Manipulating GUI Structures Declaratively

Knut Anders Stokke
University of Bergen
Bergen, Norway
knut.stokke@uib.no

Mikhail Barash
University of Bergen
Bergen, Norway
mikhail.barash@uib.no

Jaakko Järvi
University of Turku
Turku, Finland
jaakko.jarvi@utu.fi

Abstract

GUIs often contain structures that are incidental, not properly manipulatable through well-defined APIs. For example, modifying a list of items in a GUI's model may require extraneous bookkeeping operations in the view, such as adding and removing event handlers, and updating the menu structure. Observing GUIs in practice gives an indication that programmers may find it difficult or tedious to implement complete and convenient sets of operations for manipulating various structures: useful operations for adding, inserting, swapping, or reordering elements are often missing, inconsistent, or limited. This paper introduces a DSL for programming operations that manipulate such incidental structures. The programmer specifies structures via relations between elements, concretely by defining methods that unestablish and establish a relation. This gives the programmer an ability to describe structural transformations via rules that control which relations should hold before and after a rule is applied. The API for structure manipulation is generated from these rules. Our DSL can give an abstract view on ad-hoc structures, making it easier to provide the necessary set of operations for their convenient manipulation.

CCS Concepts: • **Software and its engineering → Graphical user interface languages**; **Domain specific languages**.

Keywords: declarative programming, GUIs, separation of concerns

ACM Reference Format:
Knut Anders Stokke, Mikhail Barash, and Jaakko Järvi. 2020. Manipulating GUI Structures Declaratively. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3425898.3426956

GPCE '20, November 16–17, 2020, Virtual, USA

1 Introduction

Programming is to a large extent about manipulating data and the structures that data resides in. We routinely take advantage of well-known abstract data types (ADT) that specify operations and behavior of lists, queues, trees, graphs, and other common structures. Typically data manipulated through an ADT is stored in a concrete data type implemented with the ADT's interface in mind—the standard libraries of common programming languages contain many examples.

Often structures that we encounter as programmers are, however, more ad-hoc. They are not neatly encapsulated within the boundaries of a canned implementation. This is particularly true in Graphical User Interface (GUI) programming, which is the domain that primarily motivates this paper. In this domain, data is typically split between a *model* and *view*, and operations that affect the structure of the one should be reflected on the other too. Consider a GUI for specifying a multi-city flight search, where a user specifies a list of flight segments, the departure and arrival city and a date for each. Presumably the GUI's model is a list or an array of such records, but the manipulation of the flight segments is more complex than invoking *insert* or *erase* methods of some underlying list data structure: when the model's structure changes, the view may require changes too, including adding or removing textboxes or other widgets, adding and removing event handlers on these widgets, and modifying validation logic, e.g. to ensure that dates are increasing. The programmer cannot merely consider the list of flight segments as a predefined data structure that has operations for addition, deletion and reordering—the programmer must, for each operation, ensure that the view is kept up to date with the model. We have an *incidental* data structure on our hands, lacking operations for manipulating it.

Observations about GUIs today support the view that implementing operations for structural modifications in GUIs is a challenge. It is common that features for structure modification are limited and cumbersome. Continuing with the flight search example, almost no flight booking services allow adding flight segments in the middle of a multi-city search; we checked 30 services, of which only one has this feature. Frequent travellers know that this would be a convenient feature when exploring options for complex itineraries, yet not even services that have millions of users provide it.

GUIs without adequate structural operations frustrate in all walks of life. Many encounter in-house applications whose

Figure 1. ApplyTexas' form for extracurricular activities.

GUI features for structure manipulation are very limited. As a representative example, the *ApplyTexas* [1] website for admissions to higher education institutions in the State of Texas asks, as a part of the application, the applicant to specify up to ten extracurricular activities *in the order of importance*. Each activity has 23 fields: text inputs, numerical inputs, and checkboxes, as shown in Figure 1. The GUI does not, however, let the user to reorder the activities in any way. To move an activity up or down in the form would require swapping activities by swapping (with copy-paste) each of the 23 fields of the two activities. In practice, if the user wants to reorder the activities, the choices are to fill the form again or content oneself with a suboptimal order. This time-wasting GUI, subjected to hundreds of thousands of students yearly, has been part of ApplyTexas for more than a decade. The lack of reordering operations is unfortunate and unnecessary—our mock implementation of a similar form in Figure 4 shows that implementing such operations is quite simple. In general, programmers of applications with smaller user bases can afford to put less effort on producing feature-rich GUIs, so the problem of missing structural operations is more prevalent in such applications.

This paper suggests making the incidental structures that appear in GUIs explicit to the programmer. Our DSL can provide an abstract view over such messy structures. This view is based on the programmer defining relations between elements, such as an element preceding another in a sequence, and how these relations are established and unestablished. An API for structural modifications is then specified as a set of transformation rules that control which relations hold before and after a rule is applied. Often these transformation rules are reusable for many relations. By defining a few relations for each structure that appears on a GUI, insertions, deletions, reorderings, and other restructurings can be supported with ease. Our DSL gives the programmer a principled approach for implementing operations that affect a GUIs structure, instead of resorting to whatever means of implementation feels expedient.

2 Language for Structure Manipulation

This section presents a DSL for specifying relationships between elements in structures and defining rules that modify those structures. Formally, a *structure specification* can be thought of as a triple $S = \langle C, R, T \rangle$, where

- C is a set of *components*,
- R is a set of finitary *relations* on components, and
- T is a set of *transformation rules* that express (all) possible manipulations that modify component relations.

A relation defined on components is intended to capture the structural link between component instances. While the semantics of such a link is left implicit, it is dependent on the application, one specifies explicitly how this link can be established and unestablished, and how one can test whether it exists between given component instances. These specifications are expressed as imperative code blocks in a general-purpose language; our choice is JavaScript because of its prevalence in GUI programming.

A *relation* r defined on components $c_1, ..., c_k \in C$ is a quadruple $r = \langle \{c_1, \ldots, c_k\}, r^P, r^E, r^U \rangle$, where

- r^P is an imperative code block that acts as a predicate *testing* whether the relation holds,
- r^E is an imperative code block that *establishes* the relation,
- r^U is an optional imperative code block that *unestablishes* the relation; if this block is present then we call the relation *proper*.

After relations between the components of a GUI have been defined, one can specify transformation rules to "turn relations on and off", and in this manner modify the structural links between component instances.

A *transformation rule* t is a triple $t = \langle \{c_1, \ldots, c_k\}, t^P, t^Q \rangle$, where $t^P \subseteq R$ is a sequence of *premises*, proper relations that must hold for the transformation rule to be applicable, and $t^Q \subseteq R$ is a sequence of *consequences*, relations that shall hold after the transformation rule has been applied. An *application* of transformation rule t unestablishes all relations from set t^P and establishes all relations from set t^Q.

We can now introduce a domain-specific language that enables defining structure specifications as described above. We showcase the language by specifying an assumed HTML-based GUI that supports operations on elements (`Li` nodes of the document object model) in lists (`Ul` nodes).

The structure specification $S = \langle C, R, T \rangle$ concerns components of two types: $C = \{c_{\texttt{Li}}, c_{\texttt{Ul}}\}$, corresponding to an element and a list, respectively. These components can be declared in our DSL as follows:

```
components
   Li <-> ''' «Li» instanceof HTMLLiElement '''
   Ul <-> ''' «Ul» instanceof HTMLUListElement '''
```

A declaration of a component includes its name followed by an imperative code block in JavaScript, enclosed in triple quote marks. The code tests if an instance of a component—a DOM node—is of the expected type. Guillemets can be used to refer to identifiers declared in our DSL, similarly to template strings in Eclipse Xpand [10].

Instances of components are called *placeholders*. Below we introduce two instances of component $c_{\texttt{Li}}$ and one instance of component $c_{\texttt{Ul}}$.

```
placeholders
   a: Li
   b: Li
   c: Ul
```

Next we introduce a binary relation $r_{\texttt{precedes}} \in R$, defined on any two instances of $c_{\texttt{Li}}$, expressing the fact that one instance precedes another in a given list $c_{\texttt{Ul}}$. We keep the list-container implicit, because in this case the container can be accessed from its elements. If this was not the case, $r_{\texttt{precedes}}$ could be defined between two container-element pairs or as a ternary relation on two elements and a container, to give the code blocks access to the container. The implementation defines a testing predicate and a code block that establishes the relation.

```
relations
   (precedes) a b ::=
      test
         ''' «a».nextElementSibling === «b» '''
      establish
         ''' «a».parentNode.insertBefore(«b»,
                                «a».nextSibling); '''
```

Using the relation `precedes`, a transformation rule that implements swapping of two adjacent elements `a` and `b` can now be defined. The premise of this rule requires that the relation $r_{\texttt{precedes}}$ holds for instances `a` and `b` of the component $c_{\texttt{Li}}$, whereas its consequence establishes the same relation, but with the order of arguments changed: $t_{\texttt{swapAdjacent}} = \langle \{c_{\texttt{Li}}\}, \{r_{\texttt{precedes}}(a, b)\}, \{r_{\texttt{precedes}}(b, a)\} \rangle$. The specification of this transformation rule in our DSL is as follows:

```
rules
   swap (a b) = a precedes b => b precedes a
```

When applied to two `Li` component instances `a` and `b`, this rule first checks whether the instances are indeed of type $c_{\texttt{Li}}$. After that, the relation `a precedes b` is unestablished, and the relation `b precedes a` established.

Syntactically, a rule specification is a name, followed by a list of placeholders referenced in the rule, followed by an equation sign, a list of premises, an arrow sign, and a list of consequences. Both the premise and consequence lists use comma as the element delimiter. These lists define the order in which relations are unestablished and established when the rule is applied—the order matters since (un)establishing a relation involves executing imperative code.

The following example shows a rule with two relations in the consequences list; it defines a transformation for inserting an element between two existing consecutive elements.

```
insertBetween (a b c) =
    a precedes b => a precedes c, c precedes b
```

The programmer has a complete freedom in what relations they define, so no algebraic properties between the relations can be assumed. Consider the relations `isIn` and `notIn`, defined on an element `a` and a container `l`, that capture the containment of the element in the container.

```
(isIn) a l ::=
   test
      ''' «l».contains(«a») '''
   establish
      ''' «l».insertBefore(«a», «l».childNodes); '''
```

Testing whether relation `notIn` holds is a negation of the testing predicate of relation `isIn`. The relations cannot, however, be expressed as a negation of each other because establishing them requires performing different operations on the container. We thus need a separate definition of `notIn`:

```
(notIn) a l ::=
   test
      ''' !(«l».contains(«a»)) '''
   establish
      ''' «l».removeChild(«a»); '''
```

We can now define the rule to remove a component from a list by first verifying that "the element *is* in the list", and, if so, then establishing that "the element *is not* in the list".

```
remove (a l) = a isIn l => a notIn l
```

A transformation rule for inserting an element into a container can be defined in a symmetric way:

```
insert (a l) = a notIn l => a isIn l
```

In the above example the declarative rules are obviously mere dressing for the underlying imperative operations, but the rules nevertheless make the imperative actions' effects on structural relations explicit.

From a specification of structural changes expressed in our DSL, an API in JavaScript is generated. We explain now how components, placeholders, relations, and rules are transpiled into JavaScript.

For each component c, a function `expect_`$c(a)$ is generated. This function returns `true` if its argument is an instance of component c, and throws an error otherwise. These type-checking functions are invoked on JavaScript objects whenever transformation rules are applied.[1]

[1] Many of the type-checking functions would be unnecessary if targeting a statically or gradually typed language, such as TypeScript.

For each *relation specification* r defined on instances a_1, ..., a_k of components c_1, ..., c_k, we generate three JavaScript functions: `test_`r(a_1, ..., a_k), `establish_`r(a_1, ..., a_k), and `unestablish_`r(a_1, ..., a_k). The first function is a predicate that checks, by invoking functions `expect_`c_i, that its parameters have the same type as their corresponding placeholders and then, by running the JavaScript code given in the specification of r, that r holds. The second and the third generated functions are symmetrical. They first type-check their arguments a_1, ..., a_k, and then establish (unestablish) the relation r, by executing the imperative code specified in the `establish` (`unestablish`) block of the specification of the relation r. Then these functions check that the relation indeed has been established (unestablished) by calling the testing function `test_`r.

For each *transformation rule* t, we generate a JavaScript function. This function first type-checks its arguments and checks the rule's premises. It then invokes `unestablish_`p for every premise p and `establish_`q for every consequence q. Finally, it checks whether the consequences of the rule hold. Figure 2 shows an example of the generated JavaScript function for the `insertBetween` rule.

We used the language workbench Eclipse Xtext [2, 7] to implement our DSL. Language workbenches [6, 8] let the programmer specify the syntax, typing rules, and code generators for a language, based on which they output a tailored IDE with standard services, including a syntax-aware editor, code completion, and automatic code corrections [13]. From the grammar specification for our DSL, Xtext generates a model using Eclipse Modeling Framework [14]. The model is populated during parsing, producing essentially an abstract syntax tree that can be further analyzed or transformed. We use the transformation language Eclipse Xtend [5] to generate the JavaScript code from our DSL programs. Figure 4 presents a screenshot of a working Eclipse instance of our DSL, relations and rules of a GUI similar to ApplyTexas.

3 Case Study

This section puts our DSL to use in the implementation of a simple timetable planning application. The user interface of the application, in Figure 3, shows a sequence of days, where each day contains a list of events scheduled one after another. The user can edit the starting time of each day and the duration of each event. The GUI responds to such edits by adjusting events' start and end times, so that there are no holes in the schedule.

To implement this application, the programmer defines the GUI's structures and their modification rules using our DSL, offered to the programmers as an Eclipse-based IDE. From the DSL program, the IDE generates JavaScript functions for manipulating the structures, which are then imported to the web application.

```
                    rule name          placeholders mentioned in rule
function insertBetween(a, b, c) {

  for placeholders
  expect_Li(a);
  expect_Li(b);
  expect_Li(c);
            component type
            corresponding to placeholder

  premises =
    true
    for each premise
    && test_precedes(a, b)
    ;          relation

  if (!premises) {
    console.log("Rule not applicable");
    return;
  }
  for each premise
  unestablish_precedes(a, b);
  for each consequence
  establish_precedes(a, c);
  establish_precedes(c, b);

  consequences =
    true
    for each consequence
    && test_precedes(a, c)
    && test_precedes(c, b)
    ;

  if (!consequences) {
    console.log("Failed to apply rule");
  }
}
```

Figure 2. JS code generated for the `insertBetween` rule.

To keep the code of this example simple, we delegate the value updates to a constraint system, to a library called *HotDrink* [9], that takes care of updating variables' values whenever variables that they depend on change. The programmer specifies *constraint system components* (cs-components for short), consisting of *variables* and *constraints*. We define the cs-component `event` as follows:

```
component event {
  var &previous_end, start, duration, end, title;
  constraint c1 {
    m(previous_end -> start) { return previous_end; }
  }
  constraint c2 {
    m(start, duration -> end) {
      return addDuration(start, duration);
    }
  }
}
```

This cs-component *owns* four variables (`start`, `duration`, `end`, `title`) and has one *reference* (`previous_end`), to be connected to another event's `end` variable. The HotDrink library keeps the constraints `c1` and `c2` enforced by executing

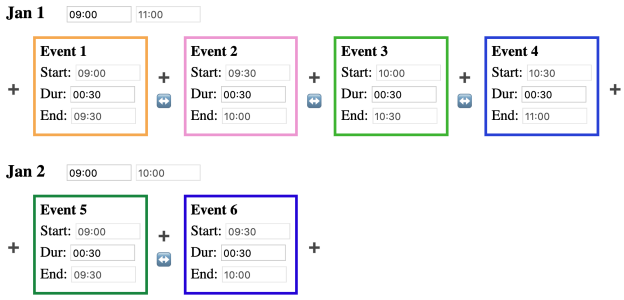

Figure 3. User interface of the timetable application.

their *methods* (named `m` in both constraints) when necessary. The constraint `c1` ensures that an event's start time is the same as the previous event's ending time. The constraint `c2` ensures the right ending time of an event. In enforcing a constraint, the return value of a method is written to the variable to the right of the arrow in the method signature.

For each pair of consecutive events (e_{k-1}, e_k), we impose the constraint `c1` between e_{k-1}`.end` and e_k`.start`. This is achieved by connecting e_k`.previous_end` to e_{k-1}`.end`, which is expressed as the assignment e_k`.previous_end =` e_{k-1}`.end`.

Thanks to HotDrink's constraint system, the code needed to realize structural changes to the GUI's model stays manageable. This is not enough, however, since the model is connected to one or more views. The structural changes in one must be kept in sync with that in the other; our DSL lets the programmer specify these changes hand in hand.

To implement the example application, we define the components `Day` and `Event`, both consisting of a view and a HotDrink cs-component bound to that view. These two members are named `view` and `model`.

The relation that specifies that two events are consecutive can then be defined as follows. The model's `vs` member gives access to a cs-component's variables and its `system` member to the underlying constraint system. The call to `system.update()` forces HotDrink to enforce all constraints.

```
(precedes) a b ::=
   test
      ''' referToSameValue(«b».model.vs.previous_end,
                           «a».model.vs.end) '''
   establish
      ''' «b».model.vs.previous_end = «a».model.vs.end;
          «a».model.system.update();
          «a».view.parentElement.insertBefore(«b».view,
                           «b».view.nextElementSibling);
      '''
```

We can now define the desired operations on the events of our application as transformation rules in our DSL. For instance, the rule

```
swapEvents (a b c d) =
   a precedes b, b precedes c, c precedes d =>
   a precedes c, c precedes b, b precedes d
```

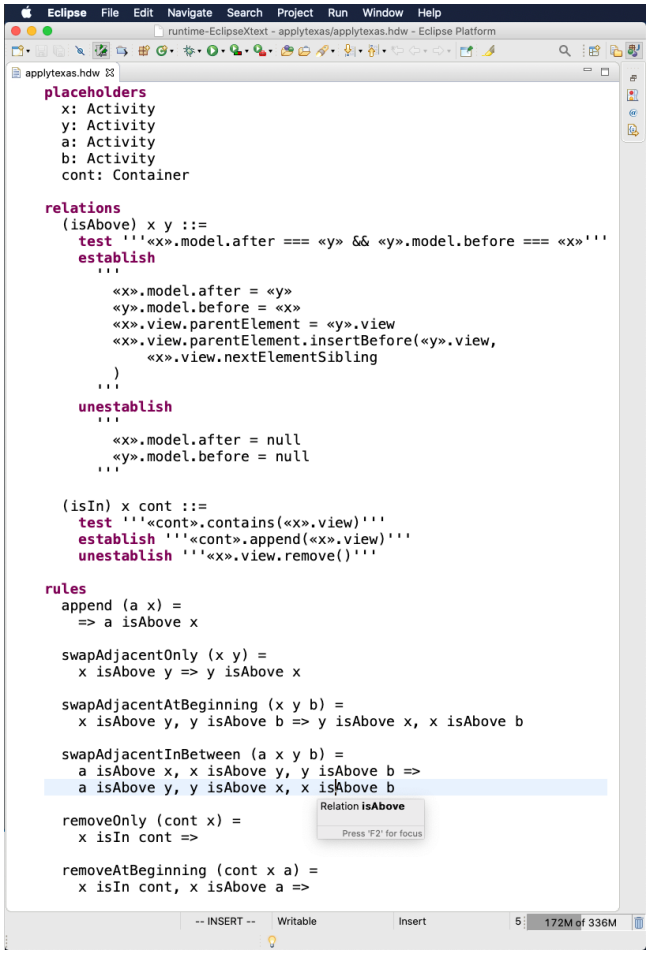

Figure 4. The Eclipse IDE for our DSL, showing code of the ApplyTexas GUI mockup. The IDE, implemented with Eclipse Xtext, supports syntax highlighting, autocompletion, hyperlinking, and hover tooltips.

specifies the operation of swapping two consecutive events `b` and `c` when they are not, respectively, the first or the last event.

In a similar way we can define the transformation rule for swapping two consecutive events `a` and `b` when `a` is the first event of its day:

```
swapEventsAtBeginning (day a b c) =
   a firstOf day, a precedes b, b precedes c =>
   b firstOf day, b precedes a, a precedes c
```

For each transformation rule, our DSL program generates a JavaScript function that can be called to apply the transformation, as explained in Section 2. For instance, to swap two consecutive events `e1` and `e2` in a day `d` where the event `e3` follows `e2`, we use the following JavaScript statement:

```
swapEventsAtBeginning(d, e1, e2, e3);
```

As can be seen, all of the complexity of a structural modification remains in the DSL program.

4 Discussion

The examples above highlight the benefits of our DSL. By explicitly defining rules according to which a structure is manipulated, we separate the code concerned with the structure from code that uses the structure. The notion of a structure in our approach is very loose, and does not necessarily align with data structures' boundaries of encapsulation. Manipulating such implicit structures directly with whatever collection of APIs they make available is ad-hoc and error prone, and does not impel the programmer to provide the best GUIs for the users. Specifying explicitly the rules of how a given structure can be modified increases the likelihood of the correctness of structural modifications, and once the transformations are in use, programming GUIs rich with features is much less laborious.

The examples above were all concerned with list structures, the planning application involving lists of lists (a list of days, each a list of events). The approach, however, generalizes to any kind of structures, such as trees and graphs, and their combinations. As mentioned in Section 2, there are no limitations on the relations the programmer defines and on what kind of components they operate.

Our approach gives no guarantee of composablity: nothing protects the programmer from declaring different sets of rules that operate on two or more structures that overlap. This could lead to one rule inadvertently breaking invariants that the programmer assumes for another rule's correctness. In this regard our DSL requires similar carefulness as GUI programming in general.

5 Related Work

Treating a user interface as a composition of components is mainstream in contemporary approaches to GUI development. Perhaps most prominently this view is embraced by React [11], a templating framework for developing GUI applications. A React user interface is essentially a hierarchy of stateful components that are responsible for updating their (and their child components') views whenever their state changes. A similar view on components is also shared by other JavaScript frameworks such as Angular [12] and Vue.js [15]. The hierarchical structure of components manifests also in GUI frameworks for implementing desktop applications, such as Windows Forms [3] and JavaFX [4].

Our DSL can complement such GUI-component solutions. As with any composition of components, the programmer can express relations on, e.g., React-components in our DSL, allowing declaratively specifying modifications of incidental structures that emerge as compositions of such components.

6 Conclusion and Future Work

The DSL presented in this paper lets the programmer express structural changes in a declarative manner. Structures, that could appear incidental and implicit without the DSL, can be explicitly declared, and an extensive set of functions for making changes to the structure can be provided with ease. Applications can then perform higher-level operations on the structure in a straight-forward manner, using the functions that hide the details of those changes.

We consider the work described in this paper as exploring a new approach for controlling the complexity of GUI programming. We can immediately identify both possible improvements to what we have reported here and avenues for new research.

As it is described in the paper, the DSL is a bit simplistic. It is evident that the same transformation rules could be applicable to more than one relation. For this purpose, we have experimented with *generic rules*, parameterized on relations. For example, the transformation rule `swap` could be made generic, so that it would work with an arbitrary binary relation that expresses order and adjacency. One rule would suffice to specify the operation of swapping once and for all, to be reused for different components in one application, or even in different applications altogether.

Generic rules have the following form:

```
swap<R> (x y) = x R y => y R x
```

The angle brackets enclosing the relation `R` state that this rule is generic; the *abstract relation* `R` and *abstract placeholders* `x` and `y` need not be declared explicitly.

From a generic rule `t<`R_1, ..., R_n`>` that uses abstract placeholders x_1, ..., x_k, we generate a JavaScript function `t(`x_1, ..., x_k, R_1, ..., R_n`)`, similar to the one in Figure 2. This function invokes functions `test`, `establish`, and `unestablish` defined as members of JavaScript objects R_i.

To apply a generic rule, we instantiate it with a relation, for example, `isAbove` defined in Figure 4. This is done by calling the corresponding JavaScript function and passing the name of the relation to it:

```
swap(elem1, elem2, isAbove);
```

Specifying structural modifications via changes to which relations hold between elements opens interesting opportunities to further study. If we annotate such relations with algebraic properties, such as symmetry, transitivity, or reflexivity, we may be able to statically analyze the rules and report to the programmer if a rule invalidates a property of a relation. E.g., assume that the relation `precedes` is annotated as anti-symmetric. If a programmer wrongly defines a transformation rule where the two relations `a precedes b` and `b precedes a` are to be established, then, because `precedes` is anti-symmetric, a report about an error can be made and communicated to the programmer.

References

[1] Accessed: 2020-07-15. ApplyTexas Sample Application: extracurricular, personal & volunteer activities (page 7). https://www.applytexas.org/adappc/html/preview20/igr_ec.html.

[2] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend: learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices.* Packt Publishing.

[3] Microsoft Corp. Accessed: 2020-07-15. Windows Forms. https://docs.microsoft.com/en-us/dotnet/framework/winforms/.

[4] Oracle Corp. Accessed: 2020-07-15. JavaFX. https://openjfx.io/.

[5] Sven Efftinge and Sebastian Zarnekow. Accessed: 2020-07-15. Xtend—modernized Java. https://www.eclipse.org/xtend/.

[6] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, and Jimi Woning. 2015. Evaluating And Comparing Language Workbenches: Existing Results And Benchmarks For The Future. *Computer Languages, Systems & Structures* 44 (08 2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007

[7] Moritz Eysholdt and Johannes Rupprecht. 2010. Migrating a large modeling environment from XML/UML to Xtext/GMF. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 97–104. https://doi.org/10.1145/1869542.1869559

[8] Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages? https://martinfowler.com/articles/languageWorkbench.html.

[9] John Freeman, Jaakko Järvi, and Gabriel Foust. 2012. HotDrink: A Library for Web User Interfaces. *SIGPLAN Not.* 48, 3 (Sept. 2012), 80–83. https://doi.org/10.1145/2480361.2371413

[10] Richard C. Gronback. 2009. *Eclipse modeling project: a domain-specific language toolkit.* Addison-Wesley.

[11] Facebook inc. Accessed: 2020-05-02. React—A JavaScript library for building user interfaces. https://reactjs.org/.

[12] Google LLC. Accessed: 2020-07-15. Angular. https://angular.io/.

[13] Voelter Markus, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL engineering: designing, implementing and using domain-specific languages.* CreateSpace Independent Publishing Platform.

[14] Dave Steinberg, Frank Budinski, Marcelo Paternostro, and Ed Merks. 2008. *EMF : Eclipse Modeling Framework.* Addison-Wesley, Upper Saddle River, N.J.

[15] Evan You. 2020. Vue.js. https://vuejs.org/.