Unanticipated Debugging with Dynamic Layers

Steven Costiou
Lab-STICC UMR CNRS 6285, Université de Bretagne Occidentale, F-29200 Brest, France
steven.costiou@univ-brest.fr

Mickaël Kerboeuf
Lab-STICC UMR CNRS 6285, Université de Bretagne Occidentale, F-29200 Brest, France
mickael.kerboeuf@univ-brest.fr

Marcus Denker
RMoD, Inria Lille, UMR CNRS 9189, CRIStAL, Université de Lille, France
marcus.denker@inria.fr

Alain Plantec
Lab-STICC UMR CNRS 6285, Université de Bretagne Occidentale, F-29200 Brest, France
alain.plantec@univ-brest.fr

ABSTRACT

To debug running software we need unanticipated adaptation capabilities, especially when systems cannot be stopped, updated and restarted. Adapting such programs at runtime is an extreme solution given the delicate live contexts the debugging activity takes place. We introduce the Dynamic Layer, a construct in which behavioral variations are gathered and activated as a whole set of adaptations. Dimensions of Dynamic Layers activation are reified to allow very fine definitions of layer scopes and a fine grained selection of adapted entities. This paper describes and discusses the Dynamic Layer solution to perform unanticipated runtime debugging. An experimental implementation with the Pharo language is evaluated through a runtime adaptation example.

KEYWORDS

Dynamic behavior adaptation, Dynamic Layers, Runtime debugging

ACM Reference format:
Steven Costiou, Mickaël Kerboeuf, Marcus Denker, and Alain Plantec. 2017. Unanticipated Debugging with Dynamic Layers. In *Proceedings of Programming '17, Brussels, Belgium, April 03-06, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3079368.3079391

1 INTRODUCTION

Autonomous systems or long running applications face unpredicted situations and they cannot adapt if behaviors were not anticipated. Thus appears the need for unanticipated adaptation [9]. The most extreme cases of unanticipated adaptation have been described by Keeney [7] as adaptations for which definitions of where, when, what and how they will be applied remain completely unknown until a problem occurs. Unanticipated adaptation can be hard to achieve and leads to complex difficulties [4]. Debugging a running program under these conditions is an extreme measure. Methodologies and tools must provide the necessary capabilities to investigate and adapt a system at runtime.

As a first step towards extreme debugging, we propose to reuse in a dynamic way the *Layer* concept from Context Oriented Programming [5]. These *Dynamic Layers* group together behavior adaptations and can be defined and applied to a running software. We provide means to express when, where and how a dynamic layer would be active and on which entities. This paper contributes on the following points:

- A Dynamic Layer construct in which behavior adaptations can be dynamically gathered to form a unit of adaptations
- A reification of layer scoping and means to express flexible and dynamic activation scopes
- A set of operators for fine grained object selection, for which layers will provide behavioral variations

The remainder of this paper is organized as follows: section 2 describes our motivation through an illustration example. Section 3 defines the Dynamic Layers and their reified scoping. Section 4 describes the early implementation of Dynamic Layers and section 5 shows an evaluation of the solution. Related works are discussed in section 6 and we conclude this paper in section 7.

2 MOTIVATING EXAMPLE

To illustrate the problem we could imagine a drone flying or moving in some urban environment, for example a postal delivery drone. It can use two systems to navigate, a GPS and wifi. If and when the GPS signal is lost, the drone is programmed to use the wifi instead. If the GPS signal is recovered, the drone stops using the wifi navigation and switches back to the GPS.

However, it is possible that the drone flies into an area where the GPS signal is constantly lost and recovered. That could happen for example in a city with a lot of high buildings and tunnels. If the GPS status changes too often, the constant switching between the two modes would not let enough time for the wifi positioning to be started and used.

Programming '17, Brussels, Belgium
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. 978-1-4503-4836-2/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3079368.3079391

This situation can lead to inefficient route plotting or stall the drone for moments. It could even be lost.

To fix the bug, the system has to be shut offline and investigated. Even if this happens during flight and mission tests, thus still under development, it may not be possible to reproduce the exact conditions under which the bug happened. Debugging the software dynamically could be an interesting solution, yet very extreme considering it means modifying a running software in a flying machine. What we really want is:

- Means to observe and investigate the system at runtime to understand what is happening
- Dynamically adapt the software's behavior to fix the problem

These are typically debug actions and concerns, but they cannot be foreseen until a problem happens, and therefore remain unanticipated. We like to call these kinds of maintenance *unanticipated debugging*, as the situations when they occur are completely unanticipated and the live context of the debug process makes it very delicate. Such extreme debugging raises questions, due to its dynamic and invasive nature:

- How can we dynamically define meaningful sets of adaptations to fit a debug context? What are the dimensions of software in which these adaptations can be expressed?
- How can we express how a set of adaptations will scope into these debug dimensions? What are the entities that can be adapted, and when will they be?

3 UNANTICIPATED DEBUGGING WITH DYNAMIC LAYERS

In this section we discuss the questions raised by unanticipated debugging [1]. We define the dimensions in which layers will live and dynamically evolve to adapt a running software. We propose a way to express how layers expand into these dimensions.

3.1 Dynamic Layers

Dynamic Layers are first class objects that can be defined and modified at runtime. They select a group of objects on which a common set of adaptations will be applied. Once created, a layer is unique and can be accessed by its name:

```
"Creating and activating a new layer"
DynamicLayer define: 'WifiPositioningLayer'.
WifiPositioningLayer activate
```

When a layer is activated, it triggers a set of behavior adaptations. Each adaptation is applied to a dynamic selection of objects, until the layer is deactivated. This period of time within which the layer is active is its scope: a layer is never active outside its activation scope. The selection of adapted entities and the scope of the layer activation are the two debug dimensions in which the layers will stretch to provide dynamic debug strategies.

[1]All source code examples are developed with Pharo Smalltalk

3.2 Debug dimensions

We are interested in particular in two dimensions. The first one is the activation scope of layers, as identified in Context Oriented Programming (COP). A layer is active only within a given scope, and to debug a running system this scope may be unknown until it is needed. As the scope defines when and how adaptations are applied at runtime, we believe it is an important aspect we must provide control for.

The second dimension is the granularity of the behavioral variations brought by a layer. Object-centric debugging [12] advocates for *"objects as the key abstraction"* in the debugging process. Following this point of view, we believe that to be flexible we need a way to express whether a layer will affect a given object, a selection of objects or all objects from the same class, and to change this selection dynamically. Figure 1 illustrates how layers expand into these two dimensions.

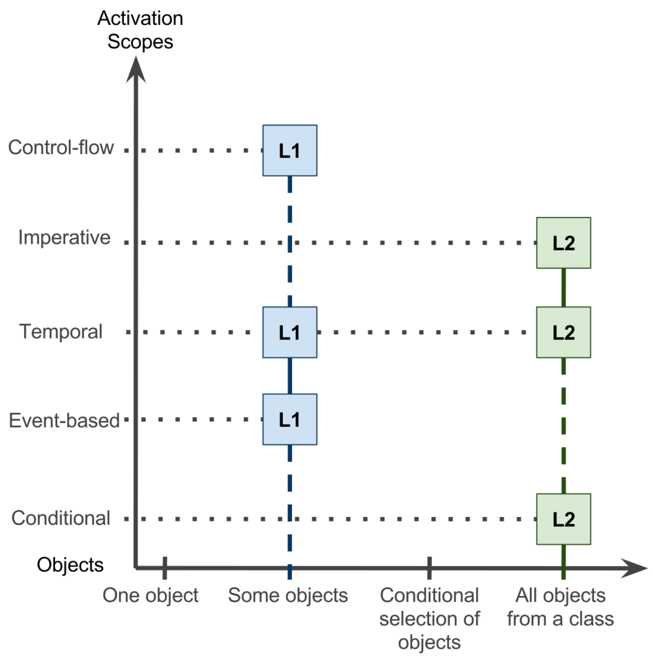

Figure 1: Dynamic Layers Dimensions: we can see two layers L1 and L2 with their selection of adapted objects and their composition of activation scopes.

The following sections describe how we propose to express layers scoping and how they select the entities for which they provide behavioral variations.

3.3 Reified Activation Scopes

We reify layer scoping to provide fine activation scopes that can be dynamically updated at runtime. Activation scopes can be based on four different scoping possibilities [6] existing in COP languages: *Control-Flow* based, *Imperative*, *Event* based or *Implicit* activation. These four scopes are:

- Control-flow: the layer would be active only during the execution of an arbitrary block of code. The scope defines both when a layer activates and when it deactivates.
- Event-based: the layer (de)activates upon events reception.
- Implicit: the layer can be activated if the system or an entity in the system meets a specific constraint (*i.e.* it is a conditional activation).
- Imperative: manual activation of the layer, *e.g.* using an *activate* keyword.

These different kinds of activation scopes already bring some flexibility to a debug process, to choose when and how to activate a layer. We add a fifth kind of activation, which is time scoped: we would like to trigger specific behaviors when debugging the software to specifically bind debug operations to time constraints. This new activation scope is described below:

- Temporal scope: the layer is active for a given duration, from a start time to an end time (possibly *"infinite"*). It also defines when a layer activates and when it deactivates. It can be put in a loop to activate the layer again after an elapsed time.

Reifications of activation scopes are described in figure 2, in which each scope is expressed and can be held as an object. We can see the particular case of the control-flow scope, for which we need to specify a start point and an end point in the source code. These points are abstract syntax tree nodes (ast) that we select manually. The developer should be able to dynamically choose these ast nodes between which the layer activation would be scoped. This dynamic manual selection makes it possible to do unanticipated control-flow scoping in arbitrary parts of the program.

Once instanciated, a scope can be composed and applied to a layer to define the conditions of its activation. Discussion of scope composition, although possible in the early implementation of our proposition, is outside the scope of this paper.

3.4 Selection and adaptation of objects

Once defined and scoped, a Dynamic Layer will adapt entities of the running system. It needs an underlying adaptation mechanism and means to select these entities to adapt. Both the adaptation mechanism and the objects' selection (which is orthogonal to the scopes dimension) are described in the following subsections.

3.4.1 A Generic Adaptation Mechanism. Layers define sets of behavioral variations for a given entity (a class, an object...). We want to dynamically inject these layers in a running software to adapt its behavior. Therefore, there must be an adaptation mechanism underneath with the capability to adapt the target entities. We chose to use *Lub* [3], a pattern for dynamic unanticipated adaptation. It can adapt objects on a per-instance basis and therefore allows flexible and dynamic groups of adapted objects. However, the chosen mechanism may vary, and could be implemented following different paradigms or solutions as long as it provides the necessary capabilities for dynamic unanticipated adaptation. We can specify these specific needs if one wants to design and implement such a mechanism by other means.

We need to adapt behaviors at a very fine grain in order to have flexibility when adapting. An adaptation must be able to target single objects as well as larger groups of objects (*e.g.* all instances of a class). It must be able to add or modify behaviors in an object. Finally, the adaptation process should leave the possibility to easily revert to the original behavior.

We chose to express this mechanism as a generic adaptation regardless of its implementation. In this paper, we use an entity called an *Adaptation*. This adaptation targets a class in which is defined new or altered behaviors. The adaptation can also select which behaviors from this class will be applied to an object. Control can be specified to choose whether the adaptation will execute behavior before, instead or after the original method. This avoids copying a full method if the adaptation is designed to insert pre/post processing to the original behavior. Excerpts of code below illustrates how an adaptation is created and how it is applied to an object:

```
| collection |
collection := OrderedCollection new.
adaptation := Adaptation
                    class: Array
                    with: #(#add:)
                    control: #instead.
collection adaptWith: adaptation.
collection add: 'test'
```

An instance of *OrderedCollection* is adapted with the *add:* method from *Array*. When sending the *add:* message to this collection, an exception is raised because the behavior from *OrderedCollection* has been replaced by the one from *Array*, which forbids its use. We could also use the *#before* or the *#after* control to execute additional behavior before or after the original one.

3.4.2 Expressing object's selection. A layer must now select for which objects it will provide behavioral variations. The design we present inverts the responsibility of the behavioral change decision from traditional context oriented practices. In most COP languages, layers are defined within classes, and objects implicitly enter a layer when an activation occurs. We chose to let the layer, as a first class object, to control which objects are going to be affected. It allows a fine customized selection of adapted entities. This is critical in an unanticipated debug context: we cannot know what are the

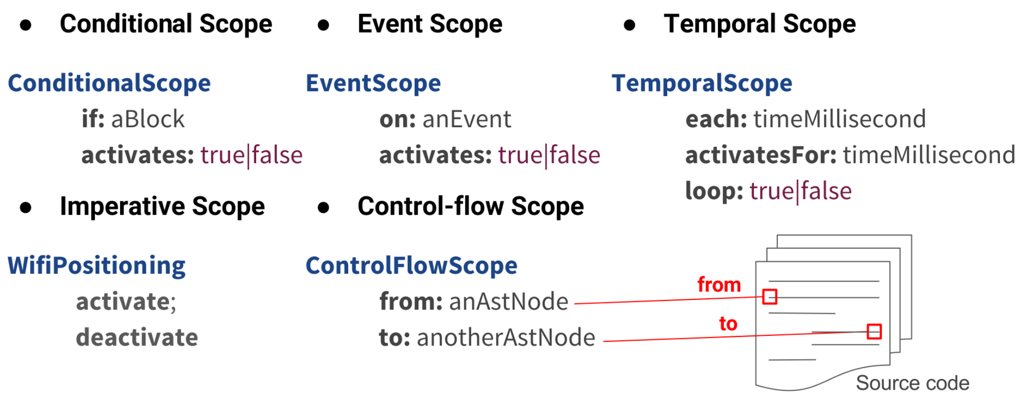

Figure 2: Reified activation scopes

objects that will require adaptation and therefore we need to keep as much flexibility as possible. We should be able to select:

- All instances of a given class
- A few instances of a given class
- An arbitrary set of objects, that may be instances of different classes
- Only one precise object

Furthermore, this selection of objects is dynamic, and therefore must be able to grow or to reduce in size, as the debugging activity progresses and requires more or less precision. The following Pharo code shows examples of selectors we designed to define these selections. It uses the previously defined adaptation:

```
"Selects one or more arbitrary objects to be
 adapted with an adaptation"
WifiPositioningLayer
        adapt: #(...objects...)
        with: adaptation.

"Selects instances of a class that are meeting specific
 constraints to be adapted with an adaptation"
WifiPositioningLayer
        select:[:object| ...selection operators...]
        from: aClass
        adaptWith: adaptation.

"Adapts all instances of an arbitrary class with an
 adaptation"
WifiPositioningLayer
        adaptAllFrom: aClass
        with: adaptation.
```

Selecting objects at runtime requires means to retrieve pointers to these objects. Examples from this paper are developed in Pharo Smalltalk, in which we can dynamically recover all instances from a given class then select the objects we are interested in. It is unpractical but it fits our prototyping purposes. New tools could be developed to provide such capabilities, like advanced debuggers.

4 IMPLEMENTATION

We developed an experimental implementation in Pharo [2], a dynamic language based on Smalltalk. It has a rich tooling environment [1] which allows for example runtime inspection of objects and runtime debugging. We implemented Dynamic Layers on top of Lub [3], a dynamic behavior adaptation mechanism with the capability of modifying behaviors of single objects at runtime. It allows to design and apply adaptations while still reasoning in terms of classes and objects. We changed the Lub backend to be based on a modified version of Reflectivity [11] to instrument abstract syntax trees (ast) nodes on specific objects. This *per-object* ast annotations bring more flexibility and control of adaptations in comparison with the Reflectivity default behavior which affects all instances of a given class.

The current implementation is partial and does not reflect all the features described in this paper. It is an early experiment as a first step in the illustration and the validation of Dynamic Layers.

5 EVALUATION: THE GPS EXAMPLE

This scenario is a toy example running in a Pharo simulation on a remote *Raspberry-pi* [2]. It models a gps in a drone that keeps switching online and offline. This constant changes prevent the wifi to take the hand in the navigation system. The developer uses a Pharo environment in which he can open a remote debugger from his development machine to debug the drone. The developer's computer and the *Raspberry-pi* (*i.e.* the drone) are connected through a dedicated network. In this example, we will use Dynamic Layers to probe the system. As the debugging process progresses, we will dynamically refine layers scoping to target specific behaviors we want to inspect and to adapt the system. The illustration code

[2]https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

below shows a block of code in the program that recovers status from five different sensors:

```
gps getSensorStatus
   ifTrue: [ ”do gps operations” ].
wifi getSensorStatus
   ifTrue: [ ”do wifi operations” ].
colorSensor getSensorStatus
   ifTrue: [ ”do color sensor operations” ].
temperatureSensor getSensorStatus
   ifTrue: [ ”do temperature sensor operations” ].
altitudeSensor getSensorStatus
   ifTrue: [ ”do altitude sensor operations” ]
```

We would like to visualize the results of the sensor checks. We propose to add *log* behavior before all executions of the *getSensorStatus* method in each sensor, to remotely print these status in the developer's environment (*e.g.* in a console). We thus define a layer with an adaptation which brings logging behavior from another class (*i.e. DLSensorLog*). The layer is dynamically applied to all the sensors:

```
1     DynamicLayer define: 'LogLayer'.
2     adaptation := Adaptation
3                         class: DLSensorLog
4                         with: #(#getSensorStatus)
5                         control: #before.
6     LogLayer adapt: sensors with: adaptation.
7     LogLayer activate
```

The *control:* (line 5) keyword allows to select whether the adaptation is triggered before, after or instead the instrumented method, which is inspired from Geppetto [14]. The layer has an imperative scope by default and can be manually activated (line 7). Each time the *getSensorStatus* method will be called, the added logging behavior will send to the developer's environment the sensor's status, then execute the original behavior.

At this point the developer can visualize all the sensors status, but is only interested in the GPS and the Wifi. Furthermore, if experimenting on a real device (for example during flight tests), he might not want the logging behavior to be active indefinitely. We can dynamically define two new scopes: a control flow scope to bound the layer activation to the part of code we are interested in and a temporal scope to allow the behavioral adaptation for only a few seconds. The control flow scope is defined to start before a first abstract syntax tree (ast) node and to stop after a second ast node. These nodes respectively start and stop at line 1 and line 3 in the code below, which is an excerpt of the larger code shown above:

```
1  gps getSensorStatus
2     ifTrue: [ ”do gps operations” ].
3  wifi getSensorStatus
4     ifTrue: [ ”do wifi operations” ]
```

The temporal scope is defined to activate every 10 seconds and only for 500 milliseconds, within the specified block of code. These two scopes are dynamically applied to the layer that automatically compose its activation scope:

```
cflowScope := ControlFlowScope from: firstAst
                                to: secondAst.
LogLayer addScope: cflowScope.
temporalScope := TemporalScope for: 500 each: 10000.
LogLayer addScope: temporalScope
```

The developer will now visualize only the GPS and Wifi status during short periods of time. It allows to watch these specific states and to understand that the problem is the constant switching between the two sensors. New layers can then be defined and applied to the sensors to prevent constant swapping between the GPS and the Wifi and fix the bug. As a final step, we can deactivate and remove the logging layer by sending the *deactivate* message, as the layer has an imperative scope.

We built a simple *view* of the running system as a first debug step, to understand what was going wrong. The noise produced by all the objects information was prevented by refining the scope of the adaptation in the control flow. Effects of the log behavior was also time bounded to avoid any extended impact on the system. Instead, we could have restrained the adapted entities to the GPS and the Wifi only. Both these strategies can be used in combination to provide very specific behavioral variations within a layer.

6 RELATED WORKS

Several mechanisms in dynamic languages can address dynamic unanticipated adaptation, for example based on full meta-objects reification [13] or on abstract syntax tree nodes annotations [11, 14]. There are also solutions for dynamic adaptation at runtime in statically typed languages like Java [16]. Such solutions for Java require modifications of the virtual machine, which induces a loss of genericity. These solutions were designed for debugging purpose in the first place, but they lack expressiveness in adaptations definitions. There are no abstraction to describe and group together adaptations, and no means to configure fine grained scopes for adaptation (de)activation.

Context Oriented Programming (COP) [5] expresses behavioral variations in *layers*. COP layers are statically defined in classes and are activated when the application meets specific contexts. This activation occurs within a given scope, which is specific to COP solutions: layer activation is bound to one scope which is either event-based, imperative, implicit or control-flow based. COP languages do not feature more than one kind of scope, and choosing one of these languages is usually driven by a specific usecase. In recent research, Kamina *et. al* [6] generalized layer activation into a unified mechanism. The four kinds of scopes can be used within the same language, which provides per-instance and global activation of layers. They also provide a model for scopes activation priorities, which is missing from the Dynamic Layers. Furthermore, COP languages aim at designing anticipated behavioral variations with a strong separation of concerns, but were not originally made for debug purposes.

Dynamic Layers combine unanticipated adaptation mechanisms flexibility and genericity with COP layers expressiveness. Layers and layer scoping are dynamic and composable, and a new kind of scope can be used to bound contexts to time constraints (temporal scope). The control-flow scope is purely dynamic and can scope a layer between two arbitrary lines of code at runtime, without modifying the base program. Dynamic Layers are not designed to build a COP language, but as a pattern for runtime debugging methodologies and practices. However the dynamic nature of scope composition raises questions about conflicts and how they could be solved. It is possible to compose irrelevant scopes and to inject layers that will never (de)activate in a running program, thus not making sense. How dynamic layers, adaptations and reified scopes behave regarding threads was also left aside. These problems were not addressed in the early design of Dynamic Layers.

Other works provide means to control changes consistency in running programs, like grouping live program modifications in layers before applying them [10], or to ensure objects consistency before applying behavioral changes [15]. These works could be interesting start points to think about safety and consistency aspects of Dynamic Layers. Chisel [8] is a framework for full dynamic unanticipated adaptation of software. It is policy based: adaptations and their contexts are specified in a high-level policy language that are injected and interpreted at runtime. While it is an effective generic solution, it is not meant as a debugging solution although it could partially be used as such. Also, its genericity has limits, as it puts constraints on software design: if the program was not developed with Chisel, it cannot be enabled at runtime.

7 CONCLUSION AND FUTURE WORKS

We defined the Dynamic Layers to debug running systems under critical conditions, which we call *Unanticipated Debugging*. The two dimensions in which layers are scoped, *i.e.* objects selection and activation scoping, are reified. We described these reifications and suggested a way to express them in a debug context. We illustrated a possible use of dynamic layers in a simple debug example.

However the solution lacks precision, in the adaptation mechanism for example, as it is not yet possible to express an adaptation of a piece of code at an arbitrary place in a method. The object selection mechanism needs a practical way to be performed, such as an extended debugger. The Dynamic Layers were only studied within one single thread, and not in multiple threaded applications. Further work will include deeper studies of these problems regarding the Dynamic Layers and their implementation. We plan to build more experiments to demonstrate *Unanticipated Debugging* on more concrete usecases. Work must also be done about Dynamic Layers consistency and their validation before being applied to a running program.

It would also be interesting to provide tools to experiment usecases of *Unanticipated Debugging*, for example to solve bugs at runtime in a small robot. It could be a remote debugger with a Dynamic Layer extension.

ACKNOWLEDGMENTS

Many thanks to Stéphane Ducasse for all the helpful discussions and to the anonymous reviewers for their reviews of this paper. This work is sponsored by the European Smalltalk User Group (www.esug.org).

REFERENCES

[1] A. Bergel, D. Cassou, S. Ducasse, and J. Laval. *Deep Into Pharo*. Lulu. com, 2013.

[2] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009. ISBN 978-3-9523341-4-0. URL http://pharobyexample.org.

[3] S. Costiou, M. Kerboeuf, G. Cavarlé, and A. Plantec. Lub: a dsl for dynamic context oriented programming. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*, page 13. ACM, 2016.

[4] P. Ebraert, T. D'Hondt, Y. Vandewoude, and Y. Berbers. Pitfalls in unanticipated dynamic software evolution. In *RAM-SE*, pages 41–50. Citeseer, 2005.

[5] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[6] T. Kamina, T. Aotani, and H. Masuhara. Generalized layer activation mechanism for context-oriented programming. In *Transactions on Modularity and Composition I*, pages 123–166. Springer, 2016.

[7] J. Keeney. *Completely unanticipated dynamic adaptation of software*. PhD thesis, University of Dublin, 2004.

[8] J. Keeney and V. Cahill. Chisel: a policy-driven, context-aware, dynamic adaptation framework. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 3–14, June 2003. doi: 10.1109/POLICY.2003.1206953.

[9] G. Kniesel, J. Noppen, T. Mens, and J. Buckley. Unanticipated software evolution. In *European Conference on Object-Oriented Programming*, pages 92–106. Springer, 2002.

[10] T. Mattis, P. Rein, and R. Hirschfeld. Transaction layers: Controlling granularity of change in live programming environments. In *Proceedings of the 8th International Workshop on Context-Oriented Programming*, COP'16, pages 1–6, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4440-1. doi: 10.1145/2951965.2951969. URL http://doi.acm.org/10.1145/2951965.2951969.

[11] O. Nierstrasz, M. Denker, and L. Renggli. Model-centric, context-aware software adaptation. In *Software Engineering for Self-Adaptive Systems*, pages 128–145. Springer, 2009.

[12] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering*, pages 485–495. IEEE Press, 2012.

[13] J. Ressia, T. Gîrba, O. Nierstrasz, F. Perin, and L. Renggli. Talents: an environment for dynamically composing units of reuse. *Software: Practice and Experience*, 44(4):413–432, 2014.

[14] D. Röthlisberger, M. Denker, and É. Tanter. Unanticipated partial behavioral reflection. In *International Smalltalk Conference*, pages 47–65. Springer, 2006.

[15] N. Taing, M. Wutzler, T. Springer, N. Cardozo, and A. Schill. Consistent unanticipated adaptation for context-dependent applications. In *Proceedings of the 8th International Workshop on Context-Oriented Programming*, COP'16, pages 33–38, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4440-1. doi: 10.1145/2951965.2951966. URL http://doi.acm.org/10.1145/2951965.2951966.

[16] T. Würthinger, C. Wimmer, and L. Stadler. Unrestricted and safe dynamic code evolution for java. *Science of Computer Programming*, 78(5):481–498, 2013.