Reactive Object Queries

Consistent Views in Object-Oriented Languages

Stefan Lehmann Tim Felgentreff Jens Lincke Patrick Rein Robert Hirschfeld

Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
{firstname}.{lastname}@hpi.uni-potsdam.de

Abstract

Maintaining consistency between data throughout a system using scattered, imperative code fragments is challenging. Some mechanisms address this challenge by making data dependencies explicit. Among these mechanisms are reactive collections, which define data dependencies for collections of objects, and object queries, which allow developers to query their program for a subset of objects.

However, on their own, both of these mechanisms are limited. Reactive collections require an initial collection to apply reactive operations to and object queries do not update its result as the system changes.

Using these two mechanisms in conjunction allows each to mitigate the disadvantage of the other. To do so, object queries need to respond to state changes of the system.

In this paper, we propose a combination of both mechanisms, called *reactive object queries*. Reactive object queries allow the developer to declaratively select all objects in a program that match a particular predicate, creating a *view*. Additionally, views can be composed of other views using reactive operations. All views are automatically updated when the program state changes. To better integrate with existing imperative systems, we provide fine-grained events signaling view updates. We implemented the proposed concepts in JavaScript.

Our initial experience with example applications shows that the combined concept eases the integration of reactive mechanisms with object-oriented environments by avoiding scattered update code.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Data-flow languages; Object-oriented languages; D.3.3 [*Language Constructs and Features*]: Data types and structures

Keywords Events, Reactive Collections, Reactive Programming, Object Queries, Object-oriented Programming

1. Introduction

Manually defining and maintaining relations between collections of objects consistently throughout the system can be tedios and error-prone. *Reactive collections* [8] address this problem by making functional dependencies among data structures explicit. User can define desired dependencies using traditional collection operations such as `map` and `filter`. The reactive framework keeps track of dependencies and automatically updates dependent collections when the initial one changes. This allows the programmer to focus on *what* dependencies should hold rather than *how* to keep the system consistent under all possible conditions.

While reactive collections allow to define transformations in a declarative manner, initial collections are often updated in an imperative and explicit fashion. *Object queries* [13] solve this problem by integrating explicit queries into programming languages. Thus, this mechanism makes the whole program space queryable. Users can define sets of objects by describing desired attributes specified in SQL-like queries. To support this feature, the framework needs to keep track of all objects in the program, for example by using means of aspect-oriented programming [5]. While this concept provides high expressiveness through declarative queries, many implementations of object queries suffer from the view maintenance problem, that is the automatic update of materialized views [2]. Despite certain advances in the field [14], manual updates are still the norm. The result is missing data consistency, which is the key issue solved by reactive collections.

Both concepts, reactive collections and object queries, mutually excel at each other's problem area. Object queries allow to declaratively define basic collections for further transformations and reactive collections provide mechanisms to solve the view maintenance problem. In this paper, we propose a combination of both concepts to benefit from each concept while mitigating their respective disadvantages. The resulting combined concept allows to model any set of objects, called *views*, and their dependencies in a declaractive manner.

Despite their advantages, the mentioned concepts usually imply a relational programming style [11]. As a result, integrating these concept with stateful, imperative environments is still challenging. To further aid the integration with imperative environments, the framework has to consider the common characteristics of imperative systems, among which are statefulness and side effect-afflicted behavior. In order to support the modification of objects in an imperative way, we provide fine-grained events on the modification of views. The combination of these approches allows the developer to manipulate views on modifications of the current system state and to modify the system state based on view updates using events.

In summary, we provide the following contributions:

- A design that implicitly handles updates to collections of objects using an integration of reactive collections with object queries

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
ACM. 978-1-4503-4033-5/16/03...$15.00
http://dx.doi.org/10.1145/2892664.2892665

- The introduction of fine-grained events representing modifications of collections to aid the integration of reactive collections with imperative environments
- A prototypical implementation of the proposed concepts in JavaScript

In the remainder of this paper, Section 2 presents the key concepts of our integration of object queries with reactive collections. An implementation of these concepts is explained in Section 3. Section 4 shows how to modify an existing object-oriented application using the proposed concepts. Related concepts are presented in Section 5. Future work and our conclusions are described in Section 6 and Section 7, respectively.

2. Reactive Object Queries

In this section we describe the concept of *reactive object queries*, an integration of reactive collections and object queries. Additionally, we describe how to integrate the concept more tightly into an object-oriented environment using fine-grained events.

2.1 Querying Objects

Central to the concept of reactive object queries is the `select` method. The `select` method takes a class and a boolean expression as parameters and returns a *view* object, consisting of every instance of the class that matches the expression. To illustrate this, **Figure 1a** depicts a program space populated by multiple circle and square objects with different properties. The user queries all circles of a certain size to generate a corresponding view.

The view object is analogous to views in conventional relational databases in that it automatically updates whenever the underlying system state changes. For example, consider that the size of a circle object changes in a way that it now matches the aforementioned query as shown in **Figure 1b**. As a result, the view is adjusted to include the modified circle in order to keep consistency with the program space.

2.2 View Transformations

Using views in conjunction with reactive operations on collections offers the possibility to specify transformations on views that are independent of the imperative control flow. As an example, one could apply a `filter` to further refine the view shown in **Figure 1a**. The derived view would automatically adapt with its base view to system state modifications. The `map` operation could be used to generate a square of equal size for each queried circle. This operation creates new objects which are populated back into the object space. Thus, they can be queried as well. All derived sets respond to changes in the object space the same way base views do. As shown in **Figure 1b**, changing the size of the circle adds it to the base view. The devired filtered view reacts to this change by including the circle as well. As a consequence, a new square is created based on the circle.

2.3 Fine-Grained Events

View transformations impose a functional style of programing in that they create new objects based on changes in the program. However, in imperative environments new objects need to be integrated into the system appropriately, for example by introducing relations to already existing objects. Currently, there in no possibility to invoke imperative behavior out of view transformations. To allow such side effects, we provide fine-grained events that emit on modifications to a view. In particular, we introduce two types of events: an `enter` event, and an `exit` event. A view emits an enter event everytime a new object is added to it. This object is passed as an argument to every registered callback. Analogous to the enter event, each view emits an exit event whenever an object is removed from the view. If multiple objects are added to or removed from a view at once, for example because the view was just constructed, or the prediate changes, we emit one event per object added or removed.

To sum up, we provide a concept of object queries that allows developers to create views for objects of interest based on arbitrary expressions. Our framework automatically updates views whenever relevant program state is modified. Further views can be derived from base views using transformation operations. To better integrate with object-oriented environments, we provide fine-grained events on each modification of a view. **Listing 1** shows the methods currently supported by our prototypical implementation as JSDoc[1]-like interface.

```
Classes
  Class: View
    Methods
      filter(filterIterator(Obect) -> Boolean) -> View
      map(mapIterator(Object) - Object) -> View
      enter(callback(Object)) -> View
      exit(callback(Object)) -> View
      now() -> Array
      size() -> Number

Global
  select(Class, predicate(Object) -> Boolean) -> View
```

Listing 1: API currently supported by our prototype.

3. Implementation

We provide a prototypical implementation[2] of the proposed concepts in JavaScript. Because enter or exit event callbacks could involve arbitrary behavior, we have to actively update views on system changes. Early prototypes showed that reevaluating all queries whenever an object is created or mutated were not performant enough[3]. So, we only track instances of specified classes. To further reduce the number of checks to do, we only keep track of modifications of objects relevant to our queries. Therefore, we perform an abstract interpretation for each query to limit the number of properties and variables to be observed. By intercepting only the accesses to relevant properties and variables, we reduce the runtime overhead of object queries and still produce correct results. To update derived views we make use of a tree structure of views. Changes to views emit enter or exit events respectively. Implementation details are provided in the following.

3.1 Object Tracking

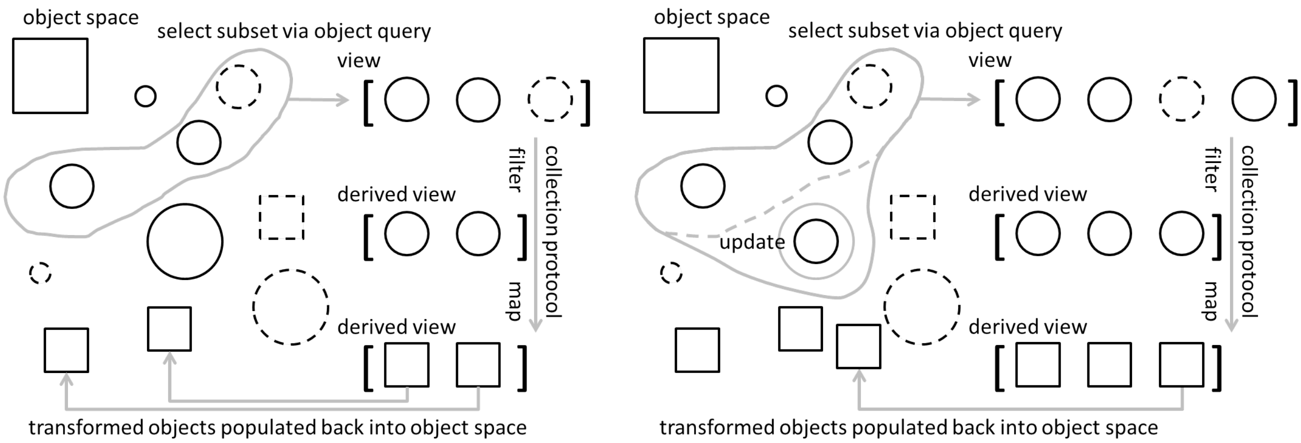
In order to continuously update query results, we need to keep track of the construction and modification of objects. To reduce performance overhead we do not keep track of every object by default. Instead, the user explicitly specifies which classes should be tracked. We use functional mixins provided by Flight[4] to define tracked classes. For example, `withLogging.call(Entity)` would instruct our framework to track all instances of the class `Entity`. The `withLogging` mixin installs an `after` advice on the objects `initialize` function. The advice adds the object to a base set of tracked objects.

[1] JSDoc `http://usejsdoc.org/` accessed on January 12th 2016

[2] Reactive Object Queries prototype `https://github.com/onsetsu/active-collection-prototype` accessed on January 3th 2016

[3] Early prototype `https://github.com/onsetsu/livsel` accessed on January 15th 2016

[4] FlightJS, Twitter `https://flightjs.github.io/` accessed on January 5th 2016



(a) (b)

Figure 1: Using reactive object queries, developers can query a subset of objects matching a particular predicate. The resulting view is always consistent to the current object space. By applying collection operations on views, developers can derive further views. Objects created by such transformations are populated back into the object space (a). As changes are introduced to the system state, the views act accordingly, reflecting the changed state by propagating modifications through the network of views (b).

3.2 Predicate Definition and Detection

Users can define an object query using the `select` method by providing a class as a base set and a Boolean expression. First, all objects of the base set that match the expression are immediately added to the result. In order to keep track on relevant changes of the objects, we intercept assignments to all variables referenced by the expression. To do so, the expression is interpreted for each object using the Lively Kernel [7] JavaScript interpreter[12]. The interpreter is customized using means of context-oriented programming [4, 6] to intercept the access to each property. Each property accessed during interpretation is wrapped with a transparent property accessor. Whenever a new value is assigned to a wrapped property or a new object is created, we check the expression result and add the corresponding object to or remove it from the result accordingly. Assignments of complex values necessitate reinterpretation. Each newly created object is automatically interpreted the same way.

Note, that the used interpreter relies on explicit access to the local scope of the expression. However, JavaScript does not support access to the local scope by default. So, we destilled and adapted the source code transformation from Babelsberg/JS [3], in order to capture the local scope of the expressions. We apply the transformation when a file is loaded to the page using a modified version of require.js[5].

3.3 Maintaining Derived Views

Calling collection protocol methods such as `map` or `filter` on query results creates further collections that need to be kept consistent to their base set. Therefore, each set maintains a list of sets that are derived from it, ultimately creating a tree structure. Whenever a set changes, it emits the `enter` or `exit` event respectively. We make use of these events to update child sets accordingly. To exemplify this, removing an object from a set emits an `exit` event. For example, a child set derived using `filter` receives this event and also removes the object if present.

[5] RequireJS `http://requirejs.org/` accessed on January 6th 2016

4. Example

To illustrate the interplay of mechanisms, we discuss an example scenario involving an object-oriented environment.

4.1 Application Scenario

We apply a modification to the Bloob soft-body physics and game engine[6]. A game in the engine is organized as multiple maps which are edited one at a time. Each map contains multiple layers which in turn contain multiple Entities. When dealing with larger maps, it is hard to keep track of all objects of interest using the built-in debugging tools. So, we want to add a simple debugging facility, called entity finder, to the engine. **Figure 2** shows a screenshot of the resulting entity finder utility[7]. The entity finder should provide the user with an input field and a dropdown list of all Entities whose name matches the input. Clicking on a list item should instruct the camera to focus on the respective Entity. The engine already provides a dropdown menu implemented as a thin wrapper around DOM elements[8]. We want to reuse this UI element.

4.2 Involving Reactive Object Queries

Listing 2 shows the complete code to create this example. First, we create the desired UI element using the `Dropdown` utility as shown in line 1 to 4. Then, we need to provide the menu with all objects of interest. Because Entities are scattered across multiple layers within a map in the engine, manually querying all objects of interest requires an implementation using nested loops. Additionally, accessing all Entities requires knowledge about internal data structures of the engine. Instead, we query the program for all objects of interest using the `select` method as shown in line 6 to 10. We provide the class of instances we are interested in, `Entity` in this case, as the first parameter. The second parameter is a Boolean ex-

[6] Bloob `https://github.com/onsetsu/bloob` accessed on January 3th 2016

[7] Reactive Object Queries example `http://onsetsu.github.io/active-collection-prototype/bloob.html` accessed on January 3th 2016

[8] Document Object Model specifications `http://www.w3.org/DOM/` accessed on January 10th 2016

pression that filters out all `Entities` whose name does not match the input string. The `select` method returns a view that consists of all Entities matching the expression. This view automatically updates whenever the input string changes, a new Entity is created, an Entity is removed, the name of an Entity is modified, and also when the implementation of the methods `includes` or `input` changes. Finally, we need to create list items for each matching Entity and attach them to the list. To do so, we derive a view from the Entity view using the `map` method. As depicted in line 13 to 18, the derived view creates a new link element for each Entity in the base view. Additionally, we register a callback to the click event in order to focus the camera on the respective Entity. The existing dropdown list is implemented statefully. So, we need to attach and remove list items explicitly. We use the fine-grained events provided by our approach to gradually modify the list. When a list item is added to the derived view, we attach it to the entity finder as shown in line 20 to 22. Analogously, we remove a list item from the DOM when the list item is removed from the view in line 23 to 25.

The presented implementation provides two major advantages over an imperative one. First, reactive object queries allow developers to specify views of interesting objects by their properties in a declarative manner. In contrast, an imperative implementation requires the developer to specify how to construct such a view explicitly. Second, the responsibility of creating and maintaining views is shifted into the framework. This results in clean, compact code, and avoids scattered code fragments to imperatively maintain multiple views.

```
1  var entityFinder = new Dropdown(
2    '#entityFinder',
3    'Blob');
4  entityFinder.show();
5
6  var matchingEntities = select(
7    Entity, function(entity) {
8      return entity.name.includes(entityFinder.input());
9    }
10 );
11
12 matchingEntities.map(function(entity) {
13   var item = document.createElement('a');
14   item.innerHTML = entity.name;
15   item.on('click', function() {
16     env.camera.track(entity.body, layer);
17   });
18   return item;
19 })
20   .enter(function(item) {
21     entityFinder.div.append(item);
22   })
23   .exit(function(item) {
24     item.remove();
25   });
```

Listing 2: Query for all `Entities` whose name contains the input string. Then show them as list items.

5. Related Work

Automatically deriving and transforming data has been investigated in research for a long time. We relate to approaches involving reactive lists, incrementalization or object queries as well as other data-centered applications.

5.1 Reactive Data Structures

Glazed Lists[9] is a Java library that allows to setup functional dependencies between data structures compatible to the Java `List` interface. The provided custom data structures can be transformed using operations such as `filter` or `sort` to create dependent lists. Additionally, the library allows to edit lists in provided GUI views in Swing or SWT applications. Like in most current approaches, dependent lists are only updated, if the input list is modified [11]. Changes to any other variables referenced in the operators do not trigger an update of the dependent lists. In contrast, our approach reacts to the changes to any variables that could affect the query results.

Flapjax [9] is a language for reactive web applications built on top of JavaScript. Flapjax provides explicit event streams as an abstraction for the communication with external web services. Streams can be transformed similar to the aforementioned reactive collections. The result are reactive UI elements that are updated with the data received from web services. Flapjax proposes reactivity through event streams instead of plain collections. Similar to Glazed Lists, reactivity is limited to the data processed using event streams.

Maier and Odersky [8] propose reactive collections. Reactive collections are created and updated automatically using data-dependency mechanisms from other lists. For example, each time the input list changes, a dependent list created with `map` updates accordingly. This is done by continuously listen to changes to signals which are specialized time-varying values. Most transformation methods have two versions, for example `map` and `sigMap`. While `map` only updates the output list when the input list changes, the `sigMap` method also updates the output list when any referenced signal changes. However, dependency tracking is limited to changes to signals referenced in the operators. So, dependent lists do not update automatically if ordinary variables change. Not supporting ordinary variables complicates the integration with existing object-oriented environments. In contrast to Maiers work, we do not introduce a seperation between reactive and ordinary variables. Instead our systems reacts to any changes to variables referenced in the `select` predicate that could potentionally affect the query result. As a result, our prototype can be used to extend existing object-oriented environments without modifications. However, we do not yet apply this behavior to our collection protocol. Here, similar to the non-prefixed operations, we only react on modifications of the input list.

5.2 Object Queries

The Java Query Language (JQL) [13, 14] allows queries over individual collections or the global set of all instanciated objects. So, views can be generated by simple, declarative query statements. However, the proposed work focusses on efficiency rather than reactivity. JQL caches queries and their results for repeated similar queries on data that has changed since the last query. As a result, JQL query results do not automatically update when the system changes but represent one-shot operations. In contrast, our approach maintains a persistent view on the program space.

Rothamel and Liu [10] present an efficient implementation to incrementalize query results. Despite using object queries, the system is more related to adaptive programming than reactive programming in that it allows to obtain efficient programs from existing non-incremental ones. In contrast, we expose reactivity to users by integrating object queries with reactive collections.

Entity Component System [1] is an architectural pattern commonly used in the context of game engine development. Every object in the scene of a game is represented as an entity. Entities are data holders for a set of components. Components are data objects representing a single aspect of an entity. Systems can query the game for entities that pocess certain aspects, respectively components. Each rendering cycle during the execution of the game, the systems perform global actions based on the queried entities. For

[9] Glazed Lists `http://www.glazedlists.com` accessed on January 4th 2016

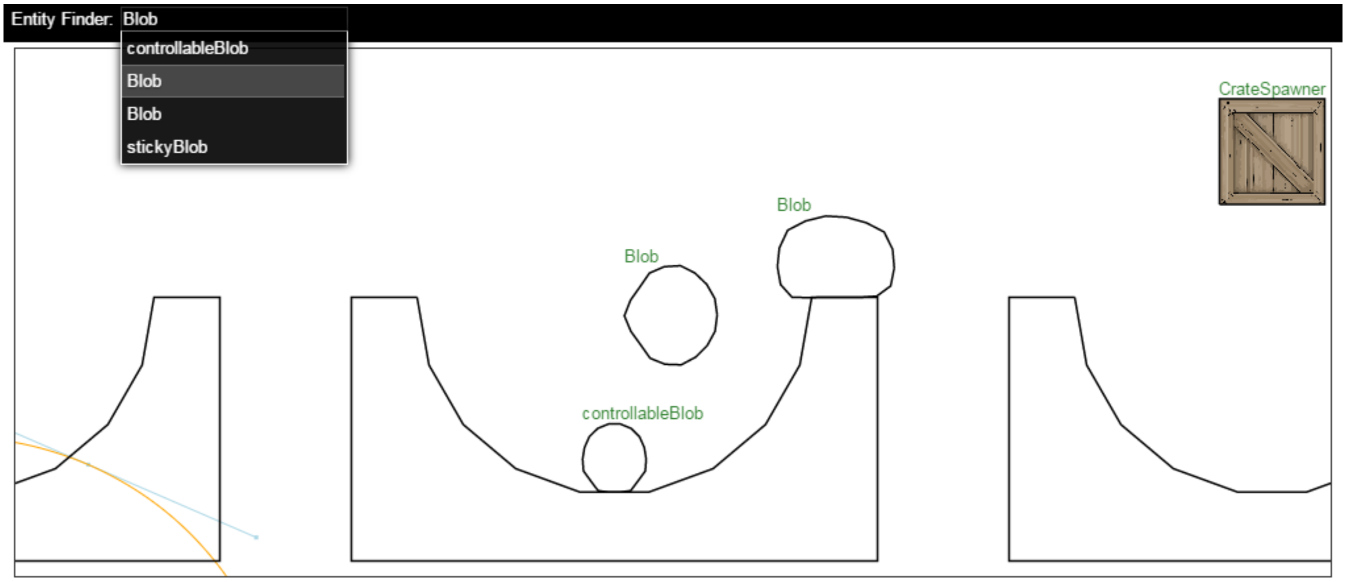


Figure 2: Screenshot of our entity finder extension (upper left corner) to the basic game view (center). The automatically updated dropdown menu is always consistent with the queried subset of `Entites`.

example, a simple physics system could query entities that have a position and a velocity component and apply a time-based physics simulation to each matching entity. Entity Component Systems offer a a simple data-driven design. Systems use queries to dynamically create a view on objects of interest. However, the used query mechanism imposes two major restrictions. First, views are only updated at fixed points in time, usually once per frame. Second, query conditions are limited to the presence and absence of components to increase performance. In contrast, our system updates queries continuously and supports arbitrary conditions.

6. Future Work

We propose to implement additional reactive operators involving multiple views as input or output. Moreover, we want to clarify operator semantics, integrate with other reactive concepts, and create dedicated debugging support.

Clear Operator Semantics for Stateful Environments Providing fine-grained events to react on modifications of views improves the integration of reactive collections with stateful environments. However, the overall problem, bridging the gap between reactive programming and stateful environments, is not solved completely. As an example, consider a map operator referencing a variable other than the list item. How should the system behave when this variable is modified? While recalculation is a valid option in context of immutable state, the semantics for mutable objects is not clear. One possiblity would be to change the properties of the mapped object. To do so, we could trace which properties of the base object lead to the values of which attributes of the resulting object and establish data dependencies between these properties.

Integration with other Concepts According to Salvaneschi [11] one main limitation of reactive collections is their limited domain. To increase the usability of this paradigm, an integration with other reactive concepts could be beneficial. Consider the conversion of views from and to observables or the usage of signals as time-varying return values for methods like `reduce` or `size`. In an object-oriented environment the adaptation of behavior based on views creates interesting possibilities. For example, one could interpret the containment in a view as an explicit context and dynamically activate a context-oriented programming layer for each object in a view [4].

Dedicated Debugging Tools Using reactive object queries leads to clean, declarative code. Yet, similar to other reactive concepts, queries are orthogonal to the control flow. The result is complex runtime behavior that is hard to debug. A dedicated debugging tool should be able to answer the following questions. Which views contain a specific object? How does a view relate to other views? Which views are potentially affected by a statement?

7. Conclusion

Manually maintaining collections of objects consistently throughout the system can be tedios and error-prone, especially in object-oriented environments. Reactive collections are a concept to define data dependencies between collections, however, initial collections have to be updated manually. Object queries allow developers to query their program for a subset of objects, but do not update as the system changes. Using both mechanisms in conjunction allows each to mitigate the disadvantage of the other.

In this paper, we have proposed the concept of reactive object queries. Those queries allow to declaratively select all objects in a program that match a particular predicate. The resulting views are analogous to views in relational databases in that they automatically update whenever the underlying program state changes. One can derive further views from these base views by applying reactive collection operation to them. As with base views, derived views automatically update in presence of changes. To better integrate into existing stateful systems, we provide fine-grained events on modifications of views.

We presented a prototypical implementation of the proposed concepts in JavaScript. Additionally, we described their usage with an explanatory example scenario.

Despite the presented future work, we think that reactive object queries already are useful in context of integrating reactivity with stateful systems.

References

[1] S. Bilas. A data-driven game object system. *Talk at the Game Developer Conference (GDC)* `http://gamedevs.org/uploads/data-driven-game-object-system.pdf`, 2002.

[2] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1991.

[3] T. Felgentreff, A. Borning, R. Hirschfeld, J. Lincke, Y. Ohshima, B. Freudenberg, and R. Krahn. Babelsberg/js - a browser-based implementation of an object constraint language. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*. Springer, 2014.

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

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*. Springer, 1997.

[6] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An open implementation for context-oriented layer composition in contextjs. *Journal of Science of Computer Programming (SCP)*, 2011.

[7] J. Lincke, R. Krahn, D. Ingalls, M. Röder, and R. Hirschfeld. The lively partsbin–a cloud-based repository for collaborative development of active web content. In *Proceedings of the Hawaii International Conference on System Science (HICSS)*. IEEE, 2012.

[8] I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*. Springer, 2013.

[9] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2009.

[10] T. Rothamel and Y. A. Liu. Generating incremental implementations of object-set queries. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2008.

[11] G. Salvaneschi and M. Mezini. Reactive behavior in object-oriented applications: an analysis and a research roadmap. In *Proceedings of the International Conference on Aspect-oriented Software Development (AOSD)*. ACM, 2013.

[12] C. Schuster. Reification of execution state in javascript, 2012.

[13] D. Willis, D. J. Pearce, and J. Noble. Efficient object querying for java. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*. Springer, 2006.

[14] D. Willis, D. J. Pearce, and J. Noble. Caching and incrementalisation in the java query language. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2008.