Zone-based Layer Activation

Context-specific Behavior Adaptations across Logically-connected Asynchronous Operations

Stefan Ramson
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
stefan.ramson@hpi.uni-potsdam.de

Jens Lincke
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
jens.lincke@hpi.uni-potsdam.de

Harumi Watanabe
Department of Embedded Software
Tokai University
Tokio, Japan
harumi-w@tsc.u-tokai.ac.jp

Robert Hirschfeld
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

Scoping behavior adaptations using dynamic extent is a crucial part of Context-oriented Programming (COP). In a synchronous execution model, dynamic extent ensures the activation of a layer for the entire duration of a block. An asynchronous execution model, however, breaks the intended semantics of dynamic extent. For example, using the await keyword postpones the execution of the block and returns to its caller. Thus, dynamic extent deactivates the behavior adaptation. Consequently, when resuming the postponed execution the layer is no longer active.

In this paper, we propose a variant of dynamic extent that activates a layer for a block and all its logically-connected asynchronous operations. We show how zones can be used to track the asynchronous dynamic extent of a block. Further, we provide an implementation of our approach as an extension to ContextJS in JavaScript.

CCS CONCEPTS

• **Software and its engineering** → *Object oriented languages*; **Control structures**; **Coroutines**.

KEYWORDS

Context-oriented Programming, Activation Means, Dynamic Extent, Asynchronous Programming, Zones, JavaScript, Promises

ACM Reference Format:
Stefan Ramson, Jens Lincke, Harumi Watanabe, and Robert Hirschfeld. 2020. Zone-based Layer Activation: Context-specific Behavior Adaptations across Logically-connected Asynchronous Operations. In *12th International Workshop on Context-Oriented Programming and Advanced Modularity (COP'20), July 21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3422584.3422764

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

ACM ISBN 978-1-4503-8144-4/20/07…$15.00
https://doi.org/10.1145/3422584.3422764

1 THE RISE OF ASYNCHRONOUS EXECUTION

Context-oriented Programming (COP) [7] dynamically extends system behavior in a cross-cutting manner. One may use partial methods to extend or override the behavior of the base system. Partial methods can be grouped together into units of modularity called *layers*. Depending on the changing context of the application, a layer may be activated dynamically at runtime, changing the system behavior by applying its partial methods on the system's base behavior. To activate the layer, developers may use one of many proposed activation means [17]. One of the most frequently used activation means is *dynamic extent* [10]. Using dynamic extent developers may activate layers only for the duration of a message send. Dynamic extent ensures to automatically deactivate the corresponding layers at the end of the message send [2]. Thus, dynamic extent allows developers to safely adapt system behavior for a well-defined portion of the execution without accidentally leaking behavior adaptations into other parts of a program.

The design of COP in general and dynamic extent in particular assume a synchronous execution model. In recent years, however, asynchronous programming became popular, especially in context of UI programming and client-server communication. In an asynchronous programming model the execution of a method may be postponed by returning a *promise*[1]. At some later point in time the promise may resolve, causing the asynchronous operation to resume its execution. As a result, asynchronous semantics may be described concisely while seemingly avoiding a "callback hell" using dedicated language concepts such as async/await.

Unfortunately, such asynchronous execution models break with the linear control flow assumed by most COP implementations [5].

[1] or a *future* if emphasis is put on the datum to be calculated by the postponed execution. In context of this paper, both can be seen as equivalent.

While we define dynamic extent through lexical scoping, asynchronous execution is not bound by the linear fashion of executing nested functions. Instead, execution scopes may overlap and interleave in time. As a result, the safe semantics of the dynamic extent activation means cannot be guaranteed anymore: layers might be missing from the current layer composition in a postponed asynchronous task, or might leak to code not intended to run with a certain behavior adaptation. Neither of the aforementioned cases is desirable.

In order to get the intended behavior of scoping the dynamic behavior exactly as specified by the lexical scope, we need to keep the layer composition consistent across all asynchronous operations logically-connected to a method. Zones[2] describe a persistent context across logically-connected asynchronous operations. Additionally, zones may control asynchronous behavior by observing and intercepting the execution of asynchronous tasks. Thus, zones can provide a suitable foundation for an extension to COP to better integrate with asynchronous execution models.

Contributions. In this paper, we propose an approach to consistently apply behavior adaptations activated through dynamic extent throughout the entire execution of an asynchronous function. In particular, we make the following contributions:

- The design of an adaptation of scopes with dynamic extent across logically-connected asynchronous operations by integrating with zones, including
  - A scheme to reify layer stacks created by dynamic extent with the possibility to capture the current layer stack and replay it at some later point.
  - An integration of COP with zones to track and apply layers consistently on logically-connected asynchronous operations.
- An implementation of this design in JavaScript as an extension to ContextJS [12].
- An extension of the Dexie.Promise library, an implementation of a zone-like concept for JavaScript, with necessary zone life-cycle callbacks.

Availability. You can find the full JavaScript implementation accompanying this work on GitHub[3]. Furthermore, a running version of the example application discussed in section 6 is available at the project's GitHub pages[4].

Organization. In the remainder of this paper, we first introduce a motivational example to further illustrate the need for an asynchronous dynamic extent in section 2. In section 3 we review the concept of zones as a persistent context across logically-connected asynchronous operations. Based on zones, we present our approach to apply dynamic extent to asynchronous operations in section 4. In section 5 we present an implementation of this approach as an extension to ContextJS in JavaScript. We discuss an implementation of our motivational example using our approach in section 6. In section 7 we highlight similar approaches to integrate COP with asynchronous programming. We describe possible directions for future work in section 8 and conclude in section 9.

[2] https://dart.dev/articles/archive/zones accessed on June 24th 2020
[3] https://github.com/onsetsu/area51 accessed on 30th June 2020
[4] https://onsetsu.github.io/area51/experiments/github-access.html accessed on 30th June 2020

2 IN NEED FOR AN ASYNCHRONOUS DYNAMIC EXTENT

To illustrate the semantic mismatch of asynchronous execution and dynamic extent consider the following example[5]. We want to write an application that lists the repositories of a user, including the author, time, and message of the last commit.

```
1 async function displayRepos() {
2   const repos = await Github.repos()
3   for (let repo of repos) {
4     const commits = await repo.commits()
5     display(repo, commits.last)
6   }
7 }
```

The above function `displayRepos` implements the desired behavior: we first query the GitHub API to list all repositories of the current user in line 2. After awaiting the response we send another Web request for each repository to query the latest commit, appending it to a result list in line 5. ① in Figure 1 shows dynamic behavior when executing `displayRepos`.

Unfortunately, this solution only shows a list of public repositories. To also list private repositories of the user, we use the layer `AuthLayer` to provide proper authorization as a cross-cutting concern throughout our application. We therefore wrap the aforementioned code into a call to withLayers to adapt the requests with the necessary access token.

```
1 withLayers([AuthLayer], async () => {
2   const repos = await Github.repos()
3   for (let repo of repos) {
4     const commits = await repo.commits()
5     display(repo, commits.last)
6   }
7 })
```

One might simply expect the above code to display information for public and private repositories of the user as shown in ② in Figure 1. However, the code actually results in an error in line 4 because the `AuthLayer` is no longer active and we try to access information of a private repository without proper authorization. To understand this misbehavior we have to look at how the asynchronous callback is executed and how and when the `AuthLayer` is activated.

Executing withLayers first activates the `AuthLayer`, then proceeds by calling the given function. A proper request to the GitHub API is created, including authorization due to the `AuthLayer` (code fragment highlighted in green). Once the request is sent, we await the response from the server. At this point the underlying JavaScript engine postpones the remainder of the function and returns control flow back to the caller of the function, ContextJS's withLayers in this case (③). Assuming the function returned properly, ContextJS removes the `AuthLayer` from the current layer stack, then, returns control flow back to its own caller. Some time later GitHub

[5] example code is given in JavaScript throughout the paper

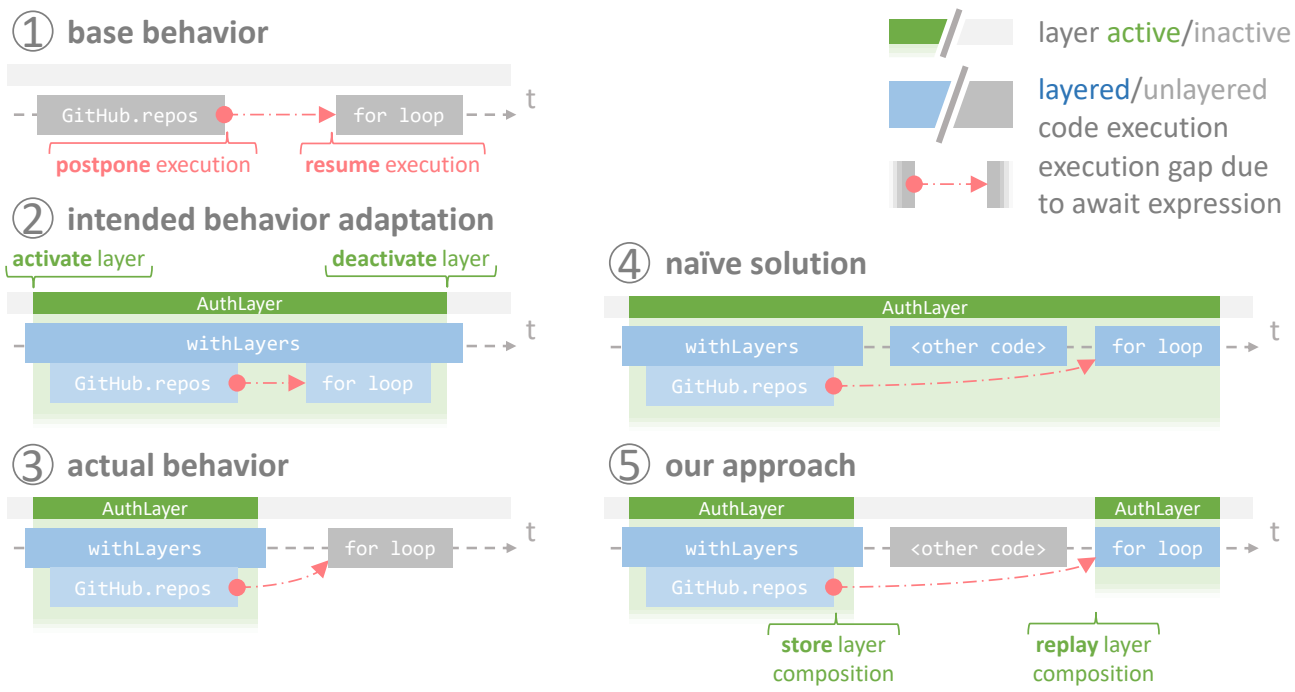

Figure 1: Annotated call traces for running the `displayRepos` function[a] discussed in section 1. The interaction of dynamic extent with asynchronous code leads to unexpected behavior.

[a] for brevity, the for-loop is summarized as a single asynchronous operation even though it itself includes an await expression

answers the request and sends a response to the browser. Thus, our postponed function resumes and attempts to loop through the `repo` array (code fragment highlighted in orange). However, as the `AuthLayer` is now deactivated, no access token is attached when requesting the commits of the first private repository in line 4. Thus, the GitHub API returns with an authorization error. As seen in this example, unawareness of asynchronous computation may result in unintended behavior.

One way to fix this behavior as an application developer is to bring the `AuthLayer` activation closer to the calls it actually adapts:

```
1 const repos = await withLayers([AuthLayer],
      () => Github.repos())
2 for (let repo of repos) {
3   const commits = await withLayers([
        AuthLayer], () => repo.commits())
4   display(repo, commits.last)
5 }
```

Unfortunately, this solution leads to architectural overhead due to code duplication. Further, the code becomes more error prone in case of code changes. Finally, this solution undermines the cross-cutting nature of COP to adapt multiple modules from a separate module: having to specify layer activations rather specifically defeats the purpose of COP itself.

From the perspective of a system developer, one solution would be to keep the `AuthLayer` active until the given asynchronous block is completely settled (④). However, this naïve solution leaks the access token to code outside the withLayers call, allowing other code fragments to access private repositories. In general, behavior adaptations activated on asynchronous functions may linger indefinitely in the system, causing unexpected and unwanted behavior.

The bigger, more general problem here is that while we define dynamic extent through lexical scoping, asynchronous execution is not bound by the linear fashion of executing nested functions. Instead, execution scopes may overlap and interleave in time. To achieve the intended behavior of scoping the behavior adaptation as specified by the lexical scope (⑤), we need to extent our notion of dynamic extent to include all logically-connected asynchronous operations as well.

3 HANDLING CONTEXT ACROSS ASYNCHRONOUS OPERATIONS USING ZONES

As described in section 2, layer compositions are not consistent across logically-connected asynchronous operations. This consistency would require COP to be aware of asynchronous operations and their relationships. Zones are a concept found in asynchronous programming to relate logically-connected asynchronous operations. In the following, we provide the necessary background on zones and describe why they are a suitable foundation for our asynchronous dynamic extent.

A *zone* is an execution context that persists across logically-connected asynchronous operations. As an example, consider the

displayRepos function in section 2 as part of a larger application. In addition to displaying the repositories of a user, the application also displays the latest contributions of that user.

```
1 async function displayRepos() {
2   const repos = await Github.repos()
3   // omitted for brevity
4 }
5 runZoned(async () => {
6   await displayRepos()
7   await displayContributions()
8 }, { user: 'onsetsu' })
```

Both parts involve requests to a remote server, thus, they are asynchronous by nature. To track which user's repositories and contributions we are interested in, we run both calls in a new zone using runZoned in line 5 with the user as a *zone property* (line 8). Zone properties act similar to a thread-local storage, allowing a developer to access resources attached to the current zone. In our example, we may access the current user through Zone.current.user from any code executed in the current zone, for example in line 7 of the following code:

```
1 class Github {
2   static async repos() {
3     const json = await this.api('/repos')
4     return json.map(r => new Repository(r))
5   }
6   static async api(path) {
7     const url = GH_API + Zone.current.user +
          path
8     const response = await fetch(url)
9     return response.json()
10  }
11 }
```

Zone properties provide even deeply nested functions, such as api in line 6, with access to the current user without polluting every subsequent function's signature with an additional parameter (here displayRepos, repos, and api).

Zone properties persist regardless of whether the invoked behavior is synchronous or asynchronous. Whenever asynchronous tasks get scheduled within a zone, the postponed code will execute in the same zone as the zone which existed at the time of invoking the asynchronous API. As a result the zone can be tracked across asynchronous invocations and, for example, let the user persist to the displayContributions call.

Each stack frame is executed within exactly one zone[6] and can never change between zones. However, a function may be executed from multiple zones. Executing the above code from multiple zones safely provides access to the GitHub data of multiple users without interfering with one another.

In addition to providing a persistent execution context, zones allow developers to *control asynchronous behavior* by observing and intercepting the execution of asynchronous tasks. In particular zones expose a variety of life-cycle callbacks, for example when entering a zone, leaving a zone or throwing an error within the execution of a zone. The latter callback is most often used to handle error across an entire thread of asynchronous tasks in a catch-all fashion. Entering and leaving a zone allows for various tracing activities, such as measuring the total time spent within a particular zone or monitoring all dangling asynchronous tasks within a zone. Particularly these tracing capabilities make zones a reasonable foundation for our proposed integration of COP with asynchronous computation as they expose the task scheduling and processing of the host environment as programmable extension points.

[6] defaults to a <root> or global zone

Listing 1: Running our GitHub-querying block with authorization but without caching. After the initial request is sent the block is postponed. Next, the CacheLayer is activated globally. Thus, the CacheLayer should cache future requests but not within the previously postponed block.

```
1 withLayers([AuthLayer], () => {
2   withoutLayers([CacheLayer], async () => {
3     const repos = await Github.repos()
4     for (let repo of repos) {
5       const commits = await repo.commits()
6       display(repo, commits.last)
7     }
8   })
9 })
10 CacheLayer.beGlobal()
```

4 MAKING DYNAMIC EXTENT AWARE OF ASYNCHRONOUS OPERATIONS

This section describes our approach to keep the layer composition consistent across logically-connected asynchronous operations. To properly integrate COP with asynchronous execution, we need to keep track of asynchronous operations and their causal relationship.

In this section we first describe how a layer composition can be saved for later restoration. Then, we describe a mechanism to relate a layer composition to a set of logically-connected asynchronous operations.

A layer composition is the combination of layers activated using various activation means, such as global activation [2], dynamic extent [8], or implicit layer activation [14, 18]. While most activation means depend on the global context of the program, dynamic extent considers only the local stack as context information. Consider the extension to our running example in Listing 1: here, we introduce another layer called CacheLayer. This layer memoizes responses to requests sent while it was active and answers subsequent requests through a cache if possible to reduce traffic and response times. However, as the AuthLayer handles sensitive information about the user, we do not want to cache this information, such as the access tokens. Thus, we wrap our requests in a withoutLayers call. When the first request is sent through GitHub.repo() in line 3, the execution of the remainder of the function is postponed and awaits its response. Even though the global context of the program might change in between (e.g., the user activates cache, and thereby the CacheLayer through a button click), we still want to continue

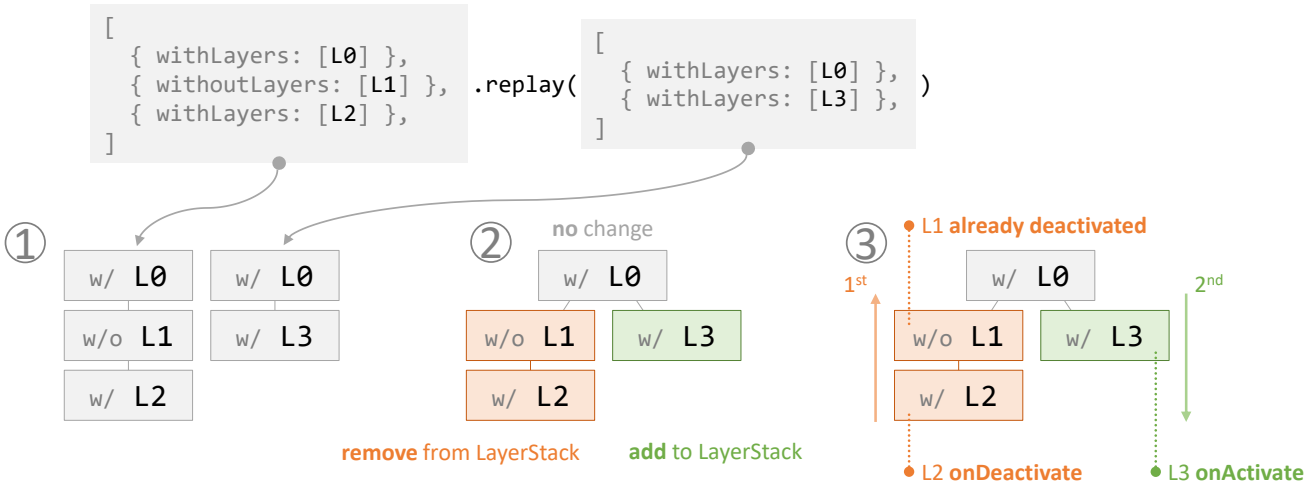

Figure 2: Replaying a layer stack with two frames (right) onto a layer stack with three frames (left). After calculating the common ancestry of both stacks (②), we first remove all additional frames from the current stack and, then, push all additional frames from the target stack onto the current stack (③).

executing the function without the `CacheLayer` (dynamic extent is prioritized over global activation). Thus, our approach has to ensure that the layer stack for dynamic extent stays consistent throughout the entirety of the asynchronous operation. In contrast, global activations happen as a side-effect in response to the global context of the program. Reverting the globally activated layers to a former state might have unintended and unexpected consequences.

Implicit layer activation binds the activation status to a boolean predicate. Reverting those layers to a former state would only introduce a short flickering in the activation state, as the bound predicate would immediately override the activation status according to its current state. To summarize, when restoring the execution context of an asynchronous operation, we just need to replay the **layer stack** given by dynamic extent activations and deactivations. The rest of the layer composition evolves with the program independent of the frame that is currently executed.

4.1 Layer Stack Reification and Replay

A layer stack is first and foremost a collection of layer stack frames representing the calls to withLayers and withoutLayers in the current call stack. As an example, the code in Listing 1 yields the following layer stack in line 3:

```
[
  { withLayers: [AuthLayer] },
  { withoutLayers: [CacheLayer] }
]
```

While typical COP implementations use a single layer stack, the integration with asynchronous operations breaks this assumption by allowing multiple execution paths to overlap and intertwine. Thus, we need to store the correct layer stack for each asynchronous operation. This requires us to be able to handle multiple layer stacks simultaneously and apply the correct one for the currently running asynchronous operation. Thus, we need to reify the layer stack into a first-class object.

In particular, we propose two methods for a layer stack to expose, `copy` and `replay`. The `copy` method creates a new layer stack with the same frames as the original one. These copies are first-class objects and, thus, can be used as parameters or return values and stored like any other value. The `replay` method takes a target layer stack as parameter and replaces the frames of the callee with the ones of the target. As a result, the callee now contains the same behavior adaptations as the target layer stack.

Replaying a layer stack. While the `replay` method correctly adapts the program behavior, simply overriding the frames neglects the life-cycle callbacks of the involved layers. The life-cycle callbacks `onActivate` and `onDeactivate` typically provide proper setup or teardown for state used by the introduced behavior adaptation [9]. It is important to consider that the `onActivate` callback should trigger **as soon as** a layer becomes active from a non-active state regardless which activation means caused this change (analog for `onDeactivate`). To do so, we extend the `replay` method with proper callback invocations.

Figure 2 exemplifies our approach for replaying a layer stack with two frames onto one with three frames (①). First, we determine the common frame ancestry containing the topmost frames common to both layer stacks (②). To prevent unnecessary layer activations and deactivations, we keep frames in the common ancestry as is. Next, all frames below the common ancestry get removed from the current layer stack starting at the bottom. While removing frames, we emit `onDeactivate` (or `onActivate`) if necessary (③). Finally, all frames below the common ancestry in the target stack get pushed onto the current stack from top to bottom. Analogous to removing frames, `onActivate` and `onDeactivate` callbacks trigger appropriately. As a result, the current stack now equals the target stack and life-cycle callbacks were emitted as if the procedure would use ordinary

withLayers and withoutLayers calls. Assuming no active global layers for our example in Figure 2, `L2` first emits `onDeactivate`, `L1` does not emit `onDeactivate` because it was not active before, and finally `L3` emit its `onActivate`.

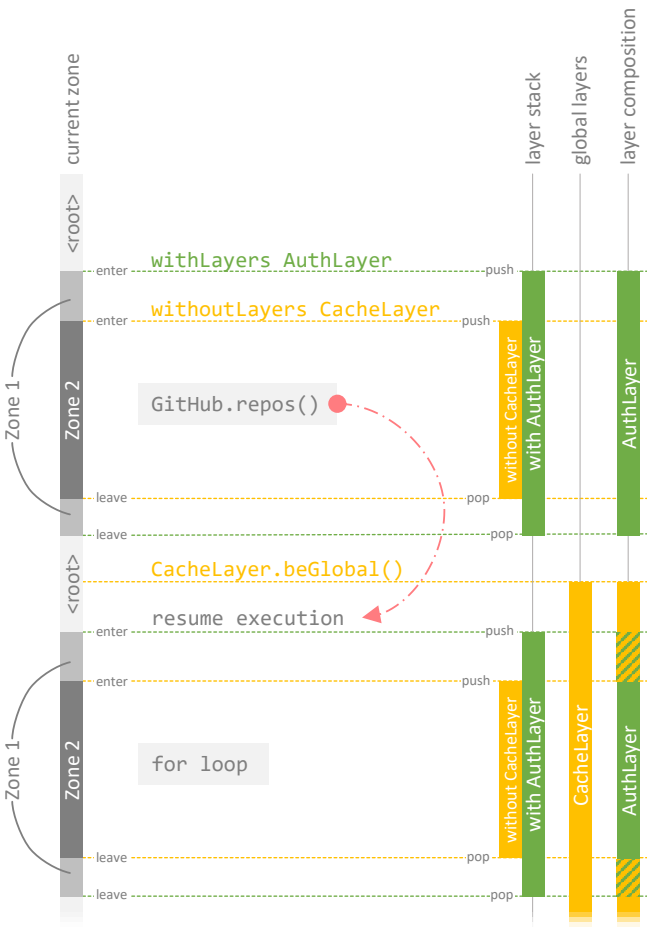

Figure 3: Interaction of zones and layer activation when executing the code from Listing 1. Using our asynchronous dynamic extent ensures that the correct layer composition is applied even for asynchronous operations.

4.2 Zone-aware Dynamic Extent

To keep the layer stack consistent throughout the entire execution of an asynchronous operation we integrate the layer replay mechanism with zones. To do so, we replace the ordinary withLayers function with asynchronous variant withLayersZoned. This variant runs the given code in a new zone. Thus, every asynchronous operation scheduled within this code is also executed within this very zone. We use the life-cycle callbacks of the zone to manipulate the layer stack so that the given layers are active when running any code inside this zone. First, we copy the current layer stack and store the copy in a local variable. Inside the zone's `leave` callback, we replay the copied layer stack to revert to the state present before the call to withLayersZoned. Additionally, we copy the current layer stack a second time. This stack represents the layer stack active within the zone. Therefore we push the given layers as a new frame onto this stack. We replay this layer stack in the zone's `enter` callback. As a result, whenever we enter the zone, the layer stack with the new frame is active and whenever we leave the zone, the layer stack is reverted to its previous state. Ultimately, this mechanism replicates the working principle of withLayers for an asynchronous execution.

Figure 3 illustrates how our asynchronous dynamic extent preserves the correct layer composition over asynchronous operations for the example code in Listing 1. withLayers([`AuthLayer`]) runs the given block in a new zone (Zone 1). Upon entering Zone 1 the `AuthLayer` is pushed onto the layer stack and, thus, activated. Analogous, withoutLayers([`CacheLayer`]) ensures the `CacheLayer` not to be active. Consequently, the `fetch` request in `GitHub.repos` is executed with an active `AuthLayer` but without the `CacheLayer`. Reaching the first await expression in line 3 postpones the remainder of the given block until receiving a response from the server. Thus, we leave Zone 2 and Zone 1 (removing frames from the layer stack accordingly) to resume normal execution of the outer code. Then line 10 activates the `CacheLayer` globally. When receiving the response from the server the execution of the postponed block continues within the appropriate zones. Upon re-entering Zone 1 the `AuthLayer` gets activated. Re-entering Zone 2 ensures that the `CacheLayer` is inactive, overriding the global activation. Thus, the for-loop runs with the same context information as the code that scheduled it.

5 IMPLEMENTATION AS A CONTEXTJS EXTENSION

This section describes a prototypical implementation of the asynchronous dynamic extent proposed in section 4. Our prototype[7] is written in JavaScript as an extension to ContextJS [12]. Thus, the extension requires a zone implementation in JavaScript as well.

5.1 Zone Libraries in JavaScript

Zone were originally designed for Dart, a language heavily focused on describing user interfaces[8] and, thus, requiring a rich tool set to deal with asynchronous operations. As zones allow developers to handle asynchronous tasks more easily, the JavaScript community quickly picked up zones as another way to deal with increasingly popular promises and async/await expressions.[9]

There are several zone libraries available for JavaScript. In the following we examine two very different zone libraries, zone.js and Dexie.Promise, for their suitability to integrate with COP.

zone.js. The library zone.js[10] originated as part of the JavaScript front-end framework Angular[11]. In context of Angular zones are

[7] https://github.com/onsetsu/area51 accessed on April 27th 2020
[8] https://dart.dev/ accessed on May 1st 2020
[9] Interestingly, zones themselves were inspired by node.js' now-deprecated domain concept along with thread-local storage (https://dart.dev/articles/archive/zones accessed on April 20th 2020).
[10] https://github.com/angular/angular/tree/master/packages/zone.js accessed on May 1st 2020
[11] https://angular.io/ accessed on 21st April 2020

used to relate asynchronously scheduled user events with their dependencies to trigger corresponding view updates.

- zone.js supports various different asynchronous APIs, including promises, `setTimeout`, and event callbacks. However zone.js does not support native await expressions.
- Zones support typical life-cycle callbacks.
- The library provides a rich set of features, for example for generating long, asynchronous call stacks.

Dexie.Promise. The IndexedDB[12] wrapper Dexie[13] includes a zone-like utility called Dexie.Promise[14]. Dexie allows developers to describe database transactions as a series of asynchronous tasks. Zones are used to keep track of transaction scopes.

- Dexie.Promise only considers promises and no other asynchronous APIs. However Dexie also supports native await expressions.
- Dexie.Promise provides no explicit object to represent zones. Instead zone properties are always attached to a promise.
- Dexie.Promise does not expose life-cycle callbacks.

Comparison. The main disadvantage of zone.js is its inability to handle native async/await code. The Angular community mitigates this issue by transpiling asynchronous functions into older versions of JavaScript maintaining similar functionality. This approach has two downsides. First, the solution introduces tooling overhead for the developer by requiring an additional transpilation step. As Angular application are typically written in TypeScript the tooling overhead for compiling to JavaScript is already impaired. However, for an extension to ContextJS this approach would require every user of the ContextJS library to integrate an additional transpilation step into their workflow. Second, the transpiled code relies on an emulator to mimic newer language features with ES2015 features. As a result, the code becomes obfuscated for the Just-in-Time compiler and, therefore, is hard to optimize. Both issues suggest that zone.js is not an optimal solution for our integration, especially considering future JavaScript versions.

In contrast to zone.js Dexie.Promise supports native await exressions. However Dexie.Promise does not expose any zone life-cycle callbacks. Yet these callbacks are crucial for our integration approach to correctly invoke `onActivate` and `onDeactivate` callbacks as described in section 4. From our available options we chose to use Dexie.Promise as a basis for our ContextJS extension and, thus, need to extend Dexie.Promise with life-cycle callbacks.

5.2 Dexie.Promise Extension

We expose four life-cycle callbacks for zones in Dexie.Promise: `beforeEnter`, `beforeLeave`, `afterEnter`, and `afterLeave`. These callbacks are defined as zone properties:

```
{
  beforeEnter(from, zone) { /* ... */ },
  afterLeave(zone, to) { /* ... */ }
}
```

[12] https://www.w3.org/TR/IndexedDB/#idl-def-IDBEnvironment accessed on May 1st 2020
[13] https://dexie.org/ accessed on May 1st 2020
[14] https://dexie.org/docs/Promise/Promise.PSD accessed on 21st April 2020

All callbacks receive two parameters: the zone we change from and the zone we change to. As an example, `beforeEnter` is invoked right before switching to the zone on which the callback is defined while the other zone is still active. To invoke these callbacks, we extend `switchZones`, the central function for zone changing in Dexie.Promise. We call `before` callbacks after internal micro task management but before the actual zone switch occurs and `after` callbacks after the actual zone switch as well as after disposing interception hooks.

We only invoke callbacks directly attached to the current zone and we ignore callbacks defined in parent zones. Additionally callbacks are also not invoked when switching from a zone to itself.

5.3 ContextJS Extension

Before integrating dynamic extent with zone life-cycle callbacks we should be able to store and replay a layer stack. In ContextJS, the active `LayerStack` is a central entity for its inner working. The `LayerStack` is essentially a collection of layer frames. Each frame contains the information to either activate or deactivate a set of layers. Manipulating the `LayerStack` adapts the layers activated through dynamic extent.

Layer Stack Reification. To copy the `LayerStack`, we simply copy all layer frames it consists. We copy layer frames by shallow copying with the exception of the withLayers and withoutLayers properties which are shallow copied individually. The copies can be passed around like any other JavaScript value.

Replaying a Layer Stack. To replay a layer stack, we apply any copy onto the active `LayerStack`. In particular, we first compute how many frames both layer stacks have in common (starting at the least significant frame). Second, we pop any additional frame from the active `LayerStack`. We remove the most significant frames first to resemble synchronous unwinding of the `LayerStack`. And third, we push the new frames onto the `LayerStack`. When popping from and pushing onto the active `LayerStack` we emit life-cycle callbacks accordingly as shown in Listing 2 in Appendix A.

5.4 Integrating Layer Activation with Zones

The functions withLayersZoned and withoutLayersZoned execute a given callback for its asynchronous dynamic extent. Both functions create a new layer frame, then call the `withFrameZoned` function as seen in Listing 3 in Appendix A. The `withFrameZoned` function directly implements the algorithm described in subsection 4.2. We create two copies of the active `LayerStack`, one to return to later and one to execute the callback in. We push the given layer frame onto the latter copy. Finally, we run the callback in a new zone. We attach life-cycle callbacks to manipulate the `LayerStack` accordingly: upon entering the zone we apply the copied layer stack with the additional frame and upon exiting we revert to the former stack. To replicate the semantics of the original withLayers as closely as possible, we use `after` callbacks for replaying the stacks and, thus, emitting life-cycle callbacks in the zone we switch to. withLayersZoned is compatible with the behavior of withLayers for synchronous code.

6 IMPLEMENTING A GITHUB APPLICATION

To check the feasibility of our approach and implementation we develop the GitHub application used as a running example:

```
1 async function displayRepos() {
2   const repos = await GitHub.repos()
3   for (let repo of repos) {
4     const commits = await repo.commits()
5     display(repo, commits.last)
6   }
7 }
```

The above function `displayRepos` first queries GitHub for all repositories of a user. Additionally, the latest commit for each repository is requested. Finally, each repository is displayed along with information on its latest commit.

```
1 await displayRepos()
2 await withLayers([AuthLayer], displayRepos)
3 await withLayersZoned([AuthLayer],
    displayRepos)
```

Calling `displayRepos` without any modifications as in line 1 of the above code lists the most recently changed repositories of the user as show in ① in Figure 4. However, these requests only consider public repositories due to missing authorization.

The `AuthLayer` represents a behavior adaptation of the application to access also private repositories. To do so, the `AuthLayer` extends methods of the `GitHub` and `Repository` classes to add an access token to their requests. Activating the `AuthLayer` using withLayers as in line 2 applies the desired behavior adaptation for the duration of the call to `displayRepos`. However withLayers only activates the behavior adaptation for the synchronous dynamic extent of `displayRepos`: after reaching the first await expression the function postpones its execution and returns synchronously. Consequently, the `AuthLayer` is deactivated and further asynchronous request will not contain an access token, thus, requesting the latest commit of a private repository fails and needs to be handled properly as seen in ② in Figure 4.

In contrast, calling `displayRepos` through our asynchronous dynamic extent variant withLayersZoned results in the desired behavior: the `AuthLayer` gets activated for the synchronous extent of the function call (until the first await expression) as well as every asynchronous operation scheduled. As a result, all necessary requests are adapted properly and have access to private information, ultimately, resulting in the list in ③ in Figure 4.

In addition to enabling a layer for any asynchronous operation logically-connected to an initial function, withLayersZoned ensures that the layers are inactive outside of its zone. This mechanism even allows to interleave multiple calls to `displayRepos` with different activation means, for example running all three calls of `displayRepos` in line 1 to 3 concurrently without accidentally leaking the access token.

7 RELATED WORK

Thread-local layer activation is a feature supported by various COP implementations [2], including ContextS [6] and ContextJ [3]. In those implementations, the thread-local storage serves a similar role

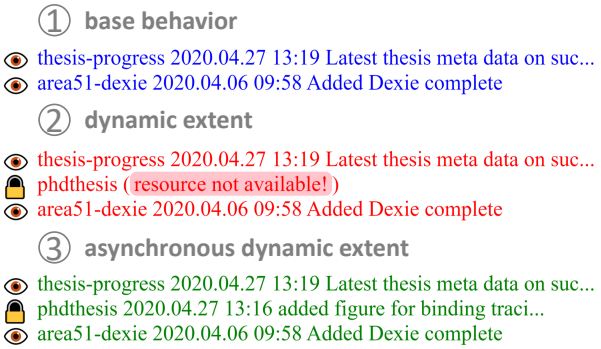

Figure 4: Results of executing the `displayRepos` function with different behavior variations.

as zones in our approach. The concept of thread-local storages even was an influence in the design of zones[15]. However, while thread-local activations prevent layers from leaking to other threads, our approach maintains the consistency of a layer stack across logically-connected asynchronous operations within a single thread.

JCop [4] heavily emphasizes context-dependent behavior on event handlers, another incarnation of asynchronous programming. As with scheduled asynchronous code, event handlers are typically not entered at a specific fixed point in the control flow but exist independent of the main control flow. Thus, JCop allows developers to activate layers for the dynamic extent of an event handler call through the usage of Aspect-oriented Programming [11]. Activating a layer this way is similar to our layer replay mechanism on an asynchronous operation. However, JCOP requires its users to specify which layer composition to activate and to explicitly state which methods (or control flow entry points) should receive that layer composition. In contrast, our approach implicitly replicates a layer stack for an event handler equivalent to the one that handler was defined in. An important distinction between JCop and our approach is that JCop only supports dynamic extent and, thus, needs to model any context in this manner. Our approach easily allows dynamic extent to interact with other activation means.

An application of JCop in the domain of Service-oriented Architecture closely resembles our approach of layer stack reification in order to preserve context between different services [1]. When calling a SOAP message to a remote service the current layer composition in the client is accessed through JCop's reflection API. The layer composition is then *enveloped* into the message call as a context description. The server *de-envelopes* the description and activates the corresponding layers to handle the request in the same context as the client. Enveloping and de-enveloping layers is very similar to storing and replaying a layer stack in our approach when scheduling and resuming asynchronous operations. However, while our approach handles context preservation implicitly, JCop requires explicit enveloping of the layer composition.

Similar to JCop, EventCJ [9] is another Java-based COP implementation that supports event-based layer activation. In contrast to JCop, EventCJ allows developers to activate layers in an instance-specific manner. While our approach does not bind layer activation

[15] https://dart.dev/articles/archive/zones accessed on June 24th 2020

to a specific instance, we bind a layer stack to a particular set of asynchronous operations, allowing a single instance to have different layer compositions within different asynchronous operations.

8 FUTURE WORK

While the integration of dynamic extent with asynchronous operations represents a useful iteration on the COP paradigm, we expect to continue improving this integration further. In particular, this work can be extended in two directions:

- Extending the integration with asynchronous operations to other meta programming concepts, such as Active Expressions [13] and Babylonian Programming [15, 16].
- Applying the concept of replaying behavior adaptations to a more general approach to control, postpone, and resume execution. For example how to generalize async/await-style asynchronous execution and continuations on coroutines?

9 CONCLUSION

In this paper we proposed zone-based layer activation, a variant of the dynamic extent activation means that activates a layer for the entire duration of a message send, including all logically-connected asynchronous operations. To do so, we capture the layer stack when defining the dynamic extent and replay that layer stack whenever a connected asynchronous operation gets executed. To track asynchronous operations we use zones, a concept from asynchronous programming languages to provide a persistent context across logically-connected asynchronous operations. Because our approach is based on zones it is applicable to all languages supporting a zone concept as part of their language definition or as a library, such as Dart. We provide a prototypical implementation of our approach in JavaScript as a ContextJS extension that integrates with the zone library Dexie.Promise.

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support of HPI's Research School[16] and the Hasso Plattner Design Thinking Research Program[17].

We want to thank David Fahlander, developer of Dexie, for his help with and explanations on micro task scheduling and zone echoing.

REFERENCES

[1] Malte Appeltauer. 2012. *Extending context-oriented programming to new application domains: run-time adaption support for Java.* Ph.D. Dissertation. University of Potsdam. http://d-nb.info/102624949X

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. 2009. A Comparison of Context-oriented Programming Languages. In *International Workshop on Context-Oriented Programming (COP)* (Genova, Italy). ACM, New York, NY, USA, Article 6, 6 pages. https://doi.org/10.1145/1562112.1562118

[3] Malte Appeltauer, Robert Hirschfeld, and Hidehiko Masuhara. 2009. Improving the development of context-dependent Java applications with ContextJ. In *International Workshop on Context-Oriented Programming, COP 2009, Genova, Italy, July 7, 2009*, Pascal Costanza, Richard P. Gabriel, Robert Hirschfeld, and Jorge Vallejos (Eds.). ACM, 5:1–5:5. https://doi.org/10.1145/1562112.1562117

[4] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. 2010. Event-Specific Software Composition in Context-Oriented Programming. In *International Conference on Software Composition (SC)*. Springer, 50–65. https://doi.org/10.1007/978-3-642-14046-4_4

[5] Sebastián González, Kim Mens, and Alfredo Cádiz. 2008. Context-Oriented Programming with the Ambient Object System. In *Proceedings of the 1st European Lisp Symposium (ELS'08), Bordeaux, France, May 22-23, 2008*, Pascal Costanza (Ed.). ELSAA, 17–32. https://european-lisp-symposium.org/static/proceedings/2008.pdf#page=23

[6] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. 2007. An Introduction to Context-Oriented Programming with ContextS. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers (Lecture Notes in Computer Science)*, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.), Vol. 5235. Springer, 396–407. https://doi.org/10.1007/978-3-540-88643-3_9

[7] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-oriented Programming. *Journal of Object Technology (JOT)* 7, 3 (March 2008), 125–151. https://doi.org/10.5381/jot.2008.7.3.a4

[8] Robert Hirschfeld, Hidehiko Masuhara, Atsushi Igarashi, and Tim Felgentreff. 2015. Visibility of Context-oriented Behavior and State in L. *Computer Software JSSST Journal* 32, 3 (2015), 149–159. https://doi.org/10.11309/jssst.32.3_149

[9] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2011. EventCJ: A Context-oriented Programming Language with Declarative Event-based Context Transition. In *International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 253–264. https://doi.org/10.1145/1960275.1960305

[10] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2015. Generalized layer activation mechanism through contexts and subscribers. In *14th International Conference on Modularity (MODULARITY), 2015* (Fort Collins, Colorado, USA). ACM, 14–28. https://doi.org/10.1145/2724525.2724570

[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings (Lecture Notes in Computer Science)*, Jørgen Lindskov Knudsen (Ed.), Vol. 2072. Springer, 327–353. https://doi.org/10.1007/3-540-45337-7_18

[12] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. 2011. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Science of Computer Programming (SCICO)* 76, 12 (2011), 1194–1209. https://doi.org/10.1016/j.scico.2010.11.013

[13] Stefan Ramson and Robert Hirschfeld. 2017. Active Expressions: Basic Building Blocks for Reactive Programming. *Journal on The Art, Science, and Engineering of Programming* 1, 2, art. 12 (2017), 49. http://arxiv.org/abs/1703.10859

[14] Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2017. The Declarative Nature of Implicit Layer Activation. In *Workshop on Context-oriented Programming (COP), 2017, co-located with the European Conference on Object-oriented Programming (ECOOP)* (Barcelona, Spain) *(COP '17)*. ACM DL, ACM, New York, NY, USA, 7–16. https://doi.org/10.1145/3117802.3117804

[15] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *CoRR* abs/1902.00549 (2019). arXiv:1902.00549 http://arxiv.org/abs/1902.00549

[16] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. 2019. Implementing Babylonian/S by Putting Examples Into Contexts: Tracing Instrumentation for Example-based Live Programming as a Use Case for Context-oriented Programming. In *Proceedings of the Workshop on Context-oriented Programming (COP) 2019, co-located with the European Conference on Object-oriented Programming (ECOOP), London, UK* (2019-07-15). https://doi.org/10.1145/3340671.3343358

[17] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. 2012. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software* 85, 8 (2012), 1801–1817. https://doi.org/10.1016/j.jss.2012.03.024

[18] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. 2007. Context-oriented Programming: Beyond Layers. In *International Conference on Dynamic Languages (ICDL), 2007* (Lugano, Switzerland). ACM, 143–156. https://doi.org/10.1145/1352678.1352688

A SOURCE CODE OF CONTEXTJS EXTENSION

The following listings are excerpts from our ContextJS extension[18] as described in section 5.

[16] https://hpi.de/en/research/research-school.html accessed on 30th June 2020

[17] https://hpi.de/en/dtrp/ accessed on 30th June 2020

[18] https://github.com/onsetsu/area51 accessed on April 30th 2020

Listing 2: Pushing a frame onto the active `LayerStack`. Life-cycle callbacks need to be handled when pushing onto this stack. Popping a frame from the `LayerStack` works analogously.

```
function pushFrame(frame) {
  const { withLayers, withoutLayers } =
      frame

  const beforePush = currentLayers()

  LayerStack.push(frame)

  withLayers && withLayers
    .filter(l => !beforePush.includes(l))
    .forEach(l => l.emit('activate'))
  withoutLayers && withoutLayers
    .filter(l => beforePush.includes(l))
    .forEach(l => l.emit('deactivate'))
}
```

Listing 3: The `withFrameZoned` function runs a callback in a new zone. Zone life-cycle callbacks ensure that the all logically-connected asynchronous operation run within the dynamic extent. Error handling is omitted for brevery.

```
function withFrameZoned(frame, callback) {
  const layerStackToRevertTo =
      storeLayerStack()
  const zonedLayerStack = storeLayerStack()
  zonedLayerStack.push(frame)

  return withZone(callback, {
    afterEnter() {
      replayLayerStack(zonedLayerStack)
    },
    afterLeave() {
      replayLayerStack(layerStackToRevertTo)
    }
  })
}
```