Usability requirements for interaction-oriented development tools

Catherine Letondal, Stéphane Chatty, W. Greg Phillips[1], Fabien André, and Stéphane Conversy

Université de Toulouse - ENAC
7 avenue Edouard Belin, 31400 Toulouse, France
firstname.name@enac.fr

[1]Royal Military College of Canada
Kingston, Ontario, Canada K7K 7B4
greg.phillips@rmc.ca

Abstract. Building interactive software is a notoriously complex task, for which many programming tools have been proposed over the years. Although the research community has sporadically identified usability requirements for such tools, tool proponents rarely document their design processes and there is no established reference for comparing tools with requirements. Furthermore, the design of most tools is strongly influenced by the design of their underlying general purpose programming languages. These in turn were designed from their own set of little-documented requirements, which adds to the confusion. In this paper, we provide a review and classification of the requirements and properties expected of interactive development tools. We review how designers of APIs and toolkits for interaction-oriented systems set the usability requirements for the programming interface of their systems. We relate our analysis to other studies in related domains such as end-user programming, natural programming, and teaching.

1 Introduction

Throughout the last several decades, programming interactive software has consistently been documented as complex and costly [59]. Dozens of tools, languages, architecture patterns and formal models have been proposed to address aspects of this complexity. However, these have seldom had a significant or lasting impact on programming practices. Most interactive software is still written in languages whose evolution was driven by other forces, and commercial user interface programming frameworks make little use of principles deriving from research – for example, few frameworks include constraints, data-flow, or standardised architecture patterns. In an era when most computers are used for interaction, it is striking that programming languages are still based on requirements derived from computation and that user interface programming still comes as an afterthought in language design.

Several reasons can be suggested for this state of affairs. First, user interfaces are a moving target: for example, principles proposed for programming command languages became partly obsolete with the advent of direct manipulation [60]. It is also possible that researchers have proposed solutions that differ too radically from industry practices. For instance, interpreted languages are thought to facilitate iterative design, but their use conflicts with intellectual property protection. However, it may be simply that *the proposed designs have failed to properly capture and address the domain's true requirements.*

As far back as the early 1970s, Weinberg suggested that we analyse programming tools in terms of their usability, proposed a simple framework for analysing the corresponding requirements, and observed that language designers rarely document the requirements they address [95]. Since then little progress has been made on the articulation of explicit usability requirements by designers, neither for mainstream programming languages nor for interactive software tools. Therefore, when addressing particular requirements, researchers may have produced solutions that ignore or even conflict with other important requirements.

We wish to explore the problem of interactive software tools from this angle. More specifically, we wish to explore the relationship between the usability requirements of programming tools for two classes of software: interaction-oriented software and computation-oriented software. Do they differ, complement, or conflict? Is one a subclass of the other? How can one combine requirements, and thus solutions? To carry out this investigation, we need a framework for comparing requirements. Development of such a framework is the principal goal of this article.

To create the framework, we begin with a top-down approach in which we analyse the nature of interaction-oriented software and its development, in order to arrive at an initial understanding of the relationship between interactive software and programming languages. We use this understanding to propose a requirement analysis framework that, hopefully, will help us understand where requirements meet or conflict. We then survey more than 50 research works on tools for interactive software, using a bottom-up approach to refine the framework and classify these works according to the usability requirements they appear to address. In the future we plan to analyse and classify programming languages in the same framework, and to use convergences and conflicts to inform the design of future programming languages and tools for interactive systems.

2 The nature of interaction-oriented programming

2.1 A subset of general-purpose programming?

Is programming interaction-oriented software a subset of general-purpose programming? Answering this question is important: if the answer is "yes", then the requirements of interactive software can be met through general-purpose programming language design. However, if the requirements of interactive software in some sense exceed those of general-purpose software, either as an intersecting set or a superset, then a different approach is required. (Spoiler: Wegner has argued convincingly that interaction is larger than computation [93]!)

The designers of programming languages more or less explicitly take this stance: each language is designed from a list of concepts that are intended to match all core requirements of programmers, and software libraries based on the language are supposed to address all specialisations of the programming activity. This has been a tremendously successful approach, even though the existence of a superset of all programming activities is only supported by the formal argument of Turing completeness.

With the notable exception of Smalltalk [48], most work on interactive systems has relied on the implicit hypothesis that there is such a thing as a general programming activity, supported by programming languages, and that programming interactive systems is a specialisation of it.

However, given the difficulties observed in designing toolkits and frameworks for interactive systems, this implicit hypothesis needs to be reassessed.

2.2 Commonalities

All programming activities, including that of interactive software, share a number of features. They are intellectual activities aimed at manipulating abstract objects, and as such are similar to mathematics, physics and similar domains. This induces requirements such as supporting the limits of human cognition – for instance in terms of working memory and visual information processing. More specifically, programming activities are creative activities aimed at building complex, dynamic, structured objects: sequences of nested actions, conditions and reactions. This induces specific requirements, some related to the links between our cognitive mechanisms and human languages, some related to our perception-action loop. And, of course, most programming activities sooner or later become collective activities with issues such as reuse and traceability; this invokes complex processes, and the corresponding requirements gave birth to software engineering. It seems evident that many software engineering issues are shared by interactive software.

These common features explain why user interface programmers find it legitimate to use generic programming language and tools, even though their experience degrades in some situations. The existence of common features also explains why tool designers, aiming at economy of design and at reaching the largest possible user base, have generally designed their tools on top of existing languages. These commonalities even explain why some graphical designers find it reasonable go beyond their traditional tasks and write interactive programs or components [61].

2.3 Discrepancies

So, in some ways, interactive software looks similar to other software. But in our experience with interactive software toolkits we often find ourselves asking "why?" Why do toolkits for interaction offer not only specialised objects such as graphics or interactors but also unique component structuring mechanisms [58] and distinct control flow mechanisms like events or data-flow instead of function calls [18]? In these ways they act on languages more as "corrective patches" than as specialisations, and in some cases we suspect the inconsistencies arising from these patches are actively harmful.

The discrepancies we see between interactive and non-interactive software are in the structure of code itself and in the nature of development processes [18]:

Contravariance in reuse and control. The control flow in interactive systems often goes from the outside to the main program. The code that transfers control (input drivers, interactors) predates the code that receives control (the application). This is the opposite of the situation that function calls were made for, thus requiring events or data-flow.

Locality of state. In interactive systems the complexity is in behaviours, that is in the change of state of objects, and not in computations. Solutions that focus on making computations as local as possible in the code tend to be counter-productive, hence the use of state machines.

Concurrency. More and more, interactive systems involve concurrent processes such as animation. Even when there is no such concurrency, applying software engineering techniques to interactive systems and splitting their code into components turns programmers into assemblers of concurrent processes: interactive components run concurrently. User interface frameworks provide this concurrency in a more or less disguised way.

Different reuse patterns. Very diverse stakeholders are involved in producing interactive applications, from programmers to graphic designers and even to end users. Many reuse scenarios for interaction are not well-supported by encapsulation mechanisms proposed by programming languages. For instance, an end user may want to modify the size of a font in a button; however, this might be considered an implementation detail to be hidden from the programmer.

3 Requirements for interaction-oriented development

The previous section provided a top-down review of the nature of interaction-oriented programming along with the large-grain commonalities and discrepancies between it and computation-oriented programming. Our ultimate intent is to achieve a fine-grained understanding of the same material. The approach we have chosen is to analyse of the requirements for each more finely, in order to understand where computation-oriented programming languages and interaction-oriented tools address similar or compatible requirements and where they address incompatible ones. For this, we enrich Weinberg's analysis framework [95] in two ways. First, we base our analysis on the following sets of activities:

- intellectual activities, where programming resides with mathematics and others;
- various sets of concept manipulation activities, because to build objects you first need to understand the nature of your building blocks and of the objects you want to build;
- construction activities, which include manipulation and evaluation; and
- collaboration activities and more generally production processes.

Second, in the following sections we consider more than fifty tools or research works and use them to refine and populate the resulting framework.

In the future, we plan to add programming language research to this analysis, in the hope that this will bring insight as to how solutions from traditional programming language design can be combined with techniques from interactive software to build better tools and languages for both.

3.1 A human intellectual activity

As an intellectual activity, building interactive systems is similar to other activities where conceptualizing, abstracting, reasoning, inducing and understanding are involved. In this sense, mathematics, logic, physics and engineering are disciplines that share common aspects with programming. Because human intellectual capacity is inherently limited, intellectual usability normally requires minimising complexity. In the case of interactive systems, targets for minimization include information complexity, access complexity and unpredictability.

Minimising information complexity. Some HCI toolkit designers advocate reducing the diversity of manipulated objects (graphical objects, input devices, behaviours, etc.) to the fewest general "atoms" that can be used to describe a whole system.

Few constructs. Occam's razor – entities should not be multiplied beyond necessity – is often cited as general rule for design. Smalltalk, which is arguably an interaction-oriented programming language, has only a few constructs such as classes, instances, and messages (Kay claims inspiration from mathematics and biology [48]). In interaction toolkits, Ubit builds a "molecular architecture" [52] from small-size components called brickgets enable to definition of any kind of user interface element.

Homogeneity. Homogeneity, sometimes referred to as "consistency" or "uniformity," is another source of simplicity. According to Weinberg [95], non-homogeneous environments discourage exploration.

Homogeneity is found in at least two dimensions, which we call horizontal and vertical. Horizontal homogeneity occurs where the same construct applies across a range of situations. For example in Sassafras the same construct is used for input/output, inter-process communication, and function calls [42]. In Flapjax, user input events and network events are described by a single abstraction, the event stream [57].

In vertical homogeneity (also known as "fractality" or "layering") the same structure recursively applies at different levels of composition. For example, Kay describes objects in Smalltalk as recursively incorporating the structure of the entire computer [48]. Brickgets in Ubit can be composed to form more complex brickgets in a hierarchical scene-graph [52]. In DIWA, the recursive decomposition of "user interface objects" provides a frame for locality and isolation [85].

Minimising access complexity. Because software is multi-dimensional, the representation of certain concerns can sometimes be spread across the system description, making them difficult to understand. This can be addressed by using structures that gather and bundle together related representations (increasing locality) or by removing extraneous representations (increasing legibility and conciseness).

Increasing locality. Locality means that the user can find elements of interest in a single place and the need for locality explains certain design choices in HCI toolkits. For instance, according to Myers

> OOP [object-oriented programming] is especially natural for user interface programming since the components of user interfaces (buttons, sliders, etc) are manifested as visible objects with their own state (which corresponds to instance variables) and their own operations (which correspond to methods). [60]

This observation forms part of the original design rationale for Smalltalk [48]. It also helps explain why object-oriented approaches seem more popular for interaction toolkits than functional approaches, in which single behaviours are typically spread across many functions. With non-local code, the user must replace visualization with interaction and memory: switching between files, searching for the related code, and mentally assembling multiple chunk of codes in order to understand it.

Other programming approaches popular in interaction toolkits also provide support for locality. Finite State Machines (e.g., SwingStates [2]) allow control aspects of interactive systems to be localized, contrasting with the "spaghetti-code" [62] inherent in the function-oriented programming paradigm [18]. Reactive programming (Esterel [20]), process-based user interfaces languages (Pike's window system [79]), data-flow (Ituikit [16] and Icon [31]), and functional reactive programming (Arrowized FRP [69]) not only avoid the need to cope with a main loop, but also enable the user to better visualize and understand the sequence of transforms from input to display.

Other examples of strong locality support include MDPC [21] in which picking (identification of the graphical object pointed to in the interface) is insulated in a single place, and Boxer [28], where each interactive feature is isolated in its own "box".

Increasing legibility and conciseness. In our context, legibility refers to the degree to which a developer is able to read and understand code in a reasonable time. Lecolinet suggests that

> GUI source code tends to be verbose and hard to read. Informative data is often hidden in a large amount of "syntactic sugar" that conveys little information but is necessary for proper compilation. This lack of conciseness tends to make programs harder to understand and to maintain. [52]

As used by Lecolinet, "conciseness" is a property that refers to the length and number of constructs needed to express the semantics implemented by a chunk of code. It is (inversely) related to the property of diffuseness in the Cognitive Dimensions Framework [37]. Cordy argues that conciseness is an important feature for interaction languages based on his experience designing Turing [22] and Kay makes a similar case in the context of Smalltalk [48].

The legibility of semantically-necessary code can be hindered by the presence of other code required for syntactic compliance. Removing this code makes the result more concise and improves legibility. This is the approach taken in Ubit, whose syntax resembles a formal specification more than a classic programming language [52]. Similarly, the Event Response Language in Sassafras was designed specifically to be more compact than an equivalent recursive transition network-based language [42].

Minimising unpredictability. Unpredictability sometimes arises when automatic algorithms are used to implement system decisions, and where either the rules by which these algorithms operate are complex or opaque, or the number of interdependent entities managed by the algorithms become large.

Many authors, including Winograd, argue that programming languages and toolkits should be primarily declarative [97]. In interaction-oriented systems an argument in favour of such algorithms is that they allow for concise declaration and automatic management of dependencies, which reduces the information and access complexity [52]. Toolkits that provide constructs for defining behaviours, graphical objects, or transformations without any control construct are sometimes described as declarative, at last in part. Many interaction-oriented systems are more or less declarative in this sense, including Garnet [58], Amulet [65], PAC-Amodeus [68], Ubit [52], and MDPC [21].

Myers notes that constraint-based systems, various flavours of model-based systems, and even simple layout algorithms are examples where the declarative aspect of the program should relieve the programmer from implementation details. However, in practice programmers have difficulty understanding declarative mechanisms well enough to to align their needs to the features that are provided [60]. Kay makes a similar observation regarding difficulties observed in skilled programmers attempting to declaratively specify an algorithm for sorting numbers into odd and even sets [48].

3.2 A concept manipulation activity

Like many activities, programming involves the use of concepts for understanding situations and possible actions as well as for defining desired results. A classical usability requirement is that tools present a consistent view of these concepts. Each domain has its own concepts, and this is one place where it appears that computing-oriented and interaction-oriented programming show interesting differences.

Graphics. Graphics has always been a major area within user interface programming, and many requirements for tools stem from the graphical nature of most user interfaces.

Historically, managing graphics has been focused on hardware and rendering processes. This started for instance with tracers: picking a pen, moving it, putting it on paper, and so on. These concepts were ubiquitous in early standards such as GKS [10]. The importance of the algorithmic approach was enforced by the direct rendering mode used in early raster displays, then by the picking algorithms used to determine which graphical objects are designated. Even today, graphics algorithms play an important role in interaction-oriented programming because developers are still required to deal with hardware limitations and to optimize their code.

A different point of view holds that managing graphics is mainly the manipulation of graphical entities: shapes, layers, visual attributes, etc. The approach was probably first taken in graphics editors and 3D programming tools with the introduction of retained rendering; later, the same concept came to user interface tools. This is an appealing perspective for interaction-based systems since it accords well with the fact that graphical designers, and not just programmers, are involved in the production of user interfaces. Tool designers have then come up with various solutions for structuring and manipulating graphical scenes, which are intended to simplify the development of structured graphics. These range from tree data structures (e.g., Piccolo [7]), to directed acyclic graphs (SVG [9, 17]), to tag-based structures (Tk [2, 74]) Other toolkits are specialized for a particular type of interactive software such as Prefuse [41], or the InfoVis toolkit [46]. In these tools simplicity is balanced with efficiency, since graphical scenes can contain large numbers of elements.

While most toolkits propose a graphics API on top of a programming language, there are some languages developed specifically to describe graphics. Baudel proposes a language that is able to describe a large class of linear visualizations such as scatterplots and treemaps [5]. The language is said to be "compact" and "canonical", in the sense that all text in a program is dedicated to graphics description and the textual description cannot be reduced further. The language uses a data-flow programming style, similar to Processing [33]. Wilkinson designed a grammar and a textual language to describe graphics, together with nVizn, a toolkit based on this language [96]. Protovis is another language dedicated to graphics description which, according to the authors, lowers the entry barriers to creating new visualizations [12].

Runtime adaptation. The execution environment of interactive programs can vary in terms of input and output devices available: mouse, keyboard, trackball, touchscreen, speech input, small or large display, etc. Programmers therefore need ways of describing what devices they wish to use and how. This was first recognized for input devices [42, 60, 84], probably because they have varied more in the past and because their structure cannot always be abstracted: a mouse has buttons, for instance. This need for describing devices includes numerous low-level pragmatic aspects, so the programmer of an interactive system must sometimes turn into a device driver programmer [18].

With these considerations comes the question of allowing programmers to choose between device-dependent and device-independent architectures. The Seeheim architecture of the mid-1980s aims to separate interaction from program logic, which at the time allowed the adaptation from text to graphical interfaces [77]. PAC-Amodeus also offers concepts to adapt to various interaction devices [68].

However, while these systems enable the programmer to abstract input and address the moving target issue as described by Myers [60], some tools may deprive programmers from the benefits of new technologies [42]. Nonetheless, the need to separate the description of the application from the description of the environment led to a series of works on adaptability or plasticity. This converged with the introduction of model-driven user interface development, where the user task and the environment are described with different language [90, 91].

Interaction modalities. Programmers of interactive software need to produce code that allows humans to interact with the machine. This potentially covers a huge range of interaction modalities and languages, from low-level perception-action loops to natural spoken language. In the absence of a grand unified theory of interaction this means many different and potentially mutually incompatible sets of concepts, from high level logic to physical models. This explains why various proposals, more or less distant from the concepts of the Turing machine, have been made for describing different modalities or concerns.

State representation. Various interaction modalities resemble network protocols in that the state of each party and its variations capture the nature of the interaction. This is true of command line dialogue, for which finite state machines were proposed early [67]. This also holds for individual widgets (buttons, menus, etc) for which state machines were reused, then improved with Statecharts [39]. State machines have also been proposed for direct manipulation and multimodal interfaces [17, 58]. More recently, they became so ubiquitous that they became a central part of UML diagrams, and adapted for mainstream languages [2].

Connections between properties. Some parts of direct manipulation, and more generally low level perception-action coupling, are best described as connections between properties of objects. This has led researchers to propose data-flow as a control mechanism [16, 19, 31, 58]. Some also saw a link with data iteration and proposed to combine this with functional programming.

Time. Animation and some types of time-sensitive input, including multimodal interaction [68], often happen in parallel and require a good representation of time [84, 61]. Specific solutions have been proposed for this, including the use of temporal logic.

Algorithms. Some specific interaction modalities, singularly gesture recognition, rely on computation and algorithms. These fit well in the functional paradigm, less in the more reactive ones. Some have proposed that incertitude in recognition be treated with mechanisms similar to errors and exceptions in imperative languages.

Things become more complex with modern user interfaces that combine all of the above modalities: the requirements add up but none of the proposed solutions satisfy all requirements. Programmers have to choose one solution and deal with the resulting complexity. Several approaches have been proposed to address this issue. One is the introduction of formalisms meant to cover the whole range of possible interactions. See Harrison and Duke [40] and Brun and Beaudouin-Lafon [14] for reviews of this approach.

Generally, there is a growing understanding that reactive programming [42, 47] and more generally parallel programming are more suited to interaction-oriented programming than the traditional sequential programming used for computations. This realization started with the debate on internal and external control [23] and reached the state where a growing number of researchers agree with Wegner that "interaction is more powerful than algorithms" [93], which means that computing is a special case of interaction and not the opposite.

Distribution. Modern interactive systems frequently include distributed users [3] and external and potentially non-anticipated interaction devices [4]. This normally necessitates a distributed system implementation, which places additional burdens on developers and the effects of which cannot be completely hidden from end users.

Important requirements for interaction-oriented distribution include entity naming [56] and concurrency control and consistency maintenance for shared and replicated artifacts [78]. Entity and device discovery and subscription features to allow dynamic adaptation to changing external environments and user needs are also required [4].

3.3 A constructive activity

The activity of constructing an interactive system can be described using Norman's model of action [71]. This provides us with a frame for addressing gulfs and discontinuities in programming tasks: how do the tools make it easier for the programmer to tell what actions are possible, to determine mappings between intention and action, to perform the action (e.g., write the code), to interactively specify the graphical user interface, and to determine the mapping from system state to interpretation? The usability requirements follow two main lines: supporting code production, thus reducing the gulf of execution, and how to helping programmers match code and execution, in order to minimize the gulf of evaluation. In this section we focus on programming as an individual activity; we address collaborative development in section 3.4.

Supporting code production. Code production can be supported by allowing easy imitation of prior systems, by supporting exploration of the potential solution space, and by automating the generation of code where appropriate.

Imitation. Programmers seldom start new programs from an empty page and without prior domain knowledge. Rather, they will often take some code they have already programmed for a similar task and modify it for the new one, or they will search for similar programs or useful program fragments on the Internet and imitate them [70]. Brandt describes this as "opportunistic programming" where programmers perform web searches for "just-in-time learning by doing" or "web auto-completions" when they do not recall the name of an API function [13]. A study of professional developers working with Alice concluded that interaction-oriented programmers need graphical copy-and-paste and histories for each graphical object to enable this type of imitation [50].

Programmers also use prior knowledge at the level of architectures, rules, design patterns and programming plans, based on existing documentation, their own experience as a programmer [26], and on discussions with colleagues [33]. High-level user interface patterns such as those documented by Schummer [83] and Borchers [11] can be used, as can the more technical patterns from the famous "Gang of Four" book [34], which contains numerous examples drawn from graphical toolkits like ET++ [94]: composite for scene graphs, abstract factory for supporting various graphical standards, command for undo facilities, etc. Imitation is not precisely reuse (see section 3.4): software reuse means referencing other elements, not copying them.

Exploration. Translating intentions into actions also requires support for exploratory manipulation of possible solutions. Weinberg argues that exploration of both problem and solution often occur at the same time [95]. This applies fully to interactive systems, where the solution is normally difficult to define without several iterations [59]. Exploration is fostered by homogeneous environments (see section 3.1) and by being able to combine elements, which is the case in Ubit [52] where "brickgets" and behaviors can be combined to construct new interaction techniques (multiple pointers, multiple remote displays, semantic zooming, multi-scale, transparency, and control-menus). The possibility of exploring also assumes progressiveness: for example, Stylos et al. the authors demonstrate that API providing constructors without parameters are easier to explore [89].

Prototype-based languages are often advocated for a more exploratory development of the user interface since changes in slots (such as a graphical aspect) can be dynamically propagated to the whole instances at run-times [63]. Noble compares using prototype-based languages to

using a document editor, where copying an existing document is more direct and easy to grasp than using a template [70]. The developers of Apple's Newton also advocate for using prototype-based languages:

> The needs of the user-interface side of the program are different. In contrast to the model, the user interface usually consists of relatively few objects, most of which appear only once in a given context, and most of which are unique in small but significant ways. [...] It is easier to reason about and control the interactions of individual objects—the usual requirement for UI programming—when the objects themselves are being programmed directly. [86]

Exploration also benefits from introspection mechanisms that help the programmer understand or access the underlying structure or manipulate the source code. ET++ provide meta-classes for building inspectors [94], InfoVis enables a deep access in the toolkit [46], and HGraph provides mechanisms for the programmer to make sense of the running system [35]. Hudson advocates for method interposition and representation of the code suited for manipulation by programs (e.g., as a tree), so the syntax is only an environmental matter [44].

Automatic programming. Some tools relieve the programmer from coding parts of the interactive system. These include interface builders like Garnet [58], or more recently tools such as Apple's Interface Builder, Qt Designer, Eclipse, Tk Komodo, and Expression Blend from Microsoft; as well as model-based interface generators such as Mickey [72] and HUMANOID [90]. In the same vein, programming-by-demonstration techniques enable the user as a programmer, to program by showing the sequence of actions to the system [24, 54]. This of course raises the issue of how to integrate automatic code generation and more conventional textual coding into the same environment.

Matching code and execution. Once code has been developed, it must be verified against the developers' original intent, validated against the users' needs, and modified as necessary.

Evaluation. Myers notes the low testability of interaction-oriented software; classical regression testing is adapted to computation-oriented programs [59]. In interactive software, it is not clear whether the evaluation steps (perception, interpretation, and comparison to the goal) should be performed on the program as a specification, by verification tools, or on the program as a dynamic artifact. Another difficulty is that the "result" of an interactive systems is not clear-cut. Interaction can lead to several potential solutions that need to be compared, but Myers also observes, while studying how designers specify interactive systems, that it is difficult to compare explored solutions, particularly for descriptions of behaviour [61]. One school of thought considers Model Driven Engineering as a means of avoiding such concerns by performing tests on models and relying on model refinement to working code [76]; however, the effectiveness of this approach is not yet proven.

Debugging. Authors observe how difficult it is to debug interactive software. In some cases, the tools that enable the perception of the program disturb its flow of control and thus its correct behaviour [45]. In others, the perceptible result of the program is exceptionally difficult to relate to the code that produces it [55].

Tools for visualizing program execution have been proposed to address some aspects of this issue. Debugging lenses provide access to the state of the system by enabling the programmer to see information about the attributes of interface elements using movable floating windows [45]. ZStep offers mechanisms for understanding the behaviour of the program by stepping through graphical changes in the user interface, as opposed to stepping through lines of code [92]. The Whyline allows developers to perceive and interpret the state of the system in terms of the actions that produced it: which statement has set which attribute, why does this window encounter this change, and so on [45]. SwingStates provides a visual depiction of its finite state machines' dynamics, allowing them to be related to interface changes [2].

Interactive coding. Of course, constructing a program does not involve sequential execution and evaluation steps, as might be implied by the previous two sections [71]. During the problem-solving process, the programmer plays with potential solutions, conducting a "conversation with the medium" as described by DiSessa [28, 29]. Progression of the work can be helped by giving an appropriate continuous feedback [71]. The gap between the representation of actions and the representations of results can be reduced by tools that enable a continuous flow through the successive actions [33]. Ideally, programming and debugging should be tightly integrated, either by designing development environments with the dual perspective in mind [8], or by having the two occur within the same representation [92, 45]. Being able to perform execution and action within the same medium enables a progressive evaluation of the program being constructed [37]. This is the general idea behind programming-in-the-user-interface (PITUI), in which the user can switch to a programming mode by using affordances in the tool to modify its behaviour [27, 29, 30, 32, 35].

3.4 A process-based activity

The requirements described so far address aspects of the objective expressivity of tools and languages with respect to to the concepts to be described, and the subjective expressivity of the construction of software pieces. Building interactive software requires support for processes where software is not only produced, but also managed over time, either individually or as collective processes including sharing, reuse and communication.

Managing the life cycle. Specific requirements of interactive systems design led to the advent of participative (user in-the-loop) iterative development processes [59]. A number of tools aim to shorten the iteration cycle, particularly those dedicated to prototyping, beginning with SILK [51] and culminating in Microsoft Expression Sketchflow.

Implementing an interactive system requires a choice of languages, toolkits, and design environment. This choice is guided by the needs discussed throughout this article, some of them closely related to life cycle. For instance, developing on top of a portable language or platform and extending it through a library is a means to reduce set-up cost of development tools. This can be implicitly targeted by tool designers, as in SwingStates' addition of state machines to Java [2]. It can also be addressed explicitly, for example in web browser-based tools like Balsamiq [88] and the Processing IDE HasCanvas [73].

Managing reuse and knowledge capitalization. Reuse is a major concern in software engineering. Its claimed benefits include improved productivity, correctness, efficiency and even safety, since a widely reused solution is likely to benefit from broad testing and maintenance. Reuse can be can be opportunistic or planned and can be seen as a concrete subset of knowledge capitalization.

In interaction-oriented system design, knowledge capitalization can be applied at different levels. At the code and architecture level, some pre-cut lines have been identified. The best known are splitting interaction from application code [23] and abstracting from input devices. This latter is a mainstay of interaction-oriented toolkits, which offer widgets to attach to application code without the requirement to address low-level input device management. In addition, these libraries contribute to the homogeneity of the final interface. However, as argued by Chatty, offering pre-defined "Lego bricks" to interface developers is not enough [18]. As innovation plays an important role in interactive system design, developers will want to build their own bricks or bypass abstraction layers to allow for, e.g., working at the driver level to support a novel input device. Tools can assist developers to address these requirements in two ways: elementary bricks can be assembled into larger ones with the same properties; and homogeneous component APIs

help designers to build their own. For reuse to be effective, it is also necessary that development environments support and encourage the creation of reusable artifacts.

In addition to code and architecture concerns, the interactive system domain is rich in models and code design patterns that provide elements of solutions for designers [2, 9, 31, 39, 6]. For example, MVC is a design pattern for code that aims to improve modularity by supporting separation of concerns between input, output, and represented state [80]. MDPC improves modularity by distinguishing the display view from the picking view [21]. Vigo is a pattern that helps implementing interactive systems based on instrumental interaction [49, 6]. Used this way, models and code design patterns allow knowledge capitalization and support best practices. Additionally, models provide formalism and semantics, and can be instrumented, or turned into a libraries.

A higher level of reuse involves creating guidelines or interaction patterns by matching common problems to known solutions [11, 83]. Codification of patterns and architectures also answers a need for shared knowledge to permit communication between team members.

Managing collective development The multiplication of roles in interaction design and development brings new problems in managing and syncing teams working on different tools and objects of interest.

Text-based revision control systems pioneered in early 1970s are widely used and have proven efficient for synchronizing the work of large development teams [81]. Unfortunately, to the best of our knowledge there are no tools of the same maturity for dealing with graphics. Some graphics, particularly vector-based graphics with textual representations like SVG, can be handled by code revision systems; however, interpreting change history for such files is error-prone and tedious.

Iterative processes also reinforce the need for tools such as sandboxes and branching, to free creativity and safely extend the design space, along with a convenient way for combining such heterogeneous artifacts as code, mock-up drawings and gestural grammars. Some propose a glue in the form of a middleware [15]; others answer this need by providing a common framework for multidisciplinary teams like IntuiKit [17] and Microsoft WPF/XAML. Ideally, each member of the development team would be presented with a role-appropriate representation of each development artifact [18].

4 Related work

There have been a number of studies of programming interfaces in areas where the difficulty of programming enforces the need for a careful design. Approaches vary by the targeted users (professional programmers, end-user programmers, novices), the types of tools studied (programming languages or APIs, generic or interaction-oriented tools), and the viewpoint taken (usability study, language or API design, survey).

Novice and End-User Programming. Some of the most prominent early studies of human aspects of programming are by Weinberg [95], Hoc et al. [43] and Soloway and Spohrer [87]. These books primarily address the psychology of novice programmers, focusing on the cognitive difficulties and educational aspects. Weinberg also describes the psycho-sociological aspects of programming [95]. The field of end-user programming is very concerned with usability issues and the design of tools and languages for unsophisticated users. Significant work in this area includes Cypher [24] and Lieberman [53, 54]. Our study targets multi-disciplinary teams which may include both professional programmers and end-user programmers such as designers.

Usability studies. Several studies have addressed the usability of languages and toolkits, including those of Myers [64] and Agarwal [1], the latter reporting on the usability of object-oriented representations. Myers' group has an ongoing project, under the name Natural Pro-

gramming (`www.cs.cmu.edu/~NatProg/`) which investigates how to design more usable programming languages and systems [66, 75]. There is a SIGCHI group [25] and related web site (`apiusability.org`) devoted to API usability.

Green's cognitive dimensions framework [37] has influenced much research on the usability of computing artifacts, including programming languages. Clarke has applied cognitive dimensions to a class library [82] and Green has investigated the cognitive dimensions of visual languages [38].

Most usability studies we are aware of target general purpose languages or APIs rather than tools for building interactive systems. Exceptions include Ko's study of Alice programmers [50] and Myers' study of the programming practices of graphical designers [61].

Other surveys. This paper is conceptually related to several reviews that have been conducted in to help state directions for future research. Brun and Beaudouin-Lafon's taxonomy attempts to compare formalisms for describing user interfaces according to their expressive power, their generative capabilities, and their extensibility and usability [14]. Our classification partly matches this work, but the proposed taxonomy does not address programming tools.

In [59], Myers addresses the challenges of programming user interfaces: this approach, by focusing on why interactive systems are difficult to build, is similar to ours, but we wanted to identify all the needs, not only the ones related to failures. In a later study, Myers et al. describe and evaluate software tools according to five themes: parts of the user interface addressed, threshold and ceiling, path of least resistance, predictability, and moving targets. Their study identifies successful and less successful approaches, with the aim to draw lessons for the design of future tools [60]. Where Myers classifies the tools themselves, we attempt to capture and classify the underlying requirements for tools.

In the book "Languages for developing user interfaces" several chapters address interaction-oriented toolkits or languages. Hudson identifies how programming languages might better support user interface tools [44]: this approach highlights technical needs, but leaves cognitive and collective aspects aside. Our approach is quite similar to that of Cordy, who looks at the design behind the Turing general-purpose programming language to discover ideas that might be applied to tools for interactive systems [22]. Singh identifies three main requirements for a user interface language: object-orientation, time as a first-class object and interactive programming, but his aim is to identify a unique best language [84]. Finally, Graham provides a summary of the book and related workshop, focussing his discussion on whether interactive system tools development needs one language, several languages or an API [36].

5 Conclusion

In this article we began with an analysis of the differences between computing-oriented programming and interaction-oriented programming in an attempt to understand why the latter appears unreasonably difficult with current tools. We then proposed a requirement analysis framework, the ultimate aim of which is to clarify where these two classes of programming have common requirements and where they diverge, and populated this framework through a broad survey of research in interaction-oriented programming.

In the future we plan to populate the framework with a survey of the requirements underlying computation-oriented programming languages. The fully-populated framework will provide a basis for analysis of the commonalities and discrepancies between requirements for interaction- and computation-oriented programming. Our ultimate aim is to identify mechanisms that will allow us to seamlessly address the requirements of both approaches, in hopes that better development tools can then be designed.

Acknowledgements

This work was supported by ANR (project Istar), Aerospace Valley and FUI (project ShareIT). Comments from the anonymous reviewers provoked significant presentational improvements.

References

1. Ritu Agarwal, Prabuddha De, Atish P. Sinha, and Mohan Tanniru. On the usability of OO representations. *CACM*, 43(10):83–89, October 2000.
2. Caroline Appert and Michel Beaudouin-Lafon. SwingStates: adding state machines to the Swing toolkit. In *Proceedings of ACM UIST'06*, pages 319–322, Montreux, Switzerland, 2006. ACM.
3. R.M. Baecker. *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration*. 1993.
4. Rafael Ballagas, Meredith Ringel, Maureen Stone, and Jan Borchers. istuff: a physical user interface toolkit for ubiquitous computing environments. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 537–544, New York, NY, USA, 2003. ACM.
5. Thomas Baudel. Visualisations compactes: une approche déclarative pour la visualisation d'information. In *Actes de la conférence IHM'02*, pages 161–168. ACM, 2002.
6. Michel Beaudouin-Lafon. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proceedings of ACM CHI'00*, pages 446–453. ACM, 2000.
7. Benjamin B. Bederson, Jesse Grosjean, and Jon Meyer. Toolkit design for interactive structured graphics. *IEEE Trans. Softw. Eng.*, 30(8):535–546, 2004.
8. O. W. Bertelsen and S. Bødker. Studying programming environments in use: between principles and praxis. In *Proceedings of NWPER'98, the Eighth Nordic Workshop on Programming Environment Research*, 1998.
9. R. Blanch, M. Beaudouin-Lafon, S. Conversy, Y. Jestin, T. Baudel, and Y.P. Zhao. Concevoir des applications graphiques interactives distribuées avec INDIGO. *Revue d'Interaction Homme-Machine*, 7(2):113–140, 2006.
10. P Bono, J. Encarnação, R. Hopgood, and P. ten Hagen. GKS – the first graphics standard. *IEEE Computer Graphics and Applications*, 1982.
11. Jan Borchers. *A Pattern Approach to Interaction Design*. John Wiley & Sons, 2001.
12. Michael Bostock and Jeffrey Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15:1121–1128, 2009.
13. J. Brandt, P. Guo, J. Lewenstein, M. Dontcheva, and S. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proc. of CHI'09*, pages 1589–1598. ACM, 2009.
14. Philippe Brun and Michel Beaudouin-Lafon. A taxonomy and evaluation of formalisms for the specification of interactive systems. In *Proceedings of HCI'95*, pages 197–212. Cambridge University Press, aug 1995.
15. M. Buisson, A. Bustico, S. Chatty, F-R. Colin, Y. Jestin, S. Maury, C. Mertz, and P. Truillet. Ivy: un bus logiciel au service du développement de prototypes de systèmes interactifs. In *Proc. of IHM'02*, pages 223–226, Poitiers, France, 2002. ACM.
16. S. Chatty. Extending a graphical toolkit for two-handed interaction. In *Proc. of UIST'94*, pages 195–204.
17. S. Chatty, S. Sire, J.L. Vinot, P. Lecoanet, A. Lemort, and C. Mertz. Revisiting visual interface programming: creating GUI tools for designers and programmers. In *Proceedings of UIST'04*, pages 267–276. ACM, 2004.
18. Stéphane Chatty. Programs = data + algorithms + architecture, and consequences for interactive software. In *Proceedings of IFIP EIS 2007*, 2007. LNCS, Springer Verlag.
19. Stéphane Chatty, Alexandre Lemort, and Stéphane Valès. Multiple input support in a model-based interaction framework. In *Tabletop*, pages 179–186, 2007.
20. Dominique Clément and Janet Incerpi. Specifying the behavior of graphical objects using Esterel. In *TAPSOFT, Vol.2*, pages 111–125, 1989.
21. Stéphane Conversy, Eric Barboni, David Navarre, and Philippe Palanque. Improving modularity of interactive software with the MDPC architecture. In *Proceedings of EIS 2007*, pages 321–338, 2008. LNCS, Springer.
22. James Cordy. *Languages for developing user interfaces*, chapter Hints on the design of user interface language features: lessons from the design of Turing, pages 329–340. A. K. Peters, Ltd., 1992.
23. Joëlle Coutaz and L. Bass. Requirements on UIMS's. In *Proceedings of the Workshop on UIMS and Environments, Lisbon*, 1990.
24. Allen Cypher. *Watch What I Do. Programming by Demonstration*. MIT Press, 1993. 652 pages.
25. John M. Daughtry, Umer Farooq, Brad A. Myers, and Jeffrey Stylos. API usability: report on special interest group at CHI. *SIGSOFT Softw. Eng. Notes*, 34(4):27–29, 2009.
26. S. Davies. The nature and development of programming plans. *Int. J. of Man-Machine Studies*, 32(4):461–481, 1990.
27. Chris DiGiano and Michael Eisenberg. Self-disclosing design tools: a gentle introduction to end-user programming. In G. Olson and S. Schuon, editors, *In Proc. DIS'95*, pages 189–197. ACM Press, 1995.
28. Andy DiSessa. *Changing Minds: Computers, Learning, and Literacy*. MIT Press, 1999.
29. Andy DiSessa and H. Abelson. Boxer: a reconstructible computational medium. In *Studying the Novice Programmer*, pages 467–481. Lawrence Erlbaum Associates, 1989.

30. Paul Dourish. Using metalevel techniques in a flexible toolkit for CSCW applications. *ACM Transactions on Computer-Human Interaction*, 5(2):109–155, June 1998.
31. Pierre Dragicevic and Jean-Daniel Fekete. Support for input adaptability in the ICON toolkit. In *Proceedings of ICMI'04*, pages 212–219. ACM, 2004.
32. Michael Eisenberg. Programmable applications: Interpreter meets interface. *ACM SIGCHI Bulletin*, 27(2):68–93, April 1995.
33. Benjamin Jotham Fry. *Computational information design*. PhD thesis, MIT, 2004.
34. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
35. Tony Gjerlufsen, Mads Ingstrup, Jesper Wolff, and Olsen Olsen. Mirrors of meaning: Supporting inspectable runtime models. *Computer*, 42:61–68, 2009.
36. T. C. Nicholas Graham. *Languages for developing user interfaces*, chapter Future research issues in languages for developing user interfaces, pages 401–418. A. K. Peters, Ltd., 1992.
37. T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of HCI'89*, pages 443–460. Cambridge University Press, 1989.
38. T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing, 7*, pages 131–174, 1997.
39. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
40. Michael D. Harrison and David J. Duke. A review of formalisms for describing interactive behaviour. In *ICSE Workshop on SE-HCI*, pages 49–75, 1994.
41. Jeffrey Heer, Stuart K. Card, and James A. Landay. Prefuse: a toolkit for interactive information visualization. In *Proceedings of ACM CHI'05*, pages 421–430. ACM, 2005.
42. Ralph Hill. Supporting concurrency, communication, and synchronization in human-computer interaction—the Sassafras UIMS. *ACM Trans. Graph.*, 5(3):179–210, 1986.
43. J-M. Hoc, T. R. G. Green, D.J. Gilmore, and R. Samurçay, editors. *The Psychology of Programming*. Academic Press, 1991.
44. Scott Hudson. *Languages for developing user interfaces*, chapter How programming languages might better support user interface tools, pages 105–113. A. K. Peters, Ltd., 1992.
45. Scott E. Hudson, Roy Rodenstein, and Ian Smith. Debugging lenses: a new class of transparent tools for user interface debugging. In *Proceedings of ACM UIST '97*, pages 179–187. ACM, 1997.
46. Jean-Daniel Fekete. The InfoVis Toolkit. In *Proceedings of the 10th IEEE Symposium on Information Visualization (InfoVis 04)*, pages 167–174, Austin, TX, October 2004. IEEE Press.
47. Jean-Daniel Fekete and Martin Richard and Pierre Dragicevic. Specification and verification of interactors: A tour of Esterel. In *Proceedings of FAHCI'98*, September 1998.
48. Alan C. Kay. The early history of Smalltalk. In *HOPL Preprints*, pages 69–95, 1993.
49. Clemens Nylandsted Klokmose and Michel Beaudouin-Lafon. Vigo: instrumental interaction in multi-surface environments. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 869–878, New York, NY, USA, 2009. ACM.
50. Andrew Jensen Ko. A contextual inquiry of expert programmers in an event-based programming environment. In *CHI'03 extended abstracts*, pages 1036–1037. ACM, 2003.
51. James A. Landay and Brad Myers. Interactive sketching for the early stages of user interface design. In *Proceedings of CHI'95*, pages 43–50. ACM Press, 1995.
52. E. Lecolinet. A molecular architecture for creating advanced GUIs. In *Proc. of UIST '03*, pages 135–144.
53. H. Lieberman, F. Paterno, and V. Wulf, editors. *End-User Development*. Kluwer/Springer, 2005.
54. Henry Lieberman, editor. *Your Wish is My Command: Giving Users the Power to Instruct their Software*. Morgan-Kaufmann, 2000.
55. Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In *ACM CHI'95 Summaries and demonstrations*, pages 480–486. ACM Press, 1995.
56. Blair MacIntyre and Steven Feiner. A distributed 3d graphics library. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 361–370, New York, NY, USA, 1998. ACM.
57. L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for Ajax applications. In *Proceedings of OOPSLA '09*. ACM, 2009.
58. B. Myers, D. Giuse, A. Mickish, B. Vander Zanden, D. Kosbie, R. McDaniel, J. Landay, M. Golderg, and R. Pathasarathy. The Garnet user interface development environment. In *CHI'94 Conference companion*, pages 457–458. ACM, 1994.
59. Brad Myers. Challenges of HCI design and implementation. *Interactions*, 1(1):73–83, 1994.
60. Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, 2000.
61. Brad Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew Ko. How designers design and program interactive behaviors. In *Proceedings of IEEE VL/HCC '08)*, 2008.
62. Brad A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proceedings of ACM UIST '91*, pages 211–220. ACM, 1991.

63. Brad A. Myers. *Languages for developing user interfaces*, chapter Ideas from Garnet for Future User Interface Programming Languages, pages 147–157. A. K. Peters, Ltd., 1992.
64. Brad A. Myers. Usability issues in programming languages. Technical report, School of Computer Science, Carnegie Mellon University, 2000. Part of the Natural Programming Project.
65. Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The amulet environment: New models for effective user interface software development. *IEEE Trans. Softw. Eng.*, 23(6):347–365, 1997.
66. Brad A. Myers, John F. Pane, and Andy Ko. Natural programming languages and environments. *CACM*, 47(9):47–52, September 2004.
67. William M. Newman. A system for interactive graphical programming. In *Proceedings of the AFIPS '68 Spring joint computer conference*, pages 47–54, Atlantic City, New Jersey, 1968. ACM.
68. Laurence Nigay and Joëlle Coutaz. A generic platform for addressing the multimodal challenge. In *Proceedings of CHI'95*, pages 98–105. ACM Press/Addison-Wesley Publishing Co., 1995.
69. Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the ACM SIGPLAN Haskell'02 workshop*, pages 51–64. ACM, 2002.
70. James Noble. Prototype based user interfaces. In *Proceedings of the ECOOP'97 Workshop on Prototype Based Object Oriented Programming*, 1997.
71. Donald A. Norman. *The Psychology of Everyday Things.* Perseus Books, 1988.
72. D. Olsen. A programming language basis for user interface. In *Proc. of CHI '89*, pages 171–176. ACM, 1989.
73. Robert O'Rourke. HasCanvas. http://www.hascanvas.com/, April 2009.
74. John K. Ousterhout. *Tcl and the Tk Toolkit.* Flatbrain Com, 1996.
75. J. Pane, C. Ratanamahatana, and B. Myers. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. of Human-Computer Studies*, 54(2):237–264, February 2001.
76. J-L. Pérez-Medina, S. Dupuy-Chessa, and A. Front. A Survey of Model Driven Engineering tools for User Interface design. In *Task Models and Diagrams for UI Design*, pages 84–97. Springer Berlin, 2007.
77. Günther E. Pfaff, editor. *User Interface Management Systems.* Eurographics Seminars. Springer, 1985.
78. W. Greg Phillips. Architectures for synchronous groupware. Technical Report 1999-425, Queen's University, Kingston, Ontario, Canada, May 1999. Available from `www.cs.queensu.ca`.
79. Rob Pike. A concurrent window system. *Computing Systems*, 2(2):133–153, 1989.
80. Trygve Reenskaug. Models - views - controllers. Technical report, Xerox PARC, December 1979.
81. M. Rochkind. The source code control system. *IEEE Trans on Soft. Engineering*, 1(4):364–370, Dec. 1975.
82. Clarke S. and Becker C. Using the cognitive dimensions framework to evaluate the usability of a class library. In *Proceedings of the First Joint Conference of EASE PPIG (PPIG 15)*, 2003.
83. T. Schummer and S. Lukosch. *Patterns for Computer-Mediated Interaction.* John Wiley & Sons, 2007.
84. Gurminder Singh. *Languages for developing user interfaces*, chapter Requirements for user interface programming languages, pages 115–123. A. K. Peters, Ltd., 1992.
85. J. Six H.-W., Voss. A software engineering perspective to the design of a user interface framework. In *CompSAC'92: Proceedings of the Computer Software and Applications Conference, Chicago 1992*, 1992.
86. Walter R. Smith. Using a prototype-based language for user interface: The Newton project's experience. In *OOPSLA '95*, pages 61–72, 1995.
87. E. Soloway and J. Spohrer, editors. *Studying the Novice Programmer.* Lawrence Erlbaum Associates, 1989.
88. Balsamiq SRL. Balsamiq Mockups. http://www.balsamiq.com/products/mockups/.
89. Jeffrey Stylos, Steven Clarke, and Brad Myers. Comparing API design choices with usability studies: A case study and future directions. In *Proceedings of the 18th PPIG Workshop*, 2006.
90. Pedro Szekely, Ping Luo, and Robert Neches. Facilitating the exploration of interface design alternatives: the HUMANOID model of interface design. In *Proceedings of CHI '92*, pages 507–515. ACM, 1992.
91. David Thevenin and Joëlle Coutaz. Plasticity of user interfaces: Framework and research agenda. In *Proc. of Interact'99*, pages 110–117. IFIP IOS Press, 1999.
92. D. Ungar, H. Lieberman, and C. Fry. Debugging and the experience of immediacy. *CACM*, 40(4):38–43, April 1997.
93. Peter Wegner. Why interaction is more powerful than algorithms. *Commun. ACM*, 40(5), May 1997.
94. Andre Weinand, Erich Gamma, and Rudolph Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, 1989.
95. Gerald M. Weinberg. *The Psychology of computer programming.* Dorset House Publishing, 1979.
96. Leland Wilkinson. *The Grammar of Graphics (Statistics and Computing).* Springer., 2005.
97. Terry Winograd. Beyond programming languages. *Commun. ACM*, 22(7):391–401, 1979.