# On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study

Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini

**Abstract**—Starting from the first investigations with strictly functional languages, reactive programming has been proposed as *the programming paradigm* for reactive applications. Over the years, researchers have enriched reactive languages with more powerful abstractions, embedded these abstractions into mainstream languages – including object-oriented languages – and applied reactive programming to several domains, like GUIs, animations, Web applications, robotics, and sensor networks. However, an important assumption behind this line of research is that, beside other claimed advantages, reactive programming makes a wide class of otherwise cumbersome applications more comprehensible. This claim has never been evaluated. In this paper, we present the first empirical study that evaluates the effect of reactive programming on comprehension. The study involves 127 subjects and compares reactive programming to the traditional object-oriented style with the Observer design pattern. Our findings show that program comprehension is significantly enhanced by the reactive-programming paradigm – a result that suggests to further develop research in this field.

**Index Terms**—Reactive Programming, Empirical Study, Controlled Experiment, Software Comprehension

## 1 Introduction

Reactive applications represent a wide class of software that needs to continuously and interactively respond to internal or external stimuli with a proper action. Examples of such applications include user-interactive software, like GUIs and Web applications, graphical animations, data acquisition from sensors, and distributed event-based systems.

Over the last few years, reactive programming (RP) has gained the attention of researchers and practitioners for the potential to express otherwise complex reactive behavior in an intuitive and declarative way. RP was first introduced in Haskell [1], [2]. Influenced by the approaches based on Haskell, implementations of RP have been proposed in several widespread languages, including Scheme [3], Javascript [4], and Scala [5], [6]. Recently, concepts inspired by RP have been applied to production frameworks, such as Microsoft Reactive Extensions (Rx) [7], which received great attention after the success story of the Netflix streaming media provider. This growing interest around RP is also witnessed by the success of the Coursera online class "Principles of Reactive Programming", in winter semester 2013-14. Finally, a lot of attention for RP and its principles can be observed in the community of front-end developers. An increasing number of libraries have been released that are inspired by the Flapjax reactive language [4], e.g., React.js, Bacon.js, Knockout, Meteor, and Reactive.coffee.

- *G. Salvaneschi, is with the Reactive Systems Group, Department of Computer Science, Technische Universität Darmstadt, Germany. E-mail: salvaneschi@cs.tu-darmstadt.de*
- *S. Proksch, S. Amann, and M. Mezini are with the Software Technology Group, Department of Computer Science, Technische Universität Darmstadt, Germany. E-mail: {lastname}@cs.tu-darmstadt.de*
- *S. Nadi is with the Department of Computing Science, University of Alberta, Canada. E-mail: nadi@ualberta.ca*

*Manuscript received December 20, 2015.*

The relevance of RP comes from the well-known complexity of reactive applications, which are hard to develop and understand, because of the mixed combination of data and control flow. The Observer design pattern [8] is widely used for such applications. It has the advantage of decoupling observers from observables. However, when it comes to program readability, it does not make things easier, because of dynamic registration, side effects in callbacks, and inversion of control [4].

In contrast, RP supports a design based on data flows and time-changing values: the programmer states which relations should be enforced among the variables that compose a reactive program and the RP runtime takes care of performing all the required updates [9]. Both programming paradigms differ in several ways. In RP, dependencies are defined explicitly, while in the Observer design pattern they result from both control and data flow. In RP, reactions can be directly composed according to their types as opposed to callbacks that are composed by sharing a common variable they read from/write to and return void. Also, in contrast to the Observer pattern, RP does not feature inversion of control and generally results in shorter code, since collecting dependencies and performing the updates is automated by the runtime.

Based on these characteristics, it has been repeatedly argued that RP greatly improves over the traditional Observer pattern used in Object-oriented (OO) programming both (i) from the *software design* perspective as well as (ii) from the perspective of facilitating comprehension of software [3], [4], [10], [11]. However, there has not been enough empirical evidence to support these claimed advantages of RP.

Preliminary empirical results seem to confirm the claimed design benefits (e.g., higher composability) of RP [5]. On the other hand, even preliminary evidence is missing regarding the claim that RP enhances comprehension. Despite the intuition about its potential, the reactive style is not *obviously* more comprehensible than the Observer design pattern. For example, in the Flapjax paper [4], a Javascript application based on the Observer pattern is compared against a functionally-equivalent RP version. The authors argue that the RP version is much easier to comprehend. However, the reader is warned that: *"Obviously, the Flapjax code may not appear any 'easier' to a first-time reader"*. Doubting, at this point, is legitimate: does RP really make reactive applications easier to read? Also, it is unclear how much expertise is required to find a RP program "easier" – if ever.

To fill this gap, this paper provides the first empirical evaluation of the impact of RP on program comprehension compared to the traditional technique based on the Observer design pattern. The experiment involves 127 subjects that were divided into an RP group and an OO group. They were shown several reactive applications and their comprehension of the reactive functionalities was measured. The experiment considers several aspects including (i) correctness of comprehension, (ii) required time, and (iii) programming-skills level needed for correct comprehension. We additionally asked the subjects for their preferred programming paradigms and for arguments that explain their choice. To the best of our knowledge, such a study has never been attempted before. Our results show that the comprehension of reactive applications is significantly improved by using reactive programming (as opposed to the Observer pattern). The answers we received from the subjects suggests that this improvement may be due to characteristics of reactive programming, such as reduced boilerplate code and better readability. In summary, the contributions of this paper are as follows:

- The first empirical study that shows that reactive programming does improve program comprehension, based on data from 127 subjects. To the best of our knowledge, this is the largest experiment ever conducted on program comprehension.
- Qualitative feedback data from subjects that provides explanations of the perceived differences of reactive programming and the Observer pattern.
- An outlook on possible directions for reactive programming language design, based on the interpretation of our results.

This paper is an extension of our previous work at FSE'14 [12], which presented results on 38 subjects. Compared to the conference version, we expanded the experiment to include an additional 89 subjects. This allowed us to run the statistical tests on a much larger set of subjects. The new data confirms the conclusions drawn in the conference paper, while even increasing the statistical significance of the obtained results and our confidence in their validity. The additional data also allowed us to extend the analysis of correctness to individual comprehension tasks. In addition to the program comprehension tasks, our new experiments added a questionnaire that sheds light on the interpretation of the controlled experiment and helps shape directions for future research.

The rest of the paper is organized as follows: Section 2 motivates the work. Section 3 introduces the design of the study. Section 4 describes the results. Section 5 provides a qualitative interpretation. Section 6 discusses areas of future research. Section 7 describes threats to validity. Section 8 presents related work. Section 9 concludes the paper.

## 2 Motivation

Reactive programming provides dedicated language abstractions for implementing reactive applications. Over the years, several RP languages have been proposed. Despite some differences between concrete languages, the following principles are shared among most approaches:

**Declarativeness** With RP, programmers state *how* components functionally depend on each other, rather than providing the computational steps that derive the new state of a component based on the state of another one.

**Abstraction over change propagation** RP makes the propagation of change in reactive applications implicit. Developers do not need to manually update dependent values, because the language runtime takes care of change propagation.

**Composability** RP provides abstractions (e.g., operators over event streams) to directly compose reactive computations into more complex ones.

**Favoring data flow over control flow** In RP, computation is driven by new data/events flowing into the system, rather than by the execution following the flow of control of the application.

These principles result in a diverse set of concrete languages: Functional Reactive Programming (FRP) [1], [2] focuses on modeling time in pure functional programming languages. In FRP, programs consist of a pure composition of time-changing values: event streams (i.e., discrete time-changing values) and behaviors (i.e., continuous time-changing values). Languages such as FrTime [3], Flapjax [4], Scala.react [5], [6] and REScala [5] embed reactive abstractions into existing (non-pure) languages. The resulting programming style mixes reactive and traditional imperative abstractions. Additionally, side effects are allowed, for example, to update the state from an event handler attached to an event stream. Rx [7] and Bacon.js [13] also embed reactive abstractions in (non-pure) general-purpose languages. They provide event streams and operators that transform, filter, and combine event streams, but do not attempt to model continuous time-changing values.

We focus on the RP flavor promoted by REScala, the language we adopted for the experiment. The possible impact of this choice on the results is discussed in Section 7. REScala supports first-class constraints among program values. These constraints are automatically enforced by the language runtime. Throughout the rest of the paper, we adopt the terminology of REScala [5] and Scala.react [6], which we will introduce in the following. Constraints are expressed as *signals*, a language concept for expressing functional dependencies among values in a declarative way. A signal can depend on reactive values with no further dependencies (referred to as `Vars`, which are updated imperatively) or on other signals. Any change to a source on which a signal depends results in the language runtime automatically recomputing the expression defining the reactive value. This is done to keep the reactive value up-to-date. The general form of a signal `c` is `Signal{`*expr*`}`, where *expr* is a standard expression. When *expr* is evaluated, all `Signal` and `Var` values

```
1  val a = Var(1)
2  val b = Var(2)
3  val c = Signal{ a() + b() }
4
5
6
7
8
9
10
11 println(c.getVal())  // 3
12 a()=4
13 println(c.getVal())  // 6
```

**(a)**

```
1  val a = Observable(1)
2  val b = Observable(2)
3  val c = a + b
4
5  a.addObserver( _ =>{
6    c = a + b
7  })
8  b.addObserver( _ =>{
9    c = a + b
10 })
11 println(c.get())  // 3
12 a.set(4)
13 println(c.get())  // 6
```

**(b)**

Figure 1: Functional Dependencies using Signals in RP (a) and the Observer Design Pattern in OO Programming (b).

it refers to are registered as dependents of *c*; any subsequent change in them triggers a reevaluation of *c*.

Consider the example in Figure 1a. The code snippet defines two vars `a` and `b`. When `c` is declared (Line 3), the runtime collects the values it depends on (i.e., `a` and `b`). When `a` is updated (Line 12), all signals that (even indirectly) depend on it – `c` in this case – are automatically updated as well. As the reader probably noticed, the syntax of the example is a bit cluttered, because of the implementation of RP as an embedded Scala DSL. Assigning a var requires `()=`, which is translated into a call to the `apply` method on the var object. Similarly, vars and signals must appear with the method call notation `()` in signal expressions. More details can be found in the original publications [5], [6].

**Old-school Reactivity: The Observer Pattern.** In OO languages, reactive applications are usually developed using the Observer design pattern [8]. This solution has gained wide popularity, because it decouples observers from observables, i.e., observables do not know (hold a static reference to) observers. Instead, observers register themselves to the observable they are interested in at runtime. Detractors argue that programs based on the Observer pattern are hard to reason about [10]. Below, we summarize the main points that support this argumentation. For easier illustration, we refer to the example in Figure 1b, which implements the same functionality as Figure 1a, following the Observer pattern.

*(i)* The natural dependency among program entities is inverted. Conceptually, changes flow from observables to observers, but in code, observers call the observable to register. For example, `c` is registered by calling `addObserver` on `a` (Line 5).

*(ii)* Programmers need to inspect a lot of code to figure out the reactive behavior, because functional dependencies are implicit. To define the dependency from `a` to `c`, programmers register a handler to `a` that updates `c`. When readers encounter `c` in the code for the first time, there is no sign that the value permanently depends on another one, since the update is performed by a side effect in the handler potentially anywhere in the program.

*(iii)* Code is cluttered. Reactive applications that make use of the Observer pattern are more verbose. The application logic is hidden behind the machinery required by the Observer design pattern.

The real impact of the previous issues is not clear though and each one could be countered with another claim. For example, (i) contributes to make OO applications too complex to read at first sight, but, with experience, programmers are likely to get used to inversion of control. For (ii), the handler still expresses the functional dependency, even if indirectly. Concerning (iii), there is no evidence that the Observer design pattern clutters programs up to the point that they are significantly harder to read than with an alternative design.

In summary, the claims that RP addresses the aforementioned issues and improves program comprehension need to be evaluated empirically.

## 3 Study Design

The claimed advantages of RP include increased composability, abstraction over state, enforcement of consistency guarantees during change propagation, and ease of comprehension [3], [4], [5], [10], [11]. In this work, we limit the scope to program comprehension. We argue that this aspect is crucial, because programs are written once, but read many times: *"Programs must be written for people to read, and only incidentally for machines to execute"* [14]. We consider only a single alternative to RP, i.e., OO programming and the Observer design pattern, because it is the most common solution for reactive applications.

### 3.1 Research Questions

The overall purpose of our study is to empirically investigate the impact of RP on the comprehension of reactive applications. We divide this into two goals: (1) investigating *if* RP has an impact on program comprehension, when compared to the OO style, and (2) *why* such a difference exists. We formulate our research questions as follows, where the first three questions address the first goal and the last question addresses the second goal.

*RQ1: Does reactive programming impact the correctness of program comprehension?*

We are interested in how developers comprehend programs written in the RP and in the OO styles. We analyze the correctness of answers to comprehension questions.

*RQ2: Does reactive programming impact time for program comprehension?*

In addition to correctness, we want to investigate if the required time to comprehend RP programs is different to OO programs.

*RQ3: Does comprehending RP programs require a different programming-skills level than the OO style?*

We are interested in the relation between programming-skills level and the level of program comprehension for both RP programs and OO programs. It is important to measure the impact of programming expertise on the use of a programming language to know whether it is only useful for advanced programmers or also exhibits significant benefits for beginners.

*RQ4: What are the reasons for a difference – if any – in comprehending RP programs and OO programs?*

The last question investigates the underlying reasons for differences we find with respect to RQ1-3.

```scala
object Squares_Reactive extends SimpleSwingApplication {

  // -- APPLICATION LOGIC ---------------------
  object square1 {
    val position = Signal {
      Point(time().s * 100, 100)
    }
  }
  object square2 {
    val v = Signal {
      time().s * 100
    }
    val position = Signal {
      Point(time().s * v(), 200)
    }
  }










  // painting components
  (square1.position.changed || square2.position.changed) += {
     _ => Swing onEDT { top.repaint }
   }

  // -- Graphics ---------------------
   lazy val panel: RePanel = new RePanel {
    override def paintComponent(g: Graphics2D) {
      super.paintComponent(g)
      g.fillRect(
        square1.position.getValue.x.toInt - 8,
        square2.position.getValue.y.toInt - 8,
        16, 16)
      g.fillRect(
        square1.position.getValue.x.toInt - 8,
        square2.position.getValue.y.toInt - 8,
        16, 16)
    }
  }
  lazy val top = new MainFrame {
    preferredSize = new Dimension(800, 400)
    contents = panel
  }
}
```

**(a)**

```scala
object Squares_Observer extends SimpleSwingApplication {

  // -- APPLICATION LOGIC ---------------------
  object square1 {
    val position = Observable { Point(0, 0) }
    addTimeChangedHandler { time =>
      position set Point(time.s * 100, 100)
    }
  }
  object square2 {
    val v = Observable { 0.0 }
    val position = Observable { Point(0, 0) }

    addTimeChangedHandler { time =>
      v set time.s * 100
      updatePos(time, v.get)
    }
    v addObserver { v =>
      updatePos(now, v)
    }
    def updatePos(time: Time, v: Double) {
      position set Point(time.s * v, 200)
    }
  }

  // painting components
  square1.position addObserver { _ => repaint }
  square2.position addObserver { _ => repaint }
  def repaint = Swing onEDT { top.repaint }

  // -- Graphics ---------------------
  lazy val panel: RePanel = new RePanel {
    override def paintComponent(g: Graphics2D) {
      super.paintComponent(g)
      g.fillRect(
        square1.position.getValue.x.toInt - 8,
        square2.position.getValue.y.toInt - 8,
        16, 16)
      g.fillRect(
        square1.position.getValue.x.toInt - 8,
        square2.position.getValue.y.toInt - 8,
        16, 16)
    }
  }
  lazy val top = new MainFrame {
    preferredSize = new Dimension(800, 400)
    contents = panel
  }
}
```

**(b)**

Figure 2: The Squares Application Implemented with RP (a) and the Observer Design Pattern in OO Programming (b).

While previous research has speculated on RP overperforming OO programming w.r.t. program comprehension, we take a neutral approach. Consequently, in Section 4, we use 2-tailed statistical tests to analyze our results.

## 3.2 Work Method

To answer the above research questions, we design an experiment that is composed of 10 comprehension tasks and a short questionnaire. In each task, a subject is shown a reactive application (written in RP or OO) and asked a question about the application's behavior.

In the rest of this subsection, we will present the reactive programs we used in the experiment, the details of the comprehension tasks and questionnaire, as well as how we executed the study.

### *3.2.1 Reactive Programs Used*

To analyze the difference between RP and OO, the experiment focuses on 10 reactive programs we developed. Each program is implemented in two versions. The RP version is based on reactive programming, i.e., signals and, when needed, events. The OO version adopts the Observer pattern to implement reactivity. The applications belong to three categories: synthetic, animated, and interactive applications. Each category addresses a different comprehension problem.

**Synthetic applications** (applications 1-4) define functional dependencies among values and propagate changes when certain values are updated. These applications look similar to Figure 1 except that the functional dependencies are more complex.

**Graphical animations** (applications 5-7) display shapes on a canvas and move them in regular patterns. Graphical animation is a traditional domain for reactive programming [1], [2].

**Interactive applications** (applications 8-10) require the user, for example, to click buttons, insert text in a form, or drag the mouse over a shape. These functionalities are common in GUIs – another traditional domain for reactive programming [3], [4].

We present an example of the *animation* category in Figure 2, which draws two moving squares on a canvas. The RP version of the application is presented in Figure 2a, the OO version is presented in Figure 2b. Please note that the code is reduced for presentation purposes. To give an intuition of

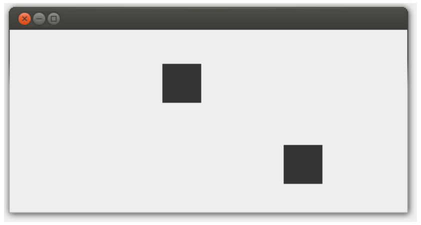

Figure 3: The Squares Graphical Animation.

the execution, a screenshot is provided in Figure 3 that was not available to the subjects. When running the application, the upper and the lower square move horizontally from left to right at constant and increasing speed, respectively.

In the RP version (Figure 2a), the `position` signal in Line 5 models a time-changing value of type `Point`. The `x` coordinate of the point depends on the `time` signal. The `y` coordinate is fixed. Every time the `time` signal changes (defined elsewhere), a new point is generated and assigned to the content of the `position` signal. The `position` signal for the second square (Line 13) works similarly, except that the `x` coordinate of the point (Line 14) depends on both time and a speed signal `v` defined on Line 10. Line 27 triggers a repainting in the asynchronous Swing events loop every time either position of `square1` or the position of `square2` changes. Lines 32-47 setup the canvas and display the initial squares.

In the OO version (Figure 2b), the same functionalities are implemented using the Observer pattern. In Line 6, a handler is registered to the observable `time` (defined elsewhere). The handler updates the `position` variable in Line 5. The `position` variable is observable and the handler in Lines 27-28 repaints the view every time `position` changes. The definition of `square2` follows the same principles; the GUI setup in Lines 29-48 is the same as with RP.

As the reader may have noticed, some functionalities, such as time, are defined elsewhere and accessed via `import` (not shown). This is also the case for the reactivity machinery (signals and observers). This choice helps to keep the application code short and is consistent across the RP and the OO version. In the experiment, subjects are shown fully working code, including `import` statements.

### 3.2.2 Comprehension Tasks and Questionnaire

For each reactive program in the study, we ask a question that requires comprehension of the application's behavior. Crucially, questions and alternative choices are formulated in a way that finding the correct answer requires understanding the (whole) reactive logic of the application. An example, taken from the synthetic applications group, is to ask for the sequence of values assumed by a variable that depends on other values in the program once these other values are updated. Answering the question requires to inspect the application to detect functional dependencies among values, i.e., which values are affected by a change and in which order. An example from the interactive group comprises a quick description of the application in the question (a canvas with a box drawn on it) and asks which combination of actions from the user produces a change of the color of the GUI. The correct answer is "crossing two borders of the box while dragging the mouse." An example for the animations group is shown in Figure 4 and refers to the application in Figure 2.

After subjects finished their comprehension tasks, they were presented with a questionnaire that investigates what subjects think of program comprehension with RP and OO. This questionnaire helps us answer RQ4 and is composed of five questions. One of these questions allows free-text answers. We asked the subjects to express their opinion about the differences between RP and OO programs. We later use open-coding [15] to analyze the provided answers. The other four questions elicit subjects' opinions about more specific aspects that – we hypothesize – may contribute to a difference between OO and RP. For example, conciseness or ease of following data flow. Answers to these questions are in the form of a rating (strongly disagree - disagree - neither agree nor disagree - agree - strongly agree). The exact question texts will be provided in Section 5.2.

### 3.2.3 Study Execution

To practically run the experiment, we needed a tool that allows users to complete the tasks in a Web browser that supports fine-grained limitation of task duration and that tracks timing information of the users. Some existing tools for controlled experiments we are aware of are standalone applications (e.g., Biscuit [16] and Purity [17]), which considerably increase the effort of running the experiment as they require a preliminary installation on each machine. We discarded using existing Web platforms for in-browser homework and exams (e.g, WebLab [18]), because they lack fine-grained time management, such as setting an upper bound to each individual task and recording the task completion time – which are important for our experiment (more on this later). We ended up developing WebCompr, a Web application for experiments on program comprehension that is tailored to our specific requirements.

We use comprehension tasks as described in Section 3.2.2, because this methodology provides objective results and scales well (most controlled studies hardly include more than 15 subjects [19], [20] – we have 127 subjects). Yet, the experiment requires a controlled environment with on-site execution and staff supervision to make sure that developers perform the task without external help. Also, we wanted to control the training of the subjects, which prevented us from making the experiment publicly available on the Internet and from publicly calling volunteers for participation.

We briefly discuss the alternatives we evaluated. In the *think aloud* approach, subjects comment on the actions they are performing [19], [20]. A subsequent interview can clarify the motivations behind each action [21]. This approach, however, does not allow to collect objective measures and apply statistical tests. Other studies measure software comprehension by artificially introducing a bug in an application. Then, they measure the time subjects need to fix a bug or perform a modification task [20]. However, bug fixing requires not only to understand the application, but also to write code that solves the problem. Thus, measurements would include *code-writing skills* as an additional factor of the experiment, besides comprehension. This approach is suitable if the factor is balanced for the groups such that it does not influence the main factor we want to observe. In our experiment, however, the code-writing skills of the subjects

The following application draws two squares.
Which of the following sentences is true ?

a - The squares are moving at the same speed
b - The first square is moving at constant speed, the second is moving at increasing speed
c - The squares have a fixed position
d - The first square is fixed, the second is moving

Figure 4: Example Question.

in the RP and in the OO style may be very diverse and there would be no way to separate the influence of this factor from the main factor we want to observe.

We decided to measure both time and correctness of the results. Subjects were encouraged to provide an answer as quickly as possible, but it was also clarified that time becomes relevant only in case the answer is correct. The reason for this design choice is to simulate a realistic coding session in which developers inspect a large project and can spend only a limited amount of time on each class, to keep the analysis of the whole software feasible.

Previous work found that, without constraints, subjects may spend the entire time of the experiment on a single task [22]. To avoid this, we defined an upper bound to the time subjects can spend on each task and fixed the available time to 5 minutes for synthetic applications and to 10 minutes for the animations and interactive applications. The latter are slightly longer and require the inspection of more code. The experiment was designed to require no more than 2 hours in total (preliminary tasks + experiment tasks) to avoid having subjects loose their focus. In practice, our estimations turned out to be rather conservative. None of the subjects required the full amount of time and most subjects finished the questionaire much earlier.

Controlled experiments can be conducted with between-subjects design or within-subject design. In *between-subjects* designs, two versions of the same application are proposed to different subjects. In *within-subjects* designs, each subject is given both versions [23]. Empirical studies in software engineering usually prefer within-subjects designs to balance the effect of the individual-skills factor, i.e., to reduce the variability due to heterogeneous skills levels among the subjects. On the other hand, within-subjects designs introduce learning effects, because subjects can apply the knowledge gained when solving a task with one factor to the solution of the same task with the other factor – a problem solved by between-subjects designs. It has been shown that, if the learning effect is small enough, proper experiment design still allows a within-subject approach [24], [25]. In our case, however, this was not an option. The learning effect is likely to dominate the effect we want to measure, because it is extremely easy for subjects to remember previous findings in the experiment – a known challenge in controlled experiments on program comprehension [22]. Therefore, we chose to design the experiment as a *between-subjects*. Subjects are randomly assigned to the OO group or the RP group as soon as they login in WebCompr for the experiment.

### 3.3 Study Subjects

The subjects of our study are students from a Software Engineering course, held in the fourth year of study in Computer Science. The study was run in two rounds: Round 1 in Fall 2014 (38 subjects) and Round 2 in Spring 2015 (89 subjects). Note that the questionnaire investigating RQ4 was only available to subjects in Round 2.

All of our study subjects have similar academic background. They have been exposed to Scala programming for at least the semester of the course. All subjects learned Java since CS101 in their first semester and used Java as their primary language in the rest of their studies. They learned the Observer pattern since their Java first-year course on programming. Subjects were taught RP for an amount of two lectures (1.5h+1.5h) and were assigned homework (8h+8h estimated) that required them to use RP to develop a reactive application from a given specification.

We used WebCompr to collect information about the subjects, such as their programming experience, and asked them for a self-evaluation of their programming-skills level. Since self-evaluation cannot be considered reliable [26], subjects were also given 18 preliminary tasks, which we used to measure their programming knowledge. We tested a variety of different topics in several very short tasks – the maximum amount of time available for each task was one minute. Questions include concepts of OO programming, such as inheritance and polymorphism, functional programming, e.g., high-order functions and pattern matching, and also RP, including events and signals. A secondary goal of these tasks was to train students to correctly use WebCompr.

Subjects were given multiple answers and a *Don't know* option. The *Don't know* option ensures that subjects do not attempt to guess the correct answer in case they have no experience with the object of the question. Overall, the questions we asked also included advanced topics not necessarily covered by the course. The assumption behind this approach is that good programmers are likely to master *advanced* features of the language. For example, a subject that knows the behaviors of the `zip` function[1] on lists is likely to be a more advanced programmer than a subject that has never heard about it.

### 3.4 Statistical Tests

For all statistical tests in this paper, we chose a $p$-value of less then $0.05$ to reject the respective null-hypothesis. We use the **Shapiro-Wilk test** to check whether a sample comes from a normally-distributed population. When the normality hypothesis cannot be verified (which is the case for our data), we need to use non-parametric statistical tests. We use the **Mann-Whitney U test** to check whether two sets of samples of ordinal variables of unknown distribution come from the same population and **Cliff's delta** to estimate the effect size. We adopt the **Peason $\mathcal{X}^2$ test** and **Fisher's Exact Test**, which is especially robust on small sample sizes, to check whether two sets of samples of categorical values of unknown distribution come from the same population and **Cramer's V** to estimate the effect size. To check correlation between variables, we provide the values of **Spearman's $\rho$ Coefficient** and **Kendall's $\tau$ Coefficient**. In both cases, a positive coefficient indicates a direct correlation, a negative

1. The `zip` function takes two lists and returns a list of pairs. Given the input lists $[l_i, i \in (0..n)]$ and $[r_i, i \in (0..n)]$ `zip` returns the list $[(l_i, r_i), i \in (0..n)]$.

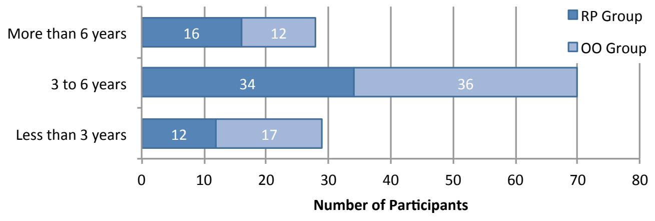

Figure 5: Years of Programming Experience of Both Groups.

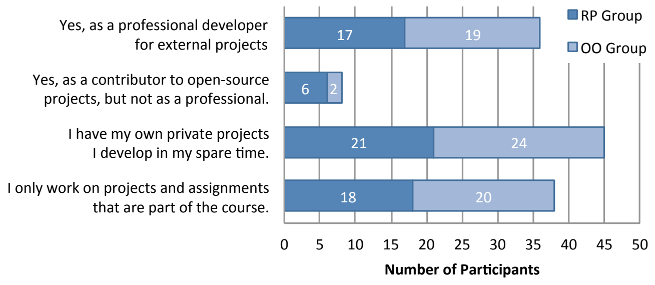

Figure 6: Subjects Experience of Both Groups

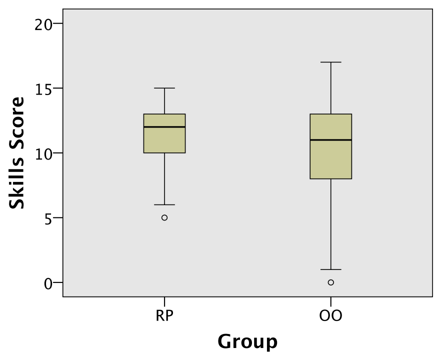

Figure 7: Skills Score of RP Group and OO Group.

coefficient an inverse correlation. None of the tests requires normality of the sample.

## 4 What is the Impact of RP on Program Comprehension?

In this section, we present the main contribution of our work: the analysis of the statistical difference between the subjects in the RP group and the subjects in the OO group. For the analysis, we used the SPSS statistical tool [27].

We asked the subjects questions about their personal background and experience as developers. Figure 5 provides the subjects distribution based on years of programming experience (*PQ1: How long have you been programming?*) Most subjects have a 3 to 6 years experience, which in a number of cases is likely to correspond to the amount of years they have been enrolled in a CS program. The OO group and the RP group exhibit similar profiles.

Figure 6 shows the distribution of the subjects in terms of programming experience (*PQ2: Do you work on any project besides those you are assigned in classes?*). Besides a small group of subjects that work on open-source projects as non-professional contributors, subjects are almost equally distributed among the remaining categories, namely those that work as professional developers on external projects, those that also develop own projects in their spare time, and those that work only on class assignments. Again, the OO group and the RP group are similar.

The subjects' results in the tasks on programming-skills level allowed us to validate the hypothesis of equal distribution among the RP group and the OO group. Validating this assumption is required to make sure that the groups are not unbalanced with respect to the programming-skills level. We calculated a *skills score* for each individual subject that is based on the correct answers achieved in all tasks that tested programming-skills level. We implemented two strategies to calculate this score: a simple strategy that treats all questions the same, regardless of their complexity, as well as a more complex strategy that takes the difficulty of the question into consideration, i.e., the complexity of each question is estimated by considering the number of correct answers for a question over all subjects. The simple strategy rewards one point as a partial score for each correct answer to a question $i$, while the weighted strategy rewards $127/c_i$ points, where $c_i$ is the number of correct answers among all 127 subjects. A wrong answer yields a partial score of 0 in both cases. The overall skill score for an individual subject is the sum of all partial scores of the subject.

Inspection of the data derived with the simple strategy shows similarity between the groups (Figure 7). With the simple strategy, a Mann-Whitney U test on the skills score shows that equality of the groups cannot be rejected ($p = 0.13$, Cliff's delta -0.16). With the weighted strategy, a Mann-Whitney U test on the skills score also leads to a non-significant difference ($p = 0.07$ , Cliff's delta -0.19). The choice of one strategy over the other does not significantly affect the results in the rest of the paper (most notably Section 4.3). Hence, we decided to use the simple strategy in the rest of the paper, because it reflects a straightforward interpretation of the results.

### 4.1 Correctness

First, we analyze the data to provide an answer for RQ1 – whether RP impacts the correctness of program comprehension. As the measure of correct understanding, we consider the *comprehension score*, i.e., the cumulative number of correct answers given by each student in all tasks.

We initially provide an overview of the results using descriptive statistics. The mean score for the OO group and the RP group are $\mu_{(score,OO)} = 7.05$ and $\mu_{(score,RP)} = 8.13$, respectively. Figure 8 shows the comprehension scores obtained by the subjects in the RP group and in the OO group. The plot seems to support the claim that the RP group obtains a better result than the OO group. To check if the difference between the scores of the RP group and the OO group is significant, we formulate the following hypothesis to be tested:

$H_0$: *The scores for the RP group and for the OO group are drawn from the same population.*

The Shapiro-Wilk test tells us that we cannot assume normal distribution for the distribution of the score in either group ($p_{\text{RP}} = 0.001$ and $p_{\text{OO}} = 0.000$). Therefore, we use the Mann-Whitney U test to check our hypothesis. The result allows us to reject $H_0$ with high significance ($p = 0.000$),

| Group | N | Avg. Rank | Rank Sum | p-Value | Cliff's delta |
|---|---|---|---|---|---|
| RP | 62 | 75.83 | 4701.5 | 0.000 | -0.36 |
| OO | 65 | 52.72 | 3426.5 | | |

Table 1: Mann-Whitney U Test for Comprehension Scores.

which allows us to use the rank sum as an indicator for which approach dominates the other. As shown in Table 1, the rank sum for RP is significantly higher than the rank sum for OO.

The correctness results for each application are shown in Figure 9. The RP group outperforms the OO group in all but tasks 8 and 9. To analyze the difference in the correctness results for each application, we use the Pearson $\mathcal{X}^2$ test and Fisher's exact test to disprove $H_0$ for each application (see Table 2). We find that RP performs significantly better than OO for tasks 3-4-5-7 and that RP never performs significantly worse than OO.

*Finding 1:* RP increases the correctness of program comprehension.

It is interesting to further inspect the correctness results to gain some insights about the applications where the RP group did not outperform the OO group. The most striking cases are Task 8 and Task 9, where the average for the OO group is higher than the average for the RP group – even if this difference is not statistically significant.

An insight is that Task 8 is the only task where the GUI logic is mixed with the application logic. In the RP style, this approach requires some comprehension of how an OO GUI library interacts with RP code. The basic idea is that when the elements of the GUI are instantiated, they receive as input signals instead of traditional values. For example, the following code creates a Swing button.

```
1 val rightButton: ReButton = new ReButton(
2 text = Signal { rightClicks() + " times clicked" },
3 background = Color.GREEN)
```

Crucially, the text displayed on the button changes over time, based on the signal expression defined in Line 3. In this example, the GUI elements are directly created to depend on reactive abstractions (i.e., the application logic). The reader can compare this snippet with the example in Figure 2b where the application logic is completely separated from the GUI code. It is possible that the poor performance of the RP group is caused by the lack of knowledge about how RP interoperates with an OO GUI library, rather than by the comprehension of the application logic itself.

In the case of Task 9, the application displays a vertical bar that slides from left to right and is visible only for some time after the mouse clicks. The visibility of the bar is regulated by the `e.switchOnce(s1,s2)` conversion function, which initially returns the signal `s1`, but always returns the signal `s2` after any occurrence of the event `e`. A possible explanation of the poor result of the RP group is that the `switchOnce` function is not known to some subjects, making it hard to deduce the logic of the application. In contrast, the equivalent OO code does not require any specific knowledge.

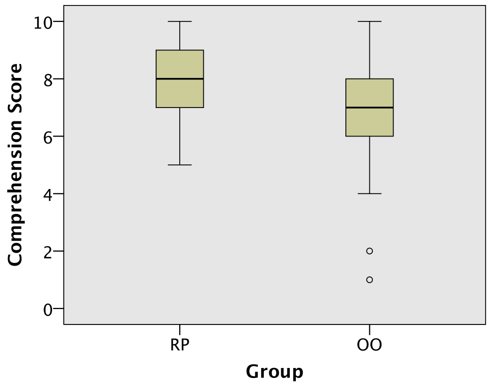

Figure 8: Comprehension Score of RP Group and OO Group.

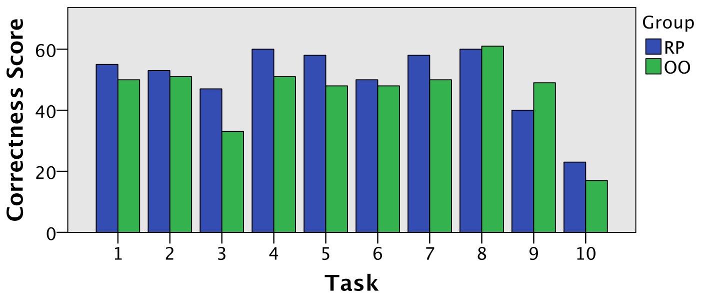

Figure 9: Comprehension Score for Individual Tasks.

## 4.2 Timing

In this section, we investigate the result of the experiment concerning RQ2 – whether comprehension with RP requires more or less time than with OO. This research question is closely related to the results obtained for RQ1. Based on Finding 1, we already know that, in terms of correctness, the RP group *does perform significantly better* than the OO group. However, timing remains an open issue. One may wonder if the previous result can be biased by a significant difference in time to understand an RP program or an OO program. RP can lead to more correct results on average, but may be much more time consuming for developers.

Descriptive statistics suggest that the RP group requires less time to complete each task. Figure 10 shows a box plot comparing the time required by the RP group and the time required by the OO group for each task. The maximum duration differs between tasks (cf. Section 3). To make the measured times comparable, we map the maximum duration for each task to 1000 units and normalize the measured times on this scale. To inspect significance with statistical tests, we formulate the following null hypothesis:

$H_0$: *The time required to complete the RP and the OO tasks are drawn from the same population.*

In contrast to the correctness case, in which binary data required a cumulative score over all the tasks, time data is continuous and we can analyze each task separately. Like before, the underlying distribution is unknown and we perform a Mann-Whitney U test. The results in Table 3 show the available data points for each group (N), the average rank, the rank sum, and the $p$-value for the test. For the

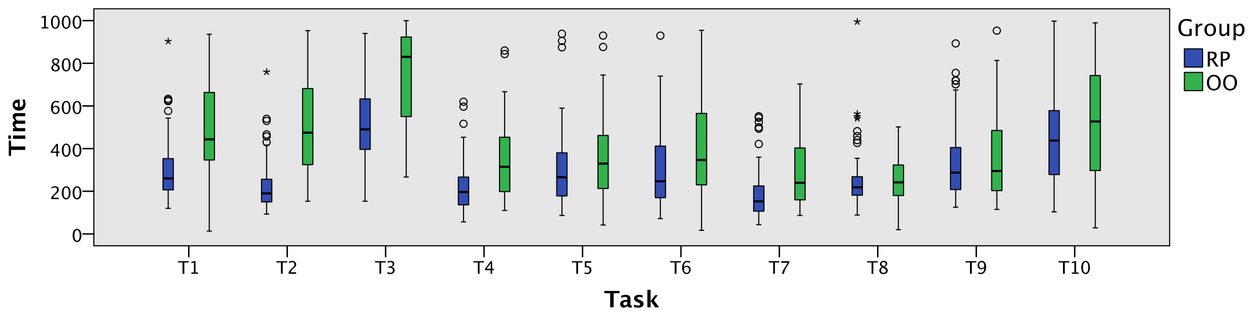

Figure 10: Time Spent on Individual Tasks in RP Group and OO Group.

| | Group | Correct | % | Pearson $\mathcal{X}^2$ | Fisher's Test | Cramer's V |
|---|---|---|---|---|---|---|
| Task 1 | RP | 55 | 88.70 | 0.102 | 0.102 | 0.17 |
| | OO | 50 | 76.90 | | | |
| Task 2 | RP | 53 | 85.50 | 0.361 | 0.361 | 0.09 |
| | OO | 51 | 78.50 | | | |
| Task 3 | RP | 47 | 75.80 | *0.006 | *0.006 | 0.26 |
| | OO | 33 | 50.80 | | | |
| Task 4 | RP | 60 | 98.80 | *0.002 | *0.002 | 0.28 |
| | OO | 51 | 78.50 | | | |
| Task 5 | RP | 58 | 93.50 | *0.004 | *0.004 | 0.27 |
| | OO | 48 | 73.80 | | | |
| Task 6 | RP | 50 | 80.60 | 0.403 | 0.403 | 0.08 |
| | OO | 48 | 73.80 | | | |
| Task 7 | RP | 58 | 93.50 | *0.012 | *0.012 | 0.23 |
| | OO | 50 | 76.90 | | | |
| Task 8 | RP | 60 | 96.80 | 0.680 | 0.680 | 0.07 |
| | OO | 61 | 93.80 | | | |
| Task 9 | RP | 40 | 64.50 | 0.245 | 0.245 | 0.12 |
| | OO | 49 | 75.40 | | | |
| Task 10 | RP | 23 | 37.10 | 0.251 | 0.251 | 0.12 |
| | OO | 17 | 26.20 | | | |

Table 2: Person $\mathcal{X}^2$ Test and Fisher's Exact Test for Correctness. Starred Differences are Significant.

| | Group | N | Avg. Rank | Rank Sum | p-Value | Cliff's delta |
|---|---|---|---|---|---|---|
| Task 1 | RP | 61 | 42.98 | 2621.5 | *0.000 | -0.63 |
| | OO | 65 | 82.76 | 5379.5 | | |
| Task 2 | RP | 62 | 39.19 | 2430 | *0.000 | -0.76 |
| | OO | 64 | 87.05 | 5571 | | |
| Task 3 | RP | 61 | 46.12 | 2813.5 | *0.000 | -0.51 |
| | OO | 62 | 77.62 | 4812.5 | | |
| Task 4 | RP | 62 | 50.35 | 3121.5 | *0.000 | -0.41 |
| | OO | 64 | 76.24 | 4879.5 | | |
| Task 5 | RP | 62 | 58.07 | 3600.5 | 0.076 | -0.18 |
| | OO | 65 | 69.65 | 4527.5 | | |
| Task 6 | RP | 62 | 54.85 | 3401 | *0.006 | -0.28 |
| | OO | 65 | 72.72 | 4727 | | |
| Task 7 | RP | 62 | 50.25 | 3115.5 | *0.000 | -0.42 |
| | OO | 65 | 77.12 | 5012.5 | | |
| Task 8 | RP | 61 | 61.56 | 3755 | 0.565 | -0.06 |
| | OO | 65 | 65.32 | 4246 | | |
| Task 9 | RP | 62 | 63.31 | 3925 | 0.837 | -0.02 |
| | OO | 65 | 64.66 | 4203 | | |
| Task 10 | RP | 62 | 57.77 | 3581.5 | 0.083 | -0.18 |
| | OO | 63 | 69.05 | 4419.5 | | |

Table 3: Mann-Whitney U Test for Answer Times. Starred Differences are Significant.

tasks 1-2-3-4-6-7, $H_0$ can be rejected with high significance. In all those cases, the rank sum indicates that times for the OO group are higher than times for the RP group (Table 3, column “Ranks sum”), i.e., *the RP group is faster*. For the tasks 5-8-9-10, $H_0$ cannot be rejected.

The above finding might only hold for the time it takes a programmer to give the answer *she believes* is correct. Therefore, we test our hypothesis also on the subset of times that subjects needed to provide the *correct answer*. The results of the Mann-Whitney U test are in Table 4. For tasks 1-2-3-4-6-7 we can reject $H_0$ with significance. The rank sum indicates that RP group is faster than the OO group. For the other tasks, $H_0$ cannot be rejected. Thus, considering only correct answers still leads to the same conclusion.

*Finding 2:* Comprehending programs in the RP style *does not* require more time than comprehending their OO equivalent.

### 4.3 Programming-skills Level

In this section, we inspect the experimental data to answer RQ3 – whether comprehending RP applications requires a more advanced programming-skills level. To this end, we correlate the programming-skills level we measured for each subject with the correctness results in the experiment. Figure 11 shows a scatter plot of the skills score and the comprehension score in the tasks for both the RP group and the OO group. There are less data points than subjects, because some data points are identical for different subjects. For those cases, the size of the dots reflects the number of subjects that fall in this category. The figure suggests that in the OO group, skill correlates with correctness, i.e., subjects with low programming-skills level perform poorly on the tasks, while subjects with advanced programming-skills level perform better. In contrast, subjects in the RP group reach high scores, independently of their programming-skills level. In other terms, the figure suggests an absence of correlation for the RP group and a positive correlation for the OO group. This intuition is further investigated in the following. We estimate the significance of the correlation between subjects' score and skills level by testing the following hypotheses:

$H_0$: *There is no correlation between the score of the subjects in the RP (respectively, OO) group in the experiment and their measured programming-skills level.*

We use Kendall's $\tau$ coefficient and Spearman's $\rho$ coefficient to test for a correlation between skills level and score. Table 5 shows the correlation coefficient $r$ and the $p$-value for each case. For the RP group, we detected no significant

| | Group | N | Avg. Rank | Rank Sum | p-Value | Cliff's delta |
|---|---|---|---|---|---|---|
| Task 1 | RP | 54 | 34.11 | 1842 | *0.000 | -0.74 |
| | OO | 50 | 72.36 | 3618 | | |
| Task 2 | RP | 53 | 33.08 | 1753.5 | *0.000 | -0.76 |
| | OO | 50 | 72.05 | 3602.5 | | |
| Task 3 | RP | 46 | 30.11 | 1385 | *0.000 | -0.56 |
| | OO | 30 | 51.37 | 1541 | | |
| Task 4 | RP | 60 | 45.5 | 2730 | *0.000 | -0.40 |
| | OO | 50 | 67.5 | 3375 | | |
| Task 5 | RP | 58 | 48.77 | 2828.5 | 0.082 | -0.18 |
| | OO | 48 | 59.22 | 2842.5 | | |
| Task 6 | RP | 50 | 43.34 | 2167 | *0.028 | -0.26 |
| | OO | 48 | 55.92 | 2684 | | |
| Task 7 | RP | 58 | 44.7 | 2592.5 | *0.000 | -0.39 |
| | OO | 50 | 65.87 | 3293.5 | | |
| Task 8 | RP | 59 | 57.46 | 3390 | 0.348 | -0.10 |
| | OO | 61 | 63.44 | 3870 | | |
| Task 9 | RP | 40 | 48.19 | 1927.5 | 0.296 | -0.13 |
| | OO | 49 | 42.4 | 2077.5 | | |
| Task 10 | RP | 23 | 17.26 | 397 | 0.073 | -0.34 |
| | OO | 16 | 23.94 | 383 | | |

Table 4: Mann-Whitney U Test for Correct-Answers Times. Starred Differences are Significant.

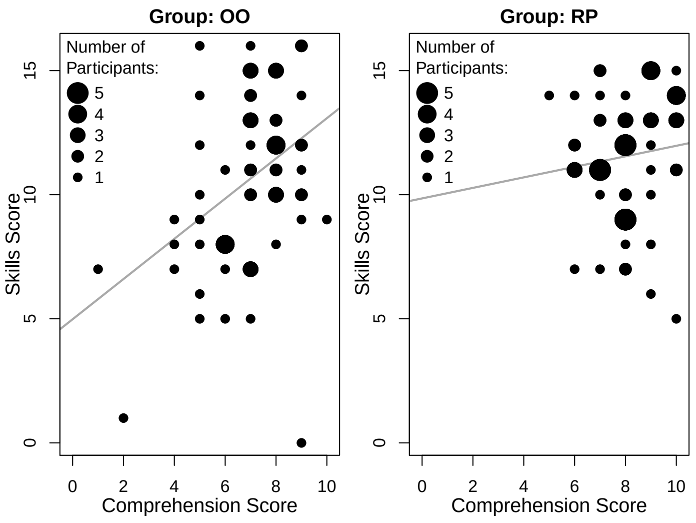

Figure 11: Skills Score vs. Comprehension Score.

correlation, i.e., $H_0$ cannot be rejected. For the OO group, however, both tests are statistically significant ($p = 0.003$, $p = 0.002$) and $H_0$ can be rejected. We conclude that in the OO group, there is evidence of a positive correlation ($r = 0.280$, $r = 0.383$) between subjects' scores and skills level.

*Finding 3:* In contrast to OO, where score results are correlated to programming-skills level, with RP, a higher programming-skills level is not needed to understand reactive applications. This suggests that, with RP, reactive applications are easier to understand for non-expert programmers.

Note that the correlations above and their significance do no change when considering the simple strategy or the weighted strategy for evaluating subjects skills levels.

| | | Comprehension-Skills Correlation | |
|---|---|---|---|
| | | RP Group | OO Group |
| Kendall | coefficient $r$ | 0.130 | 0.280* |
| | $p$-value | 0.190 | 0.003 |
| Spearman | coefficient $r$ | 0.182 | 0.383* |
| | $p$-value | 0.157 | 0.002 |

Table 5: Correlation of Comprehension Score and Skills Score with Kendall's $\tau$ and Spearman's $\rho$. Starred Correlations are Significant.

## 5 RP VS OO: PERCEIVED DIFFERENCES

The results of our empirical study show that RP outperforms OO in terms of program comprehension and, hence, should encourage researchers to further explore RP. However, the evidence presented in Section 4 does not explain the root causes of the difference between RP and OO with respect to program comprehension. In this section, we attempt to provide an explanation of our findings – hence answer RQ4 – supported by our experience with RP and the additional questionnaire data collected in Round 2 of our study.

### 5.1 Informal Observations

In Round 1, we collected informal feedback from the experiment subjects in a non-systematic fashion by talking to them at the end of the experiment. Using different wording, subjects reported that, with OO versions, *one has to follow the flow of the whole application* to infer the reactive behavior. On the other hand, with RP, *the flow associated to the reactive behavior is explicit* (in practice, a signal is defined together with its expression). Based on this feedback and on our experience, we believe that the main effect we observed comes from RP making it easier to reason about data flow, i.e., points (*i*) and (*ii*) from Section 2. This conjecture is supported by previous findings showing that data and control flow play a fundamental role in the way programmers develop a mental representation of programs [28] and that developers understand programs exactly by building such mental models [29].

Data flow determines whether a variable depends on another, a fundamental aspect of comprehending reactive applications. Detecting dependencies among variables is also a reachability question, because it requires checking whether a change to a variable propagates to the other. Previous research [30] noticed that reachability questions are (i) extremely common for developers, (ii) hard to answer correctly, and (iii) time-consuming. These results support the conjecture that RP improves correctness in the comprehension of software by making data flow easier to understand.

### 5.2 Questionnaire

The subjects involved in Round 2 answered a questionnaire to explore the perception that developers have of program comprehension with RP and OO. The questionnaire allows us to perform both quantitative and qualitative analysis of the data.

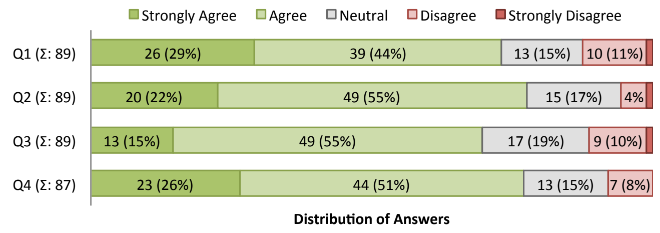

Figure 12: Answers of the Subjects to the Four Questions

### 5.2.1 Quantitative Analysis

Our quantitative analysis is based on the four rating questions presented to subjects in the questionnaire. The answers to these four questions are summarized in Figure 12.

The first of these questions directly asks the subjects for their opinion about the main research question of this study:

*Q1: A program implemented using Reactive Programming is easier to comprehend than an equivalent program based on the Observer Pattern.*

The answers to *Q1* show that, of the subjects who answered the question, 65 (73%) strongly agree or agree that RP is easier to comprehend compared to an equivalent program that implements the Observer pattern. Eleven subjects (12%) disagree or strongly disagree with this statement, while 13 subjects (15%) are neutral (neither agree nor disagree). Recall that our subjects received only 3 hours of RP training (Section 3.3).

*Observation 1:* Even developers with minimal RP experience believe that RP is easier to comprehend.

The next question collects evidence for the hypothesis that data and control flow are easier to follow in RP – a hypothesis suggested by the informal feedback collected after Round 1.

*Q2: In Reactive Programming, compared to the Observer design pattern, data and control flow are easier to follow.*

Answers to *Q2* show that 69 subjects (77%) strongly agree or agree that following data and control flow in RP is easier than in an Observer implementation, which confirms the conjecture formulated after Round 1. Only 4 subjects (4%) disagree and only a single subject strongly disagrees. 15 subjects (17%) neither agree nor disagree.

*Observation 2:* Developers believe that following data and control flow is easier in RP.

The next two questions attempt to more directly investigate the reason for better comprehension in RP. The first question addresses the matter of conciseness of RP programs. In previous work, we showed that RP programs are shorter than their OO counterpart [5]. We hypothesize that this aspect affects program comprehension. Since conciseness does not necessarily make comprehension easier, we asked the subjects to evaluate how a specific aspect of RP languages impacts comprehension.

*Q3: In Reactive Programming, compared to the Observer design pattern, programs are easier to understand, because they are shorter.*

The answers to *Q3* show that 62 subjects (70%) believe that conciseness of RP has an impact on program comprehension. Ten subjects (11%) disagree or strongly disagree with the statement, 17 subjects (19%) neither agree nor disagree.

*Observation 3:* Developers believe that conciseness makes comprehension of RP easier.

The next question investigates the role of conversion functions. RP languages provide a rich set of operators that combine reactive values and convert between signals and events [4]. For example, in Figure 2, the `changed` operator in Line 14 converts from a signal to an event that fires when the signal changes its value. RP-language designers hypothesize that these operators express programmers' intentions and make code more comprehensible [10]. Note that conversion functions have been explained to subjects in the class so that "conversion function" is known terminology for them.

*Q4: In Reactive Programming, compared to the Observer design pattern, programs are easier to understand, because conversion functions express programmers' intentions more directly.*

The answers to *Q4* show that 77 subjects (77%) strongly agree or agree that conversion functions have a positive impact on comprehension of RP programs. Only 7 subjects (8%) disagree or strongly disagree to this, while 13 subjects (15%) neither agree nor disagree. A comparison with *Q3* indicates that code length has an important role in the comprehension of RP programs, but not as important as conversion functions.

*Observation 4:* Developers believe that conversion functions make comprehension of RP easier.

### 5.2.2 Qualitative Analysis

To further understand what factors impact program comprehension and what characteristics of RP or OO affect program comprehension, we used an open question, *QOp*, to allow subjects to freely express their opinions. Note that in the questionnaire, *QOp* is actually located after *Q1* to avoid that the answer is influenced by the hypotheses presented in *Q2*, *Q3*, and *Q4*.

*QOp: Considering your choice in question Q1, how would you motivate your answer?*

We now discuss the answers we received for *QOp*, first in terms of arguments for RP and then in terms of arguments against RP. For each argument we find, we provide some of the subjects' quotes that discuss the problem. For each quote, we also provide the subject's answer to Q1.

#### Points in Favor of RP

To identify points in favor of RP, we performed an open-coding session on the answers to *QOp* that we received from the 65 subjects (73%) that agreed or strongly agreed to *Q1*. As a result, we found seven categories of reasons for finding RP more comprehensible. The reported percentages in the following section are computed with respect to the 65 answers that are considered here.

**Reduced Boilerplate Code.** A total of 28 subjects (43%) either explicitly mentioned RP having less boilerplate code or used words such as "less code-overhead". The following subject's comment provides some intuition of what subjects mean when using the term boilerplate code:

*"Programmers can focus on the implementation rather on additional boilerplate code needed to get the reactive behavior of the observer pattern."* (Strongly Agree, Id 89)

**Better Readability.** Twelve subjects (18%) indicated that they found programs written in RP easier to understand, because they provided better readability. It is interesting to note that most of these subjects also mentioned reduced boilerplate code, either as a separate point in their answer or as a reason to why they found the programs easier to read. Subject 56 makes this explicit by saying

*"RP is easier to understand because it deals with concise code. It deals with events so code is more clear to understand."* (Agree, Id 56)

**Automatic Consistency of Reactive Values.** Ten subjects (15%) mentioned that they liked the fact that RP automatically tracks dependencies among signals, as well as automatically updates their values. Subjects liked such support since it ensured the consistency of the reactive values, without interference from the programmer. For example, one subject comments that she finds RP easier to understand, because,

*"the annoying notify mechanism is completely dropped [... and] the Signals are automatically updated."* (Agree, Id 25)

**Shorter Code.** Seven subjects (11%) mentioned shorter code as one of the advantages of RP. This aspect clearly relates to *reduced boilerplate code* mentioned previously. However, shorter code only refers to the amount of code required to express a functionality, while boilerplate code refers to repeated code that clutters the application, obfuscating its behaviour.

*"Instead of many cluttered classes in the Observer Pattern, Reactive Programming enables a short, readable syntax."* (Strongly Agree, Id 38)

**Declarative Nature.** Five subjects (8%) describe RP's declarative nature as as one of the reasons that makes the code easier to understand. Three of these explicitly mention the term *declarative*, while the other two explain characteristics of declarative programming that helped them understand the code, as the following quote illustrates:

*"I think it is easier to reconstruct the path of signals and events, than to follow the notifications to the different observers and to evaluate the side-effects of the methods of the observers."* (Agree, Id 23)

**Ease of Composition.** Four subjects (6%) list the ease of composition of different events in RP as one of the factors that leads to better comprehension. This suggests that using a shorter, more intuitive syntax for composing events helps developers understand the logic involved more easily.

*"[RP is] more composable through declarative events (`||`, `&&` , ...)"* (Strongly agree, Id 87)

For example, in REScala, an event `e3` that occurs whenever one of the source events `e1` and `e2` occur can be defined as `e3 = e1 || e2`. To express the same functionality with the Observer design pattern, instead, one should call `addObserver` on each source, binding the same observer. While the functionality is the same, the `||` operator makes the programmer's intention explicit.

**Separation of Concerns.** Three subjects (5%) mention separation of concerns as one of the advantages of RP. With RP, the *change propagation* concern is separated from the application logic. In contrast, when using the Observer design pattern, change propagation is tangled with the rest of the application.

*"Observer pattern does not provide separation of concerns."* (Agree, Id 52)

### Points Against RP

While the above points provide arguments for why reactive applications in RP are more comprehensible than OO, we are also interested in the counter-arguments provided by some subjects. In the following, we look only at the opinions of the twenty-four subjects who answered *neither agree nor disagree*, *disagree*, or *strongly disagree* to Q1. This is crucial to guide future research on reactive applications.

**Learning Curve.** The results presented in Section 4 suggest that the learning curve required to increase software comprehension is quite small. Experience with the RP paradigm compared to the OO paradigm is a recurring argument in the answers of the subjects that do not fully agree with the statement in *Q1* (7 subjects, 29%). In some cases, subjects mention that they consider RP superior, but harder to master.

*"[...] if the person achieves the considerable skills to understand the programming constructs based on RP, then it would be the quickest & fastest way of programming."* (Neither agree nor disagree, Id 61)

*"We need the RP style of thinking and get used to it then only it is easier."* (Neither agree nor disagree, Id 123)

The subjects in the experiment have significant background in OO programming – mostly in Java. Subject 42 notes that previous experience in another programming paradigm can influence the learning curve of RP.

*"Readability is good in [the] normal Observer Pattern, Scala or REScala is cumbersome and the code is not easily understood. [...] I have worked for 5 years on Java and I am not able to adapt to the new programming paradigm [...]."* (Disagree, Id 42)

This subject quote also seems to suggest that the readability problems are related to functional programming in general, i.e., to general Scala features rather than REScala specific features.

**Level of Abstraction.** An aspect that may negatively affect the comprehension of RP programs is the higher level of abstraction the RP paradigm promotes (2 subjects, 8%). One of the effects of abstraction is that the details about change propagation are not directly visible to the developer. In the case of RP, programmers need to *trust* the programming runtime for propagation concerns.

*"I can't follow the Codeflow in RP. I want to know, what method is called when."* (Strongly disagree, Id 83)

Instead, with the Observer design pattern, developers can directly see the method calls that propagate the change. This aspect is expressed in the following comment:

> *"Using the Observer pattern is really clear, although it introduces a bit of boilerplate code. When you get around how REScala works, you don't think about the details anymore and it becomes easier. But [you have] to get to this point."* (Neither agree nor disagree, Id 43)

Again, a crucial aspect of the learning curve of RP seems to be mastering abstraction.

**Relation with Functional Programming.** RP has a functional flavor in that it is based on functional composition of signals, which abstract over state. As such, RP inherits advantages (e.g., conciseness) and drawbacks of functional programming (2 subjects, 8%). In addition, subjects with Java background perceive functional programming as harder than OO programming.

> *"The observer pattern is well established and everybody knows about it and how to use it. Reactive programming is less known, but introduces less boilerplate and shorter code. Given that RP code is not always easy to understand (the same holds for functional programming also) without extensive comments, they are both equally easy to understand."* (Neither agree nor disagree, Id 84)

A well-known issue in functional programming is the proliferation of combinators that address the issues that arise in the development process. This issue is mentioned in the answer of subject 121 who warns against the difficulty of mastering too many combinators. As we already discussed, our results for the correctness of Task 9, where RP does not outperform OO, may be explained by the presence of a complex combinator (Section 4.1).

> *"It depends on [the] abstraction of the RP. E.g. If there are functions that encapsulate more than one functionality, it may become complex to understand the result of the function. There could be functions that do the nearly the same [...] By adding more of functions like these it may become hard to keep an overview. [...]"* (Agree, Id 121)

Currently, REScala supports 20 combinators. Yet, this problem is even more exemplified by the Rx reactive framework, which lists more than 70 operators (almost 400, if we consider their variants) [31]. While many operators are mostly highly-specialized versions of other, more-general operators, we believe that the breadth of such an API can pose a serious challenge for a programmer that is interested in adopting RP. We believe that this aspect should drive the design of future RP languages, suggesting to keep the reactive API small, by favoring composablility over specialization of operators.

### 5.2.3 Relating Quantitative and Qualitative Analysis

It is interesting to compare the qualitative data we obtained from the free-text answers to the Observations 2–4 from the quantitative data.

The points for RP in the *automatic consistency of reactive values* and *separation of concerns* categories support Observation 2. These comments suggest that participants find it easier to understand data and control flow in RP or at least appreciate the fact that some of this logic is automatically handled for them. On the other hand, in the points against RP, one participant (Id 83) stated that she dislikes the hidden change propagation, because it forces her to trust the program without seeing an explicit, trackable flow. While only two participants had such concerns, it is worth investigating whether they can be addressed, for example, by respective debugging concepts. We started addressing these issues in a recent line of work on dedicated debugging for reactive applications [32].

Observation 3 seems to be widely supported by the qualitative data we gathered. Thirty-five out of the 65 subjects (54%) who agreed or strongly agreed to *Q1* spontaneously mention reduced boilerplate or shorter code as a major motivation for better comprehension of RP programs.

Interestingly, while Observation 4 is supported by 77 subjects (77%), who strongly agree that conversion functions make comprehension of RP easier, only one participant mentioned conversion functions, among other points, in the free-text comments. Given that *QOp* is presented to participants before *Q4*, it might be that conversion functions are simply not what participants think about first when considering the comprehension of RP programs, but something they acknowledge as important, if prompted. This is in contrast to, for example, spontaneously stating conciseness as a main advantage of RP programs.

## 6 Outlook

The results in the study suggest interesting directions for future work in different areas.

**Interpretation of the results.** This study provides a first attempt to assess the impact of the RP paradigm on software comprehension. Further systematic investigation of this aspect should proceed along different lines. More evidence should be collected on how developers reason about RP programs. Exploratory studies, e.g., using either the think-aloud approach or interviews [20], [33], could help to understand the causes of the issues developers face with RP. Such studies could also reveal which aspects of RP languages' design are responsible for the effects observed in the study. We hypothesize that syntactic binding of signals inside signal expressions play a crucial role, as it makes dependencies among reactive values explicit and easy to detect by inspecting only the portion of code with the signal's definition. Recently, an interesting line of work has applied eye-tracking techniques to understand how developers inspect, debug, and comprehend source code [34], [35], [36], [37], [38]. This kind of analysis seems to be promising to investigate with high precision the effect of RP abstractions on programmer activity for code comprehension.

Previous work studied the mental processes that are activated during programming tasks to infer cognitive models for the comprehension process. In the *top-down model* by Soloway and Ehrlich [39], developers who already possess domain knowledge (*knowledge base*) understand an application by mapping programming structures to such existing (high-level) knowledge. For example, if a programmer is an expert on operating systems, she expects the presence of a scheduling algorithm and she looks for the queue that implements it. In the *program model* proposed by Pennington [29], a programmer understands an application by building a representation of what the application is doing. Such a representation consists of a control-flow abstraction. For example, the programmer deduces that a code section implements a linked list. In the *bottom-up model* (or *situation* *model*) by Letovsky [40], the program model is used to create data-flow and functional abstractions. For example, a linked list can be recognised as a queue for process scheduling.

An open question is how the difference between OO and RP can be interpreted in the light of existing cognitive models. Mayrhauser and Vans [41] show that the above models are active at the same time, but that the program model and the bottom-up model are predominant for small applications – which is the case in our experiment. An argument in favor of the activation of the top-down model in our experiment is that subjects have domain knowledge about GUIs and animations – they have seen examples of these applications during the lecture. Also, their knowledge base is enhanced by the context provided in each question (e.g., "The following application draws two squares"). Despite these arguments, we do not believe that the top-down models plays a primary role in our experiment. The questions in the tasks focus on details that can be inferred only with a fine-grained inspection of the code (e.g., "The first square is moving at constant speed, the second is moving at increasing speed"). Hence, we hypothesize that the program model and the bottom-up model are the ones that are predominantly involved in the comprehension process triggered by our experiment. In the light of these considerations, a possible interpretation of our results is that RP enhances comprehension because (1) it significantly reduces the effort to understand control flow (i.e., the goal of the program process) and (2) it makes data flow explicit, simplifying the bottom-up process. However, investigating in-depth how cognitive models are affected by the difference between RP and OO is beyond the scope of this paper and requires further study.

**Language Design.** A language design that supports RP abstractions has been embraced by widespread programming languages, such as Scala, with the Akka Scala framework introducing support for event streams, and Java 8, which provides the `Stream` interface for collections. Our findings suggest that this is the right direction. Yet, keeping the surface of the reactive API small is an important issue. We believe that future RP-language designs should focus on identifying a small set of fundamental abstractions and combinators thereof and express the others as their composition. This effort is essential to keep comprehension of RP code accessible also for non-experts (Section 5.2).

An open issue in RP-languages design is whether both signals and events are needed. Research RP languages, such as FrTime [3], Flapjax [4], Scala.react [6], and REScala [5], foster a design with both signals and events. On the other hand, more practical languages, such as the .NET reactive extensions and Bacon.js, adopt a simpler model that only supports event streams. We believe that future experiments should explicitly address the concern of wether the presence of both signals and events make it harder to master an RP language and study the effect on comprehension of a design that supports both events and signals compared to a design with events only.

**Tool support.** Understanding how programmers work is fundamental to provide better tool support, especially in IDEs [42], a research area which is vastly unexplored for RP. Using the dependency graph as the reference model to reason about reactive programs is a promising research direction. Students involved in projects on RP at our university independently developed small applications to visualize the dependency graph. These systems adopt various technologies (e.g., Graphviz, Flash) and offer different levels of refinement, but all of them focus on displaying the evolution of the graph over time. Also, practitioners developed similar representations [43], [44], [45]. Conceptually, the structure of a program is similar in RP and OO as in both cases it can be effectively represented as a graph. Interestingly, one of our subjects followed the same line of argumentation in her answer to question QOp:

> *If you use the Observer Pattern, you have to keep track of the observers to see who gets notified and when. Using reactive programming you have to keep track of how the signals depend on each other. In a complex program, the best way [...] to display this is to use graphs. So neither version has an advantage, as the graphs will look more or less the same.* (Neither agree or disagree, Id 22)

This observation can be especially interesting for implementing tools (e.g., debuggers) that support RP and OO abstractions in the same application. Visualizing the dependency graph for debugging purposes has been explored by the Event Flow Debugger – the debugging tool of Microsoft StreamInsight CEP system [46] – and by Gedik et al. [47] for the Spade IBM stream-processing language. We also investigated dedicated debugging for reactive applications [32]. However, this approach has never been attempted for treating RP and OO abstractions uniformly.

## 7 Threats to Validity

In this section, we discuss factors that may affect the validity of our results and the countermeasures we adopted to reduce such risks.

**Construct Validity.** Threats to *construct validity* refer to the extent to which the experiment does actually measure what the theory says it does.

Our approach to measure program comprehension requires careful formulation of questions and candidate answers. Questions that are too specific may not require a significant comprehension of the program. For example, the correct answer *“3 clicks”* for the question *“How many clicks are needed to activate functionality X”* can be found (with low confidence, admittedly) just by spotting a line such as `if(numberOfClicks == 3)[...]` in the program code. We tackled this problem in two ways. First, we formulated questions and candidate answers in a way that requires a broad comprehension of the reactive behavior of the program. The questions we formulated are basically equivalent to “what does this reactive program do?”, as discussed in Section 3 and shown in Figure 3. We ensured that subjects could not simply spot the correct answer by “pattern matching” over the code. Second, we performed renamings to change too-meaningful names into more neutral ones. For example, in the application in Figure 2a, the variables `position` in Line 5 and `position` in Line 13 were originally named `constantSpeedPosition` and `increasingSpeedPosition`, which would have immediately provided an answer to the question in Figure 4.

Another issue concerns the use of the Web-based Web-Compr application to complete the tasks. Such a platform could be unintuitive for some subjects, which may potentially affect the results, especially timing. To mitigate this effect, we presented the interface of WebCompr in detail before the experiment started and showed example screenshots to give the subjects a clear feeling of what to expect. Also, the experiment started with preliminary questions about the background of the subjects, for them to become familiar with the platform while answering these questions that are unrelated to program comprehension.

Finally, a possible threat is that showing code in WebCompr might not reflect the comprehension process that developers follow in practice. For example, many developers use debuggers as a means to understand how an application behaves. Similar considerations apply to other functionalities supported by IDEs, but unavailable in WebCompr, e.g., API-doc popups, the "show definition" function for variables, a navigation view, or advanced search tools. Allowing programmers to inspect code with a fully-fledged IDE, however, would introduce a major threat to internal validity – discussed hereafter.

**Internal Validity.** Threats to *internal validity* relate to factors – other than independent variables under control – that can affect dependent variables, i.e., influence the results.

A first concern is about a possible difference between the skills of the OO group and the RP group. As discussed in Section 4 there is no significant difference, even though the $p$-values are relatively small ($p = 0.13$ or $p = 0.07$, depending on the strategy). It should be noted that the effect size is small too (Cliff's delta is $-0.16$ or $-0.19$, respectively).

While IDEs play a relevant role in programmer activity, the influence of the IDE on different programming paradigms is unclear and should be the subject of a different study. Other studies facing the same problem [24], [48], [49] adopted a minimal programming environment to minimize the effect of the IDE. This problem is especially relevant for program comprehension, since advanced search tools and outlines of the program structure may significantly speed up comprehension of the application behavior. It must also be considered that there is currently no dedicated tool support for RP, which would unfairly disadvantage this paradigm in a comparison involving an IDE. Our approach based on the WebCompr platform has the advantage that the impact of tool support is not a factor. The text search of the browser and syntax highlighting are the only *help* subjects receive.

**External Validity.** Threats to *external validity* relate to the generalizability of our findings.

A first issue concerns the training subjects received. The comparison of two programming techniques is definitely influenced by the skills level of the subjects in each. In our case, training on RP was minimal (cf. Section 3). In summary, the experiment is not biased towards RP, yet, it leads to observable differences in favor of RP – suggesting that the effect may be even higher if subjects had comparable experience in OO and RP. For our study, we chose Scala as a representative of OO languages and REScala as a representative of RP languages. It is possible that our findings are specific to those languages and do not generalize to other languages. We chose Scala, because all subjects were exposed to the language throughout the course. We chose REScala, because this way the RP extensions are the only difference between both languages. Moreover, we conduct our study in an advanced Software Engineering course, which makes us confident that students understand the abstract concepts behind the specific syntax.

Another issue concerns the types and sizes of the applications we adopted in the experiment. Regarding the type, the applications we selected are representative for typical domains of RP. We argue that these are also representative for a wide class of reactive applications. Also, synthetic applications capture the issues of reactivity in general and are not bound to a specific domain. Regarding the size, small-size applications are necessary to keep the experiment feasible. However, for what reactivity concerns, we tried to reflect the structure of bigger applications. For example, in Figure 2, the signal in Line 10 could be removed, collapsing its signal expression with the one in Line 13. Similar considerations apply for the OO counterpart. The correctness of such a design is disputable, yet, it is functionally equivalent to the presented solution. More importantly, however, intermediate observables are likely to appear in larger applications, because programmers are more likely to use these intermediate values in multiple places and, therefore, introduce them in the first place. To make sure that both the RP version and the OO version of the applications are representative of the respective style, we asked the members of our research group to review our code and modified it according to the feedback we obtained.

The subjects of our experiment are students. Although using students for empirical studies is common practice, this can affect the result of the experiment [50]. Professional developers may have more expertise in a programming technique after applying it on a daily basis for years. Yet, recent work observed no significant difference between professional developers and students [51]. As a final consideration regarding subjects' skills, Kleinschmager and Hanenberg, in a preliminary study [26], presented a negative result on using pretests as a valid criteria for measuring programming-skills level. However, we use 18 preliminary tasks to increase statistical validity, while Kleinschmager and Hanenberg use a single task to predict the outcome of their other 14 tasks, which significantly increases our confidence on the methodology we adopt.

Finally, while this work aims at evaluating the effect of a *programming paradigm* (RP), the study is limited to one *specific language* (REScala) that implements this paradigm. Each language has semantic features and specific syntax that may significantly influence program comprehension. In the design space of RP languages, REScala is more similar to research languages, such as FrTime, Scala.react, and Flapjax, than to languages popular among practitioners, such as *Rx* and *BaconJS*. We now analyse the main differences between REScala and Rx – chosen because of its popularity – and discuss how they may impact the results of our study.

The most obvious difference between Rx and REScala is that REScala distinguishes between signals and events, while Rx supports only events. This leads to a different way of thinking about the application logic as well as syntactic differences. For example, to indicate a dependency in REScala, a signal expression like `s2 = Signal{ s1.toString }` would be used. In contrast, a statement like `e2 = e1.map(_.toString)` is used in Rx. From a semantic standpoint, lack of signals means that the language does not distinguish between *continuous* (signals) and *discrete* (events) time-changing values [3]. Such distinction may help developers who see an application for the first time to get a better intuition of the expected use of a reactive value. Signals are usually adopted to express the case where the important aspect is the value that the variable *holds*, for example the mouse position. Events are rather used for cases where a *change* is important, for example a mouse click. On the one hand, this distinction may help developers in the comprehension process, on the other hand, a language in which events are the only reactive abstraction may be simpler to master.

In general, it is hard to estimate the impact of these differences – hence, whether our results generalise to Rx and similar languages. Our qualitative analysis (Section 5) suggests that reduced boilerplate code, better readability, automatic consistency, shorter code, declarative nature, ease of composition, and separation of concerns are among the causes of the advantages of RP against OO in the experiment. We speculate that these advantages also apply to Rx and similar languages given that their design is close to REScala's. Similar to REScala, other RP languages also make dependencies explicit and, hence, improve comprehensibility, even if their syntax differs (e.g., the `Signal` vs. `map` case described above). Other RP languages also enforce automatic consistency among event values, because event propagation is automatically handled by the runtime and are additionally declarative and easier to compose thanks to composition operators (names of operators may be of course different than REScala). We believe that further experiment should compare the different features of RP languages. This would help in determining which features are the best to have in an RP language.

## 8 Related Work

We organize related work as follows. First, we outline recent research on RP. Second, we provide an overview of existing empirical studies on program comprehension. Last, we enlarge the scope to other studies and controlled experiments on programming techniques. We are not aware of any controlled experiment on RP.

**Reactive Programming.** Reactive extensions of existing languages include FrTime [3] (Scheme), FlapJax [4] (Javascript), Scala.react [6] (Scala), and REScala [5] (Scala). An overview of the available solutions and of the advanced features each language adds to those presented in Section 2 can be found in the survey by Bainomugisha *et al.* [11].

Current reactive languages have been influenced by several approaches, often from quite different domains. Functional-reactive programming was proposed in the strictly-functional language Haskell and applied to graphic animations [1], [2], robotics [52], and sensor networks [53]. Graphical libraries, such as Garnet and Amulet (Lisp) [54], apply concepts of dataflow programming to relieve the user from manually updating dependent values. Researchers also investigated languages in which developers can specify bidirectional constraints [55]. In case not all constraints can be satisfied, a priority rank is applied. More recent research on constraints languages focuses on the integration of object-oriented programming and declarative constraint solving to provide the advantages of both paradigms [56], [57].

Finally, current research directions in RP include extension to the distributed setting [58], [59], integration with the OO paradigm [60], and advanced type systems to provide, e.g., bounded memory-consumption guarantees [61].

**Studies on Program Comprehension.** An introduction to the issues of designing studies on program comprehension (e.g., methodology, experiment designs, threats to validity) is in the work by Di Penta *et al.* [50]. Storey [62] surveys the *theories* that have been formulated for program comprehension and their implications on tool design.

Pennington [29] shows that different language designs influence whether control flow or data flow questions are easier to understand for programmers. Ramalingam and Wiedenbeck [63] organize an empirical study on program comprehension in the OO and in the imperative style. They find that novice developers achieve better program comprehension in the imperative style. However, in contrast to the mental representation of imperative programs, which focuses on program-level knowledge, the mental representation of OO programs focuses on domain-level knowledge. Corritore and Wiedenbeck substantially confirm these results [64].

Other researchers organized controlled experiments to investigate the effect of tools on program comprehension. Quante [22] studies the impact of dynamic object graphs on program comprehension. Wettel *et al.* [65] show that the CodeCity 3D software visualization tools significantly increase task correctness and reduce task completion time. Similarly, Cornelissen *et al.* [66] evaluate the enhancement of program comprehension by visualizing execution traces with tools.

**Studies on Programming Techniques.** Pankratius *et al.* [67] organize an empirical study to compare the use of Scala and Java to develop multicore software. Contrary to a common belief, they find that Scala does neither reduce development effort nor debugging effort. Prechelt presents an empirical study that compares seven programming languages along directions that include working time to complete a task and productivity [68]. Among other results, Prechelt observes that the language choice may cut the writing time for a program by half.

Hanenberg organized a series of controlled experiments to evaluate the effect of types in programming languages, focusing on various aspects. These experiments find no positive effects of static type systems on development time [24] and show that similar uncertainties hold for the influence of static type systems on the usability of undocumented software [25]. Also, the experiments show that generic types improve documentation, do not significantly change development time, and reduce extensibility [49]. To the best of our knowledge, systematically studying the effect of RP and, more broadly, of data flow languages is largely an open problem [69].

Beside comparing languages or programming paradigms, researchers focused on specific language abstractions and API design. Stylos and Clarke find that APIs requiring object initialization via setter methods are preferred and less problematic then APIs using constructor parameters [70]. Stylos and Myers study the effects of method placement on API learnability [71]. They find that programmers are significantly faster using the APIs in which methods combining different objects are placed in one of the objects' classes rather than the API in which methods for combining different objects are placed in a helper class. Ellis *et al.* study the effect on usability of the Factory design pattern [72]. They conclude that creating objects from factories is significantly more time consuming for programmers than object creation from constructors – regardless of the level of experience of the programmer using the API.

## 9 Conclusion

Reactive Programming is a paradigm that specifically addresses reactive software that needs to respond to internal or external stimuli with a proper action. The advantages of RP over the traditional Object-oriented paradigm, especially with regards to program comprehension, have been advocated for some time now, but little evidence has been provided in practice. In this paper, we presented a controlled experiment to evaluate the impact of RP on program comprehension. Our experiment involved 127 subjects and our results suggest that RP significantly outperforms OO. The RP group provided more correct results, while not requiring more time to complete the tasks. In addition, we found evidence that comprehension of RP program is less correlated to programming-skills level than comprehension of OO programs. Finally, through qualitative analysis of the feedback provided by subjects, we identified the main reasons behind the ease of comprehending RP programs (e.g., reduced boiler plate code and better readability). On the other hand, we also identified reasons that might prevent the adoption of RP (e.g., use of functional programming concepts). Besides providing the first empirical evidence with regards to the advantages of RP in program comprehension, our results can help guide future research on reactive applications.

## Acknowledgments

This research has been funded by the European Research Council (grant No. 321217) and by the German Federal Ministry of Education and Research (grant No. 01IS12054).

The authors want to thank Stefan Hanenberg for the inspiring discussions on controlled experiments and the useful suggestions on the analysis of the data presented in this work.

## References

[1] C. Elliott, "Functional Implementations of Continuous Modeled Animation," in *International Symposium on Principles of Declarative Programming*. Springer-Verlag, 1998.

[2] C. Elliott and P. Hudak, "Functional Reactive Animation," in *International Conference on Functional Programming*. ACM, 1997.

[3] G. H. Cooper and S. Krishnamurthi, "Embedding Dynamic Dataflow in a Call-by-value Language," in *European Conference on Programming Languages and Systems*. Springer-Verlag, 2006.

[4] 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 *Conference on Object-oriented Programming Systems Languages and Applications*. ACM, 2009.

[5] G. Salvaneschi, G. Hintz, and M. Mezini, "REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications," in *International Conference on Modularity*. ACM, 2014.

[6] I. Maier and M. Odersky, "Higher-Order Reactive Programming with Incremental Lists," in *European Conference on Object-oriented Programming*. Springer-Verlag, 2013.

[7] J. Liberty and P. Betts, *Programming Reactive Extensions and LINQ*. Apress, 2011.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object-oriented Software*. Addison-Wesley, 2000.

[9] G. Salvaneschi, P. Eugster, and M. Mezini, "Programming with implicit flows." *IEEE Software*, 2014.

[10] I. Maier and M. Odersky, "Deprecating the Observer Pattern with Scala.React," EPFL, Tech. Rep., 2012. [Online]. Available: http://infoscience.epfl.ch/record/176887

[11] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A Survey on Reactive Programming," *ACM Computing Surveys*, 2013.

[12] G. Salvaneschi, S. Amann, S. Proksch, and M. Mezini, "An Empirical Study on Program Comprehension with Reactive Programming," in *International Symposium on Foundations of Software Engineering*. ACM, 2014.

[13] "Baconjs." https://baconjs.github.io.

[14] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. MIT Press, 1996.

[15] J. W. Creswell, *Qualitative Inquiry and Research Design: Choosing Among Five Approaches*. Sage, 2012.

[16] F. Olivero, M. Lanza, M. D'ambros, and R. Robbes, "Tracking Human-centric Controlled Experiments with Biscuit," in *Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, 2012.

[17] S. Hanenberg, "An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time," in *International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2010.

[18] "WebLab Site." http://department.st.ewi.tudelft.nl/weblab/.

[19] S. D. Fleming, E. Kraemer, R. E. K. Stirewalt, S. Xie, and L. K. Dillon, "A Study of Student Strategies for the Corrective Maintenance of Concurrent Software," in *International Conference on Software Engineering*. ACM, 2008.

[20] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers, "Program Comprehension As Fact Finding," in *Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering*. ACM, 2007.

[21] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do Professional Developers Comprehend Software?" in *International Conference on Software Engineering*. IEEE Press, 2012.

[22] J. Quante, "Do Dynamic Object Process Graphs Support Program Understanding? - A Controlled Experiment." in *International Conference on Program Comprehension*. IEEE, 2008.

[23] N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*. Springer-Verlag, 2010.

[24] S. Hanenberg, "An Experiment About Static and Dynamic Type Systems: Doubts About the Positive Impact of Static Type Systems on Development Time," in *International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2010.

[25] C. Mayer, S. Hanenberg, R. Robbes, E. Tanter, and A. Stefik, "An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software," in *International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2012.

[26] S. Kleinschmager and S. Hanenberg, "How to Rate Programming Skills in Programming Experiments?: A Preliminary, Exploratory, Study Based on University Marks, Pretests, and Self-estimation," in *Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, 2011.

[27] "IBM SPSS website," http://www.ibm.com/analytics/us/en/technology/spss/.

[28] F. Détienne, *Software Design—Cognitive Aspects*, F. Bott, Ed. Springer-Verlag, 2002.

[29] N. Pennington, "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs," *Cognitive Psychology*, 1987.

[30] T. D. LaToza and B. A. Myers, "Developers Ask Reachability Questions," in *International Conference on Software Engineering*. ACM, 2010.

[31] "Rx operators." http://reactivex.io/documentation/operators.html.

[32] G. Salvaneschi and M. Mezini, "Debugging for reactive programming," in *International Conference on Software Engineering*. ACM, 2016.

[33] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *International Conference on Software Engineering*. ACM, 2006.

[34] R. Turner, M. Falcone, B. Sharif, and A. Lazar, "An Eye-tracking Study Assessing the Comprehension of C++ and Python Source Code," in *Symposium on Eye Tracking Research and Applications*. ACM, 2014.

[35] Y.-T. Lin, C.-C. Wu, Y.-C. Lin, T.-Y. Hou, F.-Y. Yang, and C.-H. Chang, "Cognitive Processes During Program Debugging: An Eye-Movement Approach," *Transactions on Education*, 2015.

[36] T. Busjahn, C. Schulte, B. Sharif, Simon, A. Begel, M. Hansen, R. Bednarik, P. Orlov, P. Ihantola, G. Shchekotova, and M. Antropova, "Eye Tracking in Computing Education," in *Conference on International Computing Education Research*. ACM, 2014.

[37] A. Jbara and D. Feitelson, "How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking," in *International Conference on Program Comprehension*, 2015.

[38] K. Kevic, B. Walters, T. Shaffer, B. Sharif, T. Fritz, and D. Shepherd, "Tracing Software Developers' Eyes and Interactions for Change Tasks," in *Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering*. ACM, 2015.

[39] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," in *Software Reusability*. ACM, 1989.

[40] S. Letovsky, "Cognitive processes in program comprehension," in *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Ablex Publishing Corp., 1986.

[41] A. von Mayrhauser and A. M. Vans, "Comprehension Processes During Large Scale Maintenance," in *International Conference on Software Engineering*. IEEE Computer Society Press, 1994.

[42] A. J. Ko, H. Aung, and B. A. Myers, "Eliciting Design Requirements for Maintenance-oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks," in *International Conference on Software Engineering*. ACM, 2005.

[43] "Demo for the internals of the ELM debugger," http://www.youtube.com/watch?v=FSdXiBLpErU.

[44] "Labview interacting debugger," http://www.ni.com/getting-started/labview-basics/debug.

[45] "Dependency graph visualization in the unreal engine 4, tools demonstration gdc 2014, minutes 6:46 and 8:45," http://www.youtube.com/watch?v=9hwhH7upYFE\#t=384.

[46] M. Ali, "An Introduction to Microsoft SQL Server StreamInsight," in *International Conference and Exhibition on Computing for Geospatial Research & Application*. ACM, 2010.

[47] B. Gedik, H. Andrade, A. Frenkiel, W. De Pauw, M. Pfeifer, P. Allen, N. Cohen, and K.-L. Wu, "Tools and Strategies for Debugging Distributed Stream Processing Applications," *Software: Practice and Experience*, 2009.

[48] S. Endrikat and S. Hanenberg, "Is Aspect-Oriented Programming a Rewarding Investment into Future Code Changes? A Socio-technical Study on Development and Maintenance Time," in *International Conference on Program Comprehension*. IEEE, 2011.

[49] M. Hoppe and S. Hanenberg, "Do Developers Benefit from Generic Types?: An Empirical Comparison of Generic and Raw Types in Java," in *International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2013.

[50] M. Di Penta, R. E. K. Stirewalt, and E. Kraemer, "Designing your Next Empirical Study on Program Comprehension," in *International Conference on Program Comprehension*. IEEE, 2007.

[51] I. Salman, A. Misirli, and N. Juristo, "Are Students Representatives of Professionals in Software Engineering Experiments?" in *International Conference on Software Engineering*. IEEE, 2015.

[52] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, Robots, and Functional Reactive Programming," in *Summer School on Advanced Functional Programming 2002, Oxford University*. Springer-Verlag, 2003.

[53] R. Newton, G. Morrisett, and M. Welsh, "The Regiment Macroprogramming System," in *International Conference on Information Processing in Sensor Networks*. ACM, 2007.

[54] B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane, "The Amulet Environment: New Models for Effective User Interface Software Development," *Transactions on Software Engineering*, 1997.

[55] B. N. Freeman-Benson, "Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming," in *European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*. ACM, 1990.

[56] T. Felgentreff, A. Borning, R. Hirschfeld, J. Lincke, Y. Ohshima, B. Freudenberg, and R. Krahn, "Babelsberg/JS," in *European Conference on Object-oriented Programming*. Springer-Verlag, 2014.

[57] T. Felgentreff, T. Millstein, A. Borning, and R. Hirschfeld, "Checks and Balances: Constraint Solving Without Surprises in Object-constraint Programming Languages," in *International Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 2015.

[58] G. Salvaneschi, J. Drechsler, and M. Mezini, "Towards Distributed Reactive Programming," in *Coordination Models and Languages*. Springer-Verlag, 2013.

[59] A. Margara and G. Salvaneschi, "We have a dream: Distributed reactive programming with consistency guarantees," in *International Conference on Distributed Event-Based Systems*. ACM, 2014.

[60] G. Salvaneschi and M. Mezini, "Towards Reactive Programming for Object-oriented Applications," in *Transactions on Aspect-Oriented Software Development*. Springer-Verlag, 2014.

[61] N. R. Krishnaswami, N. Benton, and J. Hoffmann, "Higher-order functional reactive programming in bounded space," in *Symposium on Principles of Programming Languages*. ACM, 2012.

[62] M.-A. Storey, "Theories, Tools and Research Methods in Program Comprehension: Past, Present and Future," *Software Quality Control*, 2006.

[63] V. Ramalingam and S. Wiedenbeck, "An Empirical Study of Novice Program Comprehension in the Imperative and Object-oriented Styles," in *Workshop on Empirical Studies of Programmers*. ACM, 1997.

[64] C. L. Corritore and S. Wiedenbeck, "Mental Representations of Expert Procedural and Object-oriented Programmers in a Software Maintenance Task," *International Journal of Human-Computer Studies*, 1999.

[65] R. Wettel, M. Lanza, and R. Robbes, "Software Systems As Cities: A Controlled Experiment," in *International Conference on Software Engineering*. ACM, 2011.

[66] B. Cornelissen, A. Zaidman, and A. van Deursen, "A Controlled Experiment for Program Comprehension Through Trace Visualization," *Transactions on Software Engineering*, 2011.

[67] V. Pankratius, F. Schmidt, and G. Garreton, "Combining Functional and Imperative Programming for Multicore Software: An Empirical Study Evaluating Scala and Java," in *International Conference on Software Engineering*, 2012.

[68] L. Prechelt, "An Empirical Comparison of Seven Programming Languages," *Computer*, 2000.

[69] G. Salvaneschi, "What do we really know about data flow languages?" in *International Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, 2016.

[70] J. Stylos and S. Clarke, "Usability Implications of Requiring Parameters in Objects' Constructors," in *International Conference on Software Engineering*. IEEE, 2007.

[71] J. Stylos and B. A. Myers, "The Implications of Method Placement on API Learnability," in *International Symposium on Foundations of Software Engineering*. ACM, 2008.

[72] B. Ellis, J. Stylos, and B. Myers, "The Factory Pattern in API Design: A Usability Evaluation," in *International Conference on Software Engineering*. IEEE, 2007.

**Guido Salvaneschi** is an assistant professor at TU Darmstadt. His current research interests focus on programming language design of reactive applications, such as event based languages, dataflow languages and functional reactive programming. His work includes the integration of different paradigms, incrementality and distribution. He completed his PhD at Dipartimento di Elettronica e Informazione at Politecnico di Milano, under the supervision of Prof. Carlo Ghezzi.

**Sebastian Proksch** is a doctoral candidate at TU Darmstadt in the group of Prof. Mira Mezini. His research is focused on the structured development of recommender systems in software engineering and includes work on static analyses, mining software repositories, and human factors in software engineering. The work is driven by the idea to combine different sources of input data to improve the quality of recommender systems and their evaluation.

**Sven Amann** is a PhD candidate at TU Darmstadt in the group of Prof. Mira Mezini. He is project lead of MuBench and MuDetect. His research interest is on API-misuse detectors and other recommender systems for software engineering. To tailor solutions, he investigates on the habits and problems of software developers in the wild. This includes work on big code, human factors in software engineering, machine learning, and static analyses.

**Sarah Nadi** is an assistant professor at the University of Alberta, Canada. She received both her MMath (2010) and PhD (2014) degrees from the University of Waterloo, Canada, and spent approximately two years as a postdoctoral researcher at TU Darmstadt, Germany. Her research interests include providing automated support for developing and maintaining highly configurable software, empirical software engineering, mining software repositories, software evolution, and code recommender systems.

**Mira Mezini** received the diploma degree in computer science from the University of Tirana, Albania, and the PhD degree in computer science from the University of Siegen, Germany. She is a professor of computer science at the Technische Universität Darmstadt, Germany, where she heads the Software Technology Lab.