A Process for Evidence-Based Engineering of Domain-Specific Languages

Michael Felderer*
University of Innsbruck
Department of Computer Science
Innsbruck, Innsbruck
michael.felderer@uibk.ac.at

Fabian Jeschko
University of Innsbruck
Department of Computer Science
Innsbruck, Austria
fabian.jeschko@uibk.ac.at

ABSTRACT

Domain-specific languages (DSLs) are mainly designed ad-hoc and gut feeling resulting in languages that are often not well suited for their users and engineers. In this paper we develop a process for evidence-based language engineering to design domain-specific languages based on empirical evidence to support decision in language engineering. The developed process comprises an iterative execution of the phases DSL engineering, issue identification, data collection and evidence appraisal. We exemplify the concept by designing a DSL for Gherkin, a language test-driven acceptance testing in Xtext. The required evidence is derived by mining and analyzing all GitHub projects until July 1, 2017 that apply Gherkin.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Empirical software validation**;

KEYWORDS

DSL engineering, domain-specific languages, repository mining, empirical research, evidence-based software engineering

ACM Reference Format:
Michael Felderer and Fabian Jeschko. 2018. A Process for Evidence-Based Engineering of Domain-Specific Languages. In *EASE'18: 22nd International Conference on Evaluation and Assessment in Software Engineering 2018, June 28–29, 2018, Christchurch, New Zealand*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3210459.3210479

1 INTRODUCTION

An approach to empirical studies by which the researcher seeks to identify and integrate the best available research evidence with domain expertise in order to inform practice and policy-making.

Domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application [21]. DSLs are a popular and powerful means to build customized processable languages for certain problem domains within the software and systems development lifecycle [26]. The availability of sophisticated language workbenches like Xtext [7] facilitates the development of DSLs making them increasingly more popular. The adoption of DSLs at large comes at the risk that a poorly designed DSL can be too hard to adopt by its domain users. Language engineers mostly rely on their experience and gut feeling, while making essential language engineering decisions. DSLs are in most cases designed ad-hoc and neither available DSL frameworks like Xtext nor research provide sufficient support to design languages in an evidence-based way taking the needs and the actual behavior of language users into account. A recent systematic mapping study on DSL engineering [19] revealed that within the DSL community even researchers focus on language design and implementation, but rarely evaluate their languages or even provide methodological support how evaluation and evidence-based language design can be performed.

The objective of this paper is to develop a process to engineering domain-specific languages in an evidence-based way taking available data into account. We present the main steps of evidence-based DSL engineering based on the development and refinement of Gherkin, which is a DSL for behavior driven development providing a test-driven approach to acceptance testing. We implement the DSL in Xtext and derive the empirical evidence from GitHub data, i.e., we extract and analyze all Gherkin features that were created and stored on GitHub until July 1, 2017.

The paper is structured ad follows. Section 2 gives an overview of background and related work on DSL engineering, evidence-based software engineering and behavior driven development. Section 3 describes the process of evidence-based language engineering by the example of developing a DSL for Gherkin based relevant data mined from GitHub. We conclude the paper and discuss directions of future work in Section 4.

2 BACKGROUND AND RELATED WORK

A DSL is optimized for a given class of problems, called a domain. It can be internal if it is embedded into a general-purpose language as a host or external otherwise. DSL engineering comprises the design, implementation, validation, deployment and maintenance of domain-specific languages [26]. It requires tool support which is provided by the development environment of a general-purpose language in case of an internal DSL or by a language workbench in case of an external DSL. A prominent language workbench for textually-represented DSLs is Xtext [7]. Xtext generates not only a parser for a DSL, but also a class model for the abstract syntax tree, as well as providing a fully featured, customizable Eclipse-based IDE including validation support.

*Also with Blekinge Institute of Technology, Department of Software Engineering.

EASE'18, June 28–29, 2018, Christchurch, New Zealand
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6403-4/18/06…$15.00
https://doi.org/10.1145/3210459.3210479

DLSs has become an important area of research within the field of software engineering. In a recent systematic mapping study on domain-specific languages Kosar et al. [19] identified and classified 390 papers on DSLs. Most papers address the design and implementation of DSLs, but only one their validation and nine their maintenance. Regarding the type of research, only about 7 % of the papers cover evaluation research. They confirmed an observation of Carver et al. [4] that within the DSL community, researchers are much more interested in creating new techniques and languages, than performing empirical evaluations, e.g. on the comprehension of DLSs compared to general-purpose languages [20]. Also a systematic mapping study on DSL evolution [24] from 2017 shows the same result and identifies hardly any evaluation research on DSLs or tools supporting DSL evaluation.

However, there are several tool-based approaches available that actively involve the user perspective in the design of DSLs. Cánovas Izquierdo and Cabot propose a community-aware DSL design process by enabling the active participation of all community members (both developers and end-users) from the very beginning [3, 15] supported by the collaborative meta modeling tool Collaboro. Häser et al. [12, 14] propose an experimentation environment for DSL engineering, which implements the design of DSLs as well as the definition and conduct of related experiments in the language workbench MPS. Finally, Barišić et al. [1] present a conceptual framework, called USE-ME, which supports the iterative incremental engineering process of DSLs concerning the issue of their usability evaluation based on experiments and surveys.

All these approaches focus on the active user feedback during DSL design. Our approach considers different sources of evidence, especially also from mining repositories and does not only focus on DSL design, but takes all phases of DSL engineering into account, i.e., DSL design, implementation, validation, deployment and maintenance.

Finally, one closely related paper [25] already uses the term "evidence-based" for the presented experiment on DSL evolution. The authors perform experiments whether a DSL for digital forensics supports corrective maintenance. Similar to the tool-based approaches presented before, the approach focuses on experimentation, but not on the whole process including issue identification, data collection and evidence appraisal as in our approach. However, it does not only address DSL design, but another phase, i.e., DSL maintenance.

In this paper we present an approach to evidence-based DSL engineering that goes beyond existing approaches by collective not only active user feedback but covers the whole iterative process of evidence-based DSL engineering including all phases of DSL engineering itself (i.e., design, implementation, validation, deployment and maintenance) as well as issue identification, data collection from different sources of evidence and evidence appraisal.

Evidence-based software engineering [18] aims to improve decision making related to software engineering by integrating best evidence with practical experience and human values. Empirical evidence, also known as sensory experience, is the knowledge or source of knowledge received by means of the senses, particularly by observation and experimentation [17]. Evidence-based software engineering comprises the five steps of (1) converting a relevant problem or information need into an answerable question, (2) searching for the best available evidence to answer the question, (3) critically appraising evidence for its validity, impact, and applicability, (4) integrating the appraised evidence with practical experience, stakeholder values and circumstances to make decisions about practice, and (5) evaluating performance and seeking ways to improve it.

Behavior Driven Development (BDD) [27] provides a test-driven specification approach to acceptance testing. It is widely used in modern software development. It combines the principles of test-driven development on the system level with ideas from domain-specific language engineering as it is largely facilitated through the use of a DSL that can express the technical test domain concepts of test behavior as well as the expected outcomes [13]. A DSL for testing helps to convert structured natural language statements into executable tests. Language support is therefore essential for the success of BDD. In this paper we use Gherkin, which is a DSL for BDD, as a running example to present our evidence-based language engineering approach.

Gherkin is a DSL based on the Given-When-Then canvas to describe preconditions, events as well as outcomes in the form of behavior centered tests that support the communication between business and technical stakeholders. Tests in Gherkin are interpreted by the framework Cucumber [5]. Figure 1 shows a Gherkin example feature for adding two values.

```
@Tags
Feature: Feature Name
    Some Description.
    What this Feature is for, often in the form of
    "As a", "In order to" and "I want to".

    Background:
        Given i have some global preconditions

    @ScenarioTags
    Scenario: Scenario Name
        Scenario Description

        @StepTags
        Given is the precondition
        And another condition
        When is the event
        Then is the outcome
        But not this

    Scenario Outline: Tables like this
        Given these values
            | a |
            | 1 |
            | 5 |
        When i add <a> and <b>
        Then i should get <sum>
            """
            If i need to add info that requires more than 1 line,
            i can write it as a Doc String.
            """

        @ExamplesTags
        Examples:
            Examples Description.
            | b | sum |
            | 1 | 2   |
            | 2 | 7   |
```

Figure 1: Gherkin Example

The example contains a description of the feature purpose, the background setting the context and several scenarios. Each scenario follows the Given-When-Then pattern describing the initial state of the system, the event or action taken, and the expected outcome, respectively. There are two more types of steps called And and But, used after Given and When-steps, respectively, to add additional pre- or post-conditions. Scenario outlines have one or more examples defined in additional data tables that contain concrete test values for several test runs. The table headers define variable names that can be used in the scenario text via between the brackets <> to refer to specific values. Finally, tags, that start with @, allow to organize features and scenarios.

Empirical investigations on DSL aspects of BDD are rare [23]. According to our knowledge, the only study was performed by Häser et al. [13], who investigate the role of domain concept in DSLs for BDD. However, the authors do not investigate empirical approaches to domain-specific language engineering in general and to not consider the mining of repository information to improve language design.

3 EVIDENCE-BASED DSL ENGINEERING

As highlighted before, the evidence-based paradigm is in the meanwhile well established in software engineering [17] and Dyba et al. [6] show how it can also be applied by practitioners to make informed decisions about software engineering problems. Here, we want to apply the paradigm to the special case of DSL engineering which also requires informed decision support, both from the user's and language engineer's perspective.

Figure 2 shows the process of evidence-based DSL engineering. The process iteratively executes a DSL engineering process that is complemented by an evidence cycle to continuously improve phases of the engineering process based on empirical observations. The DSL engineering cycle comprises the phases of DSL design, implementation, validation, deployment and maintenance. The evidence cycle comprises the identification of issues, the collection of data and the appraisal of evidence that results in a refinement of the DSL design, implementation, validation, deployment or maintenance.

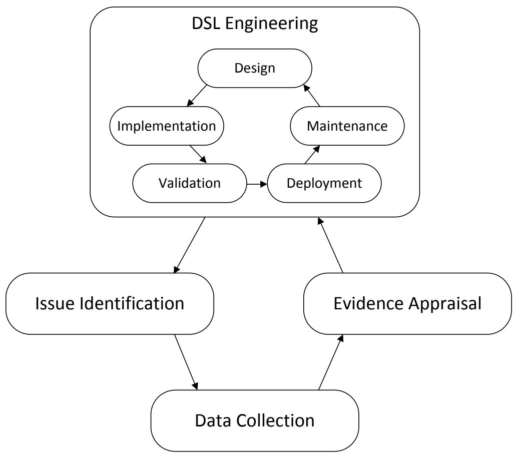

Figure 2: Process of Evidence-Based DSL Engineering

In the following section we discuss all phases of evidence-based DSL engineering in more detail. Each phase is presented by a running example, i.e., the refinement of a DSL for Gherkin newly implemented in Xtext by validation rules based on empirical evidence extracted from mining all GitHub repositories that contain Gherkin tests until July 1, 2017.

3.1 DSL Engineering

The DSL engineering cycle comprises the phases of DSL design (including definition of concrete and abstract syntax as well of the semantics and pragmatics of the language), implementation (as internal or external DSL), validation (of the syntax, semantics and pragmatics of the DSL), deployment (in the runtime environment) and maintenance (over the entire DSL lifecycle).

Example. In our example case we engineered a DSL for Gherkin (see Figure 1) in Xtext. Figure 3 shows an excerpt (i.e., the rules for a Feature, FeatureHeader, Background, Scenario, and Step) of the resulting abstract syntax for Gherkin implemented in Xtext. The Xtext sources are available online at http://bit.ly/gherkin-dsl-eng.

```
Feature:
    {Feature}
    NL*
    header=FeatureHeader?
    desc=Description?
    background=Background?
    scenarios+=Scenario*
;

FeatureHeader:
    {FeatureHeader}
    tags=Tags?
    keyword=FEATURE_HEADER_TYPE name+=GenericText* NL+
;

Background:
    {Background}
    tags=Tags?
    keyword=BACKGROUND_TYPE name+=GenericText* NL+
    desc=ScenarioDescription?
    steps+=Step*
;

Scenario:
    {Scenario}
    tags=Tags?
    keyword=ScenarioType name+=GenericText* NL+
    desc=ScenarioDescription?
    steps+=Step*
    examples+=Examples*
;

Step:
    tags=Tags?
    keyword=STEP_NAME name+=GenericText* NL+
    (docString+=CODE NL+)? table=Table?
;
```

Figure 3: Example Xtext Grammar Rules for Gherkin

Initially, the grammar only contains an abstract and concrete syntax but validation rules defining the static semantics of the language are missing. For instance, warnings if the number of scenarios per feature or of steps per scenario are extraordinarily high are not implemented due to missing knowledge about meaningful limits.

3.2 Issue Identification

The evidence-based software engineering approach starts with converting a relevant problem or information need into an answerable question. In our case, we have to identify language issues with respect to the design, implementation, validation, deployment or maintenance of the investigated DSL and turn them into answerable questions. The main challenge in this step is therefore to convert a language issue into a question that is specific enough to be answered but not so specific that you don't get any answers. According to Sackett [22] well-formulated questions usually cover the main intervention or action of interest, the context or specific situations of interest as well as the main outcomes or effects of interest. The priority of the question to be answered should be driven by importance and relevance for users and language engineers and take into account whether it is possible to answer the question in the available time.

Example. The basic Gherkin DSL does not contain suitable validation rules showing warnings if the number of specific elements is extraordinarily high. We therefore have to formulate questions on the upper limit of specific language elements. The lower limit is naturally given and is one for mandatory elements (e.g., scenarios) and zero for optional elements (e.g, tags). One type of element that is especially of interest and that we take as running example in this and the following sections is the number of scenarios per feature. The respective question that we want to answer based on evidence is "What is the typical upper limit of the number of scenarios for acceptance tests specified as scenarios in Gherkin?".

3.3 Data Collection

Finding an answer to the stated questions includes selecting appropriate information sources and executing a search strategy to collect relevant data. There are several information sources each with specific search strategies that can be used in software engineering in general in and in DSL engineering in particular. One can for instance interview or observe language users and engineers by surveys or experiments [12], search for research-based evidence in the scientific literature by performing or consuming a systematic literature review [17], collecting and analyzing grey literature like white papers or web sites [8] or mining public or private source code, bug or other software repositories [16]. As mentioned before the search strategy depends on the type of information source and has to take efficiency (available resources) and effectiveness (information needs to properly answer the stated questions) considerations into account.

Example. As the DSL is newly implemented and we assume that there is no previous experience with Gherkin in the organization, we do in this example not consider interviews or surveys of language users or engineers. Furthermore, according to our knowledge there are no systematic literature reviews or other scientific studies on BDD available, where limits for the number of scenarios are investigated. In general, scientific studies on language issues are rather rare. Therefore repository mining and grey literature analysis are especially important information sources for evidence-based DSL engineering. By applying the information mined from code repositories, language engineers and researchers do not need to depend primarily on their intuition, but on field data.

In our case we performed a comprehensive mining of Gherkin features available on GitHub. We extracted all Gherkin Features on GitHub as of July 1, 2017. To be more specific, all repositories on GitHub tagged with the language "gherkin" ("cucumber" works too, as GitHub considers them to be the same) was considered and each file ending with the "feature" extension inside those repositories was downloaded. GitHub uses the open source library called Linguist to determine file languages and then tags repositories with that language automatically. [10] files that do not end in the "feature" extension or could not be detected with Linguist could not be included in our data set. Overall we collected 8,492 features across 1,172 repositories inside a total of 960 projects. The search yielded 1,231 repositories tagged with Cucumber, but 59 did not have any features inside or contained features with an incorrect extension (which is anyway an indication for an anomaly) and were ignored. The Java code written for mining Gherkin features from GitHub is available online at http://bit.ly/gherkin-dsl-eng.

In addition, we also collected recommendations how to use Gherkin from the grey literature. We analyzed both the Cucumber Wiki[1] and the Cucumber reference guide[2]. From these two grey literature sources we could extract guidelines concerning steps per scenario, steps per background and names. We discuss them further in the evidence appraisal step discussed next.

3.4 Evidence Appraisal

Evidence appraisal comprises data quality assessment and analysis. As the quality of data and their sources as well as the rigor and relevance of studies differs, a quality assessment has to be performed before data is further analyzed. There are respective checklists for different types of data and studies available. For instance, Garousi et al. [9] propose a quality assessment checklist for grey literature in software engineering which considers the criteria authority of the producer, methodology, objectivity, date, position w.r.t. related sources, novelty, and impact. Depending on the type of data, one can apply quantitative (e.g., descriptive and inferential statistical methods) as well as qualitative data (e.g., grounded theory) analysis methods to synthesize evidence.

Example. As mentioned before, we consider two types of data, i.e., all Gherkin features extracted from GitHub in a data mining study and in addition grey literature data extracted from the Cucumber Wiki and reference guide.

Overall 8,492 Gherkin features were collected via GitHub. In the quality assessment phase 950 had to be removed because they used keywords (not the free text) in other spoken languages than English, files with wrong or inconsistent content (e.g., code following other programming languages or where the structure violated several grammar rules of Gherkin) or files that were completely commented out or too large for code generation exceeding the byte limit for Java classes. In addition, 498 files showed minor violations of the Gherkin grammar like typos in keywords or wrong usage of special

[1]The Cucumber Wiki is available online at https://github.com/cucumber/cucumber/wiki

[2]The Cucumber reference guide is available online at https://cucumber.io/docs/reference

characters like spaces, line breaks or escape symbols that could be fixed. Finally, 7,542 features remained and were further analyzed in terms of their structure to answer the stated questions.

Figure 4 shows the frequency distribution of scenarios per feature (and in addition of description lines, number of words in the name and tags per feature).

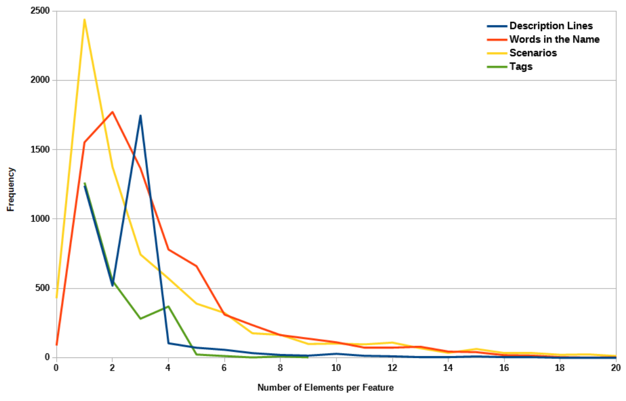

Figure 4: Distribution of Description Lines, Words in the Name, Scenarios and Tags per Feature

Figure 4 indicates that the frequency distribution of Scenarios per feature and also the second mandatory element, i.e., Words, follow a normal distribution, while the optional elements, i.e., Tags and Description, follow rather logarithmic distribution.

Table 1 shows the descriptive statistics for the number of scenarios per feature, which finally result in test cases and is therefore very essential.

Measure	All Scenarios	Without Outliers
Number of Values	7,542	7,296
Maximum	166	19
Minimum	0	0
Median	2	2
Mean	4.36	3.38
Standard Deviation	7.60	3.56

Table 1: Descriptive Statistical Measures for Scenarios per Feature with and without Outliers

If all features are considered then the highest number of scenarios per feature is 166, which is extraordinarily high. We therefore perform an outlier analysis based on Grubbs' outlier test [11] with a confidence level of 95%. It identifies 246 (3.26%) outliers, which were removed and the descriptive statistics was recalculated and is shown in the last column of Table 1. As a scenario is a mandatory element, the natural lower limit is one scenario per feature. A meaningful upper limit for the number of scenarios per feature can in our context be calculated based on the mean by adding two times the standard deviation, which results in an upper limit of approximately 10. This value can for sure later or based on other evidence be adapted.

The second source to derive guidelines are the Cucumber Wiki and reference guide. Both grey literature sources stem from a serious producer, are up-to-date, objective, comprehensive and have a high impact on other sources. We can therefore consider them as high quality additional sources to derive language recommendations. However, both sources do not contain recommendations for the upper limit of scenarios per feature, but contain other recommendations for instance for the number of steps per scenario. Table 2 summarizes all upper limits that we derived and finally implemented as validation rules in the Xtext grammer for Gherkin.

Rule	Upper Limit
Words per Feature Name	8
Words per Examples Name	4
Words per Scenario Name	9
Words per Step	11
Tags per Feature	4
Tags per Scenario	3
Scenarios per Feature	10
Steps per Background	6
Steps per Scenario	13
Table Columns	6
Table Rows	9

Table 2: Rules and their Upper Limits

DSL Reengineering. The next step is the application of the evidence, which closes the iterative cycle because it results in a DSL (re-)engineering based on the appraised evidence. In Figure 5 we show the implementation of the upper limit validation rule in Xtext. The rule ensures the number of Scenarios is limited to 10 or less per Feature and is implemented as a warning to indicate "abnormal" usage. The "@Check" annotation informs Xtext to execute this method, which is named according to what is actually, i.e., "validateScenariosPerFeature". The markImprovement method decides whether or not to display improvements based on a flag, with a given severity. The first parameter corresponds to the message displayed to users if this rule is violated, the second is the element that will be highlighted, and the third controls which part of the element to highlight. In the example of Figure 5 the element is a Feature and the part to be highlighted are the Scenarios.

```
@Check
def validateScenariosPerFeature(Feature el) {
    if (el?.scenarios != null) {
        if (el.scenarios.length > 10) {
            markImprovement("Abnormally high amount of Scenarios per Feature," +
                " should be 10 or less", el, GherkinPackage.Literals.FEATURE__SCENARIOS);
        }
    }
}
```

Figure 5: Xtext Validation Rule for Number of Scenarios per Feature

All implemented validation rules for the Gherkin grammar in Xtext are available online at http://bit.ly/gherkin-dsl-eng.

The evidence is directly integrated into the language. In our case the design of the language and the language validation are refined. In evidence-based DSL engineering the evidence is therefore directly implemented and made executable by integrating it into the engineered language. This helps to evaluate and refine the evidence and the language in another cycle as intended in the evidence-based DSL engineering process (see Figure 2).

4 CONCLUSION

In this paper we presented a process for evidence-based DSL engineering complements the DSL engineering process, which comprises the phases DSL design, implementation, validation, deployment and maintenance, with and evidence cycle to provide informed decision support for the phases of DSL engineering. The evidence cycle adds the phases issue identification, data collection and evidence appraisal. We demonstrate the approach by evidence-based engineering of a DSL in Xtext for the BDD language Gherkin. The evidence is based on a comprehensive mining of Gherkin features available on GitHub and an analysis of grey literature, i.e., the Cucumber Wiki and reference guide. We extracted all Gherkin Features on GitHub as of July 1, 2017 and investigated the question on a typical upper limit of the number of scenarios for acceptance tests specified as scenarios in Gherkin. Based on the collected and analyzed data we refined the Xtext grammar for Gherkin with a respective validation rule showing a warning if the number of scenarios per feature is greater than 10.

We think that the evidence-based approach has a high potential to significantly improve decision support in DSL engineering. The concept presented in this paper can be refined in research and practice in several directions.

As we assumed that an organization created a new DSL from scratch without prior experience, we used external open sources, i.e., data from GitHub and related references, to derive evidence. However, in practice if a language is continuously developed in an organization the internal evidence derived from statically mining own data, dynamically monitoring the usage of the DSL and interviewing language users may be more efficient and effective to derive evidence than from external sources like GitHub. However, one has to be careful when transferring findings into another setting. As with all research on empirical evidence in software engineering, drawing general conclusions from empirical studies in software engineering is difficult because any process depends to a large degree on a potentially large number of relevant context variables [2].

The evidence-based process example presented in this paper was relatively time-consuming which could limit its practical acceptance. Therefore it is essential to provide automation support to collect data and appraise evidence or to outsource activities. This could be supported by an experimentation platform for DSL engineering which allows to deploy language variants and to monitor and automatically collect runtime usage data to perform hypothesis tests. Also crowd sourcing of DSL engineering activities is promising and requires respective tool support and empirical investigation.

Finally, also the process for evidence-based DSL engineering should be refined based on empirical validation and evaluation. One can for instance make the different methods to derive evidence (like mining repositories, experimentation, surveys or systematic literature reviews) more explicit by defining specific language engineering activities and context variables that specify the circumstances under which specific empirical methods are efficient and effective.

REFERENCES

[1] Ankica Barišić, Vasco Amaral, and Miguel Goulao. 2018. Usability driven DSL development with USE-ME. *Computer Languages, Systems & Structures* 51 (2018), 118–157.

[2] Victor R Basili, Forrest Shull, and Filippo Lanubile. 1999. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering* 25, 4 (1999), 456–473.

[3] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2012. Community-driven language development. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering.* IEEE Press, 29–35.

[4] Jeffrey C Carver, Eugene Syriani, and Jeff Gray. 2011. Assessing the Frequency of Empirical Evaluation in Software Modeling Research. *EESSMod* 785 (2011).

[5] Cucumber. 2018. Cucumber. https://cucumber.io. (2018). Accessed: 2018-03-25.

[6] Tore Dyba, Barbara A Kitchenham, and Magne Jorgensen. 2005. Evidence-based software engineering for practitioners. *IEEE software* 22, 1 (2005), 58–65.

[7] Eclipse. 2018. Xtext - Language Engineering for Everyone. http://www.eclipse.org/Xtext/. (2018). Accessed: 2018-03-25.

[8] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. 2016. The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering.* ACM, 26.

[9] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. 2017. Guidelines for including the grey literature and conducting multivocal literature reviews in software engineering. *arXiv preprint arXiv:1707.02553* (2017).

[10] GitHub. 2018. About Repository Languages. https://help.github.com/articles/about-repository-languages. (2018). Accessed: 2018-03-25.

[11] Frank E Grubbs. 1950. Sample criteria for testing outlying observations. *The Annals of Mathematical Statistics* (1950), 27–58.

[12] Florian Häser, Michael Felderer, and Ruth Breu. 2016. An integrated tool environment for experimentation in domain specific language engineering. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering.* ACM, 20.

[13] Florian Häser, Michael Felderer, and Ruth Breu. 2016. Is business domain language support beneficial for creating test case specifications: A controlled experiment. *Information and Software Technology* 79 (2016), 52–62.

[14] Florian Häser, Michael Felderer, and Ruth Breu. 2018. Evaluation of an Integrated Tool Environment for Experimentation in DSL Engineering. In *International Conference on Software Quality.* Springer, 147–168.

[15] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2016. Collaboro: a collaborative (meta) modeling tool. *PeerJ Computer Science* 2 (2016), e84.

[16] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software: Evolution and Process* 19, 2 (2007), 77–131.

[17] Barbara Ann Kitchenham, David Budgen, and Pearl Brereton. 2015. *Evidence-based software engineering and systematic reviews.* Vol. 4. CRC Press.

[18] Barbara A Kitchenham, Tore Dyba, and Magne Jorgensen. 2004. Evidence-based software engineering. In *Proceedings of the 26th international conference on software engineering.* IEEE Computer Society, 273–281.

[19] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. 2016. Domain-specific languages: A systematic mapping study. *Information and Software Technology* 71 (2016), 77–91.

[20] Tomaž Kosar, Marjan Mernik, and Jeffrey C Carver. 2012. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical software engineering* 17, 3 (2012), 276–304.

[21] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[22] David L Sackett. 1997. *Evidence-based Medicine How to practice and teach EBM.* WB Saunders Company.

[23] Carlos Solis and Xiaofeng Wang. 2011. A study of the characteristics of behaviour driven development. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on.* IEEE, 383–387.

[24] Jürgen Thanhofer-Pilisch, Alexander Lang, Michael Vierhauser, and Rick Rabiser. 2017. A Systematic Mapping Study on DSL Evolution. In *Software Engineering and Advanced Applications (SEAA), 2017 43rd Euromicro Conference on.* IEEE, 149–156.

[25] Jeroen van den Bos and Tijs van der Storm. 2013. A case study in evidence-based DSL evolution. In *European Conference on Modelling Foundations and Applications.* Springer, 207–219.

[26] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL engineering: Designing, implementing and using domain-specific languages.* dslbook.org.

[27] Matt Wynne, Aslak Hellesoy, and Steve Tooke. 2017. *The cucumber book: behaviour-driven development for testers and developers.* Pragmatic Bookshelf.