-touch input.

Enact condition. Participants did not chose a tool on this condition. We provided a laptop with Enact pre-installed and connected to a mobile device.

10.2.2 Task completion.

Traditional condition. No pair completed the final version and only one pair (P4) finished the initial version of the provided interactions. P4 was also the only pair that started the implementation of the final version of the interaction in the Traditional condition. Two of the three pairs that started with *pinch-to-create* implemented its basic functionality, but none of the pairs that started with *pan-and-stamp* finished it.

Enact condition. All pairs provided the basic interactivity of the initial version of the interaction. Five out of six pairs provided the basic interactivity of the final version and one of them implemented all the details of the final interaction. All pairs implemented the basic and extended versions of *pinch-to-create* and the basic interactivity of *pan-and-stamp*. Two of the three pairs also implemented the extended version of *pan-and-stamp*. Despite our efforts to use interactions with similar complexity, we found that *pan-and-stamp* seemed more difficult to prototype than *pinch-to-create*. However, this should not pose a threat to validity because these techniques were counterbalanced across pairs.

Since Enact is explicitly designed to prototype the type of interactions provided to the participants, we expected a higher completion rate with Enact than with Traditional tools. On the other hand, no participant was familiar with Enact and they only had a short training session. Even so, participants were able to complete much more with Enact than with their Traditional tools. Since we gave participants very little time and did not expect them to finish all the tasks, we concentrate the following points on the their strategies to prototype the interaction rather than on performance measures.

10.2.3 Transitioning problems.

Traditional condition. All the designers (6/6) used the same workflow to communicate the design to the developer in the Traditional condition. Designers created a storyboard document depicting the visual states of the interface with a graphical software. Then, they sent it to the developer either in the original format, as a PDF, or through a specialized tool such as InVision. Designers illustrated the visual design with different screens and explained the user inputs with circles, icons, traces, text annotations or a combination of these. None of the designers used animations nor video to communicate interactivity. $P6_{ds}$ mentioned her intention of using video but she felt that it would require too much time with her Traditional tools. None of the designers updated their design specification at the beginning of the side-by-side phase to communicate the new design. Rather than using a design artifact, they described the final version of the interaction on top of the current implementation, verbally or by using mimicking gestures.

In the Traditional condition, developers ran into problems when interpreting the interactivity and reproducing the visual look. Developers expected text annotations. For example, $P5_{dv}$ said *"I don't understand this"* when viewing the design for the first time. When text annotations were minimal, developers expected more details. For example, $P3_{dv}$ said *"I don't know if these are multiple interactions or different steps of the same interaction"* and that she *"prefer[s] comments saying "when this happens then that happens""*. Four developers ignored the graphic

design and used either no visual elements at all (only console logs), gray buttons or wrongly colored rectangles. Two developers used external color pickers to extract the right color from the design and copy the corresponding hexadecimal string. All the developers ignored the precise size of the rectangles: for *pinch-to-create* the height of the rectangle in the design was not replicated and for *pan-to-stamp* developers generally used rectangles instead of the square shown in the design.

Enact condition. Designers used the animation feature to refine the storyboard and developers used it to understand the interaction. Thus, the generated animation worked as a contact point between the two activities while working asynchronously. Only one designer asked for icons to represent user input and another expressed the need for text annotations. Enact does not currently support text annotations but developers did not mention the lack of this feature. One explanation for this could be the use of Enact's generated animation as a communication medium instead of textual descriptions. Developers also mentioned the usefulness of showing the touch input information on the device mirror while the animation was being played.

In the Traditional condition, all developers used print logs to confirm the triggering of input events. Enact's live state machine diagram provided the same level of confidence to the developer, without extra effort and with more detail. In the Traditional condition, $P4_{dv}$ finished the basic interactivity of *pinch-to-create* but had issues to preserve the right offset between the touch point and the shape during the interaction. In the Enact condition, while using the `map` function to implement *pan-to-stamp*, he emphatically expressed that *"it is really cool that I don't have to think about the sh*tty offset"*.

The reduction in rework and redundancies was evident throughout the study. For example, four out of six developers did not follow correctly the user interface specification with the Traditional tools. In Enact there is no transition step, eliminating these issues altogether. With Traditional tools all the designers relied on copy-paste instead of using "smart objects" to create the design specification. In Enact, the storyboard propagation helps designers maintain consistency between the storyboard visual elements. Moreover, none of the designers used a design artifact to communicate the second task with their traditional tools. Instead, they relied on mimicking gestures or verbal communication. With Enact, the design artifacts are part of the prototype and were heavily used in both tasks.

10.2.4 Increasing participation. Both designers and developers performed significantly more interactions on the device with Enact than with Traditional tools (F=1.4; p<.007) (Figure 19). This suggest that designers were much more involved during the side-by-side collaboration with Enact than with Traditional tools. With Enact, all designers interacted with the target device (M = 8 times, SD = 3.65) while only three designers did it with Traditional tools (M = 1.17 times, SD = 1.46).

One reason for the increased participation is that developers were faster in creating a functional implementation with Enact, which gave designers and developers time to collaborate on the created artifact. Also, even if the implementation was not finished, with Enact, the designer had "something to play with" while with Traditional tools they usually did not. Nevertheless, we found instances, in both conditions, where designers and developers where acting out the interaction on the device even when the implentation was not finished.

Another explanation for this increased interaction with the target device could be the sense of *ownership* of the prototype. With Traditional tools, developers recreate the design with their own tools. It is not the designer's design that "comes to life" but a mere replica. With Enact, developers literally add interactivity to the artifact provided by the designer. Designers might therefore feel a stronger sense of ownership over the prototype under construction, thus increasing participation. $P4_{ds}$ said that *"you have the impression to be living in the same environment, that we share the same language"*.

10.2.5 Finding more edge cases.

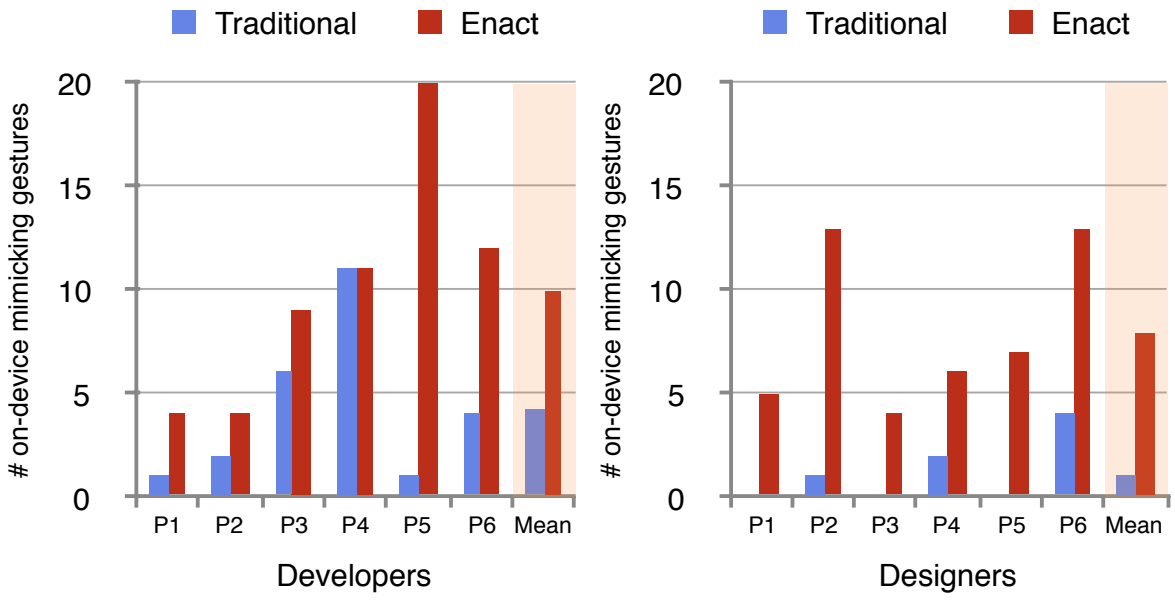

Fig. 19. Number of on-device mimicking gestures, either on the mobile device or the emulator provided by the IDE, per participant during the collaborative side-by-side phase. Designers and developers performed significantly more interactions on the device with ENACT than with TRADITIONAL tools. This suggests that with ENACT, designers participate more during the side-by-side phase and developers perform more contextualized actions on the real device during implementation.

TRADITIONAL condition. Only two pairs (2/6) found one and two of the seven edge cases –remember that only one pair finished the basic and final version of the interaction in this condition. For example, while coding *pinch-to-create* in a text editor, $P2_{dv}$ did not notice that the to-do list items should only move along the y-axis. He noticed this edge case only when he tried the interaction in the simulator. However, the simulator runs on the desktop browser instead of the target device, so that the mouse cursor events are treated as touches of a single finger, making it impossible to find edge cases related to the use of multiple fingers. In contrast, $P4_{ds}$ found an extra edge case because he was interacting with an actual multitouch device: The Apple Xcode IDE was connected to a phone, and he noticed that the items moved beyond their initial position. However, $P4_{dv}$ needed to investigate the implementation carefully in order to find which piece of code was in charge of the erroneous behavior. The lack of a clear connection between the runtime effect and the build-time code hindered $P4_{dv}$'s ability to quickly fix the issue.

ENACT condition. Five pairs (5/6) found three or more of the seven edge cases in the final version of the interaction. For example, while interacting with her first implementation attempt, $P3_{dv}$ realized that the shapes where dragged even when her finger was outside of them. Similarly, $P5_{ds}$ noticed that *pinch-to-create* behaved differently depending on which rectangle was touched first. He shared his finding with $P5_{dv}$, who added an `if` statement to determine which touch should be associated with each rectangle.

ENACT interactive representations helped designers and developers find more *edge cases*. ENACT contextualizes the interaction in the final device, giving developers and designers insights that are difficult to find in their

indirect Traditional tools: Emulators do not provide the same experience that the final context of use; IDEs connected with a real device impose a time consuming build-run-test cycle; Graphic-authoring tools are focused only in the visual properties of the interaction. Enact provides a faster design-test cycle, with more chances to find edge cases on the design by actually interacting with it. In other words, it is easier to find edge cases because the whole interaction lives less on the participants' head and more on their hands.

10.2.6 Opportunities for co-creation.

Traditional condition. Designers took a more passive role and were often just observing the developer. In the two occasions where they communicated, the participation of the designer was minimal. $P2_{dv}$ had an emulator on his screen and $P2_{ds}$ had to stand up every time she wanted to point to the screen to make a comment. In contrast, the target device was positioned between $P4_{dv}$ and $P4_{ds}$, facilitating their collaboration and participation. Nevertheless, during the side-by-side phase only the developer modified the interaction, while the designer only made suggestions.

Enact condition. Enact creates a kind of "gray area" between design and development with interesting opportunities to cross boundaries. $P2_{ds}$ was not sure she should create an input example: *"am I supposed to do this?"*. On the contrary, $P5_{ds}$ was really interested in the programming capabilities of Enact: he started adding interactivity by himself and forgot to finish the description of the interaction on the storyboard. When the developer of the same pair ($P5_{dv}$) received the design, he said *"What should I do now? This is already coded!"*.

We observed that providing *multiple viewpoints* helped break the silos between the two communities. With Enact, most of the designers were intimidated by the code editor but not by the live state machine diagram. During the side-by-side collaboration, $P5_{dv}$ built on top of $P5_{ds}$'s implementation, even extending the storyboard himself. Similarly, $P6_{ds}$ added some interactivity to the *pinch-to-create* prototype and $P6_{dv}$ directly started to fix several *edge cases* in the implementation, such as checking that the touches are inside the shapes and that the rectangle is constrained to a vertical movement. Finally we observed that intermediary representations, such as state machines, align the designer-developer vocabulary for abstract concepts, such as transition events or the state of the interaction.

10.2.7 Summary. We observed that Enact reduced rework and redundancies, allowed designers and developers participate more, helped them find more edge cases and provided a friendlier environment to co-create with the other community of practice. Both designers and developers performed significantly more mimicking gestures on the device with Enact than with Traditional tools, indicating a higher degree of participation. With Enact, five out of six pairs found three or more of the seven edge cases in the final version of the interaction. With the Traditional tools, only two out of six pairs found one or two edge cases. Finally, Enact helped designers and developers to cross their boundaries, creating more opportunities to co-create the interaction. We observed that providing *multiple viewpoints* while maintaining *a single source of truth* helped break the silos that traditionally insulate the two communities of practice. Designers were able to manipulate developer-oriented artifacts, such as the state machine and the code editor, thanks to the use of *design references*. On the other hand, developers also manipulated the storyboard and the target device, either to modify the recorded inputs, or to reveal touch and measure information on the device mirror.

11 CONCLUSION

Despite their different backgrounds and skills, designers and developers need to collaborate to create interactive systems. Our goal was to investigate how better prototyping tools can support their collaborative process.

This work makes three main contributions:

- At the empirical level, we report on three studies to better understand and classify current collaboration issues;
- At the theoretical level, we introduce a set of principles to design better collaborative prototyping tools; and
- At the technical level, we present a new tool guided by these principles to reduce collaborative breakdowns during the creation of custom touch-based interactions.

In study one we showed that current workflows and tools induce unnecessary rework. We found that designers create a multitude of redundant design documents and developers must recreate them with their own tools. This process often introduces mismatches with the original design. We proposed a classification of key design breakdowns: *missing information*, when designers do not communicate a specific detail; *edge cases*, when designers do not think about a particular case; and *technical constraints*, when designers are not aware of developer's technical limitations. The interviews also showed that when developers are not involved in the initial design phase, implementation tends to be problematic or even impossible.

In study two we found that even if the early involvement of the developer mitigated *design breakdowns*, new breakdowns appeared in subsequent meetings. We also found that the inability of current design artifacts to represent interaction generates breakdowns.

In study three we observed that the limitations on the representations used to communicate interaction result in *missing information* and ignored *edge cases*. The successful communication and representation of interactions required an iterative process, from concrete examples to more general rules.

Based on this empirical work, we introduce four principles for collaborative prototyping tools to reduce designer-developer breakdowns: *Provide multiple viewpoints*, to allow developers to participate early, *maintain a single source of truth*, to reduce reworking and redundancies, *reveal the invisible*, to avoid *missing information*, and *support design by enaction*, to find more *edge cases*. Collaborative prototyping tools based on these principles can play the role of a *boundary infrastructure* [6] that supports the flow of multiple interconnected objects between designers and developers.

To demonstrate these principles in action, we created Enact: a novel interaction prototyping tool for touch-based interaction on mobile devices. Through multiple interconnected representations of the interaction under construction, Enact reduces reworking, redundancies and design breakdowns. Storyboard propagation reduces redundancies within representations while *design references* reduce redundancies across representations. The interactive state machine diagram and the device mirror let designers and developers quickly explore the interaction under construction and detect *edge cases*. The connected target device lets designers and developers enact the interaction at each stage of the process.

Finally, we conducted two studies of Enact to validate our design goals, gather feedback from professional designers and developers, and analyze the impact of Enact during collaborative prototyping. In the first study, participants adopted *design by enaction* and appreciated the pedagogy of the tool to understand the details of the interaction. Both groups highly appreciated the reduced time-to-interaction, but wanted more powerful tools to describe the relationships between user inputs and system outputs. Based on this feedback, we added two new *viewpoints* for finer control: an interactive state machine and a code editor with *design references*.

In the second study, we assessed the new version of Enact in a collaborative setting and compared it with traditional tools. We found that participants completed more of the proposed interaction with Enact than with traditional tools. All the pairs finished the initial task while only one pair finished it with the traditional tools. Participants not only interacted more but they also found more *edge cases* in the design. We also found that with Enact, both designers and developers performed significantly more mimicking gestures on the device than with traditional tools, and designers were much more involved during the side-by-side collaboration. This shows that

Enact encourages a more active role during design and development, compared with the passive or indirect approach encouraged by traditional tools.

Our future work will focus on improving Enact, studying designer-developer collaborative prototyping in the wild, and going beyond touch-based interactions. Evaluating Enact in the wild will let us study the effects of collaborative prototyping on a long-term project, and hopefully will provide interesting findings about the use of interactive prototypes to collaborate with other stakeholders, such as clients, users, testers, and managers. We would also like to apply the design principles to other interaction styles, such as mid-air gestures, multimodal interaction or mixed reality.

We hope that these principles and Enact will inspire others to create new collaborative tools to support *communication through prototyping*. Initiatives such as the Hour of Code [14] try to spread programming to a broader audience. Undoubtedly, more and more designers will know how to code in the future. However, developer-based representations, such as code or visual programming, are not the only –neither the most adequate– representation for every aspect of an interaction. We need to provide integrated and multiple representations, e.g., symbolic, visual and enactive, to create collaborative spaces in which professionals with different skills, mindsets and values can work together. Digital tools need to provide ways of navigating these representations in a seamless way, thus reducing reworking and mismatches between design and implementation. We invite other researchers to test and extend these design principles to create new tools that better support the collaborative prototyping of interactions.

12 ACKNOWLEDGMENTS

This work was partially supported by European Research Council (ERC) grants n° 321135 CREATIV: Creating Co-Adaptive Human-Computer Partnerships, and n° 695464 ONE: Unified Principles of Interaction. We thank all the ex)**situ** team for their help and support, and the reviewers for their insightful comments and suggestions.

REFERENCES

[1] Caroline Appert and Michel Beaudouin-Lafon. 2006. SwingStates: adding state machines to the swing toolkit. In *Proceedings of the 19th annual ACM symposium on User interface software and technology - UIST '06*. ACM Press, New York, New York, USA, 319. http://dl.acm.org/citation.cfm?id=1166253.1166302

[2] Brian P. Bailey, Joseph A. Konstan, and John V. Carlis. 2001. DEMAIS: Designing Multimedia Applications with Interactive Storyboards. In *Proceedings of the ninth ACM international conference on Multimedia - MULTIMEDIA '01*. ACM Press, New York, New York, USA, 241. https://doi.org/10.1145/500141.500179

[3] Michel Beaudouin-Lafon and Wendy E. Mackay. 2000. Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces. *Proceedings of the working conference on advanced visual interfaces* (2000), 102–109. https://doi.org/10.1145/345513.345267

[4] Michel Beaudouin-Lafon and Wendy E. Mackay. 2003. Prototyping tools and techniques. In *The human-computer interaction handbook: fundamentals, evolving technologies and emerging applications*. 1017–1039. https://doi.org/10.1201/9781410615862

[5] Jan Borchers. 2001. *A Pattern Approach to Interaction Design*. John Wiley & Sons, Inc. 246 pages.

[6] Geoffrey C. Bowker and Susan Leigh Star. 2000. *Sorting things out: Classification and its consequences*. MIT press.

[7] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (jan 2006), 77–101. https://doi.org/10.1191/1478088706qp063oa

[8] Judith M. Brown, Gitte Lindgaard, and Robert Biddle. 2008. Stories, Sketches, and Lists: Developers and Interaction Designers Interacting Through Artefacts. In *Agile 2008 Conference*. IEEE, 39–50. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4599451

[9] Judith M. Brown, Gitte Lindgaard, and Robert Biddle. 2011. Collaborative Events and Shared Artefacts: Agile Interaction Designers and Developers Working Toward Common Aims. In *2011 AGILE Conference*. IEEE, 87–96. https://doi.org/10.1109/AGILE.2011.45

[10] Judith M. Brown, Gitte Lindgaard, and Robert Biddle. 2012. Joint implicit alignment work of interaction designers and software developers. In *Proceedings of the 7th Nordic Conference on Human-Computer Interaction Making Sense Through Design - NordiCHI '12*. ACM Press, New York, New York, USA, 693. http://dl.acm.org/citation.cfm?id=2399016.2399121

[11] Jerome S. Bruner. 1966. *Toward a theory of instruction*. Vol. 59. Harvard University Press.

[12] Bill Buxton. 2007. *Sketching User Experiences: getting the design right and the right design*. Morgan Kaufmann. 448 pages. https://doi.org/10.1016/B978-0-12-374037-3.X5043-3

[13] Stéphane Chatty, Stéphane Sire, Jean-Luc Vinot, Patrick Lecoanet, Alexandre Lemort, and Christophe Mertz. 2004. Revisiting visual interface programming. In *Proceedings of the 17th annual ACM symposium on User interface software and technology - UIST '04*. ACM Press, New York, New York, USA, 267. https://doi.org/10.1145/1029632.1029678

[14] Code.org. 2018. Hour of Code. (2018). https://hourofcode.com/

[15] Alan Cooper, Robert Reimann, and David Cronin. 2007. *About face 3: the essentials of interaction design*. Vol. 3. John Wiley & Sons. 610 pages. https://doi.org/10.1057/palgrave.ivs.9500066

[16] Paulo Pinheiro da Silva and Norman W. Paton. 2000. *UMLi: the unified modeling language for interactive applications*. Ph.D. Dissertation. University of Manchester. http://dl.acm.org/citation.cfm?id=1765175.1765188

[17] Thomas Erickson. 2000. Lingua Francas for design. In *Proceedings of the conference on Designing interactive systems processes, practices, methods, and techniques - DIS '00*. ACM Press, New York, New York, USA, 357–368. https://doi.org/10.1145/347642.347794

[18] Ylva Fernaeus and Petra Sundström. 2012. The material move how materials matter in interaction design research. In *Proceedings of the Designing Interactive Systems Conference on - DIS '12*. ACM Press, New York, New York, USA, 486. https://doi.org/10.1145/2317956.2318029

[19] Jennifer Ferreira, Helen Sharp, and Hugh Robinson. 2011. User Experience Design and Agile Development: Managing Cooperation Through Articulation Work. *Softw. Pract. Exper.* 41, 9 (aug 2011), 963–974. https://doi.org/10.1002/spe.1012

[20] Jason B. Forsyth and Tom L. Martin. 2014. Extracting behavioral information from electronic storyboards. In *Proceedings of the 2014 ACM SIGCHI symposium on Engineering interactive computing systems - EICS '14*. ACM Press, New York, New York, USA, 253–262. https://doi.org/10.1145/2607023.2607034

[21] Adam Fouse, Nadir Weibel, Edwin Hutchins, and James D. Hollan. 2011. ChronoViz: A System for Supporting Navigation of Time-coded Data. In *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems - CHI EA '11*. ACM Press, New York, New York, USA, 299. https://doi.org/10.1145/1979742.1979706

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.

[23] Jérémie Garcia, Theophanis Tsandilas, Carlos Agon, and Wendy E. Mackay. 2014. Structured observation with polyphony. In *Proceedings of the 2014 conference on Designing interactive systems - DIS '14*. ACM Press, New York, New York, USA, 199–208. https://doi.org/10.1145/2598510.2598512

[24] 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

[25] Björn Hartmann, Scott R. Klemmer, Michael Bernstein, Leith Abdulla, Brandon Burr, Avi Robinson-Mosher, and Jennifer Gee. 2006. Reflective physical prototyping through integrated design, test, and analysis. In *Proceedings of the 19th annual ACM symposium on User interface software and technology - UIST '06*. ACM Press, New York, New York, USA, 299. https://doi.org/10.1145/1166253.1166300

[26] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning. In *Proceedings of the 21st annual ACM symposium on User interface software and technology - UIST '08*. ACM Press, New York, New York, USA, 91. https://doi.org/10.1145/1449715.1449732

[27] Ghita Jalal, Nolwenn Maudet, and Wendy E. Mackay. 2015. Color Portraits: From Color Picking to Interacting with Color. In *Proceedings of the ACM CHI'15 Conference on Human Factors in Computing Systems*, Vol. 1. 4207–4216. http://dx.doi.org/10.1145/2702123.2702173

[28] Jun Kato, Takeo Igarashi, and Masataka Goto. 2016. Programming with Examples to Develop Data-Intensive User Interfaces. *Computer* 49, 7 (jul 2016), 34–42. https://doi.org/10.1109/MC.2016.217

[29] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, and George Fitzmaurice. 2014. Kitty: Sketching Dynamic and Interactive Illustrations. In *Proceedings of the 27th annual ACM symposium on User interface software and technology - UIST '14*. ACM Press, New York, New York, USA, 395–405. https://doi.org/10.1145/2642918.2647375

[30] Ju-Whan Kim and Tek-Jin Nam. 2013. EventHurdle: Supporting Designers' Exploratory Interaction Prototyping with Gesture-Based Sensors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '13*. ACM Press, New York, New York, USA, 267. https://doi.org/10.1145/2470654.2470691

[31] Kenrick Kin. 2012. *Investigating the Design and Development of Multitouch Applications*. Ph.D. Dissertation. http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-233.html

[32] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. 2012. Proton++: A Customizable Declarative Multitouch Framework. In *Proceedings of the 25th annual ACM symposium on User interface software and technology - UIST '12*. ACM Press, New York, New York, USA, 477. https://doi.org/10.1145/2380116.2380176

[33] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. 2012. Proton: Multitouch Gestures as Regular Expressions. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems - CHI '12*. ACM Press, New York, New York, USA, 2885. https://doi.org/10.1145/2207676.2208694

[34] James A. Landay and Brad A. Myers. 1995. Interactive Sketching for the Early Stages of User Interface Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '95)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 43–50. https://doi.org/10.1145/223904.223910

[35] Walter S Lasecki, Juho Kim, Nicholas Rafter, Onkur Sen, Jeffrey P. Bigham, and Michael S. Bernstein. 2015. Apparition: Crowdsourced User Interfaces That Come To Life As You Sketch Them. (2015). https://doi.org/10.1145/2702123.2702565

[36] 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*. https://doi.org/10.1145/3173574.3173610

[37] Sang Won Lee, Rebecca Krosnick, Sun Young Park, Brandon Keelean, Sach Vaidya, Stephanie D O'Keefe, and Walter S Lasecki. 2018. Exploring Real-Time Collaboration in Crowd-Powered Systems Through a UI Design Tool. *Proc. ACM Hum.-Comput. Interact.* 2, CSCW (nov 2018), 104:1—-104:23. https://doi.org/10.1145/3274373

[38] Yang Li, Xiang Cao, Katherine Everitt, Morgan Dixon, and James A. Landay. 2010. FrameWire : A Tool for Automatically Extracting Interaction Logic from Paper Prototyping Tests. (2010).

[39] Yang Li and James A. Landay. 2005. Informal prototyping of continuous graphical interactions by demonstration. In *Proceedings of the 18th annual ACM symposium on User interface software and technology - UIST '05*. ACM Press, New York, New York, USA, 221. https://doi.org/10.1145/1095034.1095071

[40] Jonas Löwgren. 1995. Applying design methodology to software development. In *Proceedings of the conference on Designing interactive systems processes, practices, methods, & techniques - DIS '95*. ACM Press, New York, New York, USA, 87–95. https://doi.org/10.1145/225434.225444

[41] Wendy E Mackay. 2002. *Using video to support interaction design*. INRIA Multimedia Services. https://www.lri.fr/~mackay/VideoForDesign/ . Directed by C. Leininger. Video Tutorial distributed at CHI '02.

[42] Wendy E. Mackay and Anne-Laure Fayard. 1997. HCI, Natural Science and Design: A Framework for Triangulation Across Disciplines. In *Proceedings of the conference on Designing interactive systems processes, practices, methods, and techniques - DIS '97*. ACM Press, New York, New York, USA, 223–234. https://doi.org/10.1145/263552.263612

[43] Nolwenn Maudet, Germán Leiva, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2017. Design Breakdowns: Designer-Developer Gaps in Representing and Interpreting Interactive Systems. *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing - CSCW '17* (2017), 630–641. https://doi.org/10.1145/2998181.2998190

[44] Jack Moffett. 2014. *Bridging UX and Web Development: Better Results through Team Integration*. Elsevier Science. 224 pages.

[45] Brad A. Myers, Richard G. McDaniel, and David Wolber. 2000. Programming by example: intelligence in demonstrational interfaces. *Commun. ACM* 43, 3 (mar 2000), 82–89. https://doi.org/10.1145/330534.330545

[46] Brad A. Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew J. Ko. 2008. How designers design and program interactive behaviors. *Proceedings - 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008* (2008), 177–184. https://doi.org/10.1109/VLHCC.2008.4639081

[47] Mark W. Newman and James A. Landay. 2000. Sitemaps, Storyboards, and Specifications: A Sketch of Web Site Design Practice. In *Proceedings of the conference on Designing interactive systems processes, practices, methods, and techniques - DIS '00*. ACM Press, New York, New York, USA, 263–274. http://dl.acm.org/citation.cfm?id=347642.347758

[48] Don Norman and Stephen W. Draper. 1986. *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA.

[49] Stephen Oney, Brad A. Myers, and Joel Brandt. 2014. InterState: A Language and Environment for Expressing Interface Behavior. In *Proceedings of the 27th annual ACM symposium on User interface software and technology - UIST '14*. ACM Press, New York, New York, USA, 263–272. https://doi.org/10.1145/2642918.2647358

[50] Wanda J Orlikowski. 1992. The duality of technology: Rethinking the concept of technology in organizations. *Organization science* 3, 3 (1992), 398–427.

[51] Fatih Kursat Ozenc, Miso Kim, John Zimmerman, Stephen Oney, and Brad A. Myers. 2010. How to support designers in getting hold of the immaterial material of software. In *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*. ACM Press, New York, New York, USA, 2513. https://doi.org/10.1145/1753326.1753707

[52] Sun Young Park, Brad A. Myers, and Andrew J. Ko. 2008. Designers' natural descriptions of interactive behaviors. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 185–188. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4639082

[53] Steven E. Poltrock and Jonathan Grudin. 1994. Organizational Obstacles to Interface and Development: Two Participant – Observer Studies. *ACM Trans. Comput.-Hum. Interact.* 1, 1 (1994), 52—-80. https://doi.org/10.1145/174630.174633

[54] Dina Salah, Richard F. Paige, and Paul Cairns. 2014. A systematic literature review for agile development processes and user centred design integration. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14*. ACM Press, New York, New York, USA, 1–10. http://dl.acm.org/citation.cfm?id=2601248.2601276

[55] Frank M. Shipman and Catherine C. Marshall. 1999. Formality Considered Harmful: Experiences, Emerging Themes, and Directions on the Use of Formal Representations in Interactive Systems. *Computer Supported Cooperative Work (CSCW)* 8, 4 (dec 1999), 333–352. http://link.springer.com/10.1023/A:1008716330212

[56] Susan Leigh Star. 1989. The Structure of Ill-structured Solutions: Boundary Objects and Heterogeneous Distributed Problem Solving. In *Distributed Artificial Intelligence (Vol. 2)*, M. Huhns (Ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 37–54. http://dl.acm.org/citation.cfm?id=94079.94081

[57] Susan Leigh Star. 2010. This is Not a Boundary Object: Reflections on the Origin of a Concept. *Science, Technology, & Human Values* 35, 5 (aug 2010), 601–617. https://doi.org/10.1177/0162243910377624

[58] Susan Leigh Star and James R. Griesemer. 1989. Institutional Ecology, 'Translations' and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology, 1907-39. *Social studies of science* 19, 3 (1989), 387–420. http://www.jstor.org/stable/285080

[59] Steven L Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming (LIVE '13)*. IEEE Press, Piscataway, NJ, USA, 31–34. http://dl.acm.org/citation.cfm?id=2662726.2662735

[60] Bret Victor. 2013. Media for thinking the unthinkable. *Presented at the MIT Media Lab on April 4, 2013* (2013). http://worrydream.com/MediaForThinkingTheUnthinkable/

[61] Khoi Vinh. 2015. Design Tools Survey | The Tools Designers Are Using Today. (2015). http://tools.subtraction.com/index.html

[62] Jeremy Warner and Philip J. Guo. 2017. CodePilot: Scaffolding End-to-End Collaborative Software Development for Novice Programmers. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems - CHI '17*. ACM Press, New York, New York, USA, 1136–1141. https://doi.org/10.1145/3025453.3025876

[63] Etienne Wenger. 1998. *Communities of practice: Learning, meaning, and identity*. Cambridge university press.

[64] Meike Wiemann. 2016. *Patterns as a tool for collaboration: A case study of collaboration between designers and developers through user interface pattern libraries*. Ph.D. Dissertation. Universitet Umeå. http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A919672&dswid=5919

Received September 2018; accepted January 2019