Interacto: A Modern User Interaction Processing Model

Arnaud Blouin, Jean-Marc Jézéquel

Interacto: A Modern User Interaction Processing Model

Arnaud Blouin and Jean-Marc Jézéquel

Since most software systems provide their users with interactive features, building user interfaces (UI) is one of the core software engineering tasks. It consists in designing, implementing and testing ever more sophisticated and versatile ways for users to interact with software systems, and safely connecting these interactions with commands querying or modifying their state. However, most UI frameworks still rely on a low level model, the bare bone UI event processing model. This model was suitable for the rather simple UIs of the early 80's (menus, buttons, keyboards, mouse clicks), but now exhibits major software engineering flaws for modern, highly interactive UIs. These flaws include lack of separation of concerns, weak modularity and thus low reusability of code for advanced interactions, as well as low testability. To mitigate these flaws, we propose *Interacto* as a high level user interaction processing model. By reifying the concept of user interaction, *Interacto* makes it easy to design, implement and test modular and reusable advanced user interactions, and to connect them to commands with built-in undo/redo support. To demonstrate its applicability and generality, we briefly present two open source implementations of *Interacto* for Java/JavaFX and TypeScript/Angular. We evaluate *Interacto* interest (1) on a real world case study, where it has been used since 2013, and with (2) a controlled experiment with 44 master students, comparing it with traditionnal UI frameworks.

Index Terms—user interface, user interaction, UI event processing, separation of concerns, undo/redo

1 Introduction

"*Anytime you turn on a computer, you're dealing with a user interface*" [1]. User Interfaces (UIs), and the user interactions they supply, pervade our daily lives by enabling users to interact with software systems. The user interactions provided by a UI form a dialect between a system and its users [2]: a given user interaction can be viewed as a sentence composed of predefined words, i.e. low-level UI events, such as mouse pressure or mouse move. For example, we can view the execution of a drag-and-drop interaction as a sentence emitted by a user to the system. This sentence is usually composed of the words *pressure*, *move*, and *release*, that are UI events assembled in this specific order. The human-computer interaction community designs novel and complex user interactions. As explained by [3], "*Human-Computer Interaction (HCI) researchers have created a variety of novel [user] interaction techniques and shown their effectiveness in the lab [...]. Software developers need models, methods, and tools that allow them to transfer these techniques to commercial applications.*" Currently, to use such novel user interactions in software systems developers must complete two software engineering tasks: (i) They must assemble low-level UI events to build the expected user interaction. For example, a developer must manually assemble the events *pressure*, *move*, and *release* to build a drag-and-drop. (ii) They have to code how to process such UI events when triggered by users.

- *M. Blouin and M. Jézéquel were with Univ Rennes, Inria, CNRS, IRISA, France. E-mail: firstname.lastname@irisa.fr*

To do so, developers still use a technique proposed with SmallTalk and the *Model-View-Controller* (MVC) pattern in the 80's [4]: the UI event processing model, currently implemented using callback methods or reactive programming [5] libraries. This model considers low-level UI events as the first-class concept developers can use for coding and using increasingly complex user interactions not supported off-the-shelf by graphical toolkits. The reason is that interacting with classical widgets (*e.g.*, buttons, lists, menus) is usually one-shot: a single UI event, such as a mouse pressure on a button or menu, has to be processed. For more complex user interactions such as the drag-and-drop, the current event processing model exhibits critical software engineering flaws that hinder code reuse and testability, and negatively affect separation of concerns and code complexity:

- the concept of user interaction does not exist, so developers have to re-define user interactions for each UI by re-coding them using UI events;
- the model does not natively support advanced features, such as cancellation (undo/redo), event throttling, or logging;
- developers mix in the same code the assembly of UI events and their processing, leading to a lack of separation of concerns;
- the use of callbacks to process UI events (1) can lead to "spaghetti" code [6], [7]; (2) is based on the *Observer* pattern that has several major drawbacks [8], [9], [10], [11]; (3) can be affected by design smells [12];

This paper makes the following software engineering contribution: a user interaction processing model called

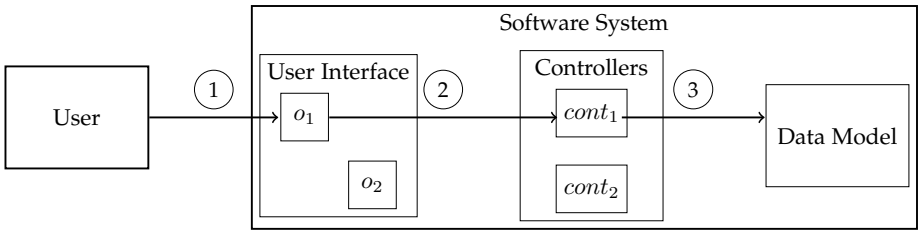

Fig. 1: Standard behavior of the UI event processing model:
(1): A user interacts with an interactive object o_1 of the user interface.
(2): The interactive object then triggers a UI event gathered by a controller $cont_1$.
(3): The controller contains an event callback that processes this UI event to possibly modify the business data.

Interacto that overcomes the above-mentioned flaws of the UI event processing model. *Interacto* reifies user interactions and UI commands as first-class objects and provides dedicated algorithms, object-oriented properties, run-time optimizations, and testing facilities to permit developers to stay focused on the core tasks of coding UIs: (i) select the user interactions they have to use; (ii) code how to turn these user interactions into undoable UI commands; (iii) reuse user interactions and UI commands in different places across software systems; (iv) write UI tests. In this model, UI events are now considered as low-level implementation concepts rarely used by developers.

To demonstrate its applicability and generality, we developed two implementations of *Interacto*: *Interacto-JavaFX* with Java and JavaFX [13], a mainstream Java graphical toolkit; *Interacto-Angular* with TypeScript [14] and Angular [15], a mainstream Web graphical toolkits. Both implementations take the form of a fluent API (*Application Programming Interface*) [16]. We evaluate *Interacto* interest:

- on a real world case study: the development LaTeXDraw, an open-source highly interactive vector drawing editor for LaTeX, downloaded 1.5 k times per month and available on more than 10 Linux distributions.
- with a controlled experiment with 44 master students, comparing it with traditional UI frameworks.

The paper is organized as follows. Section 2 introduces the background concepts and motivates the work by detailing the limits of the current UI event processing model. Section 3 details the proposed user interaction processing model and its testing support. Section 4 evaluates the proposal. Section 5 discusses the related research work. Section 6 concludes the paper and discusses future work.

2 Background and Motivations

2.1 Definitions

The standard UI event processing model involves the following concepts, as depicted by Figure 1.

User Interface. A UI allows users to control or query a software system. The most common kind of user interfaces are Graphical User Interfacess (GUIs). A UI is composed of interactive objects, such as buttons or canvases for GUIs.

UI event. When a user interacts with an interactive object, this last triggers a *UI event* such as *mouse pressed* or *key released*. A UI event embeds data, such as the position of the mouse pressure.

UI controller.[1] A *controller* registers with different interactive objects to gather the UI events these objects trigger.

Event callback. An event callback is a method associated to an interactive object. The interactive object calls such a method on UI event triggering. Developers define event callbacks in controllers. The goal of such callbacks is usually to modify the business data (but can be used to modify the state of the UI as well).

In addition to the concepts involved in the event process model, we define the concepts of user interaction and UI command as follows.

User Interaction. Users perform user interactions on a UI to control the underlying system. User interactions technically rely on one or a sequence of UI events. For example, a Drag-And-Drop (DnD) is the sequence of one pressure event, one or several move events, and one release event. A user interaction is independent of its possible usages. For example, one can use a DnD for moving, scaling, or deleting objects.

UI command. A user performs a user interaction on a UI to apply a specific UI command on the underlying system. Examples of UI commands applied using a DnD are moving, scaling, or deleting objects. UI command can take two shapes: using callbacks [12] such as in Listing 1 discussed in the next section; or using classes as discussed in Section 5.

2.2 Motivating Example

Listing 1 contains JavaScript code, adapted from [7], that illustrates the UI event processing model depicted by Figure 1. In this code example, a user can move a graphical rectangular node using a drag-lock interaction. During this drag-lock, the user interface uses a 'hand' cursor as user feedback. The JavaScript code of Listing 1 contains: the coding of a drag-lock user interaction; the use of this drag-lock interaction to move a graphical node.

The drag-lock user interaction is a special kind of drag-and-drop. A drag-lock starts by double-clicking on the node to drag. The user can then move the locked node until she double-clicks again at the dropping location. The drag-lock interaction is an interesting motivating example as it is a standard user interaction but not provided off-the-shelf by the current UI toolkits.

1. For simplicity, we use the term *controller* to refer to any kind of components that processes UI events, such as *Presenter* (MVP) [17], *ViewModel* (MVVM [18]), or *Component* (Angular [15]).

```
1  let isDragLocked = false;
2  const moveCallback = evt => {
3    draggable.attr({ x: evt.x, y: evt.y });
4  };
5  draggable.addEventListener('dblclick', evt => {
6    if (evt.button === 0) {
7      if (isDragLocked) {
8        draggable.style.cursor = '';
9        draggable
10       .removeEventListener('mousemove', moveCallback);
11     } else {
12       draggable.style.cursor = 'hand';
13       draggable
14       .addEventListener('mousemove', moveCallback);
15     }
16     isDragLocked = !isDragLocked;
17   }
18 });
```

Listing 1: A JavaScript code snippet to move a node using a drag-lock, adapted from [7]

The drag-lock of Listing 1 assembles the UI events *dblclick* (double-click) and *mousemove*. Line 5, the node to drag (*draggable*) registers to double click events. The second argument of this function is a callback method executed on each double-click on this node (Lines 5 to 18). For the first double-click, the UI uses the 'hand' cursor and the node registers to mouse move events (Lines 11 to 14). For the second double-click, the UI uses the default cursor and the node unregisters to mouse move events (Lines 7 to 10). The (un-)registration to mouse move events takes as second argument the callback method located Line 2 that moves the node using event data (Line 3). The move of the node operates only if the user uses the mouse button 0 (Line 6).

This code mixes both the definition of the user interaction and its use for moving a node. Moreover, coding a user interaction may require coding specific instructions for manually registering and unregistering to UI events (Lines 10 and 14).

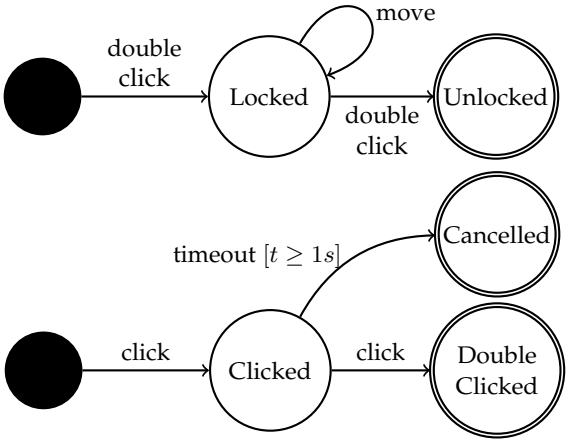

Fig. 2: Finite-State Machines (FSMs) of the drag-lock (top) and double-click (bottom) user interactions used in Listing 1. The double-click transition used in the drag-lock FSM refers to the double-click interaction.

Figure 2 (on the top) depicts an FSM that illustrates the assembly of the *mousemove* and *dblclick* events to build the drag-lock. A transition refers to a UI event or another user interaction. The execution of one user interaction ends when its FSM reaches a terminal state. One may notice that some UI events are not atomic: if the double-click is a user interaction based on several events (*pressure* and *release* that compose each click), it is sometimes considered as a UI event since it is one-shot. Also using an FSM, Figure 2 (on the bottom) depicts the assembly of UI events to build a standard double-click.

2.3 Limitations of the UI event processing model

We illustrate the current limitations of UI event processing models using the example introduced in the previous section and depicted by Listing 1. This example, that involves a drag-lock user interaction, suffers of the following flaws:

Lack of separation of concerns. Listing 1 illustrates how relying on UI events breaks the concept of separation of concerns [19] by intertwining in the same code:

- The definition of the user interaction (the drag-lock) that consists of the assembly of UI events. Current UI toolkits and approaches consider UI events as a first-class concept for coding user interfaces. UI events, however, are *low-level implementation details* that developers need to manually assemble to build user interactions, such as the drag-lock.
- The transformation of user interactions into UI commands. In the same code that assembles UI events to build a user interaction, developers have to define how to produce output UI commands. Line 3 in Listing 1 is the command instruction that moves the dragged node.
- Conditions that constraint the execution of the user interaction. For example, Line 6 checks whether the drag-lock has been done using the button 0 of the mouse.

Lack of software reuse. Listing 1 also illustrates how the UI event processing model prevents code reuse [20], [21]:

- **No user interaction reuse.** Libraries and frameworks enable software reuse by providing developers with predefined and reusable artifacts. Ignoring the concept of user interactions prevents the development of reusable interactions based on the designs established by the HCI community.
- **No user interaction substitution**. User interactions can be classified in different categories. For example, the drag-lock is a kind of DnD interactions. Following the object-oriented substitutability concept, a developer should easily be able to replace a DnD with a drag-lock as their underlying data are the same: start and end positions. Moreover, a same user interaction may have behavioral variants. Figure 3 depicts alternative behaviors of the drag-lock and double-click interactions. The double-click is now canceled on a move between the two clicks. The timeout has changed to 0.5 s. The drag-lock now requires at least one move between the two double-clicks, otherwise it is canceled. A pressure on the key 'ESC' cancels the user interaction. In such cases of user interaction variants, a developer should easily be able to replace the standard DnD by a variant, still based on substitutability.
  Developers can hardly achieve user interaction substitution with the current UI event processing model as the assembly of UI events has to be modified and this model lacks object-oriented constructs.

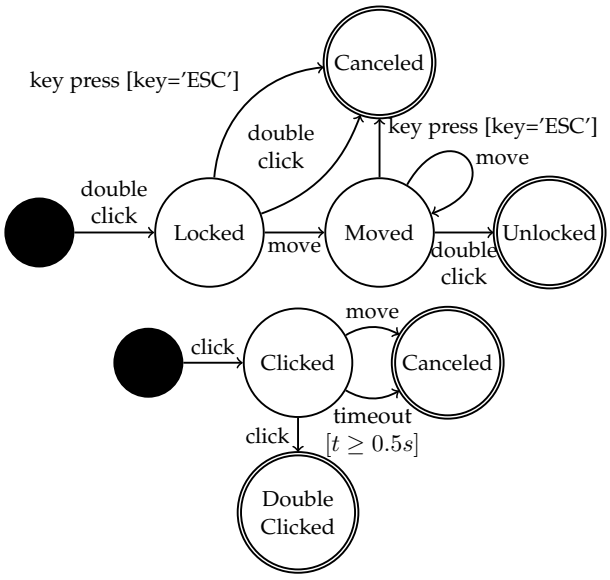

Fig. 3: Alternative versions of the drag-lock (top) and the double-click (bottom) user interactions

Lack of advanced features.

- **No undo/redo.** The code of Listing 1 modifies the business data directly in the event callbacks (Line 3). So, the changes cannot be stored to be then undone and redone. This would require glue code manually crafted by developers in the code of Listing 1 to support such a feature. Note that several UI toolkits overcome this lack with dedicated features (see Section 5).
- **No logging.** Modern systems use logs for analyzing UI usages [22] and for understanding issues. The UI event processing model does not support logging natively.
- **No throttling.** Event throttling is an optimization that permits to reduce the number of similar and successive events in order to alleviate the processing load (and possibly to gain performance). The UI event processing model does not support such a feature.

Complexity and design issues. Intertwining in the same code the assembly of UI events to build user interactions and the transformation of UI events into commands makes the code more complex: it can lead to "spaghetti" code [6], [7] and can be affected by design smells [12]. Moreover, the UI process model strongly relies on the *Observer* pattern that suffers from several major flaws [8], [9], [10], [11].

This also makes the code *more difficult to test*. For example with the code of Listing 1, developers have to test the assembly of UI events.

3 The user interaction processing model

This section describes a user interaction processing model we named *Interacto*, that overcomes the limitations detailed in the previous section. User interactions form a core concept of this model instead of events. So, we call this model a user interaction processing model instead of an event processing model. **The gist of *Interacto* is to turn user interaction executions into (undoable) commands.**

Definition 1. Interacto binding. An Interacto binding is an object that turns the executions of one user interaction into (undoable) command instances.

Definition 2. Interacto binder. An Interacto binder is an object that configures one specific Interacto binding. In the *Interacto* implementations, an Interacto binder takes the form of a fluent API.

Section 3.1 gives an overview of the *Interacto* approach. Sections 3.2 and 3.3 describes how user interactions and UI commands work in *Interacto*. Sections 3.4 and 3.5 then focus on the Interacto binding behavior and the Interacto binder syntax. Section 3.7 details the benefits of *Interacto* in terms of UI testing by proposing new UI testing oracles. Finally, Section 3.6 describes properties that characterize Interacto binders.

All the examples of this section are based on our Java implementation of *Interacto*, namely *Interacto-JavaFX*. The use of our TypeScript implementation, *Interacto-Angular*, would have led to very similar code examples.

3.1 Approach overview

Consider the example of Section 2.2: users have to use a drag-lock interaction to translate a graphical object. Listing 2 illustrates how *Interacto* works in pseudo-code. The developer: selects one user interaction (Line 1); specifies the widgets on which the Interacto binding will operate (Line 2); selects the command to produce (Line 3); details what to do during the execution of the user interaction, in particular when the interaction starts, updates, and ends or is canceled (Lines 4 to 8); defines the conditions that constraint the production and the execution of the ongoing command (Line 9).

```
1 Use the drag-lock user interaction,
2 On the interactive object 'node',
3 To produce and execute 'Translate' command instances,
4 When the interaction starts, sets a shadow to 'node',
5 When the interaction stops or is cancelled, removes
6      this shadow,
7 When the interaction updates, updates the translation
8      vector that the ongoing command will use,
9 That, only if the user uses the primary mouse button.
```

Listing 2: Pseudo-code of an Interacto binder that configures an Interacto binding that translates a node using a drag-lock

This pseudo-code illustrates the four key concepts, that Figure 4 depicts, on which *Interacto* relies. These concepts, detailed in the next sub-sections, are:

- User interactions are reusable, composable (one can build a user interaction using other user interactions), and stateful objects that graphical libraries should provide to developers instead of low-level UI events.
- UI commands are reusable and undoable objects.
- An Interacto binding transforms executions of one user interaction into output UI (undoable) commands.
- An Interacto binder has properties and a concrete syntax for configuring Interacto bindings.

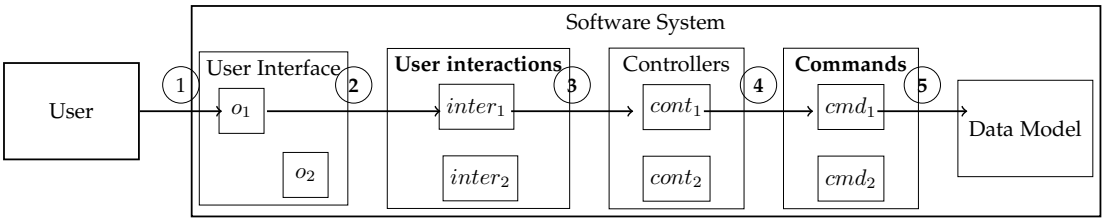

Fig. 4: Behavior of the proposed user interaction processing model (in bold what differs from Figure 1):

(1): A user interacts with an interactive object o_1 of the user interface.

(2): The interactive object then triggers a UI event **processed by a running user interaction** $inter_1$.

(3)(4): **Controllers contain Interacto bindings that turns user interaction executions into UI commands**.

(5): **The running Interacto binding executes the ongoing UI command** to modify the state of the system.

3.2 User interaction

A user interaction is composed of two separated elements: its *behavior* and its *data*.

Interaction behavior. The proposed model makes no assumption on how the behavior of a user interaction is defined. This can be, for example, using FSMs [23], [24], Petri nets [25] or reactive programming [26]. Our implementation and the description of the approach make use of FSMs. We already detailed how we model user interactions using FSMs in Section 2.2 by discussing the examples of Figures 2 and 3.

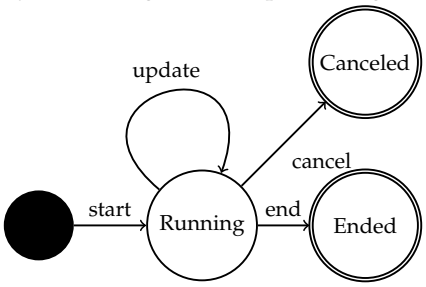

Fig. 5: User interaction life cycle

Figure 5 depicts the generic life cycle of any user interaction. An interaction starts when its FSM leaves its initial state, and is then considered as running. A transition of an interaction FSM corresponds to a UI event or to another interaction FSM (composite FSM). A transition is executed when its matching UI event is triggered (or when its sub-interaction has ended). Each time a transition of the interaction FSM is executed and the targeted state is not a terminal state, the interaction is updated: user interactions update their interaction data (based on the data of the UI event) on transitions executions. Moreover, user interactions automatically perform optimizations on entry and exit actions of states (see Section 4.1). An interaction can end in two ways. Either the interaction is canceled (state *Canceled*), *i.e.,* the user wants to abort the interaction not to produce a command. Either the interaction ends normally (state *Ended*), *i.e.,* the user completed the user interaction. We specify these two different terminal states in interaction FSM examples (Figures 2 and 3) by naming a canceling state *Canceled*. For example in Figure 2 on the right, the FSM reaches the state *Canceled* when a timeout of 1 second expires. The data of a user interaction are updated at each step (*i.e.,* on each transition execution) of their life cycle.

We define an **interaction execution** $exec_i$ as a path in the interaction life cycle, *i.e.,* a path: $start \rightarrow \ldots \rightarrow (cancel|end)$. Note that an interaction execution also corresponds to a path of the FSM of the interaction. For example with the FSM of the drag-lock interaction of Figure 3, the FSM path: $double\ click \rightarrow move \rightarrow move \rightarrow double\ click$, corresponds to the following path in the interaction life cycle: $start \rightarrow update \rightarrow update \rightarrow end$.

Interaction data. A user interaction is stateful and exposes data that an Interacto binding can use. The class diagram of Figure 6 depicts the data model shared by both the drag-lock and the DnD interactions. The drag-lock and DnD interactions are of the same type: they consist in user interactions that operate from a source position to a target position. The interface *FromToData* represents the data of such user interactions, composed of: the source position (*getSrcPosition*); the source picked object (*getSrcObject*); the target position (*getTgtPosition*); the target picked object (*getTgtObject*); the possible button if the interaction involves a mouse (*getButton*). Other user interactions complete this family, such as the *drag-and-pick*, the *drag-and-pop* [27] and the *dwell-and-spring* [28].

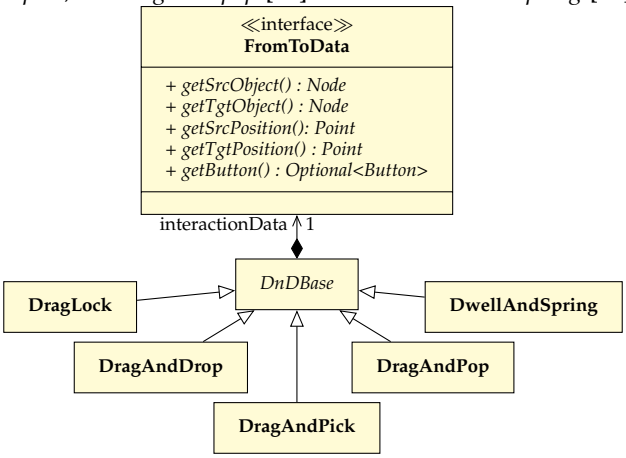

Fig. 6: The data model of user interactions that operate from a source position to a target position (*e.g.,* DnD and drag-lock)

Interaction type and substitution. During the definition of an Interacto binding, developers access the data of the selected user interaction, not its behavior. For example with Figure 6, all the user interactions of the same family as the DnD expose the same type of interaction data *FromToData*. This choice follows the same reasoning than the *bridge* design

pattern [29] that promotes the decoupling of interfaces and implementations: user interaction exposes stable interaction data interfaces. Developers base their configuration of an Interacto binding on such user interaction data so that this permits the substitution of user interactions of a same family.

Formally, we call an interaction data type $\mathcal{D}$ the interface that a user interaction i of type $\mathcal{I}$ exposes. This allows user interactions substitution: developers can replace one user interaction i of data type $\mathcal{D}$ (*e.g.*, a DnD that exposes interaction data of type *FromToData*), with another user interaction i' of the same data type $\mathcal{D}$ (*e.g.*, a drag-lock, which interaction data type is also *FromToData*) without any other change in the Interacto binder.

3.3 Undoable UI command

UI command. When a user interacts with a user interface using a given user interaction, her goal is to act on the underlying system, such as to modify its state. Developers can encapsulate such actions on the system in UI command classes. In the literature, this has two benefits:

- Enable UI commands reuse across the different controllers of the UI. One UI command may be produced from different user interactions and interactive objects of the UI (*e.g.*, buttons, menus, shortcuts).
- Support undo/redo features. developers may code UI command as undoable, so that users can undo and redo changes they apply on the system.

```
1  public class Translate extends CommandBase implements Undoable {
2    double mementoX;
3    double mementoY;
4    double newX;
5    double newY;
6    final Shape data;
7
8    public Translate(Shape shape) {
9      data = shape;
10   }
11   @Override public boolean isExecutable() {
12     return !(newX == data.getX() && newY == data.getY());
13   }
14   @Override protected void execution() {
15     data.setPosition(newX, newY);
16   }
17   @Override protected void createMemento() {
18     mementoX = data.getX();
19     mementoY = data.getY();
20   }
21   @Override public void undo() {
22     data.setPosition(mementoX, mementoY);
23   }
24   @Override public void redo() {
25     execution();
26 }}
```

Listing 3: An example of an undoable UI command

Listing 3[2] gives the Java code of a typical *Interacto* UI command coded by a developer (where *CommandBase* and *Undoable* are part of *Interacto*). The pseudo-code of Listing 2 uses this UI command. This UI command *Translate* moves the given shape (Line 6) to a new position (Lines 4 and 5). The method *execution* defines the execution of the command (Line 14). The *isExecutable* method checks whether the UI command can be executed (Line 11). Here, this method checks that the translation vector is not null. This command is undoable (its implements the interface *Undoable*) so that the developer has to implement the methods *undo* and *redo* (Lines 21 and 24). The *undo* method puts the shape back to its former location using a memento [29] produced by the *createMemento* method (Line 17). Interacto bindings automatically store the executed undoable commands in an undoable command history. *Interacto* supplies developers with predefined UI commands for helping developers in adding undo/redo features in their UIs.

Asynchronous command. A developer may want to run a command asynchronously. For example, a Web application may send queries to servers. In such a case, the *execution* method of an *Interacto* command can return a Promise object that corresponds to the pending query. In TypeScript, such a method can be written as follows:

```
protected execution(): Promise<void> | void {
    return this.http.put(...).toPromise();
}
```

Undo history. An undo history collects the executed undoable commands and implements a specific undo mechanism. In the literature, the standard undo mechanism is linear: a new command is pushed on a stack *toUndo*; on undo, the top command of this stack is popped to be undone and pushed on a second stack *toRedo*; the opposite process operates on redo.

The literature proposed several undo mechanisms such as selective ones [30], [31]. *Interacto* does not constraint the use of specific undo mechanisms and by default provides a linear undo history. *Interacto* also permits to have several undo histories in the same application to work with given Interacto binding as detailed in the next section.

3.4 Interacto binding behavior

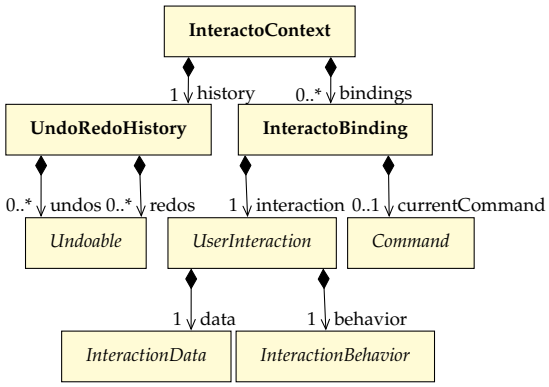

Fig. 7: The metamodel of an Interacto binding.

An Interacto binding focuses on transforming the execution of a user interaction into UI command instances. The main goal of an Interacto binding is to produce, update, execute, or cancel one UI command instance along one execution of a user interaction. Figure 7 gives the metamodel of an Interacto binding. An Interacto binding operates within an *Interacto* context that contains an undo/redo history. Formally, an Interacto binding b is set between a user interaction i of type

2. For readability, all the code that uses an implementation of *Interacto* is put in listings with a grayed background.

$\mathcal{I}$, that exposes an interaction data type $\mathcal{D}$, and a UI command of type $\mathcal{C}$. Each interaction execution $exec_i(b)$ of i, managed by b, may produce a new instance c of the command $\mathcal{C}$, such that: $exec_i(b) \rightarrow c \wedge exec'_i(b) \rightarrow c' \implies c \neq c'$. A user interaction has a behavior, such as an FSM.

```
1  onInteractionStart() {
2  // when(): states whether the binding can create the command
3     if(when()) {
4        currentCommand = toProduce(); // Creation of the command
5        first(); // Command initialisation
6     }
7  }
8  onInteractionUpdate() {
9     if(when()) {
10       if(currentCommand == null) {
11          currentCommand = toProduce();
12          first();
13       }
14       then(); // Command update
15       // The binding executes the command on each update:
16       if(continuous() && currentCommand.isExecutable()) {
17          currentCommand.execute();
18       }
19 }}
20 onInteractionEnd() {
21    if(when()) {
22       if(currentCommand == null) {
23          currentCommand = toProduce();
24          first();
25       }
26       then();
27       if(currentCommand.isExecutable()) {
28          currentCommand.execute(); // Execution of the command
29          // Registration of the command (for undo/redo)
30          registerCurrentCmd();
31       }
32    }
33    end();
34    currentCommand = null;
35 }
36 onInteractionCancel() {
37    cancel();
38    if(continuous() && currentCommand.wasExecuted()) {
39       undoCurrentCommand();
40    }
41    currentCommand = null;
42 }
```

Listing 4: The algorithm in pseudo-code of the Interacto binding behavior. An Interacto binding operates when its user interaction is running.

Listing 4 gives the algorithm of the behavior of an Interacto binding $exec_i(b)$. Method calls in italic (namely: *when*, *first*, *toProduce*, *end*, *cancel*, *then*, *continuous*) refer to methods available during the building of the Interacto binding (*i.e.*, with an Interacto binder), as detailed in the next section. The user interaction life cycle (Figure 5) drives the behavior of an Interacto binding. First, when an interaction starts (Line 1) the Interacto binding creates a new UI command if the condition of the Interacto binding (*when*, Line 3) is fulfilled. The Interacto binding creates a UI command using the function *toProduce* (Line 4). Then, the Interacto binding initializes the UI command (*first*, Line 5).

On each interaction update (Line 8) the predicate *when* conditions the update of the ongoing UI command. One may notice that the Interacto binding may not create a UI command when the user interaction starts because of the *when* predicate. So, if not already created, the Interacto binding creates a UI command and initializes it (Lines 10 to 13). The Interacto binding then updates the UI command (*then*, Line 14). The Interacto binding executes the ongoing UI command either at the end of the user interaction, or on each update of the user interaction (what we call *continuous command execution*). So, if the execution of the ongoing UI command is *continuous* and the command executable, the Interacto binding executes it (Lines 16 to 18).

When an interaction ends normally (Line 20), and similarly to the update of the user interaction, the Interacto binding checks the predicate *when* to possibly create and update the ongoing UI command (Lines 21 to 26). Then Line 27, the Interacto binding checks whether it can execute the UI command (see the method *isExecutable* Line 11 in Listing 3). The Interacto binding puts the executed UI command in a command register (Line 30) to keep in memory UI commands that may be undone by users. The Interacto binding finally *ends* (Line 33) and dereferences the ongoing UI command (Line 34).

When the user cancels the ongoing user interaction (Lines 36 to 42), the Interacto binding calls the method *cancel* (Line 37) and clears the ongoing command (if it exists).

Start and stop an Interacto binding. An Interacto binding can start and stop. On start, an Interacto binding asks its user interaction to start, *i.e.*, to listen for UI events from the selected interactive objects. On stop, an Interacto binding ask its user interaction not to listen for UI events anymore. The user interaction also flushes its interaction data.

3.5 Interacto binder syntax

The syntax of the Interacto binder definition language is summarized in Figure 8. An Interacto binder works as a builder [29] to produce one Interacto binding. A developer writes an Interacto binder using a set of routines (*config* in Figure 8) to configure how the produced Interacto binding will work. The call to the terminal method *bind* creates and starts the Interacto binding. We summarize here the main ones.

Using our JavaFX implementation *Interacto-JavaFX*, the pseudo-code of Listing 2 gives the following Java code that describes an Interacto binder that configures an Interacto binding. This Interacto binding will work using a *DragLock* interaction to produce *Translate* command instances:

```
1  binder()
2    .using(DragLock::new)
3    .toProduce(d -> new Translate(d.getSrc().getData()))
4    .on(node)
5    .first((d, c)->
6        d.getSrcObject().setEffect(new Shadow()))
7    .then((d, c) -> c.setCoord(
8       c.getShape().getX() + d.getTgtPoint().getX()
9          - d.getSrcPoint().getX(),
10      c.getShape().getY() + d.getTgtPoint().getY()
11         - d.getSrcPoint().getY()))
12   .when(d -> d.getButton() == MouseButton.PRIMARY)
13   .continuous()
14   .endOrCancel(d -> d.getSrcObject().setEffect(null))
15   .bind();
```

Listing 5: An Interacto binder to move a node using a drag-lock

Syntax:

$binding$	::=**`binder()`**$\overline{config}$**`.bind()`**	(Binding Configuration)
$config$	::=**`.using(`**$() \rightarrow i$**`)`**	(Interaction Creation)
	| **`.toProduce(`**$d \rightarrow c$**`)`**	(Command Creation)
	| **`.on(`**$\overline{w}$**`)`**	(Widgets Selection)
	| **`.first(`**$(d, c) \rightarrow void$**`)`**	(Interaction Started)
	| **`.then(`**$(d, c) \rightarrow void$**`)`**	(Interaction Updated)
	| **`.end(`**$(d, c) \rightarrow void$**`)`**	(Interaction Ended)
	| **`.cancel(`**$d \rightarrow void$**`)`**	(Interaction Canceled)
	| **`.endOrCancel(`**$d \rightarrow void$**`)`**	(Interaction Over)
	| **`.when(`**$d \rightarrow bool$**`)`**	(Command Condition)
	| **`.with(`**$\overline{k}$**`)`**	(Keyboard Keys)
	| **`.throttle(`**int**`)`**	(Throttling)
	| **`.strictStart()`**	(Strict Interaction Start)
	| **`.continuous()`**	(Continuous Execution)
	| **`.consume()`**	(Consume UI Events)
	| **`.log(`**l**`)`**	(Logging)
l	::=**`interaction`**|**`binding`**|**`cmd`**	(Logging Level)
d		(Interaction Data)
w		(Interactive Object)
k		(Keyboard Code)

Typing:

$\Gamma \vdash i : \mathcal{I}$	(Interaction Type)
$\Gamma \vdash c : \mathcal{C}$	(Command Type)
$\Gamma \vdash d : \mathcal{D}$	(Interaction Data Type)
$\Gamma \vdash i :< d$	(Interaction Substitution)

Fig. 8: The Interacto binder syntax

***using*: User Interaction Creation.** The routine *using* selects the user interaction the Interacto binding will use. In Listing 5, *using* takes as argument a function that returns a new instance of the predefined drag-lock interaction (Line 2).

***toProduce*: Command Creation.** The routine *toProduce* (Line 3) focuses on the production of a UI command. This routine takes as argument an anonymous function that returns a UI command as depicted in Listing 5 (Line 3).

***on*: Nodes Selection.** The user interaction operates on selected interactive objects to produce commands. The routine *on* allows developers to specify these interactive objects (*e.g.,* Line 4).

The *on* routine can also take as arguments an observable list of interactive objects l. This allows to dynamically register and unregister interactive objects to/from the Interacto binding: when an object w is added to a list l, the binding will work for w until w is removed from l. In the following code excerpt, the interaction will operate on each object the canvas will contain, dynamically.

```
// on can also take an observable list of nodes
.on(canvas.getChildren())
```

To support this feature using the standard event processing model, developers would have needed to manually develop the glue code that manages the (un-)registration.

***when*: Command Condition.** An Interacto binding constraints the creation, the update, and the execution of a UI command using the *when* routine (*e.g.,* Line 12). The *when* routine is a predicate that takes as argument the current interaction data (d) to state whether a UI command can be created, updated, or executed, as detailed by Listing 4.

***first*, *then*, *end*, *cancel*: Interaction Starts, Updates, Ends, Cancels**. The Interacto binding calls the routines: *first* right after the instantiation of a UI command; *then* on each update of the running interaction; *end* on each normal end of an interaction execution (if the *when* predicate is respected); *cancel* on each cancellation of the current interaction execution. They take as arguments the current interaction data (d in the following code) and/or the current UI command (c). Listing 5 illustrates the use of these routines (Lines 5, 7 and 14).

3.6 Interacto binder Properties

An Interacto binder has the following properties:

Type-safe. The return of each builder routine is typed and constrained by the previously called routines. For example, the next code selects the user interaction *DragLock*. So, this user interaction selection constraints the routines used next. In this example, the attribute d of the routine *when* has the type *FromToData*, which is the interaction data type of the drag-lock interaction.

```
binder()
  .using(DragLock::new)
  .when((FromToData d) -> ...)...
```

The same reasoning applies to the selected UI command to produce. Also, to call the method *bind*, the developer should at least have call the routines *using*, *toProduce*, and *on* beforehand.

This permits to write factory methods that shortcuts the writing of bindings. For example, using our implementations developers rarely call `binder().using(DragLock::new)` but instead `dragLockBinder()` using the factory method that partially builds a binding by selecting the drag lock interaction:

```
public static PartialBinder<...> dragLockBinder(){
  return new Binder<Interaction<FromToData>, FromToData>()
    .usingInteraction(DragLock::new);
}
```

Immutable. On each routine call, the builder clones itself to return a new builder. The goal is to ease builder reuse and factorize Interacto binders code. For example, the following code starts by defining a partial binder (`baseBinder`). This partial binder is then used to create two binders.

```
var baseBinder = buttonBinder().on(button).end(...);

baseBinder
  .toProduce(...)
  .when(...)
  .bind();

baseBinder
   .toProduce(...)
   .when(...)
   .bind();
```

3.7 Testing Interacto bindings

The proposed model does not impose testing tools to testers and does not change the way developers write UI tests. Instead, the proposed model comes with specific test oracles and testing facilities that complement classical UI testing techniques [32] and oracles [33]. We classified our proposed test oracles into two groups: Interacto binding test oracles; UI command test oracles.

Interacto binding test oracles. The goal of an Interacto binding is to transform one user interaction execution into a UI command. So, we define a testing oracle: *command produced*. The *command produced* oracle checks whether a user interaction execution produces a given command type. Listing 6 depicts this oracle using a UI test case that performs a DnD to then check whether the expected command has been created. Because Interacto bindings are first-class objects, we can observe them in a testing context to collect and analyze the UI commands they created. We built a JUnit5 extension that does this job and provides a *BindingsContext* test parameter (automatically injected) that provides testers with *command produced* assertions. For example Line 6 queries this test parameter to assess that a single command of type *AddShape* was produced during the test execution.

```
1 @Test
2 void dndToAddShape(FxRobot robot, BindingsContext ctx) {
3   // ...
4   robot.moveTo(...).press(...).moveTo(...).release(...);
5   // ...
6   ctx.oneCmdProduced(AddShape.class);
7 }
```

Listing 6: Example of UI test case that uses the *command produced* oracle

Listing 7 is another testing example that checks no command of type *AddShape* are created when executing a specific scenario:

```
1 @Test
2 void dndToAddShapeKO(FxRobot robot, BindingsContext ctx){
3   //...
4   ctx
5     .listAssert()
6     .noneSatisfy(cmd -> cmd.ofType(AddShape.class));
7 }
```

Listing 7: A second example of UI test case that uses the *command produced* oracle

UI command test oracles. Reifying UI commands as first-class objects permits the design of dedicated testing facilities. First, we define UI command oracles implemented in a dedicated testing framework. Second, we provide testers with a UI command test skeletons generator that help them in starting their UI command testing tasks.

UI command testing framework. We defined the following *UI command test oracles*: *can do*, *cannot do*, *do*, and *undo*. The *can do* oracle checks that the command can be executed. The *cannot do* oracle checks scenarios where the command cannot be executed. The *do* oracle checks the correct (re-)execution of the command. The *undo* oracle checks that the undoable command was correctly undone after its execution. Note that the *redo* oracle is the same than the *do* oracle.

A dedicated testing framework has to alleviate the job of developers by automating the execution of UI commands to help these developers in focusing on writing UI commands fixtures and oracles. The code of Listing 8 is an example of a UI command test class. Testers have to write *do* and *undo* test oracles (Lines 19 to 31). Testers must also write UI command fixtures for configuring a command that: can be executed (used for the *can do*, *do*, *undo* oracles), Lines 4 to 12; cannot be executed (for the *cannot do* oracle), Lines 14 to 16. As several *can do* and *cannot do* scenarios may exist, the methods return a set of (`Stream.of()`) oracles and fixtures.

```
1  class DelShapesTest extends UndoableCmdTest<DelShapes> {
2    List<Shape> shapes;
3    Drawing drawing;
4    @Override protected Stream<Runnable> canDoFixtures() {
5      return Stream.of(() -> {
6        shapes = List.of(Factory.newRec(),
7            Factory.newRec(), Factory.newRec());
8        drawing = Factory.newDrawing(shapes);
9        cmd = new DelShapes(drawing,
10           List.of(shapes.get(0), shapes.get(2)));
11     });
12   }
13   @Override
14   protected Stream<Runnable> cannotDoFixtures(){
15     return Stream.of(() -> {
16       cmd=new DelShapes(Factory.newDrawing(), List.of());
17     });
18   }
19   @Override protected Stream<Runnable> doCheckers() {
20     return Stream.of(() -> {
21       assertThat(drawing.size()).isEqualTo(1);
22       assertThat(drawing.getShapeAt(0))
23           .isSameAs(shapes.get(1));
24     });
25   }
26   @Override protected Stream<Runnable> undoCheckers() {
27     return Stream.of(() -> {
28       assertThat(drawing.size()).isEqualTo(3);
29       assertThat(drawing.getShapes()).isEqualTo(shapes);
30     });
31 }}
```

Listing 8: Example of a UI command test class

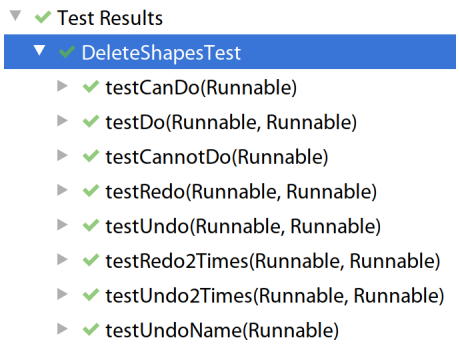

Fig. 9: The tests executed by the UI command testing framework for the test class of Listing 8

The execution of this test class runs the targeted UI command under several scenarios (test instances) to check the UI command oracles. Figure 9 shows the different tested scenarios: can execute, cannot execute, execute, undo, redo, several do/undo/redo sequences.

UI command test skeletons generator. Since UI commands are first-class objects, a static analysis can extract from them information to generate UI command test class skeletons. For example with Listing 8, testers only wrote the code inside the `Stream.of()` instructions, the rest being automatically generated. The test class attributes are copied from the UI command class attributes. The static analysis checks whether the UI command is undoable for generating methods *undoCheckers*.

4 Evaluation

In this section we evaluate five aspects of *Interacto*: Is *Interacto* implementable on different UI platforms? Does the approach scale in terms of performance and expressiveness? What are the pros and cons based on our usage? What is the scope of *Interacto*? Can students successfully use *Interacto*?

All the material of this section is available on our companion web page.[3]

4.1 Implementations

We define the first research question we address as follows:

RQ1. Can we implement the proposed model in different programming languages and graphical toolkits that support different paradigms?

We implemented the *Interacto* approach on the top of two programming languages: Java and TypeScript [14]. For each of these two languages we provide a UI platform-independent library, namely *Interacto-Java* and *Interacto-TypeScript* (this last uses the native Web graphical API[4]).

Regarding Java, we implemented an extension library, *Interacto-JavaFX*, for supporting the JavaFX UI toolkit [13], a major UI toolkit for Java. *Interacto-JavaFX-Test* complements *Interacto-JavaFX* with testing facilities for JavaFX. The support of another Java UI toolkit, such as Android, would require an new library that extends *Interacto-Java* by defining how to register to low-level UI events of the UI platform.

Regarding TypeScript, we implemented an extension library, *Interacto-Angular*, for improving the use of *Interacto-TypeScript* within the *Angular* UI toolkit [15].

The implementations contain around 8000 lines of Java 11 code and 5000 lines of TypeScript 3.8 code. Both implementations rely on the same concepts detailed in this paper. User interactions are developed using composite and concurrent FSMs. As illustrated in Section 2.3 with Figure 3, the drag-lock FSM is composite: its transitions labelled *double click* refers to the double-click interaction. Regarding concurrent FSMs, an example is the multi-touch interaction where each touch that starts corresponds to touch interaction.[5] Both implementations provide a set of around 30 predefined user interactions that developers can use. The implementations are open-source and free available.[6]

We implemented several optimizations in the implementations of user interactions for both Java and TypeScript:

- **Efficient event registration**. For example with Figure 3, the FSM on the top uses *mouse click* (for double click), *mouse move*, and *key pressure* events. When a user interaction is activated, it does not listen for all the possible UI events that this user interaction uses. Instead, when entering a state the accepted UI events at this state are identified, and the user interaction registers for these UI events only (and un-register for the other ones). For example, when the top FSM of Figure 3 starts, it only listens for mouse click events (as the fist event of a double click interaction). When entering the state *Locked*, this FSM now listens for mouse move and mouse click events only. When programming user interactions by hand, software engineers have to think about this optimization to then manually craft it.
- **Late starting**. Let us take the example of the DnD. One developer may consider that a DnD starts at the first move, not at the initial mouse pressure. In such a case, one can specify the state of a user interaction that will correspond to the starting of the user interaction (that calls the *map* and *first* routines).
- **Interacto binder shortcuts**. Our implementations provide shortcuts for initializing Interacto binders. For example, Listing 9 contains two Interacto binders coded in TypeScript with the Angular framework (to be discussed later in this section). The first Interacto binder starts with `multiTouchBinder(2)`. This code is equivalent to `binder().using(() => new MultiTouch(2))`: for most of the user interactions we defined, we provide a coding shortcut for initializing Interacto binders that use a given user interaction. In this case, the user interaction multi-touch is configured to use two touch points.

Differences between the Java and TypeScript implementations. Regarding user interactions, each graphical toolkit has its own UI events. If most of such UI events are common across the graphical toolkits, some others are platform-specific or have different parameters. For examples: the touch event of the native Web API has a force parameter (the amount of pressure the user is applying) that is not supported in JavaFX; JavaFX provides window-based UI events that the native Web API does not provide; the native Web graphical toolkit has a multi-touch support so that our TypeScript implementation provides various standard touch user interactions such as *pan*, *swipe*, *tap*. This has a limited impact of the library of predefined user interactions provided by each implementation.

Still related to user interactions, the Angular/Web API has a feature that permits to disable the default user interaction initially imposed by the Web browser. For example, a right-click shows a menu on most of Web browser, which can enter conflict with the expected behavior of a developed Web application. This feature is called *preventDefault*. We support this feature for user interactions of our TypeScript implementation. For example with Listing 9, we used this feature for the first Interacto binder *multiTouchBinder*.

3. https://github.com/interacto/research
4. https://developer.mozilla.org/en-US/docs/Web/API
5. See the companion web page for more details on the multi-touch interaction.
6. https://interacto.github.io

The last difference concerns the name of specific Interacto binder routines. The name of several concepts differ from JavaFX to Angular/Web API. For example, the concept that consists of stopping the propagation of a UI event refers to the term *consume* on JavaFX and to the term *stopImmediatePropagation* on Angular/Web API. Our implementations use the name used by their UI toolkit. Similarly, because of TypeScript method conflicts, the TypeScript implementation uses the name *onDynamic* (see Listing 9) to refer to the *on* routine that takes as argument a list to observe.

Example of how *Interacto-JavaFX* works within a JavaFX controller and an Angular component.

Interacto can work with any architectural pattern that processes UI events and provides access to the user interface elements in the code. We illustrate this point with the two following Angular and JavaFX examples.

```
1  export class AppComponent implements AfterViewInit {
2    @ViewChild('canvas') private canvas: ElementRef;
3    // ...
4    ngAfterViewInit(): void {
5      multiTouchBinder(2)
6        .toProduce(i => new DrawRect(
7          this.canvas.nativeElement as SVGSVGElement))
8        .on(this.canvas.nativeElement)
9        .then((c, i) => {
10         c.setCoords(Math.min(...i.getTouchData()
11           .map(touch => touch.getTgtClientX()))-b.x,...);
12       })
13       .continuousExecution()
14       .preventDefault()
15       .bind();
16
17     tapBinder(3)
18       .toProduce(i => new ChangeColor(
19         i.getTapData()[0].getSrcObject()))
20       .onDynamic(this.canvas.nativeElement)
21       .when(i => i.getTapData()[0].getSrcObject()
22         !== this.canvas.nativeElement && ...)
23       .bind();
24 }}
```

Listing 9: An example of how developers can code Interacto binders within an Angular component

Listing 9 is an excerpt of an Angular component (a kind of controller in the Angular framework). It shows how one can use *Interacto-Angular* within an Angular component. Currently, *Interacto-Angular* requires the widgets to be accessible from the component code. This requires the definition of a class attribute that corresponds to a widget defined in the HTML part of the component (the view of an Angular component is an HTML document). To overcome this requirement, we can improve *Interacto-Angular* with dedicated Angular facilities. In this example, *canvas* refers to such a widget (Line 2). Interacto binders are defined in the method *ngAfterViewInit* (Line 4), which is a special method of each Angular component: Angular automatically calls this method once the HTML view of the component loaded, so that Interacto binders can access its widgets. This method contains two Interacto binders: the first one (Line 5) builds an Interacto binding that uses a multi-touch interaction to produce *DrawRect* commands; the second one (Line 17) builds an Interacto binding that uses a tap interaction to produce *ChangeColor* commands. These commands are specific to this application and defined in a dedicated folder. The code of these Interacto binders and their the commands they use is similar to the Java code show in this paper to detail and illustrate the proposal.

Listing 10 gives a concrete example about how Interacto binders are defined in a JavaFX controller. The *CanvasController* class is in charge of managing a JavaFX view (not depicted here). The JavaFX dependency injection permits the controller to access the interactive objects of its view (*cf.* the annotation *@FXML*, Line 2). JavaFX calls the method *initialize* at the first use of the controller. We use this method (Line 4) to code two Interacto binders that configure JavaFX Interacto bindings (Lines 5 and 10).

```
1  public class CanvasController ... {
2    @FXML Canvas canvas;
3    // ...
4    public void initialize(...) {
5      binder()
6        .using(DragLock::new)
7        ...
8        .on(canvas.getChildren())
9        .bind();
10     binder()
11       .using(DnD::new)
12       .toProduce(d -> new Add(...))
13       .on(canvas).
14       ...
15       .bind();
16 }}
```

Listing 10: Example of how developers can code an Interacto binder within a JavaFX controller

To conclude on RQ1, these two implementations detailed in this section show that the proposed model is not tied to one specific programming language or graphical framework.

4.2 A real world use case: LaTeXDraw

RQ2. Does the implementation of the proposed model scale? We first discuss the ability of the proposal to scale for the development of a representative software system. Then, we discuss performance of the implementation compared to the use of the standard UI event processing model.

LaTeXDraw is a large open-source and highly interactive vector drawing editor for LaTeX.[7] It is downloaded 1.5 k times per month. LaTeXDraw is distributed on more than 10 Linux distributions, also available on Windows and MacOS. On its Github page, the project has 340 stars and 58 forks. LaTeXDraw is composed of around 35 000 lines of Java code. We developed LaTeXDraw in Java since 2005 using callback methods for processing UI events. As any software system, LaTeXDraw evolves: we progressively introduced the use of *Interacto-JavaFX* instead of callback methods since 2013. The current version of LaTeXDraw (4.0) now entirely relies on the fluent Interacto binder API. Note that the development

7. http://latexdraw.sourceforge.net

of LaTeXDraw precedes the development of *Interacto-JavaFX* (2005 *vs* 2013). Moreover, the goal of LaTeXDraw has no relation with *Interacto*. Because of the successive evolutions, the fully callback version and the fully *Interacto* version are not isomorphic and cannot be compared. LaTeXDraw 4.0 has the following characteristics. It is composed of 45 controllers (around 4700 lines of Java code) that contain a total of 224 Interacto bindings (around 1000 lines of Java code). These Interacto bindings produce 37 different UI commands (around 1400 lines of Java code). Regarding the used user interactions: 45 Interacto bindings use several forms of DnD (12), click (11), or keyboard interactions (22). The rest of the Interacto bindings (179) are based on standard widgets and window interactions (buttons, lists, color pickers, *etc.*). We developed all the Interacto bindings using the Interacto binder fluent API described in this paper. On the issue tracking system of the Github page of the project, eight issues, on the 30 ones opened, were related to user interaction processing.

Regarding the testing oracles and features we propose with *Interacto-JavaFX* and *Interacto-JavaFX-Test*, we tested all the UI command of LaTeXDraw using the proposed UI command oracles. We used the UI command test generator to produce test class skeletons we then completed. Before using this testing feature we wrote only few UI command test classes. The generated test classes replaced these test classes as we considered the new ones as covering more UI command concepts (undo, redo, *etc.*) and they achieve a better code coverage (currently 98.3 % of covered lines). We, however, cannot currently use the Interacto binding oracle for testing LaTeXDraw's Interacto bindings as this testing feature requires JUnit5 tests while LaTeXDraw UI test suite is written in JUnit4. We use this testing feature for testing the library *Interacto-JavaFX* and in other cases as discussed in the next section.

We use *Interacto-JavaFX* since 2013 for developing a highly interactive and widely-used open-source software system. This shows the ability of the proposal to scale for the development of such software systems.

Regarding the performance of the implementation, we evaluated the possible overhead of the use of *Interacto-JavaFX* compared to the use of UI callbacks. Because of the successive evolutions of LaTeXDraw, the fully callback version and the fully *Interacto-JavaFX* version are not isomorphic and cannot be compared. We thus ported LaTeXDraw 4.0 to use UI callback methods only. This new callback version still uses the developed UI commands. The controllers of this callback version have the following code metrics: 5600 lines of code (vs 4700 lines for the *Interacto-JavaFX* version); A cyclomatic complexity (mean per method) of 2.55 (vs 1.89); An LCOM value (Lack of Cohesion in Methods) of 1.86 (vs 1.49). We used the existing test suite of LaTeXDraw to validate the call back version: the LaTeXDraw test suite, that covers 90 % of the UI controllers code and 100 % of the Interacto binders code, assesses the isomorphic property between the *Interacto-JavaFX* and callback versions. Then, using these two implementations (*Interacto-JavaFX* and UI callback), we executed 10 times the test suite that covers the Interacto binders. We used a Linux computer with 4 CPUs of 2.6 GHz each, 16 GB RAM, an Intel HD Graphics 5500 graphical card, and Xorg 1.19.6 as graphical server (the used test suite is composed of user interface tests), and Java 8 update 161. The test execution was not parallelized. We removed all the tests not related to UI event processing. We obtained a test suite composed of 447 tests. These GUI tests simulate user interactions with the GUI to trigger (or not) the creation of commands. So these tests directly operate on Interacto binders and UI callbacks. We modified the resulting test suite to automatically measure and log the execution time of each test. We removed the assertions of these tests to transform them into executable UI scenarios only. Figure 10 summarized the measured execution times.

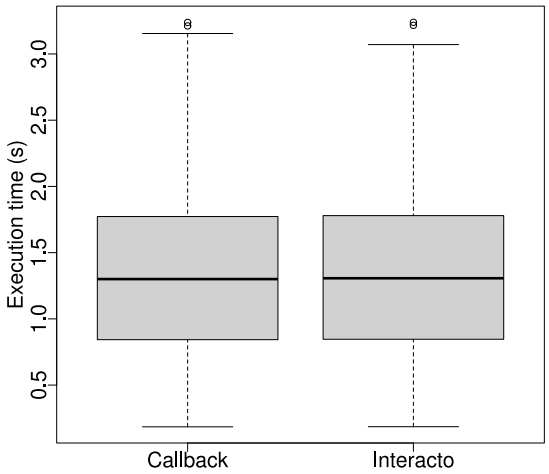

Fig. 10: Execution time comparison

For each test we computed its mean execution value based on ten executions. The mean execution time for a test is 1.331 s for the callback version and 1.334 s for the *Interacto-JavaFX* version (see Figure 10). We use the Wilcoxon signed-rank test (a paired difference test) [34] to compare the two sets of mean execution times (data do not follow a normal distribution and we used an initial confidence level of 95 %, *i.e.*, $\alpha = 0.05$): the observed differences between the callback and *Interacto-JavaFX* versions (p-value of 0.9472) are not significant and may be due to randomness. We can conclude that using the JavaFX implementation the conducted experiment gives no significant evidence regarding an execution time overhead caused by the use of *Interacto-JavaFX* compared to the implementations based on callback methods.

This experiment shows no overhead in terms of performance when using *Interacto-JavaFX* compared to standard JavaFX UI callbacks.

RQ3. What are the pros and cons of the proposed model based on our real world usage? This research question permits to discuss about the scope, the expressiveness of our proposal mainly in the context of LaTeXDraw.

The use of *Interacto-JavaFX* in LaTeXDraw completely removed the use of UI callbacks for processing UI events. We faced no situation in which we could not use *Interacto-JavaFX* to process all the heterogeneous user interactions that LaTeXDraw employs.

The use of *Interacto-JavaFX* does not remove the use of general-purpose callbacks and data binding features in

LaTeXDraw. The first reason is that the *Interacto* approach complements data binding and achieves different purposes: *Interacto* provides features for processing user interactions while data binding creates dynamic links between object values. For example, the following code statement, from LaTeXDraw, establishes a data binding that does not imply the use of any UI command or user interaction:

```
msg.visibleProperty().bind(Bindings
  .createBooleanBinding(!msg.getText().isEmpty() ...));
```

Similarly, the following reactive programming code, extracted from LaTeXDraw, listens for changes in an observable list to call the method *updateSelectionBorders*. This reactive code uses throttling to limit the number of updates and improve performance. This code does not refer to any UI event (and thus not to any user interaction) and is thus out the scope of *Interacto*.

```
JavaFxObservable.<ObservableList<Shape>>changesOf(
        drawing.getSelection().getShapes())
  .throttleLast(20, TimeUnit.MILLISECONDS)
  .observeOn(JavaFxScheduler.platform())
  .subscribe(next -> updateSelectionBorders());
```

These two remarks also concerns the Angular framework that supports data binding and reactive programming features.

The main drawback we found concerns several complex Interacto binders that require temporary variables to share data across the different routines of the binding. For example, the following Interacto binder code defines and uses two objects, namely *xgap* and *ygap*, to share data across the *first* and *then* routines. In this case, developers may prefer to define a new class instead of using an Interacto binder. This is possible with our implementations.

```
private void configureDnD2ScaleBinding() {
  final AtomicInteger xgap = new AtomicInteger();
  final AtomicInteger ygap = new AtomicInteger();
  binder()
    ...
    .first((i, c) -> {
      // This routine sets values to xgap and ygap.
    })
    .then((i, c) -> {
      // This routine uses xgap and ygap.
    })
    .bind();
}
```

The code above also shows that we placed the Interacto binder code into a specific method. Several controllers have multiple and complex Interacto binders. Putting all the Interacto binders in the same method would produce a long method code smell. For readability we split these definitions into different methods with a name that clearly describes the job of the Interacto binder.

Another point is related to coding style. There exists two ways for registering callbacks on widgets in both JavaFX and Angular:
1/ **Programmatically**, one can register as follows in this TypeScript code:

```
button.addEventListener('mousedown', evt => {...});
```

2/ One can also register **in the UI description code**, here in the HTML code of an Angular application:

```
<button (mousedown)="myCallback()"/>
```

where *myCallback* is a TypeScript method defined in the code. The goal of this style is to permit stakeholders with limited programming skills (*e.g.*, designers) to specify which user interactions to use into views.

The *Interacto* implementation for Angular supports both styles. For example, the following Angular HTML example defines an SVG rectangle on which the use of a drag-lock is specified for moving it (Interacto attribute *ioDraglock*):

```
<rect [ioDraglock]="moveRect"></rect>
```

The method *moveRect* is defined in the Angular component as follows. This method takes as arguments a partial binder: the user interaction to use is already configured.

```
moveRect(binder: PartialDragLockBinder): void {
  binder
    .toProduce(() => ...)
  ....bind();
}
```

Finally, in some specific cases a developer may not want to rely on UI events but on data binding as an alternative to UI event processing. For example, the following code establishes a binding between the text value of a text field, changed when a user interacts with the text field.

```
msg.textProperty().bind(button.textProperty());
```

Instead of processing UI keyboard events produced by this text field, a developer can bind the text value to another value. This way of binding data, however, does not support undo/redo features for JavaFX and Angular. Conceptually and technically, *Interacto* can support such a use case. To do so, we have to build a user interaction whose FSM uses the value change event of such text properties. Then, we can use this new interaction within an Interacto binding. However, the core idea of *Interacto* to focus on real user interactions.

4.3 Scope of the proposed model

We discuss here the expressiveness and extensibility of the proposal through the kinds of software systems it can support.

RQ4. What is the scope of the proposed model?
This research question aims at discussing the types of UIs *Interacto* supports and to what extent it is extensible.

The long-term use case detailed in Section 4.2 discusses about LaTeXDraw, a highly-interactive graphical software systems. LaTeXDraw employs both a large panel of standard widgets (*e.g.*, buttons, checkboxes, lists, tabs) and 2D user interactions dedicated to the handling of 2D shapes. LaTeXDraw must be used with mice and keyboards.

We use *Interacto-JavaFX* to build other yet smaller software systems. We developed and maintain *Spoon Visualisation*, a graphical tool for visualizing and interacting with the Spoon Abstract Syntax Tree (AST) of Java code. Spoon is a Java framework for developing dedicated Java code analyzers [35].

It parses Java code to build a Spoon AST that one can handle to transform or analyze Java code. *Spoon Visualisation* uses Spoon to parse Java code and display the resulting Spoon AST using tree-based and text widgets. *Spoon Visualisation* uses other user interactions and widgets that LaTeXDraw and covers a different domain. The Spoon maintainers accepted *Spoon Visualisation* and merged it in the main branch of Spoon as a tool of the Spoon ecosystem.[8]

Regarding *Interacto-Angular*, we ported *Spoon Visualisation* to Angular.[9] We also develop an illustrative Angular Web application to explain how to use the TypeScript implementation of our proposal within Angular.[10] This application mainly uses touch-based user interactions (multi-touch, swipe, *etc.*).

These developments show that *Interacto* can be used in other contexts that LaTeXDraw. The types of software systems one can develop using *Interacto* are the same as the ones targeted by the graphical toolkits we support, namely Angular and JavaFX. So, this encompasses desktop, Web, mobile software systems developed to use touch-screens, mice, keyboards, as input devices. We do not see any blocking issue for porting our proposal to other graphical toolkits that operates with object-oriented programming languages such as Android or React as they rely on the same concepts as Angular and JavaFX.

Regarding the extensibility of the proposal, UI developers can develop new user interactions for *Interacto* implementations using the atomic UI events provided by the UI toolkit. To do so, the developer has to create one class for the user interaction and a second class for its FSM. The developer may have to create another class for defining the data the new user interaction will expose (if the existing interaction data classes do not match the requirements). If a developer has to support a new UI events, he has to create a new class that represents a transition to be used in FSMs. The process is the same for Angular and JavaFX. The companion web page points to the code of a very simple example.

We think that these steps may take more time for a developer than coding the classical assembly of UI events.

Regarding the integration of *Interacto* within guidelines of UI toolkits, our implementations aim at following the same programming style that the one proposed by the UI toolkits. For example with Angular, *Interacto* works with pre-configured Angular linters, leverages Angular standard features (*e.g.*, dependency injection, services, directives), and regarding event processing permits both coding styles. Moreover, style guides of UI toolkits mainly focus on design choices to follow. For example, double-click interactions are not allowed in Gnome applications.[11] These guides have no impact on *Interacto* since they do not discuss the way user interactions are technically processed.

4.4 An empirical study with students

Five contributors developed the long-term use case we detailed in Section 4.2. To overcome this threat of generalization we now discuss the following research question:

RQ5. To what extent beginners successfully use *Interacto* for processing UI events compared to standard UI toolkits?

To study this research question we conducted an empirical study that involved students.

Objects. The object of the experiments is an Angular 9.1 Web application we created for the experiment. This application has a simple Angular service that stores data. This application also has a single Angular component that uses this service. The HTML document of this component contains several widgets: undo/redo buttons; a *div* tag that contains text ; a text area; an SVG document that contains one rectangle. The Angular component does not contain any code related to the processing of UI events produced by these widgets.

We duplicated the application to have two applications for the experiment: a classical Angular application; the same Angular application for which we added *Interacto-Angular* 5.3.0 in its dependencies.

We focus on Angular (and the underlying native Web API) and Interacto for two reasons: *Interacto-Angular* is implemented in TypeScript and works with Angular; Angular is a major Web application framework widely used in the industry. It relies on major software engineering concepts, in particular reactive programming through its data binding features for overcoming the limits of the *Observer* pattern.

Subjects. The subjects of the study are 44 master students in computer science with a strong focus on software engineering. Studies [36], [37] showed that students can be valid and well representative subjects for experiments and development tasks. The subjects are all volunteers with a background on Web development with Angular. Each subject answered three questions regarding his/her expertise in: programming in general (q1); Web programming in general (q2); Web programming with Angular (q3). They have to give a number between 1 and 10 included (1 meaning no expertise and 10 very strong expert). We used their answers to form two balanced groups of 22 students each: group G1 that performed the experiment with Interacto; group G2 that used Angular. To form G1 and G2 we paired subjects that provided similar answers to the three questions to obtain 22 pairs of subjects. G1 and G2 each contain one subject of each pair to balance the two groups. Table 1 details the mean values of the groups for the three questions.

TABLE 1: Expertise of each group of subjects (mean values)

Group	q1	q2	q3
G1	6	3.8	4.1
G2	5.8	3.7	3.8

Sub Research Questions. To discuss RQ5, we formulate the three following sub research questions:

RQ5.1 Does the use of Interacto improve the correctness to process UI events on typical development tasks?

RQ5.2 Does the use of Interacto reduce the time on those tasks?

RQ5.3 To what extent the students prefer using Interacto for completing those tasks?

8. https://github.com/INRIA/spoon/tree/master/spoon-visualisation
9. https://github.com/arnobl/spoon-web/
10. https://github.com/interacto/example-angular
11. https://developer.gnome.org/hig/stable/

Tasks. We designed three representative tasks that cover different UI development aspects, in particular user interaction usages and undo/redo support. The three tasks use different user interactions more or less complex to use or code. Each group had to do the same three tasks (namely, T1, T2, T3). We established a time limit for each task: if reached, they have to stop working on their current task and commit the changes. The total duration of the session was $95\,\mathrm{min}$.

T1. This task is composed of two sub-tasks T1.1 and T1.2. The goal of T1.1 is to use a simple user interaction, the triple click, that subjects can easily use in Angular and Interacto. The subject had to use a triple-click on a given HTML *div* tag to change its color (stored in an Angular service). The goal of T1.2 is to support the undo/redo of this color change. The time limit of T1 is $35\,\mathrm{min}$.

T2. This tasks focuses on the use of a more complex user interaction. This interaction consists in typing text in a text area. If the user stops writing after a delay of 1 second, the text data stored in the Angular service must be updated. This is a mainstream user interaction that text processing tools use to limit the number of editing actions. The time limit of T2 is $20\,\mathrm{min}$.

T3. This task is composed of two sub-tasks T3.1 and T3.2. The goal of T3.1 is to use a complex user interaction, the DnD, provided by Angular and Interacto. The subjects had to use a DnD to move a rectangle. The Angular service stores the rectangle data. A 2D SVG rectangle renders graphically these data. Angular provides DnD features for moving objects graphically by changing, for example, its graphical style (its CSS). This Angular feature, however, does not modify the possible data model a dragged object renders, here the rectangle data.

Interacto provides a DnD interaction, but subjects had to employ it to change the coordinates. The goal of T3.2 is to support the undo/redo of this move. The time limit of T1 is $40\,\mathrm{min}$.

In this section we refer to *T-UNDO* as the tasks related to undo/redo operations, namely: T1.2 and T3.2. We also refer to T-UI as the tasks related to user interactions, namely: T1.1, T2, and T3.1. The goal of these transversal tasks is to discuss results by topics rather than tasks.

Dependent Variables. We collected or computed the following variables:

- *Average Time* (TIME): measures the average time in minutes the subjects spent to complete each question. We computed this metric based on the time stamps of the commits each subject made: we asked the subjects to commit locally before and after each question.
- Correct Answer (CORR): measures the correctness of each question answered by a subject. We measured CORR by designing 10 UI tests: four tests for T1 and T3 that test the user interaction, the data changes, the undo, and the redo; two tests for T2 that test the user interaction and the data changes. The result of a test execution is a boolean value.
- Level of Difficulty (DIFF): measures the difficulty felt by each subject for each task. After each task (or subtask), the subject gave the associated DIFF between 1 and 10 included where 1 is a very easy task and 10 a very difficult one.

Experimental Protocol. The subjects had no knowledge about Interacto before the experiment. They followed several courses of Web development using Angular and may have skills on Angular acquired during internships in the industry. To reduce this knowledge gap the subjects followed one practical session ($95\,\mathrm{min}$) during which they worked on an Angular application. This preliminary session contained exercises on processing UI events using the native Web API, Angular, and Interacto. We asked the subjects to answer the three questions about their expertise after this session. We asked the subjects not to talk to or help each other during the session. The subjects were free to use any other resource to perform the tasks (online documentation, *etc.*).

Results. The TIME, CORR, and DIFF results do not follow a normal distribution. For all the statistical tests used in this section we consider a $95\,\%$ confidence level (*i.e.*, p-value<0.05).

TABLE 2: Total test results, effect size, and confidence of the correctness results

Task	Tests Interacto (pass/fail)	Tests Angular (pass/fail)	Odds Ratio	p-value
1	64 / 16	55 / 27	1.95	0.076
2	32 / 12	22 / 22	2.63	0.048
3	19 / 35	35 / 39	0.61	0.206
Total	115 / 63	112 / 88	1.43	0.093
T-UNDO	31 / 15	26 / 42	3.3	0.004
T-UI	84 / 48	86 / 46	0.93	0.898

Table 2 reports the results regarding the correctness. The columns '*Tests*' reported the total number of executed tests. From the results of the tests executions we produced a 2×2 contingency table. We thus used the Fisher Exact Test [34] to study whether the results from G1 and G2 are independent. To measure the size effect, we used the Odds Ratio [34]. We did not consider the test results related to T1.2 or T3.2 for the subjects that did not start working on each of these task.

The results of T1 tend to be in favor of *Interacto*, but are not significant enough to draw conclusions. Regarding T2, the results are in favor of *Interacto* (significant results) with an odds ratio of 2.63: on T2, the odds that a student achieves better correctness are increased by $163\,\%$ using *Interacto* compared to Angular. Regarding T3, the results do not exhibit significant results. For T3.1, most G2 subjects used the native DnD feature that permits to graphically move objects: Using *Interacto*, G1 subjects had to use an *Interacto* DnD to do such a move programmatically, which is a more complex task. Similarly to T1, the total correctness tends to be in favor of *Interacto*, but is not significant enough to draw conclusions.

Regarding the transversal task *T-UNDO* the results are in favor of *Interacto* (significant results) with an large effect size (3.3): on undo/redo features, the odds that a student achieves better correctness are increased by $230\,\%$ using *Interacto* compared to Angular. Regarding *T-UI* the results are not significant enough to draw conclusions.

RQ5.1 conclusion. First, *Interacto* helped the subjects in correctly adding undo/redo features to the application. Second, we think that an *entrance barrier* may exist for students to master *Interacto* usages. This may concern in particular the use of complex user interactions such as the DnD in T3.

TABLE 3: Means, effect size, and confidence of the time results

Task	Mean Interacto (minutes)	Mean Angular (minutes)	Means Diff. (% (minutes))	$\hat{A}12$	p-value
1.1	16.96	20.42	$-16.9\,\%$ (-3.46)	0.59	0.442
1	27.27	31.97	$-14.7\,\%$ (-4.7)	0.70	0.045
2	13.41	18.48	$-27.4\,\%$ (-5.07)	0.81	< 0.001
3.1	38.64	19.3	$+100.2\,\%$ ($+19.34$)	0.18	< 0.001
3	37.65	38.37	$-1.9\,\%$ (-0.72)	0.44	0.484
Total	77.6	88.62	$-12.4\,\%$ (-11.02)	0.85	< 0.001
T-UNDO	26.33	48.1	$-82.6\,\%$ (-21.77)	0.71	0.029
T-UI	68.3	58.28	$+17.2\,\%$ ($+10.02$)	0.40	0.097

Table 3 reports the results for the TIME variable. We applied the Mann-Whitney test [34] as this test makes no assumptions about the distributions of assessed variables. We measure the effect size using the Vargha-Delaney $\hat{A}$ ($\hat{A}12$) measure [38], [39] following $\hat{A}12$ measure nomenclature [38]: negligible: > 0.5 small: > 0.56, medium: > 0.64, large: > 0.71. With the TIME variable, the less subjects spent time the better it is in favor of the used approach.

We did not report and discuss the results of T1.2 and T3.2 as their values may be strongly biased by the time limit of T1 and T2: One subject may start T1.2 (or T3.2) several minutes before the limit affecting the relevance of studying T1.2 (or T3.2) alone. The results of T1.2 and T3.2 are however integrated in the results of T1 and T3.

When concatenating the TIME results of T1, T2, and T3, the probability that the use of *Interacto* would take less time than the use of Angular is $85\,\%$ (significant result). This result is in favor of *Interacto* and has a large effect size.

For T1, the probability that the use of *Interacto* would take less time than the use of Angular is $70\,\%$ (significant result). This result is thus in favor of *Interacto* with a medium effect size. The reason may be that using *Interacto* the use of interactions is more direct. Also, the support of undo/redo is native in *Interacto* so that subjects did not had to implement their own undo/redo facilities.

For T2, the probability that the use of *Interacto* would take less time than the use of Angular is $81\,\%$ (significant result). This result is thus in favor of *Interacto* and has a large effect size. The G2 subjects had to deal with a widespread keyboard interaction not supported by most of UI toolkits (Angular included), which takes time to support manually. Similarly with T3.1, the probability that the use of *Interacto* would take less time than the use of Angular is $18\,\%$ (significant result). This result, in favor of Angular with a large effect size, has the same origin: as we discussed with CORR, G2 subjects used the native Angular DnD that moves objects graphically. G1 subjects had to use the *Interacto* DnD to change data, which may take more time.

For T3, the data do not show significant results in favor of *Interacto* or Angular. If T3.1 required less time for G2 as discussed in the previous paragraph, T3.2 required more time to G2 for coding undo/redo facilities.

RQ5.2 conclusion. First, globally the use of *Interacto* has a large positive impact of the time spent to do the three tasks against Angular. Second, the results show that providing developers with user interactions has a benefit (for *Interacto* in T2 or Angular in T3.1) in terms development time.

TABLE 4: Means, effect size, and confidence of the difficulty results

Task	Mean Interacto	Mean Angular	Means Diff. (%)	$\hat{A}12$	p-value
1.1	3.27	3.95	$-17.2\,\%$ (-0.68)	0.56	0.51
1.2	4.66	4.85	$-3.9\,\%$ (-0.19)	0.56	0.54
2	4.08	5.76	$-29.2\,\%$ (-1.68)	0.69	0.03
3.1	7.19	4.13	$+74.1\,\%$ ($+3.06$)	0.26	< 0.01
3.2	6.16	7.53	$-18.2\,\%$ (-1.37)	0.71	0.18

Table 4 reports the level of difficulty asked to each subject for each sub-task. Similarly to RQ5.2 we applied the Mann-Whitney test [34]. This table does not contain lines *Total*, *T-UNDO*, and *T-UI*: since multiple subjects did not started (and reported the level of) T1.2 or T3.2 we cannot sum the results of the three tasks, contrary with TIME. With the DIFF variable, the less subjects found a task hard the better it is in favor of the used approach. The significant results concern T2 and T3.1 and seem to be related to the use of predefined user interactions: The probability that the subjects find the task easier with *Interacto* is $69\,\%$ for T2 and $26\,\%$ for T3.1.

Regarding qualitative data for this sub-RQ, we now discuss some of the written comments of the subjects. We identified five categories of remarks, mainly from G1 (*Interacto*).

Pros of Interacto. Multiple subjects detailed that they liked *Interacto* for several reasons. A subject from G2 wrote that "*after we've been taught a bit of Interacto it seems obvious that it can help to do complex actions more easily than only with Angular*". Other subjects wrote that: "*Interacto significantly simplifies certain tasks*"; "*Interacto very easy to handle, very easy to manage the undo/redo*".

Cons of Interacto. "*I do not like the fact I have to define a* ViewChild *in the HTML, then a property in the Angular component and then make the bind. I prefer the Angular system where you just have to bind a function in the HTML*". We agree with this remark and already discussed it with RQ3 in Section 4.2.

"*I also find that for very simple things it can quickly make the application heavier because it makes you write a lot of code.*" Indeed, the coding of undoable classes in particular may take time. A concrete benefit may appear when a developer reuses the same undoable command for several user interactions, which is a common case during the development of UIs (*e.g.*, key shortcuts, menus).

Entrance barrier. "*I think Interacto is very powerful, when you know how to use it well*". "*Struggling to understand totally* dndBinder()". "*Once we understand the principle of the different*

Binder, *the code is done without too much difficulty*". "*Knowing what the variable* i *was in the routines was the most useful part that I saw just at the end*". All these remarks pointed out that using *Interacto* requires learning its major concepts, which can hardly be done in two practical sessions.

Documentation issue. "*The presentation of the Interacto documentation is not very intuitive.*" "*The user community is not yet very developed and therefore no help is available on the forums or on the net in general*". Related to the previous point, these remarks stressed out that *Interacto* has no user community compared to Angular so that it highly relies on its official documentation.

RQ5.3 conclusion. First, the level of difficulty reported by the subjects confirmed that the use of predefined user interactions simplifies the coding activity. Second, the informal feedback are mainly positive regarding the benefits of *Interacto*. The subjects, however, noticed various issues, in particular the entrance barrier that may exist to use *Interacto* correctly.

RQ5 conclusion. This experiment with students on three representative user interaction development tasks exhibits several points. First, the use of *Interacto* is beneficial, in terms of time and correctness, for students to add undo/redo features to the application. Second, the use of predefined user interactions, from both Angular and *Interacto*, is also beneficial in terms of time and correctness. This give another path to explore to improve *Interacto*: *Interacto* may also provide developers with predefined yet partial Interacto binders to do standard actions, similar to the Angular's DnD. Third, an *entrance barrier* to correctly use *Interacto* may exist. This barrier may concern the understanding about how to use the API to turn the execution of an interaction into a command. This is normal for a novel technique and does not hamper the possible adoption of *Interacto* since software engineers tend to learn new frameworks regularly.

4.5 Threats to validity

External validity. This threat concerns the possibility to generalize our findings. The real world use case we detailed in Section 4.2 was developed by authors of this paper. To mitigate this issue we conducted an experiment with 44 master students in computer science to discuss the pros and cons of *Interacto* compared to the Web API/Angular.

Closely related, LaTeXDraw is developed in JavaFX. To mitigate this issue, we discussed in Section 4.3 about other use cases we developed in Angular and JavaFX. The experiment conducted in Section 4.4 used *Interacto-Angular* within an Angular application.

The benchmarks conducted in Section 4.2 concerns JavaFX code. Claiming that the Angular implementation has no overhead as well may require dedicated experiments. However, we cannot identify any reason for having such a difference between these two platforms as they rely on similar concepts. Similarly, the experiment of Section 4.4 used an Angular application. We selected Angular as it follows state-of-the-art practices in terms of front-end development.

We design the experiment using three tasks representative of what a UI developers can do to process user interactions. The selected user interactions (keyboard interaction, simple mouse interaction, DnD) are widespread user interactions. These three tasks are of different difficulties and cover different aspects of the processing of user interactions.

Regarding the population validity, all the subjects were volunteers. We asked the subjects to fill a questionnaire before the experiment on their knowledge in front-end development. We used those data to design two balanced groups of subjects.

Construct validity. This threat relates to the perceived overall validity of the experiments. Regarding tiredness, the duration of the experiment was 95 min. We chose this duration as it is the standard duration of the practical sessions that these students follow. The use of a time limit may introduce a threat in the time results analysis. We consider that not considering a time limit would lead to a more problematic threat to validity: the tiredness as previously discussed. We do not consider the use of a time limit with students as an issue since by their current situation students got accustomed to time limited exercises.

Regarding the learning gap, none of the subjects knew *Interacto*. We conducted an initial practical sessions to introduce Angular and *Interacto-Angular* concepts (data binding, UI event processing). Despite this effort, we are aware that subjects may feel more comfortable with Angular than with *Interacto-Angular*. This issue is accentuated with the documentation: we noted that G1 subjects (*Interacto*) strongly relied on the *Interacto* documentation as the only source of information. The G2 subjects could relied on various sources of information (Angular website, *Stack Overflow*, *etc.*).

5 Related Work

We grouped the related research work into three categories: approaches related to reactive programming and complex event processing; approaches related to UI event processing; approaches that reify user interactions as first-class concepts.

5.1 Reactive programming and complex event processing

Reactive Programming (RP) provides abstractions and mechanisms to use time-changing values in programs [5]. In a user interface development context, RP is used for two purposes: binding and transforming (user interface) data; processing UI events, where UI events are processed as data streams. Data binding is broadly used by recent graphical toolkits (*e.g.*, Angular,[12] WPF,[13] and Android[14]) to update data on other data changes following RP mechanisms. The use of RP to process UI events brings several benefits compared to the use of callback methods. First, it may reduce the size of the code thanks to various stream operators. Second, it overcomes the identified limits of the *Observer* pattern [8], [9], [10], [11]. We could use RP to code the user interactions that Interacto provides, for example using the *ReactiveX* library [40], [41] that Angular already provides and that also works within JavaFX. However, using RP alone (*i.e.*, not within Interacto) to process UI events has the following drawbacks that *Interacto* overcomes as discussed in this paper: UI event is still the

12. https://angular.io/guide/template-syntax
13. https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-wpf
14. https://developer.android.com/topic/libraries/data-binding

core concept and developers have to assemble events to build user interactions; developers still have to write glue code to transform UI events into commands. The rest of this subsection discusses the main RP approaches related to user interface development.

ReactiveUI[15] is a RP framework dedicated to the user interface development. This framework considers commands as first-class concepts. Developers can process UI events by producing commands thanks to specific routines. ReactiveUI, however, still consider UI events as first-class concepts.

[42] propose the use of RP to develop user interfaces. In particular, UI events are considered as data streams that can be processed. UI events, however, are still first-class concepts in this approach.

Ur/Web is a programming language for the Web [43]. RP is used in this approach for rendering graphical objects. Event callbacks are still used to control interactive objects.

[44] propose a programming paradigm for developing interactive event-driven systems. This approach is not specific to UIs. When applied to user interfaces, it focuses on the rendering of graphical objects and UI data binding.

Mobl is a declarative language for programming mobile web applications [45]. *Mobl* implements the *Model-View* pattern: no controller is used to link views to data models. The processing of low-level UI events and the data bindings are moved to the *View*. One goal of *Mobl* is to reduce the boilerplate code written in controllers to synchronize data models and views. *Mobl* promotes separation of concerns by supporting the separation of user interface and data model, which is a corner-stone of user interface engineering. *Mobl* provides reactive behavior mechanisms to be used directly in views to update them. *Interacto* is not tied to a specific architecture: it requires accesses to the interactive objects that compose the user interface to process their events. The current implementations of *Interacto*, however, cannot work with the *Model-View* pattern as views are usually described in an XML dialect.

Elm is a functional RP framework for programming user interfaces [26]. When used to develop user interfaces, *Elm* focuses on two aspects: building and layouting user interfaces; processing UI events. Similarly to the other RP approaches, *Elm* considers UI low-level events only. *Elm* does not consider the concept of command.

Scala.React is a Scala RP framework that aims at overcoming the limits of the *Observer* pattern [8]. This paper takes as example the development of user interfaces. UI events, however, are still first-class concepts and commands are not considered.

Flapjax is a programming language for Web applications [46]. *Flapjax* provides, on the top of JavaScript, reactive programming features to tackle various web development problems. As most of the languages or frameworks discussed in this section *Flapjax* can be used to develop user interactions. For example, the authors illustrate parts of *Flapjax* by developing a DnD. The authors notice that building user interactions bring benefits for developers: "*by separating the DnD event stream from action of moving the element, we have enabled a variety of actions*". Commands, undo/redo, and the associated glue code, however, are not considered.

15. https://reactiveui.net

[11] discuss some challenges of programming user interfaces, in particular the data synchronization and update. The authors highlight the complexity of understanding the spaghetti code provoked by the use of handlers to update data. The authors propose an approach to overcome this problem. This work focus in data binding and did not target UI event processing.

Complex Event Processing (CEP) tackles the problem of analyzing data to detect event pattern [47]. User interactions could be developed using CEP. For example, a DnD may match the event-based pattern *press-drag+-release*. The main drawback is that a pattern is matched when all the required events are processed. User interactions may require to process its events all along its execution.

5.2 UI event processing approaches

The Garnet system [48] provides developers with a set of predefined, reusable, and customizable sets of behavioral interactive objects called interactors. Interactors aim at hiding the UI event processing from developers. Following this work, the author of Garnet then proposed the use of (undoable) commands with interactors [49]. These work are certainly the closest ones that inspired *Interacto*. They, however, follow a different philosophy than *Interacto*. Interactors are interactive objects predefined (yet customizable) for specific actions. For example, the predefined *move-grow* interactor aims at moving or changing the size of an object. *Interacto* promotes the concept of user interaction as a first-class concern to replace the current usage of UI events. By using *Interacto*, developers are free to use a given user interaction to produce various commands.

More recently, *InterState* is an approach for defining user interface behavior [7], [50]. The authors motivate the limits of the current UI event-callback model to then propose the use of FSMs to describe different parts of a user interface behavior. Concretely, *InterState* is a new programming language and environment for helping developers in coding and reusing UI code. With *Interacto* we do not want developers to use another language or environment. We aim at proposing developers with a technique that seamlessly works within their UI toolkits. *InterState* does not consider UI commands. Moreover, with *InterState*, the states of a user interaction are directly bound to properties of a data object to change. This makes the job of developers more complex when they have to replace the current user interaction with another one.

Based on the work of [48], [51] propose an approach for developing interactive graphical objects. This approach proposes a DSL (*Domain-Specific Language*) embedded in Scala to develop interactors. This work shares several ideas with the concept of *Interacto*: low-level UI events are hidden from developers, to use predefined and customizable interactors, similarly to [48]. These interactors can be controlled with some routines close to the ones that form the *Interacto* fluent API, such as *when*. This approach, however, focuses on defining view templates. It also does not propose a process to automatically transform user interactions into commands since commands are not first-class concepts in this approach.

5.3 User interactions as objects

Reifying user interactions as first-class concerns is not new and largely admitted in the HCI community. [23] propose

to code user interactions as FSMs instead of using low-level UI events. This approach, however, fully focuses on user interactions and do not consider separation of concerns, code reuse, and UI commands.

[25] propose the use of Petri nets to model the behavior of user interfaces, user interactions included. This approach, however, does not propose any user interaction processing model.

[24], [52] propose an architectural design pattern where user interactions are modeled as FSMs. This design pattern also suggests the separation of concerns between user interactions and UI commands. No detail, however, is provided on a user interaction processing mechanism to transform such user interactions into commands.

Various UI modeling approaches aim at focusing on software interactivity. Task modeling approaches, such as *ConcurTaskTrees* (CTT) [53], aim at expressing user's activities by describing tasks that users can do. These approaches thus focus at a high level of abstraction on UI commands. The *Interaction Flow Modeling Language* (IFML) [54] aims at specifying UIs. IFML has an *events specification* and an *event transition specification* perspectives that detail the events (UI events included) that change the state of the UI and their impacts on this last. UsiXML [55] is a multi-level approach for building multi-platform and adaptive UIs. The top level of UsiXML concerns task modeling with the same goals than CTT. Finally, improving the interactivity or more generally the user experience of model-driven approaches and environments is a major concern [56], [57]. For example, [58] propose to complete generative DSL environment approaches with models dedicated to UIs and user interactions for improving the user experience.

5.4 UI Toolkits

Several UI toolkits provide developers with features for coding (undoable) UI commands. For example, Java Swing already proposed to associate UI commands to simple widgets (*e.g.*, buttons) and provides a linear undoable history. These features are now part of JavaFX. WPF has a similar feature for binding commands to simple widgets. To the best of our knowledge, Angular and Android rely on event handlers for processing UI events and do not have command features such as the ones provided by WPF. *Interacto* is inspired by these features and brings facilities (algorithms, a dedicated API, run-time optimizations, object-oriented properties, *etc.*) that help developers in coding how executions of a user interaction can produce command instances. Interacto improves: 1/ the expressiveness of the use of UI commands with simple widgets with the above-mentioned facilities and with multiple binder routines (when, log, consume, *etc.*) to customize the command creation; 2/ the support of more complex user interactions in their usages for producing commands.

Several UI toolkits have reuse facilities: Angular has *Directives*,[16] WPF has *Behaviors*,[17] and Android has *Slices*.[18] These features are interesting as they permit developers to extend the behavior of widgets with new features and new properties. However even when using these features, a developer still has to handle low-level UI events to produce commands and gets no specific support for turning user interaction executions into commands as in *Interacto*. These UI toolkits features can however be very useful for *implementing Interacto*. For example, our Angular implementation of *Interacto* partly relies on Angular directives as a reuse mechanism. We believe that a port of *Interacto* to WPF or Android might similarly rely on Behavior and Slice objects.

Regarding the testability of commands and of the production of command instances, core testing and UI testing frameworks (e.g. TestFX for JavaFX, Expresso for Android, or Mocha/Jasmine/Protractor for Angular) provide core features that enable us to write more complex test assertions and test frameworks: based on and complementary to these frameworks, *Interacto* proposes dedicated testing oracles implemented in tools for easing the writing of tests for commands and production of commands. Our implemented testing tool both generate skeletons of test classes and also provides a framework for helping in writing UI command tests and for testing the production of commands.

Finally, several UI toolkits try to overcome the problem of relying on basic UI events only by providing supplementary UI events. For example, the HTML API provides the *dragstart*, *drag*, and *dragend* events that respectively represent the starting, the running, and the ending of a DnD. If such UI are a progress towards the support of complex user interactions, they still rely on the UI event processing model and its limits: they do not help developer in turning UI events into commands, while *Interacto* notably provides algorithms, a dedicated API, run-time optimizations, object-oriented properties, for this purpose.

6 Conclusion

This paper presents *Interacto*, a novel user interaction processing model. Based on software engineering good practices, *Interacto* aims at better engineering UIs. Instead of providing developers with low-level UI event processing, *Interacto* reifies user interactions and UI commands as first-class concerns. The two implementations of *Interacto*, *Interacto-JavaFX* and *Interacto-Angular*, show that the proposal is not tied to a specific language or UI platform.

The long term experiment shows that the proposal scales for one very interactive and widely-used software system. The experiment conducted with students exhibited several pros and cons of the proposal. The use of *Interacto* is beneficial, in terms of time and correctness, for students to add undo/redo features to the application. The use of predefined user interactions is also beneficial in terms of time and correctness. However, an *entrance barrier* to use correctly *Interacto* may exist.

In our future work, we will investigate how to provide developers with predefined yet partial Interacto binders to do standard actions, similar to the Angular's DnD that moves objects graphically. Second, we will investigate how to help HCI designers in designing and testing novel user interactions, and how to produce concrete user interactions for integration in *Interacto*. Finally, *Interacto* may also help the design of dynamic and static code analyzing techniques for the producing of UI tests, in the continuation of the test generation technique we propose.

16. https://angular.io/api/core/Directive
17. https://github.com/Microsoft/XamlBehaviorsWpf/wiki
18. https://developer.android.com/guide/slices

REFERENCES

[1] Z. Mijailovic and D. Milicev, "A retrospective on user interface development technology," *IEEE Software*, vol. 30, pp. 76–83, 2013.

[2] M. Green, "A survey of three dialogue models," *ACM Trans. Graph.*, vol. 5, no. 3, pp. 244–275, 1986.

[3] M. Beaudouin-Lafon, "Designing interaction, not interfaces," in *Proc. of AVI '04*. ACM, 2004, pp. 15–22.

[4] G. E. Krasner, S. T. Pope *et al.*, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

[5] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, p. 52, 2013.

[6] B. A. Myers, "Separating application code from toolkits: Eliminating the spaghetti of call-backs," in *Proc. of UIST'91*. ACM, 1991, pp. 211–220.

[7] S. Oney, B. Myers, and J. Brandt, "Interstate: Interaction-oriented language primitives for expressing GUI behavior," in *Proc. of UIST '14*. ACM, 2014, pp. 10–1145.

[8] I. Maier and M. Odersky, "Deprecating the observer pattern with scala. react," EPFL, Tech. Rep., 2012.

[9] G. Salvaneschi and M. Mezini, "Towards reactive programming for object-oriented applications," *Trans. on AOSD*, vol. 8400, pp. 227–261, 2014.

[10] G. Salvaneschi, S. Amann, S. Proksch, and M. Mezini, "An empirical study on program comprehension with reactive programming," in *Proc. of FSE 2014*. ACM, 2014, pp. 564–575.

[11] G. Foust, J. Järvi, and S. Parent, "Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems," in *Proc. of GPCE'2015*. ACM, 2015, pp. 121–130.

[12] A. Blouin, V. Lelli, B. Baudry, and F. Coulon, "User Interface Design Smell: Automatic Detection and Refactoring of Blob Listeners," *Information and Software Technology*, vol. 102, pp. 49–64, 2018.

[13] Oracle, 2018. [Online]. Available: https://openjfx.io

[14] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript," in *Proc. of ECOOP'14*, 2014, pp. 257–281.

[15] Google, 2019. [Online]. Available: https://angular.io

[16] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[17] M. Potel, "MVP: Model-View-Presenter the Taligent programming model for C++ and Java," *Taligent Inc*, 1996.

[18] J. Smith, 2009. [Online]. Available: https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern

[19] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[20] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, pp. 131–183, 1992.

[21] R. E. Johnson, "Frameworks = (components + patterns)," *Commun. ACM*, vol. 40, no. 10, pp. 39–42, 1997.

[22] A. Apaolaza and M. Vigo, "Wevquery: Testing hypotheses about web interaction patterns," *Proc. ACM Hum.-Comput. Interact.*, vol. 1, no. EICS, pp. 4:1–4:17, Jun. 2017.

[23] C. Appert and M. Beaudouin-Lafon, "Swingstates: Adding state machines to java and the swing toolkit," *Software: Practice and Experience*, vol. 38, no. 11, pp. 1149–1182, 2008.

[24] A. Blouin and O. Beaudoux, "Improving modularity and usability of interactive systems with Malai," in *Proc. of EICS'10*, 2010.

[25] D. Navarre, P. Palanque, J.-F. Ladry, and E. Barboni, "ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability," *ACM Trans. on CHI*, vol. 16, no. 4, pp. 1–56, 2009.

[26] E. Czaplicki and S. Chong, "Asynchronous functional reactive programming for guis," in *Proc. of PLDI '13*, 2013, pp. 411–422.

[27] P. Baudisch, E. Cutrell, D. Robbins, M. Czerwinski, P. Tandler, B. Bederson, A. Zierlinger *et al.*, "Drag-and-pop and drag-and-pick: Techniques for accessing remote screen content on touch-and pen-operated systems," in *Proc. of Interact'03*, 2003, pp. 57–64.

[28] C. Appert, O. Chapuis, and E. Pietriga, "Dwell-and-spring: undo for direct manipulation," in *Proc. of CHI'12*. ACM, 2012, pp. 1957–1966.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[30] A. Prakash and M. J. Knister, "A framework for undoing actions in collaborative systems," *Trans. on CHI*, vol. 1, pp. 295–330, 1994.

[31] Y. Yoon and B. A. Myers, "Supporting selective undo in a code editor," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 223–233.

[32] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (gui) testing: Systematic mapping and repository," *Information and Software Technology*, pp. 1679–1694, 2013.

[33] V. Lelli, A. Blouin, and B. Baudry, "Classifying and Qualifying GUI Defects," in *Proc. of ICST'15*, 2015, pp. 1–10.

[34] D. J. Sheskin, *Handbook Of Parametric And Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, January 2007.

[35] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015.

[36] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *Proc. of ICSE'15*, vol. 1. IEEE, 2015, pp. 666–676.

[37] M. Svahnberg, A. Aurum, and C. Wohlin, "Using students as subjects-an empirical evaluation," in *Proc. of ESEM'08*, 2008.

[38] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, pp. 101–132, 2000.

[39] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[40] E. Meijer, "Reactive extensions (rx): curing your asynchronous programming blues," in *ACM SIGPLAN Commercial Users of Functional Programming*. ACM, 2010, p. 11.

[41] A. Maglie, *ReactiveX and RxJava*. Apress, 2016, pp. 1–9.

[42] A. Courtney and C. Elliott, "Genuinely functional user interfaces," in *Haskell workshop*, 2001, pp. 41–69.

[43] A. Chlipala, "Ur/web: A simple model for programming the web," in *Proc. of POPL '15*. ACM, 2015, pp. 153–165.

[44] A. Milicevic, D. Jackson, M. Gligoric, and D. Marinov, "Model-based, event-driven programming paradigm for interactive web applications," in *Proc. of Onward! 2013*. ACM, 2013, pp. 17–36.

[45] Z. Hemel and E. Visser, "Declaratively programming the mobile web with Mobl," in *Proc. of OOPSLA '11*. ACM, 2011, pp. 695–712.

[46] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: A programming language for ajax applications," in *Proc. of OOPSLA '09*, 2009.

[47] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[48] B. A. Myers, "A new model for handling input," *ACM Trans. Inf. Syst.*, vol. 8, no. 3, pp. 289–320, Jul. 1990.

[49] B. A. Myers and D. S. Kosbie, "Reusable hierarchical command objects," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '96. ACM, 1996, pp. 260–267.

[50] S. Oney, B. Myers, and J. Brandt, "ConstraintJS: programming interactive behaviors for the web by integrating constraints and states," in *Proc. of UIST'12*. ACM, 2012, pp. 229–238.

[51] O. Beaudoux, M. Clavreul, A. Blouin, M. Yang, O. Barais, and J.-M. Jézéquel, "Specifying and Running Rich Graphical Components with Loa," in *Proc. of EICS'12*, 2012, pp. 169–178.

[52] A. Blouin, B. Morin, O. Beaudoux, G. Nain, P. Albers, and J.-M. Jézéquel, "Combining Aspect-Oriented Modeling with Property-Based Reasoning to Improve User Interface Adaptation," in *Proc. of EICS'11*, 2011.

[53] F. Paterno, C. Mancini, and S. Meniconi, "Concurtasktrees: A diagrammatic notation for specifying task models," in *Human-computer interaction INTERACT'97*. Springer, 1997, pp. 362–369.

[54] M. Brambilla and P. Fraternali, *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann, 2014.

[55] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero, "Usixml: A language supporting multi-path development of user interfaces," in *Proc. of Interact'04*, 2004.

[56] A. Blouin, N. Moha, B. Baudry, H. Sahraoui, and J.-M. Jézéquel, "Assessing the Use of Slicing-based Visualizing Techniques on the Understanding of Large Metamodels," *Information and Software Technology*, vol. 62, no. 0, pp. 124 – 142, 2015.

[57] S. Abrahão, F. Bourdeleau, B. Cheng, S. Kokaly, R. Paige, H. Stöerrle, and J. Whittle, "User experience for model-driven engineering: Challenges and future directions," in *Proc. of MODELS'2017*. IEEE, 2017, pp. 229–236.

[58] V. Sousa, E. Syriani, and K. Fall, "Operationalizing the integration of user interaction specifications in the synthesis of modeling editors," in *Proc. of SLE'19*, 2019, pp. 42–54.