Interfaces for Modular Reasoning in Context-Oriented Programming

Paul Leger
Universidad Católica del Norte
Chile
pleger@ucn.cl

Hidehiko Masuhara
Tokyo Institute of Technology
Japan
masuhara@acm.org

Ismael Figueroa
Universidad de Valparaíso
Chile
ismael.figueroa@uv.cl

ABSTRACT

Different activation mechanisms for Context-Oriented Programming (COP) like implicit activations have been proposed, increasing COP opportunities to be applied in real scenarios. However, activation mechanisms and base code definitions are insufficiently decoupled, as conditionals to activate layers require base code variable references. This hinders reuse, evolution, and modular reasoning of COP and base code, and therefore, uses of COP in real scenarios. This paper proposes interfaces, which are shared abstractions to communicate activation mechanisms and base code in a decoupled manner. Using these interfaces, an object can exhibit its internal state and behaviors, and conditionals use them to (de)activate layers. As layers are planned to be (re)used in different applications, developers can use interfaces to overcome the incompatibility between values exposed by a particular base code and values required by a layer. In addition, as a layer is a plain object, it can use an interface to exhibit the conditional evaluation of its activation to other layers to resolve conflicts among activations of layers. We apply this proposal to implicit activations in which evaluations of conditionals implicitly (de)activate layers. Finally, we illustrate the benefits of this proposal through RI-JS, a practical JavaScript library that currently supports interfaces, reactive activations (implementation variant for implicit activations), global and dynamic deployment, enter and exit transition processes, and partial methods.

KEYWORDS

Interfaces, Activation Mechanisms, Context-Oriented Programming, Reactive Programming, JavaScript

ACM Reference Format:
Paul Leger, Hidehiko Masuhara, and Ismael Figueroa. 2020. Interfaces for Modular Reasoning in Context-Oriented Programming. In *12th International Workshop on Context-Oriented Programming and Advanced Modularity (COP'20), July 21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3422584.3423152

1 INTRODUCTION

Because of the proliferation of diverse technological devices, such as notebooks, smartphones, and wearables [9], there is a clear trend in the software industry towards constructions of systems that adapt their behaviors at runtime according to an identified context [2]. Context-Oriented Programming (COP) [20], through *layers*, allows developers to implement the context identification and adaptations in a modular manner. Since this programming approach was presented in 2005 [12][1] with partial methods [5, 20], COP researchers have added to layers diverse mechanisms for scope [23, 29], transition [23, 24], and widely for activation [4, 16, 20, 22–25, 32, 47, 49].

A mechanism of activation is used to determine whether a context is identified or not. When a context is identified, its associated layer is activated. In COP, we can find different activation mechanisms which can be imperative [16, 20], event-based [23], or implicit [22, 25, 32, 47, 49]. The last two mechanisms allow developers to declare with a conditional when a layer must be activated.

One modularity issue, that is still present in COP, is coupling between the conditional declaration in an activation and base code (Figure 1a-b). This is so because developers *implicitly* depend on variable references or method invocations in base code to declare a conditional, making fragile programs, hindering the reuse, evolution, and modular reasoning of layers in real applications with COP. Changes in base code or a conditional declaration may spuriously (de)activate layers. A similar problem has been identified in other areas like aspect-oriented programming [27], which is known as *fragile pointcuts* [19, 40]. As Figure 1b shows and the example used in [22], this kind of coupling is also present when the activation of a layer depends on the activation status of another one.

This paper proposes interfaces to decouple base code from conditionals to activate layers. In this proposal, developers can use interfaces to exhibit part of the internal state (*i.e.,* field values) and behavior (*i.e.,* method executions) of an object, and conditional-based activation mechanisms declare their predicates using *explicitly* identifiers available in this kind of interfaces (see Figure 1c). To ease the reuse of the same layer in different base code (*e.g.,* smartphones and tablets), interfaces allow developers to define expressions that reconcile conditional requirements with what objects, of a specific base code, expose. As a layer is also an object, it can use an interface to decouple a dependency with another layer (Figure 1d). Although our proposal should work for any conditional-based activation mechanisms, we apply interfaces to implicit activations.

We illustrate the benefits of interfaces through RI-JS, a practical JavaScript library that currently supports global and dynamic deployment, enter and exit transition processes that are executed around a layer activation, partial methods with `proceed` support. RI-JS uses reactive activations, an implementation variant for implicit

[1] In the Pervasive Computing area [38], context-oriented programming was used previously without focusing on programming language aspects [15, 26]. Rather, articles in this area referred to requirements and features of a system that depends on a context.

COP'20, July 21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8144-4/20/07…$15.00
https://doi.org/10.1145/3422584.3423152

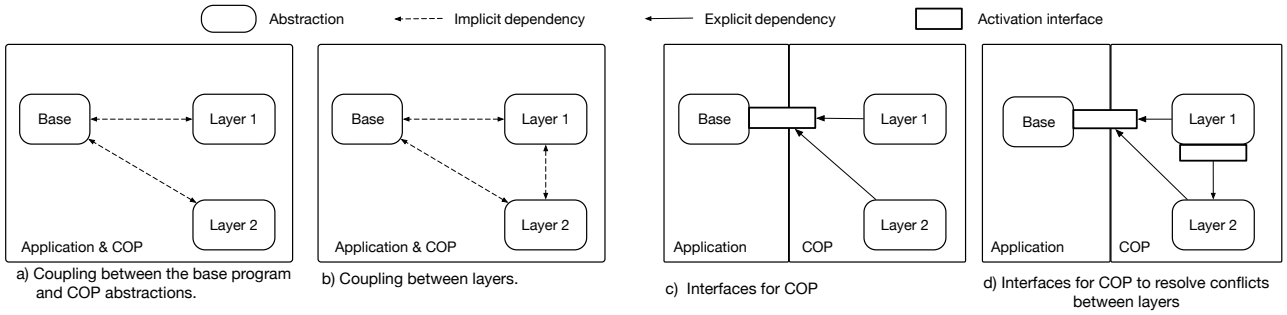

Figure 1: Dependencies between *a*) layers and base code, and *b*) layers. Using interfaces for COP to *c*) decouple layers and base code, and *d*) layers.

activations where a conditional is composed of signals, *i.e.,* time-varying values in Reactive Programming (RP) [11].

The rest of this paper is structured as follows. Section 2 motivates the use of interfaces in COP. Section 3 presents our proposal and Section 4 introduces a concrete implementation for JavaScript. Section 5 discusses related work. Section 6 concludes this paper.

Availability. The current RI-JS implementation is on http://github.com/pleger/rai-js. In addition, the example presented in this paper is available at http://pleger.cl/sites/raijs [35] (revision 4f5d7034). Our proposal currently supports Nodejs (v13.6.0) [31], Google Chrome (v83.0.4103.61) [17] and Mozilla Firefox (v77.0) [14] browsers without the need for an extension.

2 ACTIVATION MECHANISMS IN CONTEXT-ORIENTED PROGRAMMING

This section starts introducing Context-Oriented Programming (COP) [20] and its mechanisms to activate layers. Finally, we exemplify a coupling issue that appears when conditional-based activation mechanisms are used.

2.1 Layers

In systems that adapt their behaviors at runtime according to an identified context, the implementation of context identification and behavior adaptations crosscut with several other concerns of the system [20]. Although there are different visions how to achieve COP [45], COP researchers mainly work on an abstraction named *layer* [12]. This abstraction is used to modularly implement these two concerns into a single module. In object-oriented languages like JavaScript, a layer is composed of *partial methods* that vary the behavior of their original methods when this layer is activated.

Following the example shown in [25], we illustrate the use of layers. Consider a smartphone application that changes its layout according to two screen orientations: landscape and portrait. The application must adapt their behaviors to support these two previous orientations (*e.g.,* icon distribution in the screen), implying that there are crosscutting concerns that can be modularized with layers. As figure 2 shows, landscape and portrait are layers that vary the behavior of the method draw in playerView. Note these two layers should not be activated at the same time due to an unexpected application behavior; we discuss this issue in the next section.

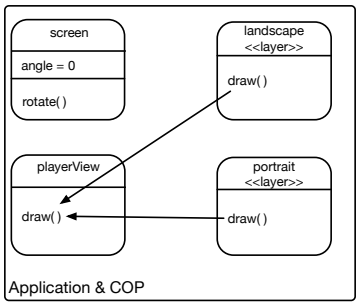

Figure 2: Two layers where each one contains a partial method of draw**.**

2.2 Activation Mechanisms

An activation mechanism establishes how and when a layer must be activated. In the body of literature on COP, we can find different kinds of mechanisms to activate layers: *Imperative* [16, 20] are which activate layers with the execution of a statement like with, *implicit* [7, 22, 25, 32, 47, 49] activate a layer when a developer-defined conditional is satisfied, *event-based* [6, 23] are which use the matching of events and a conditional to activate layers, and combinations of them [4, 24]. Using the smartphone application example, Figure 3 illustrates the three mechanisms to activate the landscape layer when the rotation angle of the smartphone exceeds a threshold. Unlike the imperative activation mechanism, activation mechanisms use a conditional that must be satisfied to activate a layer; in this example is screen.angle > THRESHOLD.

If we include the portrait layer as well, we need to deactivate one layer when another is activated. To resolve activation conflicts between layers, activation mechanisms provide *ad-hoc* constructs. In the ServalCJ extension [25], for example, developers have to use the construct when with the name of the layer:

```
activate landscape if(screen.angle > THRESHOLD);
activate portrait when !landscape;
```

Activation and Scope. In this paper, we make a subtle difference between *when a layer is active* and *when a layer has to be applied* (scope). In other words, a layer may keep active although its scope does not apply in a certain portion of a program execution. This difference is useful in two examples. First, when we need a *persistent*

```
//imperative (contextJ)
if (screen.angle > THRESHOLD) {

  //applying layer on base code
   with(landscape) {
    playerView.draw();
  }
}
```

```
//event-based (JCop)
on(* playerView.draw()) &&
when (screen.angle > THRESHOLD) {
     with(landscape);
}
```

```
//implicit (ServalCJ extension for COP)
activate landscape if (screen.angle > THRESHOLD);
```

Figure 3: Activating the landscape layer using different activation mechanisms.

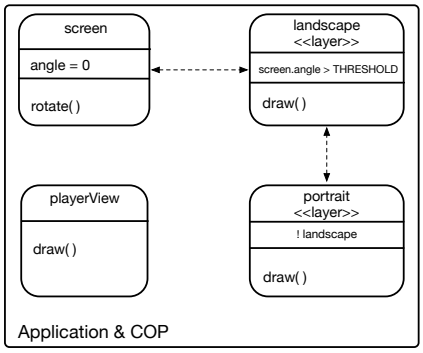

Figure 4: Dependencies between a) landscape and screen, and b) landscape and portrait.

layer, that is keeping information in an active layer (*e.g.,* the rotation angle that activated the landscape layer). Second, when we need not apply an active layer in some particular place without an execution of the enter and exit transition process (*e.g.,* a video game that does not support the landscape mode). Taking into account this observation, we consider *per-object* (*i.e.,* lexical region) and *per-control-flow* (*i.e.,* dynamic extent) as the *scope* of an already activated layer, and not as a different activation mechanism. Researchers have already discussed different scope strategies (combinations of lexical and dynamic scope) in programming languages [42], and particularly in areas related to context-oriented programming [41, 43, 44].

2.3 Coupling in Activation Mechanisms

Unlike imperative activation mechanisms, remaining two mechanisms use a developer-defined conditional that must be satisfied to activate layers. This conditional is composed of base code variable references or method executions, coupling layers and base code. Figure 4 illustrates the dependency that appears between the base code and layers in the smartphone example through the angle field of screen. This dependency can also appear between layers, for example, potrait depends on a specific name of another layer. This kind of coupling, which is studied in other areas like aspect-oriented programming [8, 19, 40], makes fragile programs, hindering the reuse, evolution, and modular reasoning. Next, we illustrate these three issues:

Reuse. Consider we now need to reuse the landscape and portrait layers in a *tablet* application, whose base code may differ from the smartphone application. Even if the tablet application contains the screen object with similar behavior, any small change in this object can make difficult the reuse of both layers. For example, the screen.angle rotation is expressed in radians in the tablet base code and degrees in the landscape layer, implying both layers have unexpected behaviors. Therefore, if developers need to use these layers, landscape must have a particular modification.

Evolution. A smartphone application is frequently being updated. For example, if the smartphone now supports a 3d rotation, screen may add the angleZ field and rename angle to angleXY, implying landscape cannot be activated or, even worse, this application does not work anymore. Hence, developers should evolve the smartphone application with the layers together due to this coupling.

Modular reasoning. Developers of base code and layers are commonly different for two reasons at least. First, as mentioned in [8], development in advanced paradigms like COP requires a high level of expertise, being done by specialized developers, leading to different roles. Second, these developers should implement layers for diverse devices like smartphones and tablets. As a consequence, for example, a base code developer can get unexpected behavior if the value of screen.angle is varied to animate a window (*e.g.,* a crash in a racing video game) because of the temporary activation of a layer. As the previous example shows, an implicit dependency between base code and layers obscure the software development reasoning.

3 INTERFACES FOR COP

This section presents our proposal to decouple base code and conditional-based activation mechanisms.

Using the core idea behind the Open Modules [1] and related proposals [8, 18, 39], we propose a shared interface abstraction that communicates base code variable references or method executions with conditionals used in most activation mechanisms (Section 2.2). Our proposed interfaces allow developers to exhibit field values and method executions (with returns) from any object, which are used in the conditional declaration to (de)activate a layer. In addition, these interfaces also accept expressions as values to exhibit to address the reuse issue (Section 2.3). For example, consider the example of the angle rotation that is exhibited in radians but the layer requires in degrees, the expression rotation: angle*180/PI can be used in the interface to reconcile both implementations. Likewise, as a layer is also an object, its conditional evaluation can be exhibited to other layers to resolve activation conflicts. With this kind of interface, developers can replace the *implicit* dependency between base code and layers with an *explicit* dependency between this interface and base code/layers. We apply interfaces to implicit activation mechanisms, but they may be applied to another conditional-based activation mechanism.

Figure 5 illustrates the use of our proposal in the smartphone application example. The screen object exhibits angle as rotation and the landscape layer uses this alias to declare its conditional. Likewise, this layer exhibits its conditional evaluation as noPriorityLayer which is used by portrait. Using interfaces, base code and layers can be reused, reasoned, and evolved in an independent manner. To evidence theses benefits that bring interfaces, Figure 6 shows how the landscape layer is used in a tablet application (Section 2.3):

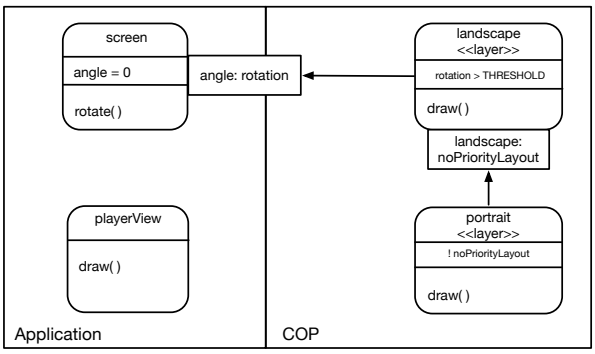

Figure 5: Interfaces to decouple layers and base code, and layers themselves.

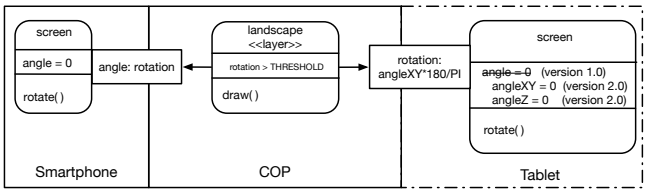

Figure 6: Using interfaces, the same layer is used in two different scenarios.

Reuse. We can observe that the landscape layer is used in a second scenario without modification.

Evolution. The tablet application is updated from version 1 to 2, and a base code developer has only to update what variable needs to be exhibited now (angleXY). In addition, this variable also changes its kind of value (degrees to radians), and interfaces allow the developer to adapt it to satisfy the layer requirements.

Modular reasoning. As we see in the previous point, a base code developer only worries about updates that occur in base code, easing modular reasoning.

4 RI-JS

We provide a concrete and practical implementation of this proposal through RI-JS [35], a JavaScript library that currently supports interfaces, global and dynamic deployment, enter and exit activation transition processes, and partial methods with proceed support. RI-JS uses reactive activations, an implementation of implicit activations where conditionals are composed of signals, *i.e.,* time-varying values in Reactive Programming (RP) [11, 13]. If a signal changes its value, the entire conditional is evaluated to determine whether a layer is activated or not. We introduce RI-JS through an implementation of the smartphone application example.

Base code and layers. Similar to SignalJ [21], RI-JS allows developers in JavaScript to create and assign signal values, *e.g.,* angle. The layers landscape and portrait are created with an activation conditional and an enter transition. Conditionals are signal expressions, also known as *composite signals* [25]. The enter transition is executed when its layer is activated; in this implementation, both layers rotate the screen when are activated. In addition, developers can add an exit transition, which is executed when the layer is deactivated. Both enter and exit transitions are optional for the creation of a layer.

```
//Base code
let screen = {
  angle: new Signal(0),
  rotate: function() {...}
};

let playerView = {
  draw: function() {...}
};

//layers
let landscape = {
  conditional: new CompSignal("rotation > THRESHOLD"),
  enter: function() { screen.rotate(); }
};

let portrait = {
  conditional: new CompSignal("!noPriorityLayout"),
  enter: function() { screen.rotate(); }
};
```

Partial methods. To create a layer, COP developers do not require to include partial method implementations. This is so because base code developers *do know* how the application must change when a layer is activated. For this reason, partial methods are added after creating a layer. In this implementation of the smartphone example, landscape and portrait layers vary the draw method in playerView.

```
RI.addPartialMethod(landscape,playerView,"draw",function() {...});
RI.addPartialMethod(portrait,playerView,"draw",function() {...});
```

Interfaces. Any object can exhibit its reactive field values or reactive method executions (*i.e.,* composite signals) and give an alias that should be used by layers. The piece of code below shows that screen and landscape exhibit a reactive field value and a reactive method execution respectively.

```
RI.exhibit(screen,{rotation: "angle"});
RI.exhibit(landscape,{noPriorityLayout: "conditional"});
```

Dynamic and global deployment. The current implementation of RI-JS supports global and dynamic (un)deployments of layers. In the smartphone example, we deploy two layers: landscape and portrait. The layer activations order follows the order of changes in exposed reactive values. For example, when landscape varies the result of its conditional, RI-JS triggers the portrait conditional evaluation.

```
RI.deploy(landscape);
RI.deploy(portrait);
```

As the previous implementation shows, RI-JS does not extend the *syntax* of JavaScript because it is provided as a library, increasing its potential use in existing JavaScript applications. For the same reason, RI-JS does not require to transform a piece of code to work.

4.1 Performance

The main goal of this paper focuses on expressiveness to decouple base code and layers. For this reason, we have not sacrificed any potentially valuable feature in RI-JS on the basis of its expected cost. Nevertheless, we are interested in making RI-JS in a practical implementation for JavaScript developers in context-oriented programming. Therefore, we carried out a preliminary performance

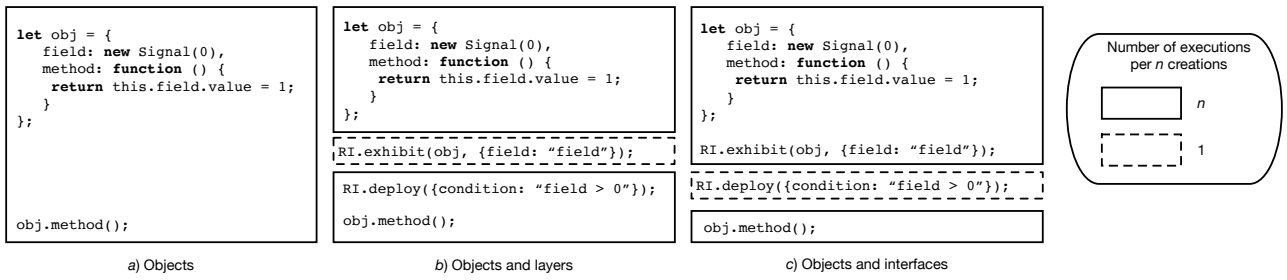

Figure 7: Three pieces of code used for the RI-JS performance evaluation: *a*) Only creates objects, *b*) creates objects and deploys layers, and *c*) creates objects and interfaces.

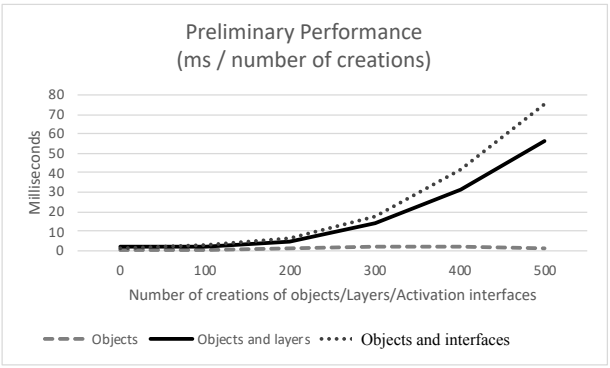

Figure 8: A preliminary performance evaluation about the current implementation of RI-JS.

evaluation. For this evaluation, we used Nodejs (v13.6.0) [31] on Macbook Pro (2017), 3.1 GHz Dual-Core Intel Core i5 with 8GB of RAM running macOS Catalina, and the RI-JS GitHub revision was `4f5d703` (June 3, 2020).

Figure 7 shows the three JavaScript programs executed to evaluate RI-JS performance. The first program only creates until 500 objects (Figure 7a), the second one creates/deploys until 500 objects/layers with one interface (Figure 7b), and the last one creates until 500 objects/interfaces with one layer (Figure 7c). Using these three programs, Figure 8 shows a chart line that represents a preliminary performance evaluation of the current RI-JS implementation: average time of 100,000 executions for each number of creations (x-axis). Although RI-JS does not observe every statement trying to activate a layer or execute a partial method, the current implementation is clearly slower than a program without RI-JS. In addition, an incremental use of interfaces affects performance. This could be due to the propagation of data stream in an unoptimized and naive reactive programming implementation [3]. In JavaScript, we can find proposals like Flapjax [30] and RxJS [36] that have also faced performance issues to extend this language with reactive programming.

5 RELATED WORK

This section briefly reviews three kinds of proposals: conditional-based activation mechanisms, group-based adaptation mechanisms, and interfaces in aspect-oriented programming (AOP) [27]. The last two kinds of proposals are important because group-based adaptations modify object behaviors when certain conditionals are satisfied, and interfaces for COP intend to address coupling issues that appear in activation mechanisms using AOP ideas.

Conditional-based activation mechanisms. As mentioned in Section 2, implicit and event-based activation mechanisms use expressions that evaluate if certain requirements are satisfied to activate a layer. However, these two mechanisms use different strategies to evaluate these expressions. For instance, *a*) implicit activation in ContextPy [48] verifies if a conditional is satisfied every time a partial method may be called, and *b*) event-based activation in EventCJ [23] uses pointcuts from aspect-oriented programming to activate a layer when certain join points are matched. However, these kinds of activations make an implicit dependency through variable references and method calls/executions in base code with layers. In our proposal, conditionals only use variables that are available in an interface, where these variables do not require a *bijective* relation to base code. For example, a variable in our interfaces can correspond to the return of a method execution or a developer-defined expression.

Group-based behavior adaptation mechanisms. In this kind of mechanisms such as Predicated Generic Functions [46] and Interface Mediations [33], an object adapts its current behavior while belongs to a group, and the membership depends on the object internal state [34]. Whereas group-based behavior adaptations (de)active the group membership of *one* object based on its internal state, interfaces work as the activator of a group based on internal states of *one or more* objects. Hence, interfaces and group-based behavior adaptations can be considered as complementary because the former (de)actives groups and the latter (de)activates the membership of objects to these groups.

Interfaces in aspect-oriented programming. In 2005, Aldrich [1] discusses issues that arise from the implicit dependency between base code and pointcuts. As an example, the *fragile pointcut* issue [19, 40] refers to any change in base code that (silently) generates spurious advice executions or unexpectedly disables them. To address these issues, the author proposes *open modules* to allow developers to explicitly specify what join points of a program execution can be advised by an aspect. Later, a number of related proposals have been published [8, 18, 39]. For example in Join Point Interfaces [8], base code developers must *explicitly* establish what

join points are exhibited by objects of a class; allowing the modular reasoning of the application of an aspect. Our proposal follows the same line of these proposals for conditional-based activation mechanisms in COP. In addition, interfaces allows developers to exhibit expressions composed of variables or method executions that come from different objects.

6 CONCLUSIONS AND FUTURE WORK

There is a strong demand for systems that can adapt their behaviors according to an identified context nowadays. COP aims to develop these systems in a modular manner. However, conditional-based activation mechanisms are not so modular because coupling of base code and conditionals declarations. This paper has proposed interfaces that help developers decouple these conditional declarations from base code through a shared interface. We applied our proposal to implicit activation mechanisms and provided a concrete implementation for JavaScript, named RI-JS. While our proposal could be applied to other activation mechanisms, we think its major challenges are related to:

Robustness. We help to decouple base code and layers. However, this kind of decoupling is not completely *robust*[2] because conditionals require a set of variables from (potentially different) objects that may not exhibit them. In the current version of our model, if one of the variables used in a conditional is not exhibited by any object, the conditional is evaluated to false and its associated layer is not activated at that moment. The previous point means that base code developers have to be concerned to satisfy conditional requirements, breaking the explicit dependency between base code and layers. This is because of the dynamic nature of our proposal, meaning that an object can make the decision of what exhibit at runtime. If we move on to a static approach, a conditional checker for these interfaces may help to address this issue.

Case study. Researchers in systems that depend on the context have mentioned the need for specialized programming paradigms [2, 37]. To evaluate the benefits of our proposal in a real scenario, we plan to use RI-JS to develop this kind of system. A potential case study where we can apply these interfaces is presented in [28], in which authors present a Web application that adapts its behavior to the particular learning rhythm of a primary school student in math.

REFERENCES

[1] Jonathan Aldrich. 2005. Open Modules: Modular Reasoning about Advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005) (Lecture Notes in Computer Science)*, Andrew P. Black (Ed.). Springer-Verlag, Glasgow, UK, 144–168.

[2] Unai Alegre, Juan Carlos Augusto, and Tony Clark. 2016. Engineering context-aware systems and applications: A survey. *Journal of Systems and Software* 117 (July 2016), 55–83.

[3] Edward Amsden. 2011. *A Survey of Functional Reactive Programming Concepts, Implementations, Optimizations, and Applications.* Technical Report. Rochester Institute of Technology.

[4] Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. 2014. Unifying multiple layer activation mechanisms using one event sequence. In *Proceedings of 6th International Workshop on Context-Oriented Programming, COP 2014*. Uppsala, Sweden, 1–6.

[2] "*A system or component that can function correctly in the presence of invalid inputs [uses] or stressful environmental conditions.*"(IEEE Standards Committee [10]).

[5] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. 2011. ContextJ: Context-oriented Programming with Java. *Computer Software* 28, 1 (Feb. 2011), 272–292.

[6] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. 2010. Event-Specific Software Composition in Context-Oriented Programming. In *Software Composition*, Benoît Baudry and Eric Wohlstadter (Eds.). Malaga, Spain, 50–65.

[7] Engineer Bainomugisha, Jorge Vallejos, Coen De Roover, Andoni Lombide Carreton, and Wolfgang De Meuter. 2012. Interruptible Context-dependent Executions: A Fresh Look at Programming Context-aware Applications. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Tucson, Arizona, USA, 67–84.

[8] Eric Bodden, Éric Tanter, and Milton Inostroza. 2014. Join Point Interfaces for Safe and Flexible Decoupling of Aspects. *ACM Transactions on Software Engineering and Methodology* 23, 1 (Feb. 2014), 1–41.

[9] Marie Chan, Daniel Estève, Jean Fourniols, Christophe Escriba, and Eric Campo. 2012. Smart wearable systems: Current status and future challenges. *Artificial Intelligence in Medicine* 56, 3 (Nov. 2012), 137–156.

[10] IEEE Standards Committee et al. 1990. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990* (Dec. 1990), 1–84.

[11] Gregory Harold Cooper. 2008. *Integrating Dataflow Evaluation into a Practical Higher-order Call-by-value Language.* Ph.D. Dissertation. Providence, RI, USA.

[12] Pascal Costanza and Robert Hirschfeld. 2005. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *Proceedings of the 2005 Symposium on Dynamic Languages*. San Diego, USA, 1–10.

[13] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (IFCP)*. Amsterdam, The Netherlands, 263–273.

[14] Firefox. 2020. A free and open-source Web browser. (2020). https://www.mozilla.org (v77.0).

[15] M. L. Gassanenko. 1998. Context-Oriented Programming. In *euroFORTH*. Schloss Dagstuhl, Germany.

[16] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. 2011. Subjective-C: Bringing Context to Mobile Platform Programming. In *Proceedings of the Third International Conference on Software Language Engineering*. Eindhoven, The Netherlands, 246–265.

[17] Google Chrome. 2020. A free and open-source Web browser. (2020). https://www.google.com/chrome (v83.0.4103.61).

[18] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. 2006. Modular Software Design with Crosscutting Interfaces. *IEEE Software* 23, 1 (2006), 51–60.

[19] Kris Gybels and Johan Brichau. 2003. Arranging Language Features for More Robust Pattern-based Crosscuts. In *International Conference on Aspect-Oriented Software Development (AOSD)*. Boston, USA, 60–69.

[20] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-oriented Programming. *Journal of Object Technology* 7, 3 (March-April 2008), 125–151.

[21] Tetsuo Kamina and Tomoyuki Aotani. 2018. Harmonizing Signals and Events with a Lightweight Extension to Java. *The Art, Science, and Engineering of Programming* 2, 3 (March 2018), 1–29.

[22] Tetsuo Kamina and Tomoyuki Aotani. 2019. TinyCORP: A Calculus for Context-Oriented Reactive Programming. In *Proceedings of the Workshop on Context-Oriented Programming (COP)*. London, UK, 1–8.

[23] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2011. EventCJ: A Context-oriented Programming Language with Declarative Event-based Context Transition. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*. Porto de Galinhas, Brazil, 253–264.

[24] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2016. Generalized Layer Activation Mechanism for Context-Oriented Programming. *LNCS Transactions on Modularity and Composition* 9800 (2016), 123–166.

[25] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2017. Push-based reactive layer activation in context-oriented programming. In *Proceedings of the 9th International Workshop on Context-Oriented Programming (COP)*. Barcelona, Spain, 17–22.

[26] Roger Keays and Andry Rakotonirainy. 2003. Context-oriented programming. In *Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile*. San Diego, USA, 9–16.

[27] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C.V. Lopes, C. Maeda, and A. Mendhekar. 1996. Aspect Oriented Programming. In *Special Issues in Object-Oriented Programming*. Max Muehlhaeuser (general editor) et al.

[28] Paul Leger, Grecia Gálvez, Lino Cubillos, Diego Cosmelli, Milton Inostroza, Éric Tanter, Gina Luci, and Jorge Soto Andrade. 2014. ECOCAM, un sistema computacional adaptable al contexto para promover estrategias de cálculo mental: características de su diseño y resultados preliminares. *Revista Latinoamericana de Investigación en Matemática Educativa* 17, 1 (March 2014), 33–58.

[29] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. 2011. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Science of Computer Programming* 76, 12 (Dec. 2011), 1194–1209.

[30] Leo Meyerovich, Arjun Guha, Jacob Baskin, Gregory Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. Orlando, Florida, USA, 1–20.

[31] NodeJS. 2020. A JavaScript runtime built for the server side. (2020). https://nodejs.org (v13.6.0).

[32] Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2017. The declarative nature of implicit layer activation. In *Proceedings of the International Workshop on Context-Oriented Programming (COP)*. London, UK, 7–16.

[33] Patrick Rein, Robert Hirschfeld, Stefan Lehmann, and Jens Lincke. 2016. Compatibility Layers for Interface Mediation at Run-Time. In *Companion Proceedings of the 15th International Conference on Modularity*. Málaga, Spain, 113–118.

[34] Patrick Rein, Stefan Ramson, Jens Lincke, Tim Felgentreff, and Robert Hirschfeld. 2017. Group-Based Behavior Adaptation Mechanisms in Object-Oriented Systems. *IEEE Software* 34, 6 (Nov. 2017), 78–82.

[35] RI-JS Website. 2019. A COP practical library that uses Interfaces for COP in JavaScript. (April 2019). http://pleger.cl/sites/raijs

[36] RxJS. 2018. Reactive Extensions for JavaScript. (2018). https://rxjs.dev

[37] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. 2012. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software* 85, 8 (Aug. 2012), 1801–1817.

[38] M. Satyanarayanan. 2001. Pervasive computing: Vision and challenges. *IEEE Personal Communications* 8, 4 (Aug. 2001), 10–17.

[39] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. 2010. Types and Modularity for Implicit Invocation with Implicit Announcement. *ACM Transactions on Software Engineering and Methodology* 20, 1 (June 2010), Article 1.

[40] Maximilian Stoerzer and Juergen Graf. 2005. Using pointcut delta analysis to support evolution of aspect-oriented software. In *IEEE International Conference on Software Maintenance (ICSM)*. Budapest, Hungary, 653–656.

[41] Éric Tanter. 2008. Expressive Scoping of Dynamically-Deployed Aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD)*. Brussels, Belgium, 168–179.

[42] Éric Tanter. 2009. Beyond Static and Dynamic Scope. In *Proceedings of the 5th ACM Dynamic Languages Symposium (DLS 2009)*. Orlando, FL, USA, 3–14.

[43] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. 2009. Expressive Scoping of Distributed Aspects. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD 2009)*. Charlottesville, USA, 27–38.

[44] Rodolfo Toledo, Paul Leger, and Éric Tanter. 2010. AspectScript: Expressive Aspects for the Web. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD)*. Rennes and Saint Malo, France, 13–24.

[45] David Ungar, Harold Ossher, and Doug Kimelman. 2014. Korz: Simple, symmetric, subjective, context-oriented programming. In *Onward! 2014 - Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Part of SPLASH 2014*. Portland, USA, 113–131.

[46] Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt, and Kim Mens. 2010. Predicated Generic Functions: Enabling Context-Dependent Method Dispatch. In *SC'10: Proceedings of the 9th international conference on Software Composition*. Malaga, Spain, 66–81.

[47] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. 2007. Context-oriented Programming: Beyond Layers. In *Proceedings of International Conference on Dynamic Languages*. Lugano, Switzerland, 143–156.

[48] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. 2007. Context-Oriented Programming: Beyond Layers. In *Proceedings of the International Conference on Dynamic Languages (ICDL 2007)*. 143–156.

[49] Takuo Watanabe. 2018. A Simple Context-Oriented Programming Extension to an FRP Language for Small-Scale Embedded Systems. In *Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-time Composition*. Amsterdam, Netherlands, 23–30.