The Declarative Nature of Implicit Layer Activation

Stefan Ramson, Jens Lincke, Robert Hirschfeld

Software Architecture Group
Hasso Plattner Institute
University of Potsdam, Germany

9th International Workshop on
Context-oriented Programming, Barcelona 2017

Implicit Layer Activation (ILA)

Activation means to **declaratively** define extent of a layer activation

Activation status of a layer bound to a **Boolean predicate**:

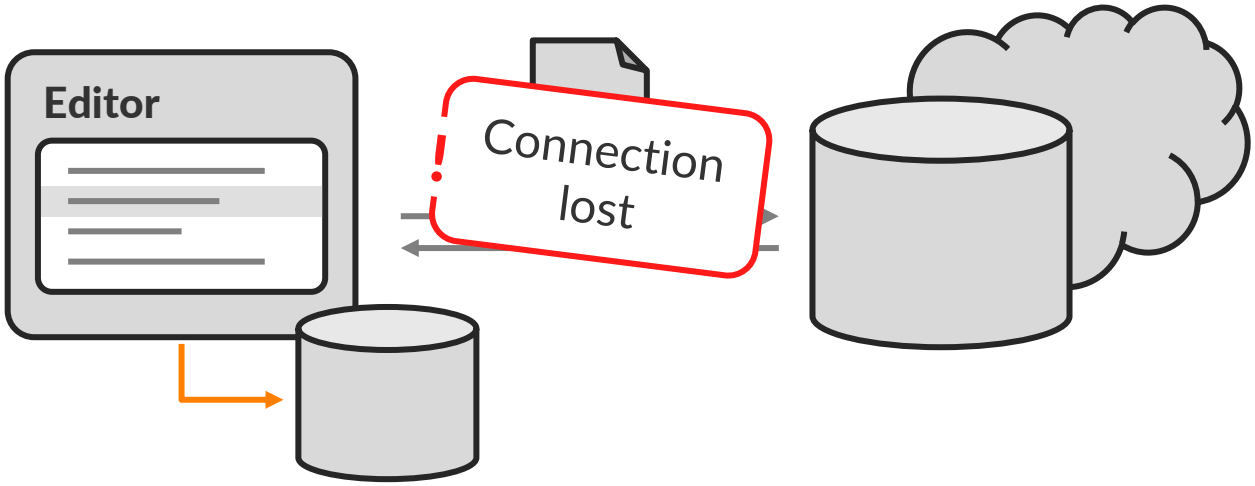
```
layer.activeWhile(condition)
```

Introduces a **constraint** to the program:

```
always: layer.isActive() === condition().?
```

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

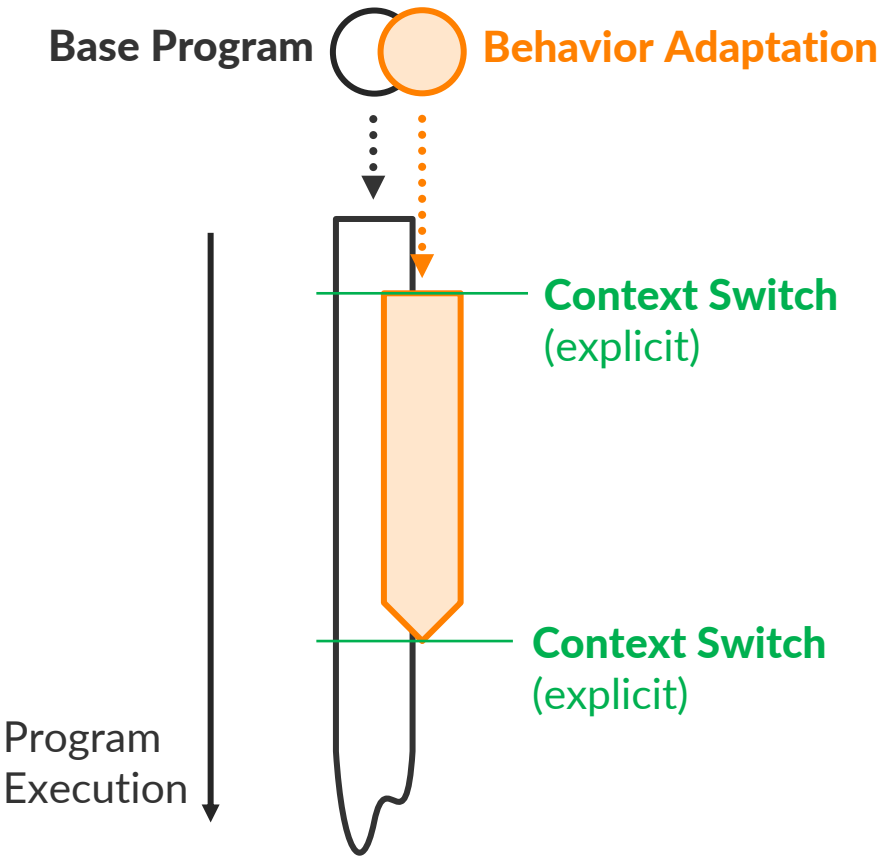
Motivational Example



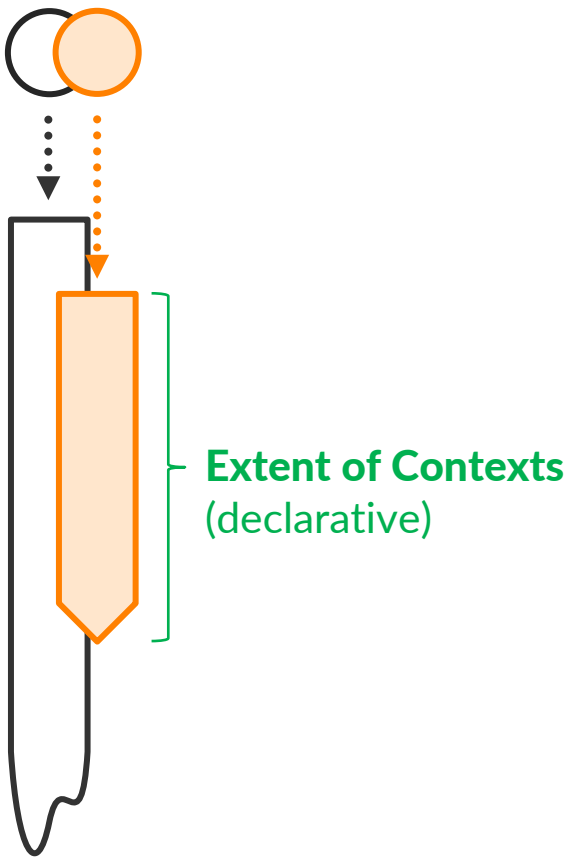

```
class Editor {
  save() { "write to remote storage" }
  [...]
}
```

```
layer('Connection lost')
  .refineObject(editor, {
    save() { "write to local storage" }
    [...]
  })
 .activeWhile(() => !webSocket.connected())
```

Transition versus Extent

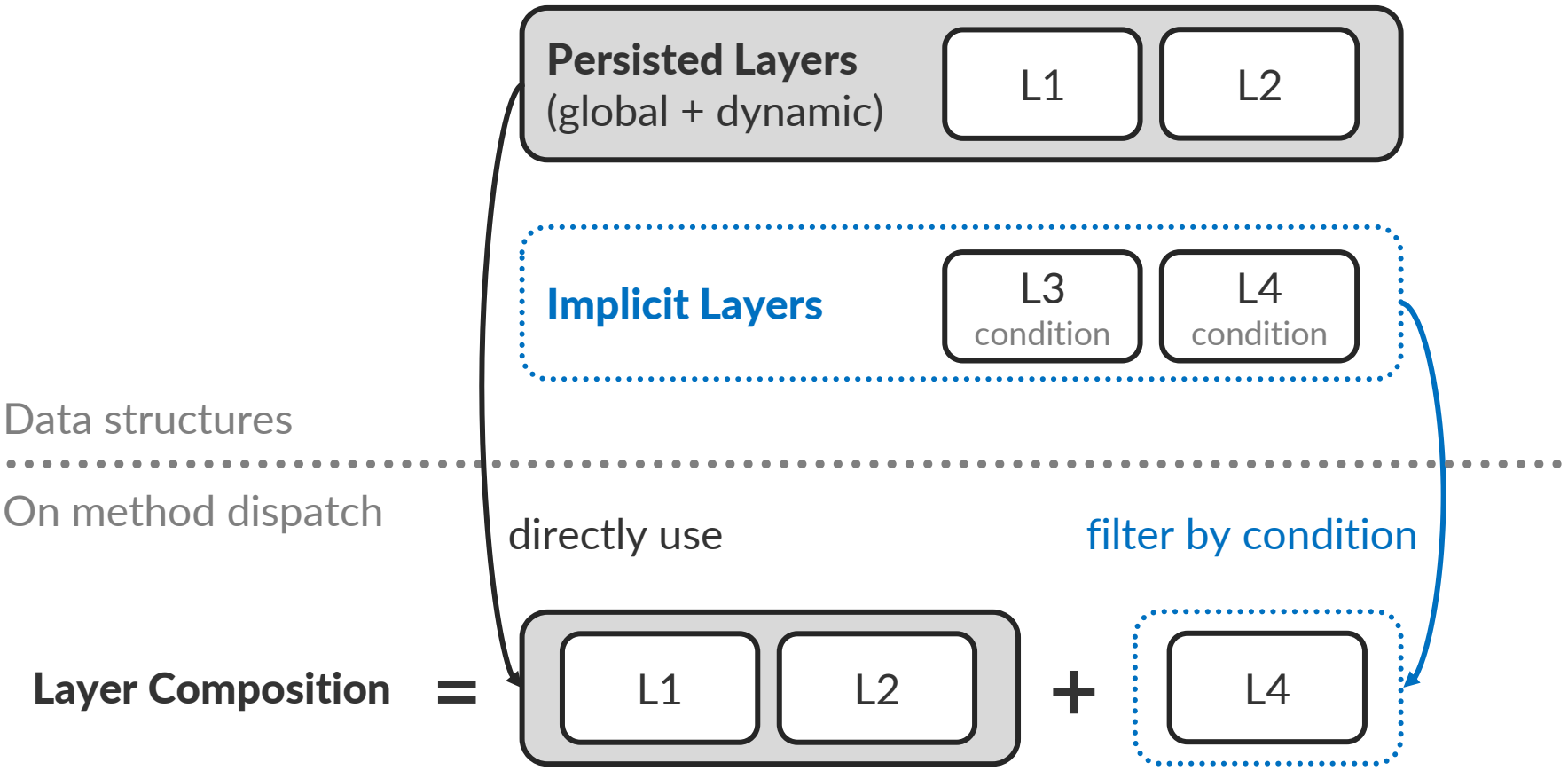


- Global Layer Activation
- Dynamic Layer Activation
- Event Transitions

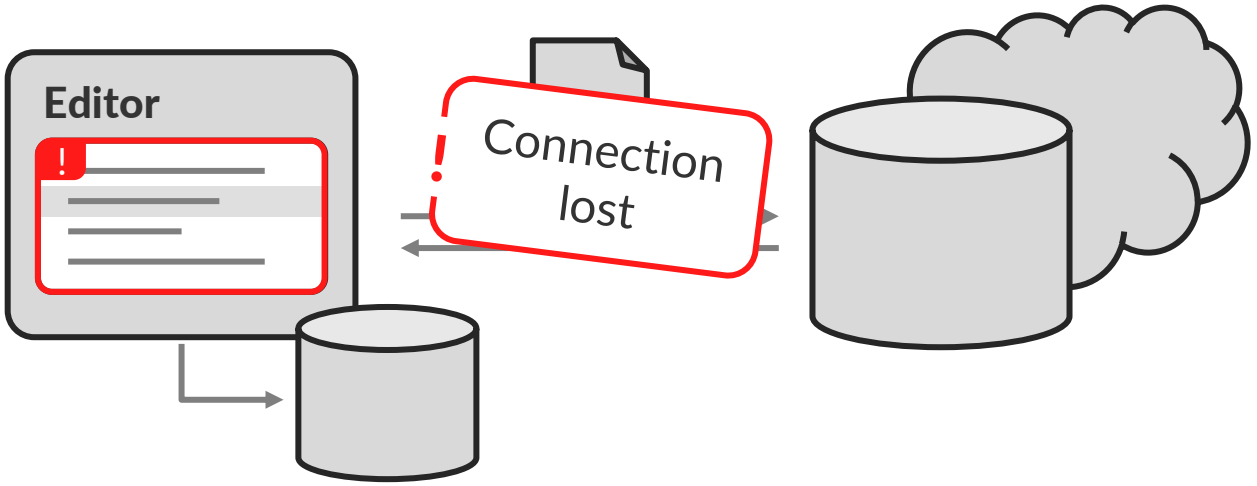


- Implicit Layer Activation

Traditional Mechanism



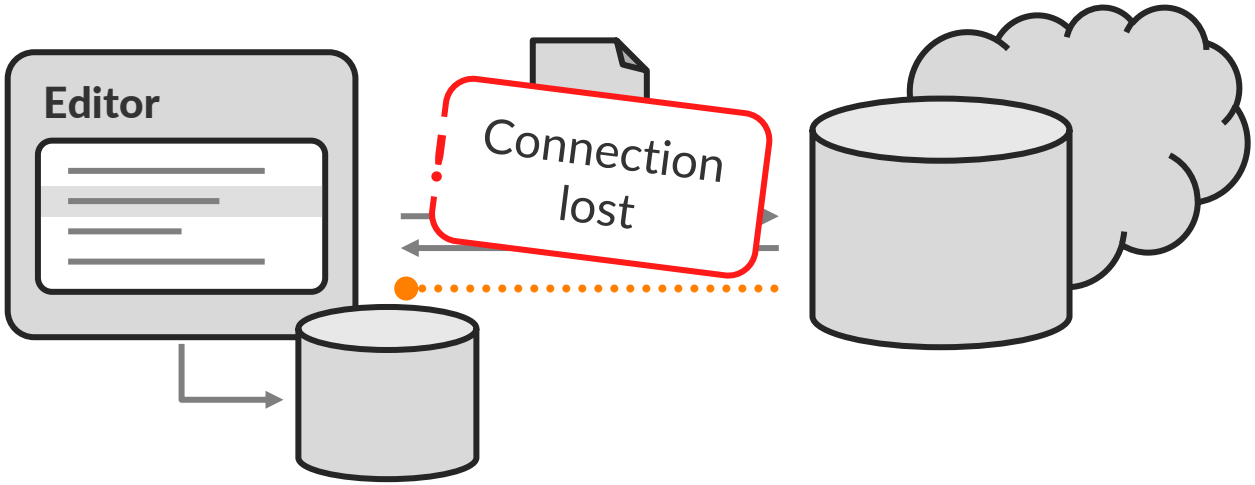
Active Entities




```
layer('Connection lost')
  .activeWhile(() => !webSocket.connected())
  .style(editor, 'offline')
```

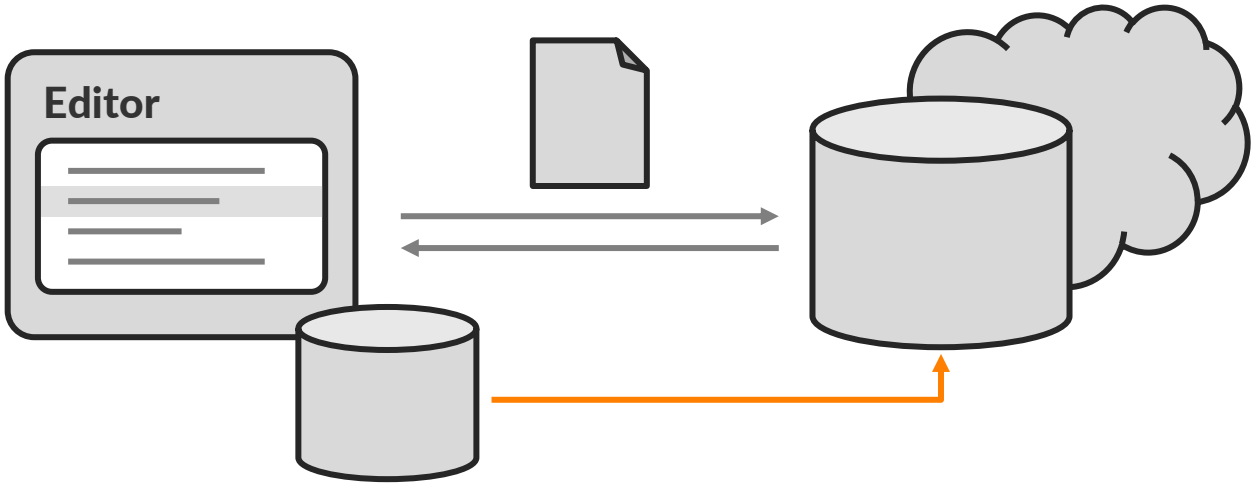
Stefan Lehmann, Tim Felgentreff, and Robert Hirschfeld. 2015. Connecting Object Constraints with Context-oriented Programming: Scoping Constraints with Layers and Activating Layers with Constraints. In 7th International Workshop on Context-Oriented Programming (COP). ACM, Article 1, 6 pages. DOI:https://doi.org/10.1145/2786545.2786549

Life-Cycle Callbacks




```
layer('Connection lost')
  .activeWhile(() => !webSocket.connected())
  .onActivate( "try to reestablish connection" )
```

Life-Cycle Callbacks




```
layer('Connection lost')
  .activeWhile(() => !webSocket.connected())
  .onActivate( "try to reestablish connection" )
  .onDeactivate( "sync remote with local storage" )
```

Problem Statement

Traditional implementation limits behavior adaptations to the Call-Return Model.

Recent trend of reactive and declarative mechanisms

Is an **integration** with such mechanisms viable?

Contributions

Extension of ContextJS with ILA in 2 Variants:

- Imperative implementation
- Reactive implementation

Comparison of the implementations regarding:

- Limitations
- Complexity
- Performance

Imperative Implementation

Existing mechanism

- When calling a layered method, the *currentLayers* function computes the layer composition.
- Either using a cached result, or by determining a new one using global and dynamic layers

```
export function currentLayers( ) {
  [...]
  if(!current.composition) {
    current.composition = composeLayers(LayerStack);
  }
  return current.composition;
}
```

Imperative Implementation

- Use a separate list *implicitLayers* to represent layers potentially activated through ILA
- Method *activeWhile* added to class *Layer*:

Add the layer to the list of implicitly activated layers, if necessary

```
activeWhile(condition) {
  if(!implicitLayers.includes(this)) {
    implicitLayers.push(this);
  }
  this.implicitlyActivated = condition;
  return this;
}
```

Store the given activation condition

Imperative Implementation

```
function getActiveImplicitLayers() {
  return implicitLayers
    .filter(layer => layer.implicitlyActivated());
}
```

filtering for layers with their conditions evaluating to true

Imperative Implementation

Extent existing computation of layer composition

```
export function currentLayers() {
  [...]
  return current.composition
    .concat(getActiveImplicitLayers());
}
```

Append all implicitly activated layers to the already computed layer composition

- Layer composition contains dynamically, globally, and implicitly activated layers
- Cannot cache implicitly activated layers

Reactive Implementation

Based on **Active Expressions**, a reactive primitive

```
aexpr(expression).onChange(callback)
```

Invoke callback whenever result of expression changes

Stefan Ramson and Robert Hirschfeld. 2017. Active Expressions: Basic Building Blocks for Reactive Programming. The Art, Science, and Engineering of Programming (<Programming>) 1, Issue 2 (2017). DOI:https://doi.org/10.22152/programming-journal.org/2017/1/12

Reactive Implementation

```
activeWhile(condition) {
  aexpr(condition)
    .onBecomeTrue(() => this.beGlobal())
    .onBecomeFalse(() => this.beNotGlobal());
  return this;
}
```

Comparison

- Implementation complexity
- Performance evaluation
- Properties and conceptual limitations

Implementation Complexity

#AST nodes (SLOC)	Complete	Difference to ContextJS
Unmodified ContextJS	2485 (568)	
Imperative Implementation	2557 (580)	72 (12)
Reactive Implementation	2525 (575)	40 (7)

Performance Evaluation

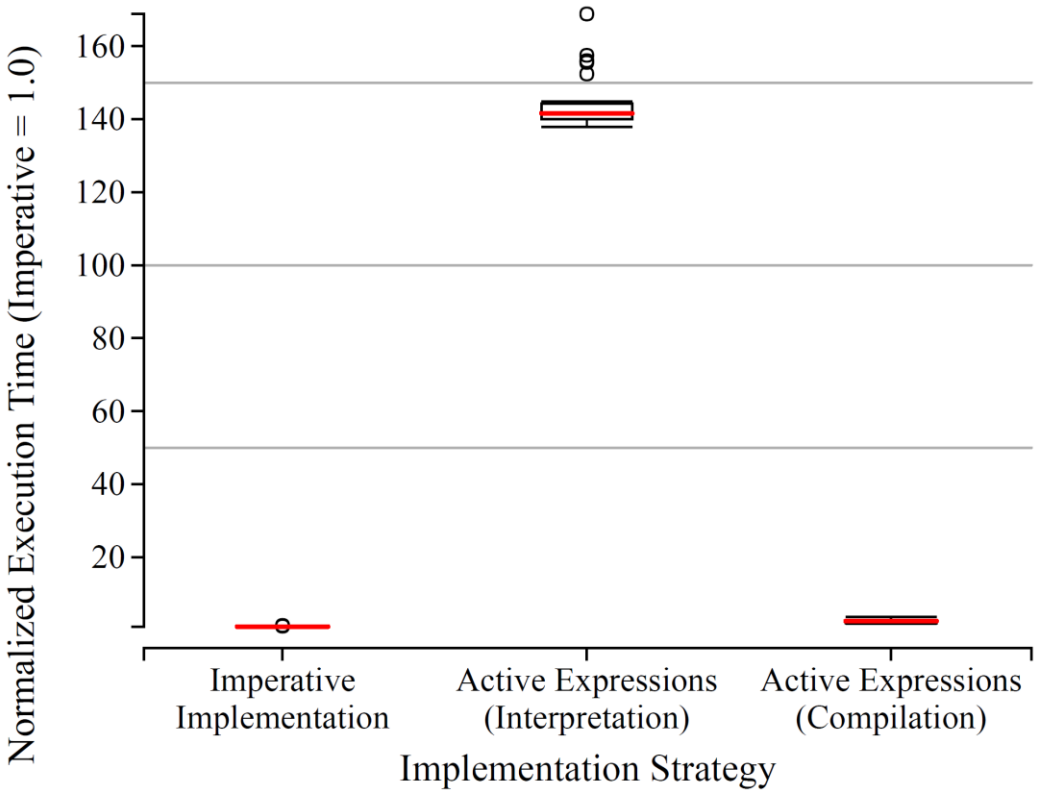
3 Variants:

- Imperative implementation
- Reactive implementation
  - Interpreter-based
  - Compilation-based

4 micro benchmarks:

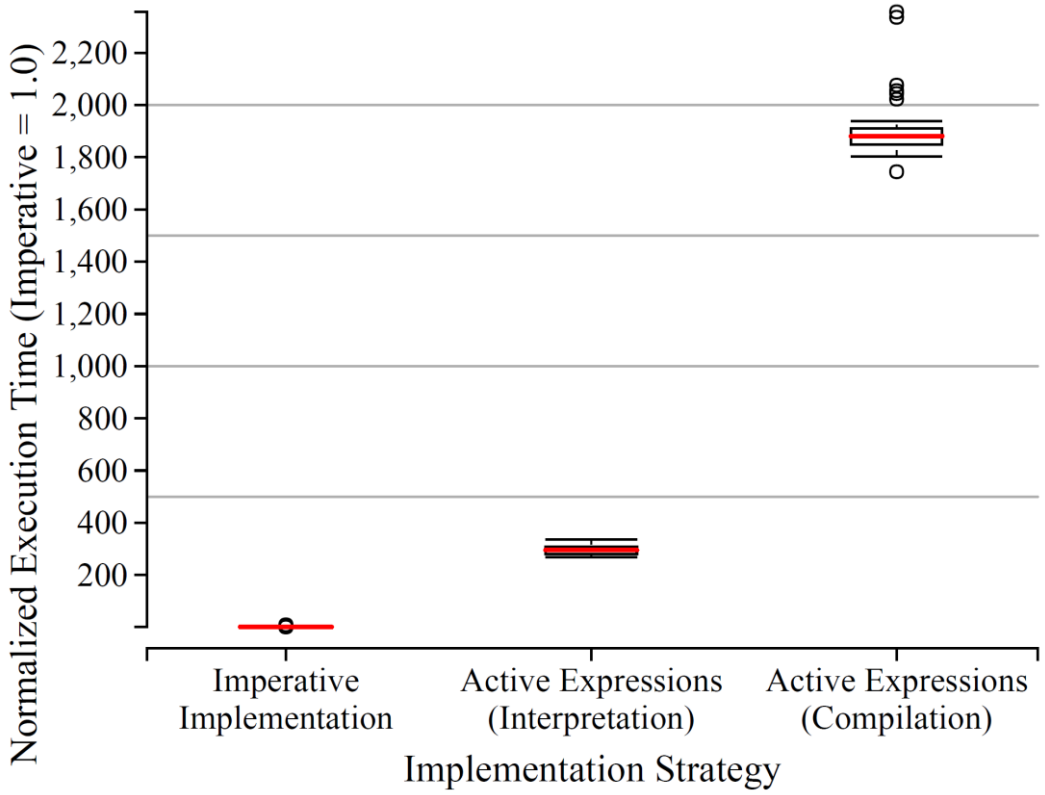
- Association of layer and predicate
- Frequently switch contexts
- Frequently invoke behavior
- Multiple activated layers

Initial association of layer with a **predicate** (10,000 layers)



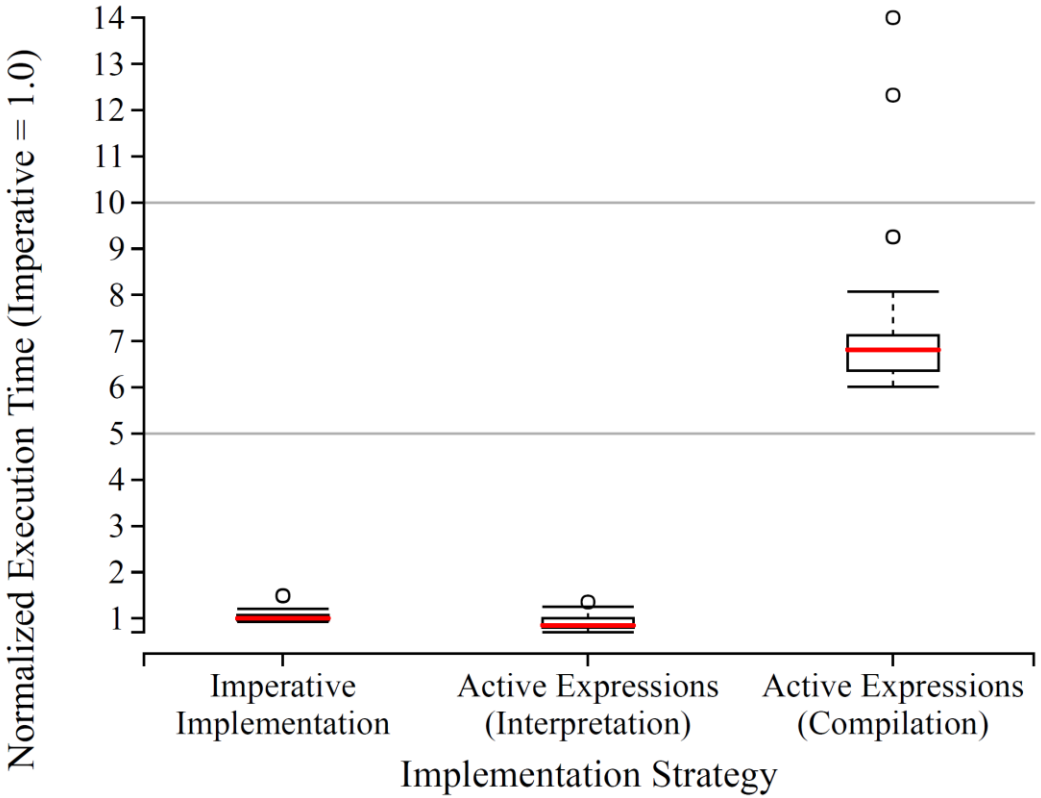
variant	timing [ms]	slowdown (vs Imperative)
Imperative	28.09	
Reactive (Interpretation)	3976.06	141.57 [138.74 - 144.77]
Reactive (Compilation)	70.29	2.50 [2.14 - 2.64]

High ratio of context switches to invocations (ratio: 1000 to 1)



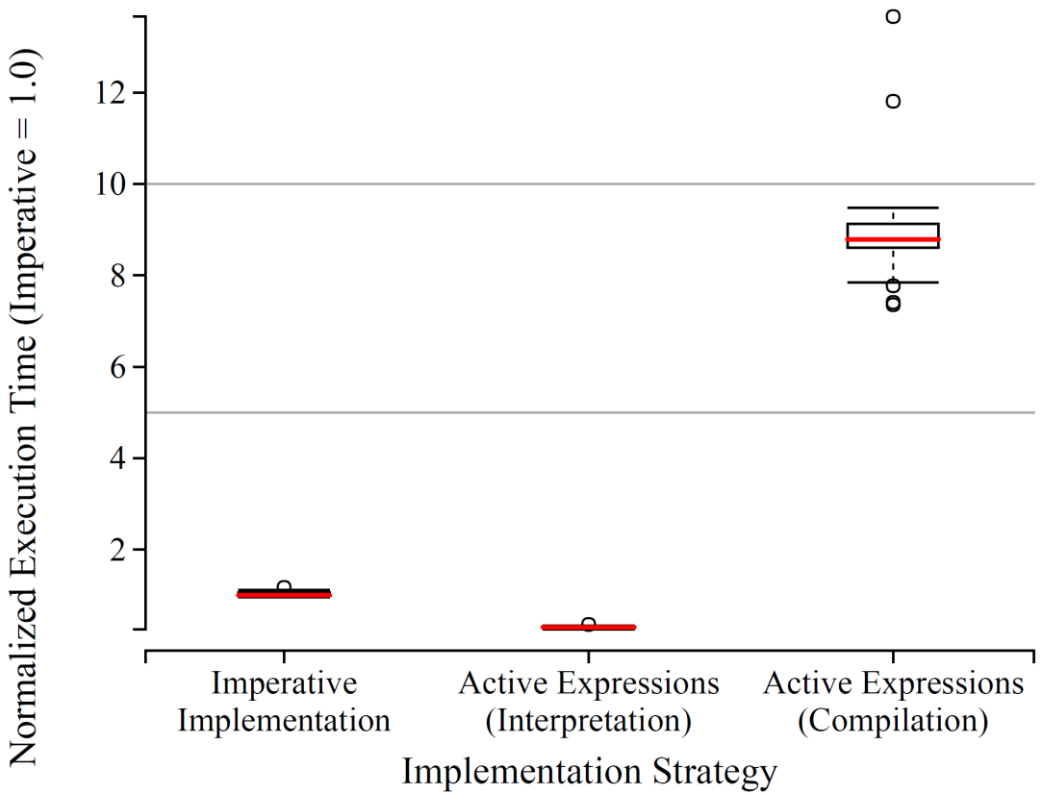
variant	timing [ms]	slowdown (vs Imperative)
Imperative	0.42	
Reactive (Interpretation)	125.63	295.61 [284.28 - 305.44]
Reactive (Compilation)	799.34	1880.81 [1851.14 - 1924.57]

High ratio of **invocations** to context switches (ratio: 1000 to 1)



variant	timing [ms]	slowdown (vs Imperative)
Imperative	17.63	
Reactive (Interpretation)	14.98	0.85 [0.80 - 0.96]
Reactive (Compilation)	120.12	6.81 [6.39 - 7.08]

Invoking a method with multiple layers (1,000 layers)



variant	timing [ms]	slowdown (vs Imperative)
Imperative	226.72	
Reactive (Interpretation)	68.80	0.30 [0.29 - 0.31]
Reactive (Compilation)	1992.30	8.79 [8.44 - 9.06]

Performance Evaluation: Insights

Imperative variant has smallest initial overhead

Computation:

- At **dispatch time** (imperative): favors frequent context switches
- On **change** (reactive): favors frequent invocations of context-dependent behavior

Reactivity slows down computation due to dependency tracking

Properties and Conceptual Limitations

Type of Implementation	Implementation Details	Resulting Properties	Supports
Imperative	on method dispatch pull-based	Lazy activation On-demand state Ephemeral layers	Call-Return Model
Reactive/ Constraint-based	on state change push-based	Eager activation Reified activation state	Call-Return Model Active Entities Life-Cycle Callbacks

Stefan Lehmann, Tim Felgentreff, and Robert Hirschfeld. 2015. Connecting Object Constraints with Context-oriented Programming: Scoping Constraints with Layers and Activating Layers with Constraints. In 7th International Workshop on Context-Oriented Programming (COP). ACM, Article 1, 6 pages. DOI:https://doi.org/10.1145/2786545.2786549

Conclusion

Neither variant is strictly better.
Decide according to your **use case**.

Reactive implementation opens up **design space** for Context-oriented Programming.