Electronic Communications of the EASST
Volume 078 (2019)

8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation

-

Doctoral Symposium and Industry Day, 2018

Evolve: Language-Driven Engineering in Industrial Practice

Tim Tegeler and Jonas Schürmann

16 pages

Guest Editors: Falk Howar, Anna-Lena Lamprecht
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Evolve: Language-Driven Engineering in Industrial Practice

Tim Tegeler[1,2] **and Jonas Schürmann**[1]

Chair for Programming Systems
TU Dortmund University, Germany[1]
Creios GmbH
Selm, Germany[2]

Abstract: In general, software projects still have a very high failure rate. We noticed that one of our projects did not gather pace. It was delayed from the beginning and on the way to fail. After investigating the development process, we located the issue in the chosen architecture of the software. Even though the used technology has many advantages, application developers were handicapped due to the cumbersome architecture. The challenge was how we could keep the advantages, but simplify the work of the application developers. We came up with the approach to build a toolkit and family of dedicated Domain-Specific Languages which are developed alongside the project. We called it *Evolve*, and it is built upon the Language-Driven Engineering paradigm. We were able to salvage the project and establish *Evolve* in the development process of related applications. With *Evolve*, we successfully brought Language-Driven Engineering to industrial practice. It will play a major role in our future software development.

Keywords: Language-Driven Engineering, Language Evolution, Domain-specific Languages, Code Generation, Industrial Practice, Web Application

1 Introduction

This paper is an experience report of a delayed industrial software project in the domain of web applications. It discusses challenges which occurred during the implementation phase and how the adoption of *Language-Driven Engineering* (LDE) helped to resolve them. The practical utilization shows the positive effect for the developing process. It introduces *Evolve*, a toolkit and family of dedicated *Domain-Specific Languages* (DSL) following the LDE approach.

In the early stage of the project we observed that it was already behind schedule. The whole project was in danger of missing its deadline. After having a more in-depth look inside the development process of the project and interviewing the application developers, we identified the major problem. The given architecture forced the application developers to write similar code over and over again. Furthermore, the code was very cryptic with complex syntax. Thus resulted in a very error-prone development process with had a negative impact on the motivation of the developing team and the progress of the project in general. The success of the project was at risk. It was clear that we had to act radically if we wanted to salvage the project. The initial idea was to tailor a *Domain-specific Language* (DSL) once for a critical part of the application to speed up the overall development process. The DSL should help to reduce the code to be written by hand and having simpler syntax than the original code. It should enable the application developers to

```
1  const toggleAction = {type: 'TOGGLE'};
2
3  const initialState = {
4      value: true
5  };
6
7  function reducer(state, action) {
8      if (typeof state === 'undefined') {
9          return initialState;
10     }
11
12     switch (action.type) {
13         case 'TOGGLE':
14             return {value: !state.value};
15     }
16 }
```

Listing 1: Simple Redux example containing store (state), reducer and action

focus on the conceptual design of the software. On first glance, one would say that the problem of the project is the given architecture and a DSL would only conceal this.

But actually, the underlying architecture, which is built with Redux[1], has advantages that outweigh its disadvantages in our use case. An application based on Redux has three main components[2], a store, a reducer, and multiple actions. The store holds the whole state of the application in an object tree and is the "one source of truth" [Bac16]. An action is a simple object that expresses an event and can carry a payload. The reducer takes the current object tree and an action as input to transform and emit the new object tree. Simple changes of values inside the store require actions and reducers. This can lead to very verbose projects.

Listing 1 demonstrates the possible verbosity of vanilla Redux projects. The code is narrowed to the essential Redux components and omits parts that are necessary to execute it (for instance, in a browser). Subject of this example is the functionality of toggling a simple boolean value. Therefore, the code includes just one action *'toggleAction'* (c.f. line 1) which describes that a toggle event was triggered. The action is a simple object and is distinguished from other actions by the *'type'* attribute. A payload is not necessary nor shown in this example. Lines 3 to 5 declare the initial state of the store. The store is a simple object containing just one attribute of type boolean called *'value'* initialized with the value *'true'*.

Central and most important part of the Redux pattern is the reducer. In most cases this can be a *simple* function. Depending on the size of the store and number of actions, the complexity of the function can grow dramatically. In this example the reducer functions spans from lines 7 to 16. It takes two parameters *'state'* (the state before emitting the action) and *'action'* (the action that describes an event). If the state is undefined, e.g. when the reducer function is called for the first time at the beginning of the program execution, it returns the initial state (line 9). But if the state is defined, the reducer handles the passed action by using a switch statement (lines 12 to 15). As

[1] https://redux.js.org
[2] For an introduction of the Redux architecture please refer [Bac16]

already noted, an action can be distinguished by its *'type'* attribute. The switch statement makes use of this identifier to handle the different cases. An action does not know about the way an event like toggling is handled. This is solely the task of the reducer and is located in the respective case of the switch statement (line 14). For handling the toggle action, the reducer creates a new state object, writes the inverted value and returns the new state. In summary, it can be stated that this pattern of a vanilla Redux project produces a lot of code for simple task like the toggling of a Boolean value. But by using this pattern, we gained benefits we have not experienced before with other architectures: Control over the state of the application at any time, maintainability of the code basis and powerful debugging of the running application. *Evolve* has the ability to retain the advantages and balance out the disadvantages. On second glance, introducing a dedicated DSL for a project that is already in danger of missing the deadline sounds unintuitive. However, we experienced enormous leverage due to bootstrapping [SGNM18].

In Section 2 we illustrate the vision and concept of *Evolve* and how we apply the concept of *Language-Driven Engineering* (LDE) to industrial practice. Section 3 addresses the realization by showing key components, like language design and code generation. Languages are exposed in detail in Section 4. Section 6 concludes the paper.

2 Vision and Concept

Evolve uses a service-oriented architecture to provide the composition, refinement, and evolution of the DSLs. Following this principle of LDE, we were able to divide "the labor on the basis of Domain-Specific Languages [...] targeting different stakeholders." [SGNM18] The DSLs focus on the differences in the code and consider the things that do not change as *Archimedean points* [SGNM18, SN16]. Not only can application developers and language developers participate in a project as a vertical composition, but different language developers can collaborate working on complementary DSLs as a horizontal composition. When creating a new tool getting the users (i.e., application developers) involved is critical. We want to involve all stakeholders of the project in a fashion of *Design Thinking* [RS12] to get feedback as soon as possible. With *Evolve*, we do not distinguish between DSL and application development which enables us to introduce new levels of reuse for similar projects [SGNM18].

Application developers requested missing features in the generated code when using *Evolve* and "express[ed] their desires in terms of what they want to achieve (WHAT/Requirement level), without worrying about possible ways of realization (HOW/Implementation level)." [SGNM18] We were able to add those features to a specific DSL and to provide them with a new version very fast by using *Continuous Practices* (CP) [TGS18, SBZ17]. Moreover, even bugs in the generated code were fixed permanently all by the language developers. The application developers were able to participate in the evolution of the DSLs [SGNM18]. This creates a *Continuous Improvement Cycle* (CIC) [MC03] (refer Figure 1). After an iteration of the CIC and receiving an improved version of *Evolve*, application developers were able to regenerate the code without knowing about the implementation of the new feature.

Based on the CIC, the evolution of the DSL drives the development of the application. This evolution of the DSL is a central part of LDE and was decisive of naming the project *Evolve*. The first DSL generated *Data Transfer Objects* (DTO) (used as API payloads) and decoder functions

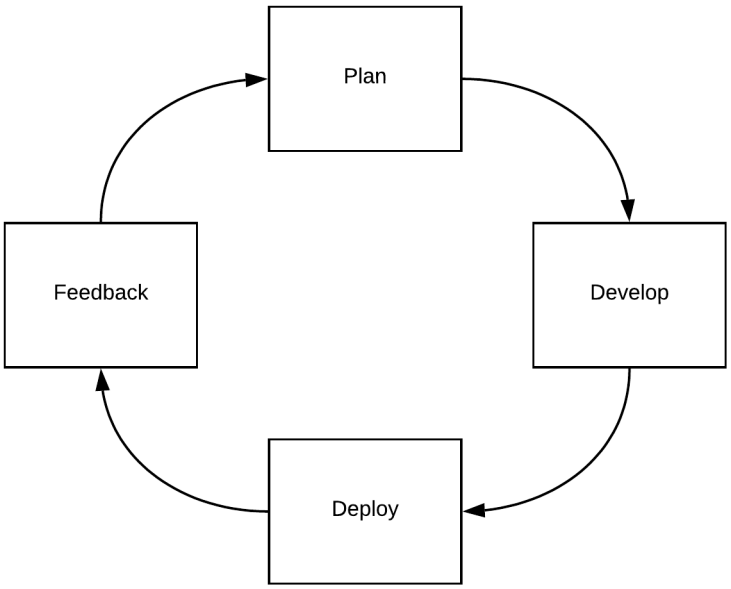

Figure 1: *Continuous Improvement Cycle* used to integrate *Evolve* in the application development

each to ensure type safety. After the first success in production and receiving positive feedback from the application developers, we decided to go a step further: Applying the idea of developing a DSL to simplify the development for more parts of the projects. This includes API requests, user input forms, and Redux stores. A detailed explanation of the languages will be given in Section 4.

3 Realization

The initial idea was to speed up the overall development process of the target project. We tried to find cheap and easy solutions that would integrate seamlessly into our already established workflow. We were worried about over-engineering *Evolve* and not being efficient enough to benefit from it. Our focus was on easily accomplishable objectives to start with the CIC and to use *Evolve* in production as soon as possible. In this section, we illustrate two different aspects of the realization. On the one hand, the implementation of the application with its central components language design and code generation. On the other hand, the operations of the development with integration into the existing IDE and the set up of our deployment process.

3.1 Language Design

After we used *Evolve* in production with the first DSL and decided to add more, we determined some fundamental requirements.

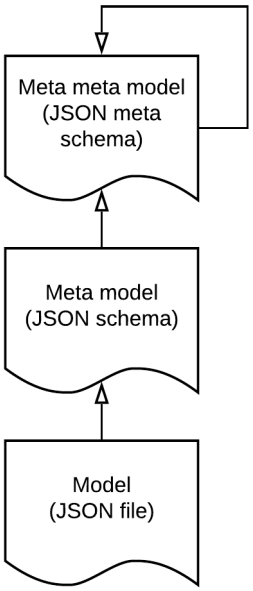

Figure 2: Levels of the language design applied to Meta-modeling

1. Languages are based on the same technology to make the introduction of new languages inexpensive.

2. Creating declarative languages to keep it as simple as possible for the application developers.

3. Languages are built on top of technology already known by the application developers to reduce the initial hurdle.

Therefore, we chose the textual JavaScript Object Notation (JSON)[3] as our carrier language to develop our external DSLs [FP11]. JSON is ubiquitous in developing modern web applications, and our application developers were very familiar with it. It is the most popular data format for HTTP payloads in APIs and used to configure dependencies and build tools for JavaScript-based applications. Our models are exclusively based on JSON files. A lot of tools and libraries exist to work with JSON; therefore, the cost of developing a DSL are reduced. This contains deserialization of the files and validation of the models. The "relevant parts of the modeling language" [KT08] are formalized in the metamodel (Figure 2). As the metamodeling language [KT08], it was apparent to use JSON schema[4] which itself is based on JSON. The JSON schema of each language holds information about the specific domain. It defines the rules and constraints of the language and can be used to assist the user while modeling (c.f. Subsec-

[3] https://www.json.org
[4] https://json-schema.org

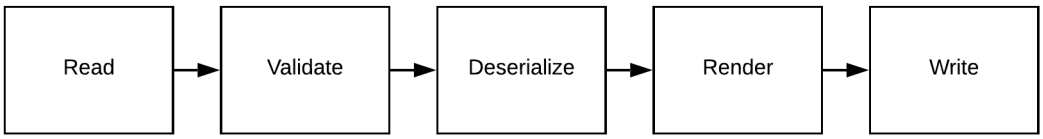

Figure 3: Universal generator pipeline used for all DSLs

tion 3.3). A JSON schema is formalized by the JSON meta-schema and can be validated against it. The JSON meta-schema is self-descriptive and formalizes itself.

3.2 Code Generation

The delayed project is the front-end part of a sophisticated web application. The front-end is delivered by a web server and runs as a client inside of the user's browser. It communicates with the back-end on the servers by an API. The front-end is written in TypeScript[5] and compiled to JavaScript code to be executable by a standard browser. Therefore *Evolve*, is focused on the front-end of web applications and generates TypeScript code.

Evolve provides only code generation for certain parts of the front-end where we saw the most leverage. It is not possible to generate the whole application, but it supports automated full code generation from the models [KT08]. Entire files are generated, which is very advantageous. The files are *atomic*, *disposable* and *opaque*. Atomic means that these files are the smallest entities to deal with. They are the building blocks for the larger application. Disposable means the files can be recreated from the models by the generator at any time. The generated code contains not only static but behavioral structures [KT08]. There is no round-trip-problem where the information is kept up-to-date between model and code [SVC06, KT08]. Generated files hide the internal code from the application developer (opaque) and only export their API to the global scope. Like using a library, the application developers can use the code without knowing the exact implementation. Thus, an application developer can use the files as black boxes. The files or building blocks must be linked to the handwritten parts of the application by using the provided API.

Although the generated code should not be relevant to the application developers, we had some claims on our code generation. The code generator should be deterministic, so that, provided with the model, it produces the same code. For this reason, we rely mainly on pure functions and avoid side effects. Because we are committing our generated code to the *Version Control System* (VCS), regenerating code would cause dirty working trees. When working with many models, this could flood the VCS, and manual changes are hard to distinguish. Furthermore, the code should be humanly readable for later debugging and avoiding the lock-in effect [ZZ12]. This includes proper code styling and meaningful symbol names for variables. Lines of code where the actual order of appearance is not important for the correctness of the source code (e.g. interface attributes) is always in alphabetical order to improve readability. In the beginning, we were not sure how successful *Evolve* would be. We wanted to have generated code that could be

[5] https://www.typescriptlang.org

```
1 <#if interface.export>export </#if>interface ${interface.name} {
2 <#list interface.attributes as attribute>
3     readonly ${attribute.name}: ${attribute.type}
4 </#list>
5 }
```

Listing 2: Template of a TypeScript interface written in Freemarker

easily extended and maintained in case abandoning *Evolve*.

Evolve is based on Java[6] and built by Gradle[7]. As template language we use Freemarker[8]. After developing the first DSL, we noticed that we should introduce general, reusable and composable code generators. Instead of creating a few large templates (e.g. one template for each DSL), we decided to compose code generators in order to enable them to work together. E.g., a code generator to create interfaces (interfaces are used in TypeScript to create types based on duck typing [MGN17]). The Freemarker template of this code generator is shown in Listing 2. The template is based on a DTO called interface. It maintains the name that should be used and whether the interface should be exported (available in the global scope) or not (line 1). Besides the meta information, it also provides a list of attributes in alphabetical order. Attributes are again simple DTOs keeping just the type (like Boolean, number and string) and the name of the attribute (line 3). The template iterates (line 2) over this list and prints the *'readonly'* modifier, the name, and the type.

Evolve provides a universal generator pipeline (Figure 3) that makes it easy to add new DSLs. It starts with the reading (Figure 3: Read) of the models on filesystem-level. The models will be validated (Figure 3: Validate) against the related JSON schema. The models will be deserialized (Figure 3: Deserialize) into DTOs. On the base of the DTOs Freemarker is used to render (Figure 3: Render) the source code. The source code is written (Figure 3: Write) back on filesystem-level.

3.3 Mindset-Supporting Integrated Development Environment

LDE introduces a new development paradigm which is heavily built upon *Mindset-Supporting Integrated Development Environments* (mIDE). These are extended *Integrated Development Environments* (IDE) enriched by the mindset of the stakeholders with native support for the related DSLs [SGNM18]. How to provide application developers with mIDE was a challenge of using LDE in industrial practices with *Evolve*. It was clear that we would not be able to develop dedicated mIDEs alongside *Evolve* without jeopardizing the actual project. Modern IDEs, based on the IntelliJ platform [9], were present on the application developers' machines for the classical development work. Our idea was to build upon the existing IDE and integrate *Evolve*.

The IntelliJ platform already has excellent support for JSON syntax and even JSON schema

[6] https://java.com
[7] https://gradle.org
[8] https://freemarker.apache.org
[9] https://www.jetbrains.com/idea

With the JSON schema support, there were many advantages. The user is guided in the process of modeling by autocomplete and errors are prevented before the actual code generation [KT08].

With the versatile run configuration of IntelliJ, we were able to integrate the code generator of *Evolve* into the project. Once having done so, the code generation was just as easy as pressing a button. This enhanced an IDE to become a *Mindset-Supporting Integrated Development Environment* (mIDE), without having to deal with "the major hurdle of LDE currently perceived: the construction of the required mIDEs" [SGNM18].

3.4 Development, Deployment and Distribution

After gaining the first experience with LDE in industrial practice, we noticed two essential requirements. These requirements are even more critical with delayed projects where reaction to future problems has to be instant. On the one hand, to enable the full potential of LDE, application and language developers must collaborate as closely as possible. On the other hand, when time is valuable, we cannot have a clumsy development, a unreliable deployment or complex distribution process.

We choose GitLab[10] to support the entire application development process of *Evolve*. It provides a repository for source control and enables us to establish a CIC (Figure 1) [TGS18]. With the CIC (Figure 1) we want to get the application developers involved. Application developers can provide feedback as issues (Figure 1: Feedback phase). After that, we discuss the request and ideas (Figure 1: Plan phase). Implement their requirements (Figure 1: Develop phase) and supply them with the next version as comfortable as possible (Figure 1: Deploy phase).

Evolve is based on *Test-Driven Development* (TDD) to speed up feature integration, find bugs early and support fearless refactoring [Bec03, Ast03]. The tests yielded by the TDD are integrated with the *Continuous Integration* (CI) solution of GitLab. Tests will be executed automatically after every change that is pushed to our VCS. After successful tests, we can deploy a new version of *Evolve* by *Continuous Deployment* (CD) strategies. The deployment is triggered by pushing the master branch to the deployment branch. This will start the compilation of the code and deployment of the executable. As deployment target and for publishing we use GitLab pages[11]. It acts as a managed web server hosted by GitLab and is integrated into the repository. Alongside the executable, we also deploy the JSON schema files. The executable together with the JSON schema files can be downloaded from the web server. Every project that depends on *Evolve* has a built-in update mechanism in the related mIDE to receive new versions. The update mechanism is based on the run configuration of IntelliJ.

4 Languages

This section gives an introduction to the DSLs that are currently supported by *Evolve*. We focused on four main parts of the project where we observed the most boilerplate code and potential of applying DSLs. In the following subsections, we are focusing on the DSLs and will not show the resulting generated code. This would push the boundaries of this paper.

[10] https://about.gitlab.com
[11] https://about.gitlab.com/product/pages

```
{
  "path": "user",
  "rest": {
    "create": {
      "responseEntity": "UserResponse",
      "requestEntity": "UserRequest"
    }
  }
}
```

Listing 3: Model that describes a create operations of a user

4.1 Request Language

Modern web applications are based on communication with a server by an API. In the related project, this API was built loosely on the style of *Representational State Transfer* (REST) [Fie00]. REST is not a standard based on a Request for Comments (RFC) [Dog15], but it is widely used and accepted. Although it is protocol-independent [Dog15], we are assuming that the HTTP protocol is used.

We introduce a *Request Language* to generate the API layer of our web application fully. The DSL is resource-oriented and uses the proposed actions of REST in a combination of HTTP verbs to read and alter. A resource is everything that can be named, like a document or user [Fie00]. Not only high-level operations like CRUDL (create, retrieve, update, delete, list) [Dog15] are supported, but also low-level descriptions of HTTP requests to create custom operations.

Listing 3 shows an example of this *Request Language*. It describes a create operation of a user resource. The model is an object and consists of two main parts, the *'path'*, and *'rest'* attributes. The *'path'* attribute is a simple string and is used to describe the endpoint for the resource. Mostly this will be concatenated with the root path of our API (e.g. *'/api/user'*). The *'rest'* attribute is an object itself. It supports a subset of the attributes *'create'*, *'retrieve'*, *'update'*, *'delete'* and *'list'*. However, this example shows only the *'create'* attribute. The *'create'* attribute is again an object itself. It has only the two attributes *'requestEntity'* and *'responseEntity'* attributes. They define the TypeScript interfaces that are used for the payload in the body of the HTTP request and response.

With the *Request Language*, we were able to refactor our API rapidly during the development process.

4.2 Entity Language

When data is exchanged via an API, sender and receiver must ensure that the data is valid. As already mentioned, the front-end is written in TypeScript and strongly based on types. Interfaces (c.f. Subsection 3.2) are used to define types. JSON is compatible with interfaces and serves as a data format. However, JSON is independent of any programming language and has no native support to ensure types. It is in the assignment of the programming language to ensure it. The

```
{
  "active": {
    "type": "boolean"
  },
  "age": {
    "type": "number",
    "null": true
  },
  "firstName": {
    "type": "string",
    "origin": "first_name"
  },
  "lastName": {
    "type": "string",
    "origin": "last_name"
  }
}
```

Listing 4: Model that describes a user entity

test of the data has to take place during runtime when data is received. We use JSON Bouncer[12] to build validators that can ensure that a specific JSON string is of a given type.

The *Entity Language* generates not only to the entity but to the decoder as well. The basic structure of an entity model is an object where the metamodel does not define the attributes. Therefore, however, every attribute has to be an object of a given form itself. The metamodel requires only the attribute *'type'* as mandatory. It can have the values *'string'*, *'number'* or *'boolean'*. Beside the mandatory attribute *'type'*, two optional attributes *'null'* and *'origin'* are supported.

Listing 4 is a model of a user entity. Every first level attribute (i.e. *'active'*, *'age'*, *'firstName'* and *'lastName'*) stands for an attribute of the generated interface. The attribute *'active'* is of the type *'boolean'*. The attribute *'age'* is of the type *'number'*. The attributes *'firstName'* and *'lastName'* are of the type *'string'*. A typical use case for *'origin'* is to adapt between different spelling practices of attributes. In this example, *camel case* [SM10] (e.g., *'firstName'* or *'lastName'*) is used for the internal representation, but the origin uses *snake_case* [SM10] (e.g. *'first_name'* or *'last_name'*).

The *Entity Language* allowed us to scale our entities and rely on the structure of the received data.

4.3 Form Language

A well understood, but vulnerable domain of web applications are form-based inputs [SP]. Inputs enable users to interact with the application and enter data. In general, user input cannot be trusted [SP] and therefore, has to be validated and sanitized. Since our generated code is executed on the user's machine and leaves the server API untouched, we have to secure our server as well.

[12] https://gitlab.com/MazeChaZer/json-bouncer

```
{
  "page": "Registration",
  "inputs": {
    "email": {
      "type": "email",
      "mandatory": true
    },
    "password": {
      "type": "string",
      "mandatory": true
    },
    "birthday": {
      "type": "date"
    }
  }
}
```

Listing 5: Model that describes a registration form

However, this is out of the scope of this project. We do not try to defend against malicious attacks but handle mistaken inputs by using generated code for common problems.

The DSL features four parts related to form based inputs. Storing the raw and potential flawed data entered by the user. Validation of the input is based on input types. Generating proper error messaging for invalid data that can be used to guide the user during the input process. Sanitizing the data for further usages, like sending the data to an API and storing validated and sanitized data. Supported input types are the basic types of *'string'*, *'integer'* and *'boolean'* and higher-level types like *'date'*, *'datetime'*, and *'email'*.

Listing 5 shows an example model to describe a registration form. The *'page'* attribute is a simple string and defines on which page the form is used. This is used internally by the generator to reduce the glue code to connect the generated and handwritten code. The *'inputs'* attribute is an object itself and stores the input objects. The Name of the input (e.g., *'password'*) can be chosen freely and is not a keyword of *Evolve*. With the attribute *'mandatory'*, which is Boolean, an input is required to be filled. The *'type'* attribute determines the actual type of the input (c.f. above).

Through the *Form Language*, we were able to create standardized forms and recognizable user experience.

4.4 Store Language

The central part of a Redux based web application is the store where the actual state of the application is saved in an object tree (please refer to Section 1). In the delayed project, most of the data in the object tree consist of the simple types like integers for storing the current page of a paginated list or Booleans to describe if GUI elements are active. For changing a value inside of the store, an action and a reducer are needed. This results in much handwritten boilerplate code which led to very verbose projects. Our approach to simplify the usage of Redux is the *Store*

```
{
  "page": "UserList",
  "states": {
    "leftMenuIsOpen": {
      "default": "true",
      "type": "boolean",
      "toggle": true
    },
    "activePage": {
      "default": "1",
      "type": "integer",
      "setter": true
    }
  }
}
```

Listing 6: Model that describes a simple Redux store

Language. It provides modeling and generating of stores to handle simple data types.

In our web application, we preferred an architecture that is not made up of layers. Instead of having static layers (like in the model-view-controller [LR01] architectural pattern), it is structured like slices. Every page of our web application defines a slice and has distinct scope. This DSL supports this approach and generates a store, a reducer, and actions for each.

Listing 6 shows an example model to describe a Redux store for the *'UserList'* page. A model of a Redux store is divided into the two parts, *'page'* and *'states'*. States is a simple object and can have multiple attributes. The name of a state is the attribute name. It can be of the types *'integer'* or *'boolean'*. This is defined in the *'type'* attribute. With the *'default'* attribute, a default value for the state in the store can be determined. Based on the type different actions and related reducer-functions can be generated. For the Boolean type, a toggle action can be set. The toggle method enables the user to change the value of the Boolean to the opposite of the current value. For the integer type, setter, increment, and decrement actions can be set. The setter method enables the user to set and overwrite the existing value. The increment and decrement methods increment respectively decrement the integer value.

The *Store language* simplified the repetitive work of the Redux pattern and significantly reduced the handwritten code.

5 Evaluation

In this section, we evaluate the impact of introducing *Evolve* to our project.

To get a first impression of the success of *Evolve*, we look at the proportion of *lines of code* (LOC) between model- and generated files in the project after it went into production. Table 1 shows that the project contains *1508 LOC* of model files which automatically generates to *6625 LOC*. This means an expansion factor of *4.4* which results in a reduction of *77%* of LOC in the parts of the project that are generated and *18%* of LOC in the overall project. Table 2 indicates the overall number of files (*418 files*) and code (*28029 LOC*) in the project. Hand written files

	Model	Generated (Factor)
Lines of Code	1508	6625 (4.4x)

Table 1: Proportion of model- and generated LOC in the project

	All	Handwritten	Generated (Percentage)
Number of Files	418	332	86 (21%)
Lines of Code	28029	21404	6625 (24%)

Table 2: Number of generated files and LOC in the project

could be reduced by *21%* and the LOC by *24%*.

Admittedly, using the number of files and LOC as a metric for source code is vague, especially when analyzing generated code. Even so, one of our major goals was to reduce the amount of hand written code. Since *Evolve* generates humanly readable code that is very close to a manually implementation, it makes the generated code more comparable (c.f. Subsection 3.2). A fifth of the handwritten code (*6625 LOC*) could be replaced by generated code which saved valuable developing time and reduced error probability not only once, but also in the future. It is effortless to introduce new functions or fix bugs in that part of the application. To sum up one could say, that *Evolve* had a major impact on the codebase of the project and the development.

But not only formal metrics like the amount of code indicated the success of *Evolve*. During the adoption of *Evolve*, we noticed positive effects on intangible indicators like team spirit and teamwork as well. It increased the overall mood of the team. The application developers experienced an *eureka moment* comparable to the time of their first successful compiling and running program. Many of them had never actively generated code before. But this *eureka moment* happened on both sides. It was also the first time for the language developers to create and push a DSL (or more precisely, a family of DSLs) into production and experiencing the success of it at first hand.

During the first days we were able to quickly remove error-prone handwritten code by generated code and decrease the pressure on the application developers. They quickly came up with ideas how to improve *Evolve* and their DSLs. With the leverage of the CIC, we were able to get these ideas very fast into production and receive feedback if the introduced alterations worked or not. *Evolve* was developed independently from the initial use case of our application and is compatible with similar frontend projects. Therefore, it was adopted with success by several other frontend projects in the company on short notice. This led to a very fast evolution of the DSLs and introduced a new mindset in our overall team that goes beyond *Evolve*. Instead of just trying to fix problems by hand, our team tried more and more to find elegant ways to solve them or prevent them in the first place: using existing tools, building libraries, and improving our processes. This includes automation for repetitive tasks, stricter code style or introducing more sophisticated testing techniques.

6 Conclusion and Perspective

In this paper, we have featured an approach to LDE in industrial practice. The subject of the study was a delayed project in the domain of web applications. We introduced *Evolve*, a toolkit, and a family of dedicated DSLs. With *Evolve*, we were able to salvage the project by significantly reducing the manual effort of the application developers. We illustrated that LDE was easy to integrate into our established workflow. DSLs became first-class citizens of the development process. This example shows that LDE can have a significant impact on real-world projects. It helps to reduce the manual effort and speed up the overall development process. Furthermore, it contributes to raising the moral and motivation of the team by offering simple solutions for error-prone tasks. By establishing *Evolve*, similar projects can build upon the principles of LDE more easily. They will benefit from this new level of reuse [SGNM18]. We will continue to apply LDE to upcoming projects in the future.

While developing and using *Evolve* we faced two related drawbacks for the time being. The task of generating code has to be triggered manually by the application developer, and the generated code is checked into our VCS. In future, we want to have this task automated by the mIDE and integrate the generation in our CI pipeline. After that, we can version only the models and the generated code can indeed be classed as disposable. In the next step of the development of *Evolve*, it would be desirable to revise the syntax of the DSL. Using JSON (together with JSON schema) as the carrier language turned out to be very cumbersome. Instead of forcing a file format specialized for data transfer, a custom language should be considered. The syntax can be precisely tailored to the domain without having syntactic overhead. To follow the principles of LDE even more, *Evolve* should support the interaction between the DSLs. At the moment, every DSL is independent of each other.

In the future, we would like to go a step further by automating the generation even more. Modern web-based APIs are self-describing, e.g., based on the technology of the OpenAPI Initiative[13]. They provide human and machine-readable documentation. We want to use this documentation to generate API requests and entities. Considering the generator pipeline (Figure 3), this can be done by two different approaches. Either by introducing model-to-model-transformation in front of the reader or by substituting read, validate and deserialize stages or to generate from API by using documentation.

Considering that *Evolve* was aimed to speed up other projects, it was built upon basic solution and developed dedicated to related work. In the future, we intend to revise parts of *Evolve* and combine it with already established and successful projects in the area of LDE.

Acknowledgements

We thank Luke Thienemann for assistance with developing the concept of *Evolve* and his comments that greatly improved the manuscript.

[13] https://www.openapis.org/about

Bibliography

[Ast03] D. Astels. *Test Driven Development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003.

[Bac16] A. Bachuk. Redux · An Introduction. https://www.smashingmagazine.com/2016/06/an-introduction-to-redux/, 2016. [Online; accessed 04-March-2019].

[Bec03] K. Beck. *Test-driven Development: By Example*. Kent Beck signature book. Addison-Wesley, 2003.

[Dog15] F. Doglio. *Pro REST API Development with Node.Js*. Apress, Berkely, CA, USA, 1st edition, 2015.

[Fie00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[FP11] M. Fowler, R. Parsons. *Domain-specific languages*. Addison-Wesley / ACM Press, 2011.
http://books.google.de/books?id=ri1muolw_YwC

[KT08] S. Kelly, J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008.

[LR01] A. Leff, J. T. Rayfield. Web-application development using the Model/View/Controller design pattern. In *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*. 2001.

[MC03] P. Murray, R. Chapman. From continuous improvement to organisational learning: developmental theory. *The Learning Organization* 10(5):272–282, 2003.

[MGN17] N. Milojković, M. Ghafari, O. Nierstrasz. It's Duck (Typing) Season! In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. Pp. 312–315. May 2017.
doi:10.1109/ICPC.2017.10

[RS12] R. Razzouk, V. Shute. What Is Design Thinking and Why Is It Important? *Review of Educational Research* 82:330–348, 09 2012.
doi:10.3102/0034654312457429

[SBZ17] M. Shahin, M. A. Babar, L. Zhu. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *CoRR*, 2017.

[SGNM18] B. Steffen, F. Gossen, S. Naujokat, T. Margaria. Language-Driven Engineering: From General Purpose to Purpose-Specific Languages. 2018.

[SM10] B. Sharif, J. I. Maletic. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *2010 IEEE 18th International Conference on Program Comprehension*. June 2010.

[SN16] B. Steffen, S. Naujokat. *Archimedean Points: The Essence for Mastering Change*. Pp. 22–46. Springer International Publishing, Cham, 2016.

[SP] D. Stuttard, M. Pinto. *The Web Application Hacker's Handbook: Finding And Exploiting Security Flaws, 2nd Ed*. Wiley India Pvt. Limited.

[SVC06] T. Stahl, M. Voelter, K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, Inc., USA, 2006.

[TGS18] T. Tegeler, F. Gossen, B. Steffen. A Model-driven Approach to Continuous Practices for Modern Cloud-based Web Applications. 2018.

[ZZ12] K. Zhu, Z. Z. Zhou. Research Note - Lock-In Strategy in Software Competition: Open-Source Software vs. Proprietary Software. *Information Systems Research* 23:536–545, 2012.