An Empirical Study on Program Comprehension with Reactive Programming

Guido Salvaneschi, Sven Amann, Sebastian Proksch, Mira Mezini
Software Technology Group
Technische Universität Darmstadt
lastname@informatik.tu-darmstadt.de

ABSTRACT

Starting from the first investigations with strictly functional languages, reactive programming has been proposed as *THE programming paradigm* for reactive applications. The advantages of designs based on this style over designs based on the Observer design pattern have been studied for a long time. 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 – that, beside other advantages, reactive programming makes a wide class of otherwise cumbersome applications more comprehensible – has never been evaluated. In this paper, we present the design and the results of the first empirical study that evaluates the effect of reactive programming on comprehensibility compared to the *traditional* object-oriented style with the Observer design pattern. Results confirm the conjecture that comprehensibility is enhanced by reactive programming. In the experiment, the reactive programming group significantly outperforms the other group.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

Reactive Applications, Reactive Programming

1. INTRODUCTION

Reactive applications are a wide class of software that needs to 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 intuitive and declarative way. RP has been firstly introduced in Haskell. Influenced by these approaches, implementations of RP have been proposed in several widespread languages, including Scheme [3], Javascript [29] and Scala[40, 27]. Recently, concepts inspired by RP have been applied to production frameworks like Microsoft Reactive Extensions (Rx) [25], 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 in the front-end developers community is revealed by the increasing number of libraries, inspired by the Flapjax reactive language [29], that implement RP principles, among the others React.js, Bacon.js, Knockout, Meteor and Reactive.coffee.

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 [14] is widely used for such applications. It has the advantage of decoupling observers from observables. But, when it comes to program readability, it does not make things easier, because of dynamic registration, side effects in callbacks, and inversion of control.

In contrast, RP supports a design based in 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. Dependencies are defined explicitly instead of being hidden in the control flow, combination can be guided by types opposite to callbacks that return void, contrarily to the Observer pattern control is not inverted and less boilerplate is required since collecting dependencies and performing the updates is automatized by the framework. Based on these arguments, it has indeed been argued that RP greatly improves over the traditional Observer pattern used in OO programming both (i) from the *software design* perspective as well as (ii) from the perspective of facilitating the comprehensibility of the software [26, 3, 29, 2].

Yet, not enough empirical evidence has been ever provided in favor of the claimed advantages of RP. Preliminary empirical results seem to confirm the claimed design benefits (e.g., more composability) of RP [40]. However, even preliminary evidence is missing regarding the claim that RP enhances comprehensibility. 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 [29], a Javascript application based on Observer is compared against a functionally equivalent RP version. The authors argument 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 the 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 38 subjects that where divided into a RP group and an OO group. They were shown a reactive application and their understanding of the reactive functionalities was measured. The experiment considers several aspects including (i) correctness of comprehension, (ii) required time and (iii) programming skills needed for correct comprehension. To the best of our knowledge, such a study has never been attempted before. All experimental data and the artifacts developed for this work are available for download[1], including the tasks that compose the experiment, the raw results and the complete statistical analysis not included in this paper for space reasons.

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 discusses threats to validity. Section 6 presents related work. Finally, Section 7 outlines open research challenges and concludes the paper.

2. MOTIVATION

In this section, we introduce reactive programming and discuss the weaknesses of the Observer pattern that can potentially reduce program understandability.

Reactive Programming in a Nutshell.

Reactive programming supports first-class constraints among program values. These constraints are automatically enforced by the runtime. In the rest, we adopt the terminology of REScala [40] and Scala.react [27]. Constraints are expressed as *signals*, a language concept for expressing functional dependencies among values in a declarative way. A signal can depend on variables without further dependencies (i.e., vars) or on other signals. When any of the dependency sources changes, the expression defining the reactive value is automatically recomputed by the language runtime to keep the reactive value up-to-date.

The general form of a signal `s` is `Signal{`*expr*`}`, where *expr* is a standard expression. When *expr* is evaluated, all `Signal` and `Var` values it refers to are registered as dependents of *s*; any subsequent change of them triggers a reevaluation of *s*.

Consider the example in Figure 1a. The code snippet defines two vars `a` and `b`. When `s` is declared (Line 3), the runtime collects the values it depends on (i.e., `a` and `b`). When `a` is updated (Line 6), all signals that (even indirectly) depend on it – `s` in this case – are automatically updated.

As the reader has probably already 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 [40, 27].

Reactivity the Old School: The Observer Pattern.

In OO languages, reactive applications are usually developed using the Observer design pattern. This solution has gained wide popularity because it decouples observers from observables, i.e., observables do not know (hold a static reference to) observers in

[1]http://goo.gl/8Qn4cx

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

(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).

advance. Instead, observers register themselves to the observable they are interested in at runtime [14]. Detractors argue that programs are hard to reason about. Below, we summarize the main points that support this argumentation. For convenience, we refer to the example in Figure 1b, which implements the same functionality as Figure 1a but with the Observer design pattern.

(i) The natural dependency among program entities is inverted. Conceptually, change flows from observables to observers, but in code, observers call the observable to register. For example, `c` is registered 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 `b`, 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 are more verbose and the application logic is hidden behind the machinery required by the Observer design pattern.

Executive Summary: Call to Arms.

It is, however, not clear, what the impact of the previous issues is. 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 should be evaluated empirically.

3. STUDY DEFINITION

The claimed advantages of RP include increased composability, abstraction over state, automatic memory management, enforcement of consistency guarantees during change propagation and ease of comprehension [40, 26, 3, 29, 2]. 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"* [1]. We also consider a single alternative to RP,

```
1  object Squares_Reactive extends SimpleSwingApplication {
2
3    // -- APPLICATION LOGIC ----------------------
4    object square1 {
5      val position = Signal {
6        Point(time().s * 100, 100)
7      }
8    }
9    object square2 {
10     val v = Signal {
11       time().s * 100
12     }
13     val position = Signal {
14       Point(time().s * v(), 200)
15     }
16   }
17
18   // painting components
19   (square1.position.changed || square2.position.changed) += {
20      _ => Swing onEDT { top.repaint }
21    }
22
23   // -- Graphics ----------------------
24    lazy val panel: RePanel = new RePanel {
25     override def paintComponent(g: Graphics2D) {
26       super.paintComponent(g)
27       g.fillRect(
28         square1.position.getValue.x.toInt - 8,
29         square2.position.getValue.y.toInt - 8,
30         16, 16)
31       g.fillRect(
32         square1.position.getValue.x.toInt - 8,
33         square2.position.getValue.y.toInt - 8,
34         16, 16)
35     }
36   }
37   lazy val top = new MainFrame {
38     preferredSize = new Dimension(800, 400)
39     contents = panel
40   }
41 }
```

(a)

```
1  object Squares_Observer extends SimpleSwingApplication {
2
3    // -- APPLICATION LOGIC ----------------------
4    object square1 {
5      val position = Observable { Point(0, 0) }
6      addTimeChangedHandler { time =>
7        position set Point(time.s * 100, 100)
8      }
9    }
10   object square2 {
11     val v = Observable { 0.0 }
12     val position = Observable { Point(0, 0) }
13
14     addTimeChangedHandler { time =>
15       v set time.s * 100
16       updatePos(time, v.get)
17     }
18     v addObserver { v =>
19       updatePos(now, v)
20     }
21     def updatePos(time: Time, v: Double) {
22       position set Point(time.s * v, 200)
23     }
24   }
25
26   // painting components
27   square1.position addObserver { _ => repaint }
28   square2.position addObserver { _ => repaint }
29   def repaint = Swing onEDT { top.repaint }
30
31   // -- Graphics ----------------------
32   lazy val panel: RePanel = new RePanel {
33     override def paintComponent(g: Graphics2D) {
34       super.paintComponent(g)
35       g.fillRect(
36         square1.position.getValue.x.toInt - 8,
37         square2.position.getValue.y.toInt - 8,
38         16, 16)
39       g.fillRect(
40         square1.position.getValue.x.toInt - 8,
41         square2.position.getValue.y.toInt - 8,
42         16, 16)
43     }
44   }
45   lazy val top = new MainFrame {
46     preferredSize = new Dimension(800, 400)
47     contents = panel
48   }
49 }
```

(b)

Figure 2: The Squares application implemented with RP (a) OO programming (b).

i.e., OO programming and the Observer design patterns, because it is the most common solution for reactive applications.

Object of the Experiment.

The experiment focuses on 10 reactive programs. Each program is implemented in two versions. The RP version is based on reactive programming, i.e., signal and, when needed, events. The OO version adopts the Observer pattern to implement reactivity.

The applications we developed belong to three categories. The first category consists of **synthetic** applications (applications 1-4) that 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. The second category consists of graphical **animations** (applications 5-7), where shapes are displayed on a canvas and move according to regular patterns. Graphical animations is a traditional domain for reactive programming [9, 8]. The last category consists of **interactive** applications (8-10) that require the user, e.g., to click buttons or drag the mouse over a shape. These functionalities are common in GUIs, another traditional domain for reactive programming [29, 3].

An example of the applications we used is the Squares application in Figure 2a (RP version) and Figure 2b (OO version). The example belongs to the animations category. Code is reduced for presentation purposes. The Squares application draws two moving squares on a canvas. To give an intuition of the execution, a screenshot is provided in Figure 3 (note that subjects did not have access to it). The upper and the lower square move horizontally from left to right at constant speed – respectively, at increasing speed.

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 in Line 10. Line 19 triggers a repainting in the asynchronous Swing events loop every time either position of `square1` or the position of `square2` changes. Lines 24-39 setup the canvas and display the initial squares.

In the OO version (Figure 2b) the same functionalities are implemented using the Observer pattern. Line 6 register a handler

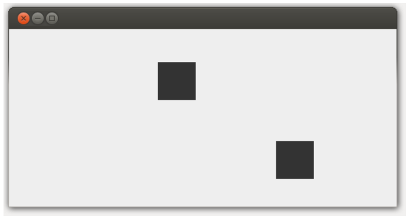

Figure 3: The Squares graphical animation.

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 unchanged from the RP case.

As the reader may have noticed, some functionalities, like 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.

Methodology.

The experiment is composed of 10 tasks. In each task, a subject is shown a reactive application and asked a question about the application behavior. Crucially, questions and alternative choices are formulated in a way that finding the correct answer requires to understand the (whole) reactive logic of the application. An example taken from the synthetic applications group is to ask the sequence of values assumed by a variable that depends on other values in the program once such 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 is shown in Figure 4 and refers to Figure 2.

To practically run the experiment, we developed WebCompr, a Web application for experiments on program comprehension that allows users to complete the tasks via a Web browser. Existing tools for controlled experiments we are aware of are not Web-based (e.g., Biscuit [32] and Purity [15]) which considerably increases the complexity of running the experiment. We relinquished using existing Web platforms for in-browser homework and exams (e.g, WebLab[2]) because they lack fine-grained time management, such as setting an upper bound to each single task and recording the task completion time – which are important for our experiment (more on this later). WebCompr is highly customizable, collects events from the user in a generic way and supports different kinds of tasks, including some not used here, for example, *blind* tasks where subjects can inspect an application for a limited amount of time and are asked to complete a task afterward, without access to the code anymore. WebCompr is available to the community to run experiments similar to ours.

We use tasks as described above as a research instrument, because they provide objective results and scale well (most controlled studies hardly include more than 15 participants [12, 22] – we have 38 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 ask for volunteers.

We briefly discuss the alternatives we evaluated. In the *think loudly* approach subjects comment the actions they are performing [12, 22]. A subsequent interview can clarify the motivations behind each action [38]. 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 [22]. 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, not only comprehension (i.e., *code-writing skills* is an additional factor of the experiment). This approach is suitable, if the factor is balanced for the groups. In such a case, it doesn't influence the main factor we want to observe. In our experiment however, the programming skills of the subjects 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 – program comprehension.

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 spontaneously force themselves to spend on each portion class only a limited amount of time to keep the analysis of the whole software feasible.

For practical reasons, we needed to define an upper bound to the time subjects can spend on each task. Also, it has been shown that, without constraints, subjects may spend the entire time of the experiment on a single task [36]. We fixed the available time to 5 minutes for synthetic applications and to 10 minutes for the animations and interactive applications, because the latter are slightly longer and require to inspect more code. In practice, our estimations turned out to be rather conservative. None of the subjects required the full amount of time and most subjects provided that answer way earlier. In any case, to avoid that attention decreases too much, the experiment was designed to require no more than 2 hours (preliminary tasks + experiment tasks).

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 [20]. In most empirical studies in software engineering, within-subject design is usually preferred to balance the effect of the individual skills factor, i.e., to reduce the variability due to heterogeneous skills 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 a factor to the solution of the same task with the other factor – a problem solved by between-subjects designs. It has been shown [16, 28] that, if the learning effect is small enough, proper experiment design still allows a within-subject approach. In our case, however, this was no 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 [36]. Therefore, we chose to design the experiment as a *between-subjects*.

[2]http://department.st.ewi.tudelft.nl/weblab/

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

Figure 4: Example question.

Study Context.

The subjects of our study are students from a Software Engineering course, held in the 4th year of study in Computer Science. 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 require to use RP to develop a reactive application from a given specification.

We used WebCompr also to collect information about the subjects, such as their programming experience, and required them to self-evaluate as programmers. Since self-evaluation cannot be considered reliable [21], subjects were also given 18 preliminary tasks to solve as a measure of their programming knowledge. To test a variety of different topics we necessarily kept each task very short – the maximum amount of time available is a minute. Questions include concepts of OO programming, like 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 the Web application we used for the experiment.

Overall, the questions we asked were quite advanced and not necessarily covered by the course. Subjects were given multiple answers and a *Don't know* option to avoid guessing. 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 `zip` function[3] on lists is likely to be a better programmer than a subject that has never heard about it.

Research Questions and Hypotheses.

We want to investigate the impact of RP on the comprehension of reactive applications. Our research questions can be formulated as follows. The fist research questions concerns correctness of program comprehension in the RP and in the OO style:

RQ1: Does reactive programming increase correctness of program comprehension?

The second research question investigates the role of time (besides correctness) in comprehending RP programs on OO programs:

RQ2: Is comprehending reactive applications in the RP style more or less time-consuming than in the OO style?

The third question concerns the relation between programming skills and the level of comprehension of RP programs compared to OO programs. The impact of programming expertise on the use or RP is important to know whether RP is only useful for advanced programmers or exhibits significant benefits also for beginners.

RQ3: Does comprehending RP programs require more advanced programming skills than the OO style?

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

[3]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 $[(l_i, r_i), i \in (0..n)]$.

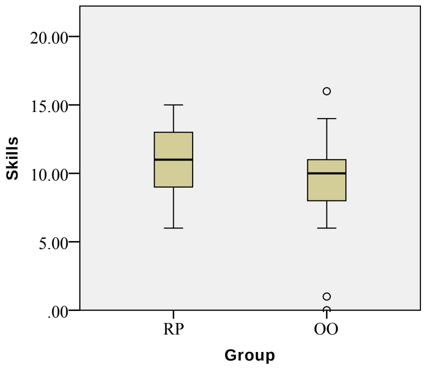

Figure 5: Subjects skills in RP group and OO group.

4. STUDY RESULTS

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.

As a preliminary step, the tasks on programming skills allow us to validate the hypothesis of equal distribution among the RP group and the OO group. This assumption is needed to make sure that the groups are not unbalanced for what programming skills concern. A first inspection of the data shows similarity between the groups (Figure 5). We performed a Mann-Whitney 2-tailed U test on the cumulative result obtained in the skills questions. The result shows that equality of the groups cannot be rejected (p=0.17).

4.1 Correctness

First, we analyze the data to provide an answer for RQ1. Correctness analysis concerns the amount of tasks for which the subject correctly understood the behavior of the application and provided the right answer. As the measure of correctness, we consider the score (i.e., the cumulative number of correct answers) provided by each student in all tasks.

We initially provide an overview of the results with descriptive statistic. The mean score for the OO group and the RP group are $\mu_{(score,OO)} = 6.65$ respectively $\mu_{(score,RP)} = 8.17$. Figure 6 shows the cumulative scores obtained by the subjects in the RP group and in the OO group. The plots seem to support the claim that the RP group obtains a better result than the OO group. However, to conclude that the difference is significant, it is necessary to apply an appropriate statistical test.

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:

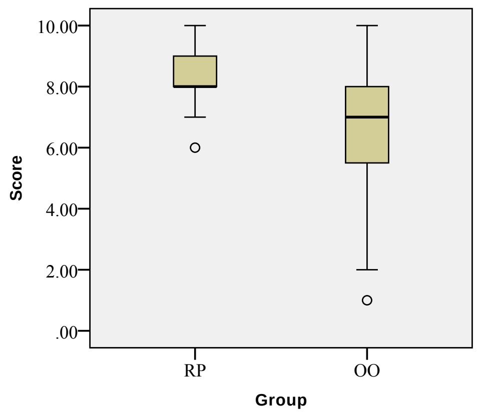

Figure 6: Box plot of the correctness Ranks.

Group	N	Rank avg	Rank sum	p-value
RP	18	23.69	426.5	0.023
OO	20	15.73	314.5	

Figure 7: Mann-Whitney U test for the Scores.

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

The distribution of the score for both groups cannot be safely assumed to exhibit a normal distribution. Testing normality with the Shapiro-Wilk test gives p_{RP}=0.127 and p_{OO}=0.070, which does not allow to assume the normality hypothesis. Since the underlying distribution is not known, we performed a non-parametric Mann-Whitney U test. The result shows that, with high significance (p=0.026) H_0 can be rejected. After this result is established, the rank sum indicates which approach dominates the other. As shown in Figure 7, the rank sum for RP is significantly higher than the rank sum for OO.

This result provides an answer to RQ1 and allows us to conclude that the RP increases correctness of program comprehension.

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. As a matter of fact, 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 be much more time consuming for developers.

Descriptive statistics suggests that the RP group requires less time to complete each task. Figure 8 shows a box plot comparing the time required by the RP group and the time required by the OO group for each task. Since some tasks require 5 mins and other 10 mins (cf. Section 3), to make the results comparable, we normalize the times for all exercises to a 0-1000 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, where binary data require a cumulative score over all the tasks, for time data are continuous and we can analyze each task separately. Similar to the previous case, the underlying distribution is unknown and we perform a non-parametric Mann-Whitney U test. The results are in Figure 9. For exercises 1-2-3-4-7, H_0 can be rejected with high significance (p<0.05). In all those cases, the rank sum indicates that OO times are higher than RP times (Figure 9, column "Ranks sum"), i.e., *the RP group is faster*. In exercises 5-6-8-9-10, H_0 cannot be rejected (p>0.05).

One may argue that the last result measures the time it takes a programmer to give the answer *she believes* is correct. We further inspected our results to restrict the analysis to only the times that the subjects needed to provide the *correct answer*. The results of the Mann-Whitney U test are in Figure 10. For exercises 1-2-3-7-10, the difference is significant (p<0.05) which rejects H_0. The rank sum indicates that OO dominates RP. In the other exercises H_0 cannot be rejected. These observations confirm that the previous conclusion still holds even when the analysis considers only the correct answers. In summary, we answer RQ2 concluding that comprehending programs in the RP style *does not* require more time than comprehending their OO equivalent.

4.3 Programming Skills

In this section, we inspect the experimental data to answer RQ3 – were more advanced programming skills required for comprehending RP applications. To this end, we correlate the programming skills we measured for each subject with the correctness results in the experiment. Figure 11 shows a scatter plot of the programming skills and the cumulative 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).

The figure suggests that in the OO group lower-skilled subjects perform poorly in the tasks. Subjects that performs better are also advanced programmers. In the RP group, instead, subjects do not need advanced programming skills to reach high scores. In other terms, the figure suggests absence of correlation for the RP group and a positive correlation for the OO group, which we further investigate in the rest. At this stage of the analysis, the inspection of the plots in Figure 11 shows a significant difference between the RP group and the OO group.

We use a statistical test to estimate the significance of the correlation of subjects' score and skills. We define the following hypotheses:

$H_{0(RP)}$: *There is no correlation between the score of the subjects in the RP group in the experiment and the measured programming skills.*

$H_{0(OO)}$: *There is no correlation between the score of the subjects in the OO group in the experiment and the measured programming skills.*

The skills results cannot be safely assumed normally distributed. We use the Kendall's (τ) test and Spearman's rank correlation (ρ) test. Even if this result is unlikely, we do not exclude negative correlation and perform 2-tailed tests. The results are presented in Figure 12, where we show the correlation coefficient r and the p-value p for each case. For the RP group, we detected no significant correlation, i.e., H_0 cannot be rejected. Surprisingly, in this case, the correlation coefficient is even negative. For the OO group, instead, both tests are statistically significant (p=0.039, p=0.045) and H_0 can be rejected. We conclude that in the OO group there is ev-

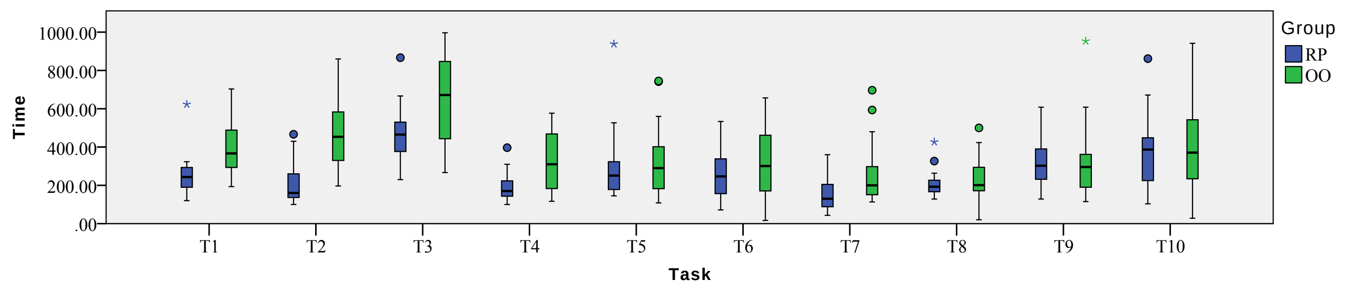

Figure 8: Box plots for time in RP and OO Tasks.

	Group	N	Rank avg	Rank sum	p-value
Task 1	RP	18	13.17	237	0.001
	OO	20	25.2	504	
Task 2	RP	18	11.39	205	0
	OO	20	26.8	536	
Task 3	RP	18	15.19	273.5	0.023
	OO	20	23.38	467.5	
Task 4	RP	18	14.03	252.5	0.003
	OO	20	24.43	488.5	
Task 5	RP	18	18.14	326.5	0.483
	OO	20	20.73	414.5	
Task 6	RP	18	18.11	326	0.474
	OO	20	20.75	415	
Task 7	RP	18	14.75	265.5	0.011
	OO	20	23.78	475.5	
Task 8	RP	18	18.86	339.5	0.745
	OO	20	20.08	401.5	
Task 9	RP	18	20.42	367.5	0.638
	OO	20	18.68	373.5	
Task 10	RP	18	19.11	344	0.845
	OO	20	19.85	397	

Figure 9: Mann-Whitney U test for Answer Times.

	Group	N	Rank avg	Rank sum	p-value
Task 1	RP	16	9.66	154.5	0
	OO	12	20.96	251.5	
Task 2	RP	15	9.2	138	0
	OO	15	21.8	327	
Task 3	RP	14	9.43	132	0.023
	OO	9	16	144	
Task 4	RP	17	12.26	208.5	0.01
	OO	14	20.54	287.5	
Task 5	RP	17	15.53	264	0.992
	OO	13	15.46	201	
Task 6	RP	14	13.75	192.5	0.643
	OO	14	15.25	213.5	
Task 7	RP	18	13.64	245.5	0.028
	OO	15	21.03	315.5	
Task 8	RP	18	17.86	321.5	0.542
	OO	19	20.08	381.5	
Task 9	RP	13	17.12	222.5	0.236
	OO	16	13.28	212.5	
Task 10	RP	5	3.6	18	0.03
	OO	6	8	48	

Figure 10: Mann-Whitney U test for Correct Answers Times.

idence of a positive correlation (r=0.358, r=0.452) between scores and subjects skills.

The analysis provides an answer for RQ3. We conclude that, in contrast to OO where score results are correlated to programming skills, with RP (advanced) programming skills are not needed to understand reactive applications. This result suggests that RP lowers the *entrance barrier* required to understand reactive applications.

5. DISCUSSION

5.1 Threats to Validity

In this section, we discuss factors that menace the validity of our result and which countermeasures we adopted to reduce this risk.

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 understanding requires careful formulation of questions and candidate answers. Questions that are too specific may not require a significant understanding 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 in the program code a line like `if(numberOfClicks == 3)[...]`. We tackled this problem in two ways. First, we for-

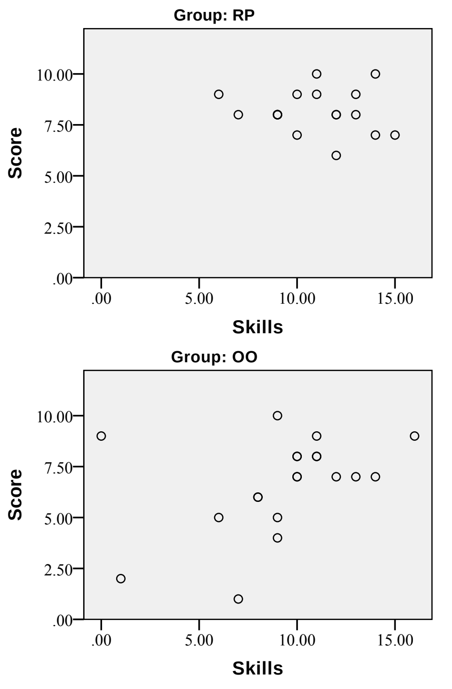

Figure 11: Scatter plots of Score and programming Skills.

			RP		OO	
			Score	Skills	Score	Skills
Kendall	Score	r	1	-0.102	1	*0.358
		p	0	0.597	0	0.039
	Skills	r	-0.102	1	*0.358	1
		p	0.597	0	0.039	0
Spearman	Score	r	1	-0.139	1	*0.452
		p	0	0.584	0	0.045
	Skills	r	-0.139	1	*0.452	1
		p	0.584	0	0.045	0

Figure 12: Correlation of Score and Skills with Kendall's τ test and Spearman's ρ test. Starred correlations are significant.

mulated questions and candidate answers in a way that requires a broad understanding 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 made sure that subjects cannot 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 immediately provides an answer to the question in Figure 4.

Another issue concerns the use of a Web-based WebCompr application to complete the tasks. Such a platform can be not immediately intuitive 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 starts with *warm up* tasks to help subjects getting confidence with the platform.

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.

While IDEs play a relevant roles in programmer's activity, the influence of the IDE on different programming paradigms is unclear and should be subject of a different study. Other studies facing the same problem [11, 16, 18] 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 can significantly speed up understanding 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 that takes the IDE into account. Our approach based on the WebCompr platform has the advantage that the impact of tool support is a removed factor. Text search and syntax highlighting are the only *help* subjects receive.

External Validity.

Threats to *external validity* relate to what extent our findings are generalizable.

A first issue concerns the training subjects received. The comparison of two programming techniques is definitely influenced by the skills of the participants in each. For example, a previous study on parallel programming in Scala concluded that programmers are no more productive in Scala than in Java, but subjects trained in both Scala and Java multicore programming for the study had 4 years experience with Java and no previous knowledge of Scala. In our case training on RP was minimal (cf. Section 3), yet leads to observable differences in favor of RP.

Another issue concerns the types and size of the applications we adopted in the experiment. The type of applications we selected are representative for typical domains for 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. Concerning 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 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 [7]. Professional developers can have more expertise in a programming technique after applying it on daily bases for years. A final consideration concerns subjects skills. Kleinschmager and Hanenberg, in a preliminary study [21], presented a negative result on using pretests as a valid criteria for measuring programming skills. However, we use 18 preliminary tasks to increase statistical validity, while in [21] a single task is used to predict the outcome of other 14 tasks which significantly increases our confidence on the methodology we adopt.

5.2 Preliminary Interpretation and Outlook

The results of our empirical study show that RP overcomes OO for program comprehension and, hence, should encourage researchers to further explore RP. However, the work presented in this paper does not test the root causes of the difference between RP and OO w.r.t program comprehension. Further systematic investigation of this issue should proceed along two lines.

First, more evidence should be collected on how developers reason about RP programs. Exploratory studies, e.g., using either the think loudly approach or interviews [24, 22] help understanding the causes of the issues developers face. Far form being systematic, we collected informal feedback from the subjects in the experiment. With 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*) in Section 2. This conjecture is supported by previous findings on program comprehension showing that developers understand programs by building a mental model based on control flow [34] and data and control flow play a fundamental role in the way programmers develop a mental representation of programs [6]. 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 to check whether a change to a variable propagates to the

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
L.Ratio	.038	.527	.036	.042	.008	.019	.586	.253	.573	.88
Fisher	.067	.697	.052	.093	.048	.045	.719	1	.709	1

Figure 13: Likelihood Ratio and Fisher's Exact Test.

other. Previous research [23] noticed that reachability questions are extremely common for developers, they are (i) hard to answer and time-consuming and (ii) error prone. These findings also provide an explanation of our results.

Second, further studies are needed to investigate the use and effect of RP for certain domains in a more systematic way. In our case, this point concerns the relation between programming tasks and the effect of RP on program comprehension. In our experiment design, correctness is a binary variable. Hence, it is unlikely that splitting the analysis for correctness provides significant results for each task. For this reason, in Section 4 we analyzed the aggregated data. However, a preliminary comparison of the correctness in the OO group and in the RP group for each task is depicted in Figure 13. Since the hypothesis on the size of expected frequencies for a Chi-squared test is not met, we used the Likelihood Ratio and Fisher's exact test. The tests are still able to detect a significant advantage of RP for tasks 3, 5 and 6. The difference in the other tasks is not significant. Interestingly, two out of three significant cases belong to animations, which has been the domain for which functional-reactive programming has been originally proposed as a suitable programming paradigm [9].

6. RELATED WORK

We organize related work as follows. First, we outline recent research on RP. Second, we provide and 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 [29] (Javascript), Scala.react [27] and REScala [40] (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 [2].

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 [8], robotics [19] and sensor networks [31]. Graphical libraries, like Garnet and Amulet (Lisp) [30] applied concepts of dataflow programming to relieve the user from manually updating dependent values. Other researchers proposed languages where developers can specify constraints. Bidirectional constraints are allowed and in case not all constraints can be satisfied, a priority rank is applied [13]. Current research directions include RP in the distributed setting [39] and integration with the OO paradigm [41].

Studies on Program Comprehension.

Pennington [34] shows that different language designs can influence whether control flow or data flow questions are easier to understand for programmers. Ramalingam and Wiedenbeck [37] 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 [5]

Other researchers organized controlled experiments to investigate the effect of tools on program comprehension. Quante [36] study the impact of dynamic object graphs on program understanding. Wettel *et al.* [45] show that the CodeCity 3D software visualization tools significantly increase task correctness and reduce task completion time. Similarly, Cornelissen *et al.* [4] evaluate the enhancement of program comprehension by visualizing execution traces with tools.

From a methodological perspective, Di Penta *et al.* discuss the issues of designing studies on program comprehension [7]. Storey [42] reviews the *theories* that have been formulated for program comprehension and their implications on tool design.

Empirical Studies on Programming Techniques.

Pankratius *et al.* [33] organize an empirical study to compare the use of Scala and Java to develop multicore software. Contrarily 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 [35]. In [17] researchers investigate variation of development time using Aspect-oriented Programming. Hanenberg organized a series of controlled experiments to evaluate the effect of types in programming languages. Among the other results, these experiments do not find a positive effects of static type systems on development time [16] and show that similar uncertainties hold for the influence of static type systems on the usability of undocumented software [28]. Also, it was found that generic types improve documentation, do not significantly change development time and reduce extensibility [18].

Beside comparing languages or programming paradigms, researchers focused on specific abstractions and API design, including requiring parameters to objects constructors [43], effects of method placement on API learnability [44] and the effect on usability of the Factory desing pattern [10].

7. CONCLUSION

RP is a paradigm that specifically addresses reactive software. The advantages of RP over the traditional OO paradigm 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. Results suggest that RP outperforms OO. The RP group provided more correct results, while not requiring more time to complete the tasks. In addition, we found evidence that understanding RP program requires less programming skills than OO programs.

In the future, we will continue working on the evaluation of RP. First, we plan to repeat the experiment with other subjects to further strength statistical significance. Second, we want to further investigate which aspects of RP are crucial for improving comprehension extending the analysis to the domain of the applications.

Acknowledgment

This research has been funded by the European Research Council, grant No. 321217 and by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under grant No. 01IC12S01V and under grant No. 01IS12054.

	Max time	300	300	300	300	600	600	600	600	600	600																			
		TASKS											**PRELIMINARY TASKS**																	
	Subject	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**	Score																		Skills
	1	60	30	*148	30	92	64	26	77	*113	240	8.00	1	1	1	1	1	1	1	1	0	0	1	0	0	1	1	1	0	0	12
	2	81	47	117	*66	116	131	78	131	*284	*253	7.00	1	1	1	0	-1	0	0	1	0	0	1	0	1	1	1	1	1	0	10
	3	88	49	94	72	107	142	115	88	157	232	10.00	1	1	1	0	1	0	1	1	1	0	1	0	1	1	1	1	1	1	14
	4	70	*74	200	93	200	*320	67	131	243	*376	7.00	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	1	15
	5	57	38	113	59	94	*64	31	113	135	135	9.00	1	1	1	0	1	1	0	1	1	0	1	1	1	1	1	1	0	0	13
	6	93	54	149	46	107	*200	124	116	192	*123	8.00	1	1	1	0	-1	0	1	1	-1	0	1	0	0	1	1	1	0	0	9
	7	*36	41	111	41	164	43	69	115	*77	123	8.00	0	1	1	0	-1	0	1	1	1	0	1	1	1	1	1	1	1	1	13
	8	41	*78	*140	33	127	164	51	126	*172	*159	6.00	1	1	1	1	1	1	0	1	1	0	1	0	0	1	1	0	0	1	12
RP	9	91	140	148	56	231	94	123	146	*197	*180	8.00	0	1	1	0	0	-1	0	0	0	0	1	0	1	1	1	1	1	1	9
	10	54	98	137	80	194	203	53	196	235	*232	9.00	1	1	1	-1	1	0	0	0	0	0	-1	0	1	1	0	0	0	-1	6
	11	84	*125	159	67	316	133	191	113	157	*517	8.00	-1	1	1	1	1	1	0	1	1	0	1	0	1	0	0	0	0	0	9
	12	*187	129	260	119	563	299	50	256	365	*62	8.00	0	1	1	0	0	0	1	1	1	0	1	0	0	0	1	-1	0	0	7
	13	97	42	173	45	*164	162	135	79	173	*403	8.00	0	1	1	0	0	1	1	1	0	1	-1	0	1	1	0	1	0	0	9
	14	64	46	69	35	166	117	87	158	139	*119	9.00	1	1	1	1	1	-1	1	0	1	0	1	1	0	1	0	-1	1	0	11
	15	51	45	*108	43	87	69	78	100	196	*166	8.00	1	1	1	1	0	0	1	1	1	0	1	0	1	1	1	1	0	0	12
	16	66	31	*139	45	137	*154	95	109	190	*269	7.00	1	1	1	1	1	1	1	1	0	0	1	0	1	1	1	1	0	1	14
	17	76	57	138	56	174	318	216	136	234	*342	9.00	1	1	1	0	0	0	0	1	0	0	1	0	1	1	1	0	1	1	10
	18	88	36	160	46	111	289	58	98	123	235	10.00	1	0	1	0	1	1	1	1	1	1	0	0	-1	1	1	1	0	0	11
Correct	Median	73.00	46.00	143.00	46.00	137.00	137.50	78.00	115.50	190.00	232.00	8.00																	Median		11
	Average	72.56	58.87	144.86	56.82	175.65	159.14	91.50	127.11	195.31	193.00	8.17																	Average		10.89
All	Median	73	48	139.5	51	150.5	148	78	115.5	181.5	232																				
	Average	76.89	64.44	142.39	57.33	175.00	164.78	91.50	127.11	187.89	231.44																				
	19	95	157	*165	95	232	176	170	254	292	*384	8.00	1	1	1	0	0	1	1	1	1	0	1	0	0	1	1	0	0	1	11
	20	165	126	*95	94	99	285	96	150	188	165	9.00	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21	211	232	129	46	*336	208	*356	*46	117	*563	6.00	1	1	1	-1	0	1	0	0	0	0	1	0	1	1	0	1	0	0	8
	22	132	102	*155	59	137	243	71	199	100	332	9.00	1	1	1	-1	1	1	0	0	1	0	1	0	1	1	1	1	0	0	11
	23	104	142	282	*65	282	*135	108	107	206	*224	7.00	1	1	1	1	0	1	1	1	1	0	1	0	1	1	1	1	1	0	14
	24	176	151	247	129	250	107	128	98	122	297	10.00	1	1	1	1	1	0	0	-1	-1	0	1	0	1	1	0	-1	1	0	9
	25	164	132	*137	151	105	302	288	108	69	*365	8.00	1	0	1	0	1	0	1	1	1	0	1	0	1	1	1	0	0	0	10
	26	*96	167	299	72	69	98	68	131	111	*149	8.00	1	1	1	0	1	0	1	1	0	0	1	0	1	1	1	1	0	0	11
	27	*60	*98	*207	*129	198	*392	*121	174	*213	*210	2.00	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	1	1
OO	28	*129	*183	*112	*145	92	269	198	300	221	*162	5.00	1	1	-1	1	-1	0	1	1	0	0	1	0	1	0	1	0	0	1	9
	29	*123	*89	211	*141	222	128	124	179	154	303	7.00	1	1	1	0	1	-1	0	1	1	0	1	-1	1	1	1	0	0	0	10
	30	*79	100	251	*140	*150	*185	*119	146	365	319	5.00	0	1	1	0	1	0	0	-1	0	0	0	0	1	1	0	1	0	0	6
	31	160	258	*257	51	114	*320	145	198	181	*121	7.00	1	-1	1	0	1	1	1	1	1	0	0	-1	1	1	1	0	0	0	10
	32	*89	97	164	49	65	52	92	113	131	*132	8.00	1	0	1	0	-1	0	1	1	1	0	1	-1	-1	1	1	1	0	1	10
	33	97	200	*294	173	445	394	187	106	101	565	9.00	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	0	1	16
	34	*83	*140	115	92	*131	94	*77	86	*206	*121	4.00	1	1	1	0	1	0	1	1	0	0	1	0	0	0	1	1	0	0	9
	35	133	118	196	150	*200	*219	89	110	174	*312	7.00	0	1	1	1	0	0	1	1	1	1	1	0	1	1	1	0	0	1	12
	36	58	59	*222	74	*211	141	102	128	270	*221	7.00	1	1	1	1	-1	1	1	1	1	0	1	0	1	1	1	1	0	0	13
	37	87	89	*80	35	*128	77	80	100	*92	*88	6.00	1	1	1	0	1	-1	0	1	0	0	1	0	0	1	0	1	0	0	8
	38	*116	*233	*269	*38	*447	*10	*418	12	*572	*17	1.00	-1	0	1	0	1	0	1	1	1	0	1	0	1	0	0	0	0	0	7
Correct	Median	132.50	132.00	211.00	83.00	137.00	158.50	108.00	128.00	164.00	311.00	7.00																	Median		10
	Average	132.23	141.36	210.50	89.54	176.33	174.73	127.71	134.61	165.73	354.50	6.65																	Average		9.25
All	Median	110	136	201.5	93	174	180.5	120	120.5	177.5	222.5																				
	Average	117.85	143.65	194.35	96.40	195.65	191.75	151.85	137.25	194.25	252.50																				

Figure 14: Data and descriptive statistics.

8. APPENDIX: DATA

For reproducibility of the experiment and the analysis, we report the collected data in Figure 14. For each task (Columns 3-12), we show the amount of time in seconds required for each subject to complete the task (Lines 4-21 for the RP group and Lines 26-45 for the OO group). Values with an asterisk "*" indicate a wrong answer. We show descriptive statistics for each tasks, considering all answers (Lines 24-25 and 48-49) and correct answers only (Lines 22-23 and 46-47). Column 13 reports the amount of correct answers (Score) for each subject. Columns 14-31 show the results of the preliminary tasks to asses subjects skills. 1 denotes correct answers, -1 represents wrong answers, 0 indicates a *Don't know* answer. Column 32 displays the skills results for each subject. Column 32 also reports the median and the average for the skills.

Figure 15 shows some metrics for the applications used in the tasks including size as non-comment lines of code counted with CLOC[4]. For the RP version: Number of signals/vars, number of events and number of handlers. For the OO version: Number of handlers and number of observables.

Task	**RP**	**OO**	**RP**			**OO**	
	LOCs	**LOCs**	**sig/vars**	**events**	**hdlr**	**obs**	**hdlr**
T1	9	24	6	0	0	6	5
T2	10	28	4	0	0	4	4
T3	9	38	6	0	0	6	8
T4	12	31	7	0	0	7	8
T5	46	54	5	4	1	5	8
T6	41	49	5	4	1	5	8
T7	38	47	4	2	1	4	5
T8	32	43	7	2	0	2	2
T9	42	49	5	2	1	6	4
T10	61	82	6	6	1	7	7

Figure 15: Main metrics for the tasks.

[4]http://cloc.sourceforge.net/

9. REFERENCES

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, Aug. 2013.

[3] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. ESOP'06, pages 294–308. Springer-Verlag, 2006.

[4] B. Cornelissen, A. Zaidman, and A. van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Trans. Softw. Eng.*, 37(3):341–355, May 2011.

[5] C. L. Corritore and S. Wiedenbeck. Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *Int. J. Hum.-Comput. Stud.*, 50(1):61–83, Jan. 1999.

[6] F. Détienne. *Software Design—cognitive Aspects*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[7] M. Di Penta, R. E. K. Stirewalt, and E. Kraemer. Designing your next empirical study on program comprehension. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, pages 281–285, 2007.

[8] C. Elliott. Functional implementations of continuous modeled animation. In *Proceedings of the 10th International Symposium on Principles of Declarative Programming*, PLILP '98/ALP '98, pages 284–299, London, UK, UK, 1998. Springer-Verlag.

[9] C. Elliott and P. Hudak. Functional reactive animation. ICFP '97, pages 263–273. ACM, 1997.

[10] B. Ellis, J. Stylos, and B. Myers. The factory pattern in api design: A usability evaluation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 302–312, Washington, DC, USA, 2007. IEEE Computer Society.

[11] 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 *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 51–60, 2011.

[12] 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 *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 759–768, New York, NY, USA, 2008. ACM.

[13] B. N. Freeman-Benson. Kaleidoscope: mixing objects, constraints, and imperative programming. OOPSLA/ECOOP '90, pages 77–88, New York, NY, USA, 1990. ACM.

[14] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2000.

[15] S. Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 22–35, New York, NY, USA, 2010. ACM.

[16] S. Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. *SIGPLAN Not.*, 45(10):22–35, Oct. 2010.

[17] S. Hanenberg, S. Kleinschmager, and M. Josupeit-Walter. Does aspect-oriented programming increase the development speed for crosscutting code? an empirical study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 156–167, Washington, DC, USA, 2009. IEEE Computer Society.

[18] M. Hoppe and S. Hanenberg. Do developers benefit from generic types?: An empirical comparison of generic and raw types in java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 457–474, New York, NY, USA, 2013. ACM.

[19] 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*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[20] N. Juristo and A. M. Moreno. *Basics of Software Engineering Experimentation*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[21] 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 *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '11, pages 15–24, New York, NY, USA, 2011. ACM.

[22] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 361–370, New York, NY, USA, 2007. ACM.

[23] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 185–194, New York, NY, USA, 2010. ACM.

[24] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[25] J. Liberty and P. Betts. *Programming Reactive Extensions and LINQ*. Apress, Berkely, CA, USA, 1st edition, 2011.

[26] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.

[27] I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. In G. Castagna, editor, *ECOOP 2013 - Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 707–731. Springer Berlin Heidelberg, 2013.

[28] 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 *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 683–702, New York, NY, USA, 2012. ACM.

[29] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper,

M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. OOPSLA '09, pages 1–20. ACM, 2009.

[30] 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. *IEEE Trans. Softw. Eng.*, 23(6):347–365, June 1997.

[31] R. Newton, G. Morrisett, and M. Welsh. The Regiment Macroprogramming System. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 489–498, 2007.

[32] F. Olivero, M. Lanza, M. D'ambros, and R. Robbes. Tracking human-centric controlled experiments with biscuit. In *Proceedings of the ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '12, pages 1–6, New York, NY, USA, 2012. ACM.

[33] V. Pankratius, F. Schmidt, and G. Garreton. Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 123–133, 2012.

[34] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295–341, 1987.

[35] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, Oct. 2000.

[36] J. Quante. Do dynamic object process graphs support program understanding? - a controlled experiment. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 73–82, June 2008.

[37] V. Ramalingam and S. Wiedenbeck. An empirical study of novice program comprehension in the imperative and object-oriented styles. In *Papers presented at the seventh workshop on Empirical studies of programmers*, ESP '97, pages 124–139, New York, NY, USA, 1997. ACM.

[38] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 255–265, Piscataway, NJ, USA, 2012. IEEE Press.

[39] G. Salvaneschi, J. Drechsler, and M. Mezini. Towards distributed reactive programming. In *Coordination Models and Languages*, pages 226–235. Springer, 2013.

[40] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. AOSD, To appear, 2014.

[41] G. Salvaneschi and M. Mezini. Reactive behavior in object-oriented applications: An analysis and a research roadmap. In *AOSD'2013*, 2013.

[42] M.-A. Storey. Theories, tools and research methods in program comprehension: Past, present and future. *Software Quality Control*, 14(3):187–208, Sept. 2006.

[43] J. Stylos and S. Clarke. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 529–539, Washington, DC, USA, 2007. IEEE Computer Society.

[44] J. Stylos and B. A. Myers. The implications of method placement on api learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 105–112, New York, NY, USA, 2008. ACM.

[45] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 551–560, New York, NY, USA, 2011. ACM.