Declarative Software Development

Distilled Tutorial *

Ralf Lämmel[1]
Andrei Varanovich[1] Martin Leinberger[1] Thomas Schmorleiz[1]
Jean-Marie Favre[2]

[1] Fachbereich Informatik, Universität Koblenz-Landau, Germany
[2] Laboratoire d'Informatique de Grenoble, Université de Grenoble, France

Abstract

Software development could be said to be declarative, if declarative programming languages were used significantly in the development of a software system. Software development could also be said to be declarative, if lightweight or heavyweight formal methods or model-driven engineering and model transformation were used as the primary development methods. This tutorial discusses another view on 'declarative software development'. That is, we promote the use of declarative methods for understanding software systems, software languages, software technologies, and software concepts. More specifically, we discuss a method package of a software ontology, automated software analysis, a modeling approach for software technologies, and *Linked Data*-based publication and exploration of software data.

Categories and Subject Descriptors D.2.2 [*SOFTWARE ENGINEERING*]: Design Tools and Techniques; D.2.4 [*SOFTWARE ENGINEERING*]: Software/Program Verification; D.2.11 [*SOFTWARE ENGINEERING*]: Software Architectures

Keywords Software Chrestomathy, 101companies, 101, Software Reverse Engineering, Software Ontology, SoLaSoTe, Software Technology, Feature Model, Technology Model, Megamodel, *MegaL*, Linked Data, Linked Software Data

* This online version is a very minor revision of the official publication. Specification regarding the author list from the point of view of the first author: Ralf Lämmel (faculty) delivers the tutorial and leads the research activities behind the tutorial. Andrei Varanovich (PhD student) carries out much of the research underlying the tutorial. Martin Leinberger (PhD student) and Thomas Schmorleiz (MSc student) contribute to the research in important ways. Jean-Marie Favre (faculty) has been involved in the research underlying the tutorial, especially at an earlier stage; he deserves much credit for the underlying vision.

PPDP'14, September 08 -10 2014, Canterbury, United Kingdom.

http://dx.doi.org/10.1145/2643135.2643163

1. Motivation of the Tutorial

Software development is breathtakingly heterogeneous in terms of the involved entities and methods; here are some dimensions or variation points—just to mention a few:

- Many different programming languages of different paradigms are used in development. Programming technologies are diverse, perhaps even more than languages.
- Many different concepts are involved. Beside the domain concepts of a system under development, there are many general concepts related to programming techniques, algorithms, data structures, and design.
- Different technological spaces [13] (such as Javaware or Modelware or Xmlware)[1] may be relevant, thereby effectively challenging language and technology choices, required developer skills, applicable development methods, etc.
- Different methodologies may be used, e.g., lightweight or heavyweight formal methods, or model driven engineering, or more conservative approaches with another dimension regarding agile versus traditional development management.

This heterogeneity is also manifest in actual software development projects in the sense that a substantial cocktail of entities of the aforementioned types may be simultaneously at play, e.g., multiple programming languages and a significant number of design concepts. The assumption is now that developers need to understand all the involved entities. To this end, they need to be provided with appropriate knowledge.

In this tutorial,[2] we discuss declarative methods for providing software developers with structured knowledge of software systems, software languages, software technologies, and software concepts. These methods are adapted from reverse engineering, semantic web, and modeling (megamodeling). More specifically, we discuss a method package of a software ontology, automated software analysis, a modeling approach for software technologies, and *Linked Data*-based publication and exploration of software data.

[1] We quote [13]: '*A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. It is often associated to a given user community with shared know-how, educational support, common literature and even workshop and conference meetings. It is at the same time a zone of established expertise and ongoing research and a repository for abstract and concrete resources.*'

[2] Webpage of tutorial: `http://softlang.uni-koblenz.de/ppdp14/`

Feature	Description	Data requirement	Functional requirement	Nonfunctional requirement	UI requirement	*OR* feature	Feature implication [2]
Company	Data model of companies with employees	✓				✓	
Flat company	Companies without departmental structure	✓					
Hierarchical company	Companies with nested departments	✓					
Flattened company	Flattened (normalized) representation	✓					Hierarchical company
Singleton	The constraint for a single company	✓					Company
Total	A query to total (sum up) the salaries of all employees		✓				Company
Median	A query to compute the median of the salaries		✓				Company
Depth	A query to determine departmental nesting depth		✓				Hierarchical company
Cut	A transformation to cut all salaries by half		✓				Company
COI	The ability to model conflicts of interest for employees	✓					Company
Mentoring	The ability to associate mentors and mentees	✓					Company
Ranking	The constraint for salaries to align with nesting level	✓					Hierarchical company
History	The ability to access company data over the timeline	✓					Company
Serialization	Serialization of company data in files		✓			✓	Company
Closed serialization	Technology-specific representation		✓				Company
Open serialization	Technology-oblivious representation		✓				Company
Persistence	Database-based persistence		✓				Company
Parsing	Parsing of text-based syntax for company data		✓				Company
Unparsing	Unparsing of text-based syntax for company data		✓				Company
Visualization	Visual syntax for company data		✓				Company
Mapping	Mapping company data across technological spaces			✓			Company
Distribution	Distribute data / functionality on client / server			✓			Company
Parallelism	Data parallelism for totaling huge companies			✓			Company
Logging	Logging of data changes			✓			Company
Browsing	Browsing company data				✓		Company
Editing	Editing company data such as names and salaries				✓		Browsing
Restructuring	Restructuring such as moving departments				✓		Editing
Web UI	Web-based user interface				✓		Browsing

Figure 1. Features of the 101system

Road-map of the distilled tutorial overview

Said declarative methods are sketched in §3. Prior to that, the software chrestomathy 101 [9, 15] is introduced in §2, as the chrestomathy is systematically used for illustration of the methods and some of the required infrastructure is effectively part of the 101 project. The layout of the actual tutorial, as planned for the PPDP audience, is briefly explained in §4.

2. The Software Chrestomathy '101'

We quote from [15]: "A *software chrestomathy* is a collection of software systems ('contributions') meant to be useful in learning about or gaining insight into software languages, software technologies, software concepts, programming, and software engineering." Software chrestomathies are similar to program chrestomathies (such as Rosetta Code[3]) except that (little) software systems rather than programs are collected. We continue quoting: "Such a step from programs to systems naturally expands the scope in which chrestomathies may be useful for learning and gaining insight. That is, the scope may include now software development or software engineering in addition to just programming. Further, software engineering ideas may be applied to the development (the assembly), the documentation, the maintenance, and the use of software chrestomathies."

The 101companies chrestomathy [9], or just '101',[4] collects 'contributions' which implement certain features of a feature model [2] for an imaginary human-resources management system; see Fig. 1 for an overview. The feature model is designed to touch upon many issues in programming and the use of software technologies. There are two parts to the contributions:

- 101repo: A federated repository to maintain all source code of the contributions with a version control system.
- 101wiki: A 101-specific semantic wiki (inspired by [12]) to maintain semi-structured documentation of the contributions and related entities.

The following Prolog code for the feature 'Total' is taken from a Prolog-based contribution. It implements a traversal over a term-

[3] `http://rosettacode.org/`

[4] `http://101companies.org/`

Top-level classification of entities

- Entity — *Everything in the scope of the software ontology*
  - Instrument — *Entities usable in software development*
    - Language — *Software languages such as Java or XML*
    - Technology — *Software technologies such as JUnit or Eclipse*
    - Concept — *Software concepts such as Visitor pattern or Unit testing*
  - Feature — *Features of 101's imaginary human resources management system*
  - Contribution — *Implementations of 101's imaginary system*
  - Contributor — *Contributors of code and documentation*
  - Theme — *Designated containers of related contributions*
  - Vocabulary — *Designated containers of domain-specific terms*
  - Resource — *External resources such as standards and specifications*

Some illustrative properties grouped by subject entity

Subject/Predicate	Object	Description	Example
Entity			
instanceOf	Entity	*An instance/type relationship*	*Prolog* **instanceOf** *Logic programming language*
isA	Entity	*A specialization relationship*	*Programming language* **isA** *Software language*
partOf	Entity	*A whole-part relationship*	*ghci* **partOf** *Haskell platform*
dependsOn	Entity	*Dependence relationship*	*Hackage* **dependsOn** *Cabal*
sameAs	URL	*Equivalence relative to external resource*	*Haskell* **sameAs** `http://www.haskell.org`
similarTo	URL	*Similarity relative to external resource*	*Monad* **similarTo** *wikipedia ... Monad*
documentedBy	Contributor	Authorship of documentation	*haskellSyb* **documentedBy** *Ralf Lämmel*
Contribution			
uses	Instrument	*Instrument usage*	*haskellSyb* **uses** *Data.Generics*
implements	Feature	*Feature implementation*	*haskellSyb* **implements** *Total*
developedBy	Contributor	*Developer of contribution*	*haskellSyb* **developedBy** *Ralf Lämmel*
varies	Contribution	*Indication of variation*	*haskellSyb* **varies** *haskellComposition*
moreComplexThan	Contribution	*Indication of complexity*	*haskellEngineer* **moreComplexThan** *haskellStarter*
Technology			
uses	Instrument	*Instrument usage*	*Hackage* **uses** *Versioning*
implements	Resource	*Compliance with a standard, et al.*	*ghci* **implements** *Haskell 2010 Language Report*
supports	Concept	*Support of a protocol, et al.*	*Ruby on Rails* **supports** *REST*

Figure 2. A sketch of the schema of *SoLaSoTe*

based representation of company data to total (i.e., sum up) all salaries:

```
% Total all salaries in a company
total(X,R) :- collect(getSalary,X,L), sum(L,R).

% Helper for salary extraction
getSalary(employee(_,_,S),S).

% Higher-order traversal for accumulation
collect(P,X,L) :-
  apply(P,[X,Y]) ->
    L = [Y];
    X =.. [_|Xs],
    maplist(collect(P),Xs,Yss),
    append(Yss,L).
```

The code illustrates higher-order style and generic term manipulation in Prolog [14, 20]. The following Haskell code, again for the feature 'Total', is taken from a Haskell-based contribution, while the technology *Data.Generics* (a library) is used for SYB style of generic programming [16].

```
module Company.Total where
import Company.Data
import Company.Generics
import Data.Generics

-- Total all salaries in a company
total :: Company -> Float
total = everything (+) (extQ (const 0) id)
```

The shown source-code units are taken from 101repo. We will see some bits of 101wiki-based documentation in a second. The tutorial uses 101's contributions for demonstrating the kind of knowledge that the tutorial's methods can provide to software developers, as discussed in the next section.

3. Declarative Methods of the Tutorial

We will discuss now several declarative methods for use in software development. These methods are not restricted to any particular discipline of development (such as formal methods). Rather these methods enrich programming or development with means for comprehension and knowledge management.

3.1 The Software Ontology SoLaSoTe

Ontologies are increasingly used in software engineering for analysis, design, implementation, documentation, testing, and maintenance of software systems [1]. The software ontology *SoLaSoTe*, which is being developed in tandem with 101, serves for the classification and other forms of characterization of software languages, software technologies, and all kinds of software concepts relevant for programming and development (e.g., design patterns, programming techniques, data structures, algorithms). The ontology is maintained on 101wiki.

Fig. 2 sketches *SoLaSoTe*'s schema: the major entity types and available properties (say, 'predicates' in the RDF terminology). For instance, 'isA' and 'instanceOf' are used for classification.

- *this* **developedBy** *Contributor:DerJackel*
- *this* **implements** *Feature:Cut*
- *this* **implements** *Feature:Depth*
- *this* **implements** *Feature:Total*
- *this* **uses** *Language:Java*
- *this* **uses** *Language:SQL*
- *this* **uses** *Technology:Eclipse*
- *this* **uses** *Technology:H2*
- *this* **uses** *Technology:JDBC*

Figure 3. *SoLaSoTe* properties of one of 101's contribution

SoLaSoTe and 101 are intertwined in so far that the semi-structured documentation of contributions on 101wiki refers to *SoLaSoTe* entities and uses designated properties; see the schema's block with 'Contribution' as subject in Fig. 2. Documentation may thus declare the features implemented by a contribution as well as the languages, technologies, concepts (e.g., design patterns) used by the contribution. For instance, Fig. 3 shows such properties for a Java-based contribution, as part of its documentation, as rendered on 101wiki. The contribution exercises database technologies as well as SQL while implementing a number of 101's features. 'this' is the subject of the properties, i.e., the actual contribution.

The schema of Fig. 2 can be represented more formally in RDFS and OWL and the consistency of the actual ontology (i.e., use of the right predicates for the right subject and object types) can be checked by SPARQL queries that are obtained as interpretation of OWL. In this manner, 101wiki is also partially validated. This mix of declarative methods is inspired by Semantic Web research.

The expectation is that the ontology helps developers understanding the languages and technologies and concepts they are dealing with. Importantly, developers can query the ontology through 101triples—a SPARQL endpoint. For instance, the following query identifies 'popular concepts', i.e., it sorts concepts referred to in the documentation of contributions by the number of referring contributions.

```
SELECT ?concept (COUNT(DISTINCT ?contribution) AS ?count)
WHERE {
  ?concept rdf:type onto:Concept .
  ?contribution rdf:type onto:Contribution .
  ?contribution ?p ?concept .
}
GROUP BY ?concept
ORDER BY DESC(?count)
```

As of writing, the top-ten most popular concepts with number of referring contributions in parentheses are the following: MVC (11), GUI (11), Task parallelism (9), POJO (7), Data composition (7), Data parallelism (6), Object model (6), In-memory XML processing (6), Visitor pattern (5), Record (4).

3.2 Automated Software Analysis with 101meta

We use a computational infrastructure, 101worker [8], to apply diverse automated analyses in the realm of software reverse engineering to the source code and the documentation of 101 (i.e., 101repo and 101wiki). For instance, ontology validation, fact extraction, feature detection, and metrics calculation are performed. We focus here on one line of analyses—namely those that are expressed in a rule-based language 101meta [8] for associating metadata automatically with source-code files.

That is, 101meta rules analyze source-code files to identify implemented features as well as used languages, technologies, and concepts. The documentation, as discussed in §3.1 (see Fig. 3), may also identify such characteristics, but it does not associate such data with specific source-code files or even fragments thereof. Also, the apparent redundancy between documentation and analysis helps with validation of the available knowledge.

A 101meta rule, when applied to a file, *checks* conditions for the file and *associates* metadata with the file, if the conditions are met. For example, the following rule models *feature detection* [6] for 'Total'; 101meta is rendered here in JSON-based syntax:

```
{
  "check" : {
    "run" : {
      "name" : "check-token",
      "args" : "total"
    }
  },
  "associate" : {
    "comment" : "automated feature detection",
    "feature" : "Total"
  }
}
```

The shown rule runs a programmatic test, 'check-token', on the file. This program actually checks the file's tokenized content so that only files with the token 'total' are accepted. If so, a metadata unit, attesting to the detected feature, is associated with the file. The tokenization of source code may also include splitting of compound identifiers and stemming.

3.3 The *MegaL* Language for Technology Models

The *MegaL* language [10, 17] for technology models is developed in tandem with 101, *SoLaSoTe*, and 101meta. A *MegaL* model describes important characteristics of a software technology in relation to relevant software artifacts, software languages, and yet other entities. Thus, *MegaL* models are simple declarative models; in fact, they are entity-relationship models with entities for languages, technologies, concepts, and artifacts and with relationships to express data flow, dependencies, conformance, and others. Such technology modeling is a form of megamodeling; other forms of megamodeling have seen much recent interest in MDE, model management, software architecture, and software language engineering [7, 21, 22].[5]

The illustrative *MegaL* model of Fig. 4 concerns the JAXB technology of XML-data binding, as part of the Java platform. The megamodel declares JAXB as technology, the involved languages for XML and OO types, the corresponding artifacts (files), the actual code-generation or mapping component 'xjc' (part of JAXB), and ultimately the data flow of applying code generation. In the model, rather than saying the mapping result is element of the Java

[5] In simple terms, a megamodel [4] is a model whose model elements are again models by themselves. We should apply a broad interpretation of 'model'—to include models as in software modeling, metamodels, transformation models, but also similar artifacts from other technological spaces, e.g., programs or grammars.

```
Technology JAXB // The XML−data binding technology

/* Involved languages */
Language XML // XML
Language XSD // XML schemas
XSD subsetOf XML // XSD being an XML language
Language Java // Full Java
Language JavaJaxb // Java subset targeted by JAXB
JavaJaxb subsetOf Java // A subset indeed

/* Involved artifacts */
File xmlTypes // The initial XML types
File ooTypes // The derived classes
xmlTypes elementOf XSD // XSD−based XML types
ooTypes elementOf JavaJaxb // Classes of JAXB's Java subset

/* The code generator / mapping */
xjc: Function[XSD → JavaJaxb] // A function
xjc partOf JAXB // Generator component of JAXB
xjc(xmlTypes) ↦ ooTypes // Its application
xmlTypes correspondsTo ooTypes // The correspondence
```

Figure 4. A technology model for some aspect of Java's XML-data binding approach with JAXB

language, we designate a technology-specific subset 'JaxbJava' to better convey that the generated Java code exercises a rather limited Java subset: classes rather than interfaces; trivial getters and setters for private attributes rather than arbitrary methods; use of JAXB-specific annotations. Further, there is a correspondence relationship between XML types and Java classes: one class for each XML type (and vice versa) with essentially the same name and attributes (children).

Ontological knowledge based on *SoLaSoTe* and *MegaL* models complement each other. The former focuses on classification and characteristics overall. The latter focuses on the use of the technology in terms of the involved artifacts, their characteristics, and related data flows. *MegaL* has an interesting semantics [17] such that one can verify whether a given software system appears to conform with the assumed model of technology usage. This form of verification relies on binding *MegaL* models to actual systems and custom evaluators for interpreting the involved relationships such as 'elementOf', '↦', and 'correspondsTo' in the example.

3.4 Linked Software Data

At this point, we have covered various kinds of 'software data': actual source code (such as 101's contributions), the *SoLaSoTe* ontology, derived software data based on reverse engineering functionality (such as metadata derived by the rule-based language 101meta), and *MegaL* models. These kinds of data are clearly related. For instance, ontology entities are referenced by metadata and megamodels; metadata is associated with source-code units.

The question arises as to how to integrate all such data in a way that different stakeholders may access the data conveniently. We use *Linked Data* [5, 11] to this end. *Linked Data* is a method of publishing structured data so that it can be interlinked and accessed conveniently including computer-based remote access. The basic *Linked Data* principes are these [3]:

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using standards (RDF*, SPARQL).
4. Include links to other URIs. so that people can discover more things.

These principles are applied in a non-trivial way to our situation; see [19] for details:

- There is a central HTTP URI service, 101explorer,[6] which allows exploration of all entities grouped by type ('namespace'). That is, there are groups for languages, technologies, concepts, contributions, and yet others. A lookup returns links to related resources, e.g.:
  - All ontological entities are manifested on 101wiki, e.g., `http://101companies.org/wiki/Language:Haskell` for Haskell.
  - As 101's contributions are physically hosted with a version control system (in fact, with GitHub), the corresponding repo URLs serve as related resource.
- The *SoLaSoTe* ontology is represented as an RDF triplestore accessible through the SPARQL endpoint 101triples. In this manner, one can query the ontology in a browser or programmatically (through an API). Query results are essentially URIs which can be looked up again with 101explorer.
- The tree-like structure of source-code folders for 101's contributions is also discoverable through 101explorer. In fact, even individual source-code files are further taken apart based on language-parametric fact extraction machinery. Thus, every source-code unit (class, function, data type, etc.) has a resolvable URI. For instance, the Haskell function *total* of the Haskell file shown in §2 has this URI: `http://101companies.org/resources/contributions/haskellSyb/src/Company/Total.hs/pattern/total`. In this manner, one can programmatically process all source code.
- Data computed by reverse engineering functionality is published on a designated file server 101data.[7] Data may come in the form of *dumps* such as the set of all 101meta rules gathered from different repositories. Data may also come in the form of 'tandem' files complementing source-code files. For instance, for each source-code file f, there are tandem files like these:
  - f.metrics.json: lines of code (LOC) and other basic metrics.
  - f.extractor.json: basic extracted facts, e.g., imports.
  - f.fragments.json: a list of fragment URIs.
  - f.matches.json: metadata units assigned via 101meta rules.

  For instance, the '.metrics.json' file for the Haskell file shown in §2 looks like this:

```
{"size":198,"loc":10,"ncloc":10,"relevance":"system"}
```

The *Linked Data* approach enables queries that can simultaneously apply to *SoLaSoTe*, 101's contributions, and derived resources. In [18], we describe a case study to compare feature implementations across languages, technologies, and styles. To this end, we rely on automatic feature detection to associate source-code units with 101's features; we rely on computed metrics calculation for comparing different implementations; we rely on the triplestore for validation of detected features relative to declared features.

For illustration, consider Fig. 5. It visualizes the non-comment LOC metric for seven of 101's contributions; four written in Java, two written in Haskell; one written in Python. These contributions implement features 'Total', 'Cut', and 'Hierarchical company' in a modular fashion, thus allowing precise measurement of lines of code for the feature-implementation code. Among the Java-based

[6] `101companies.org/resources?format=html`

[7] `http://data.101companies.org/`

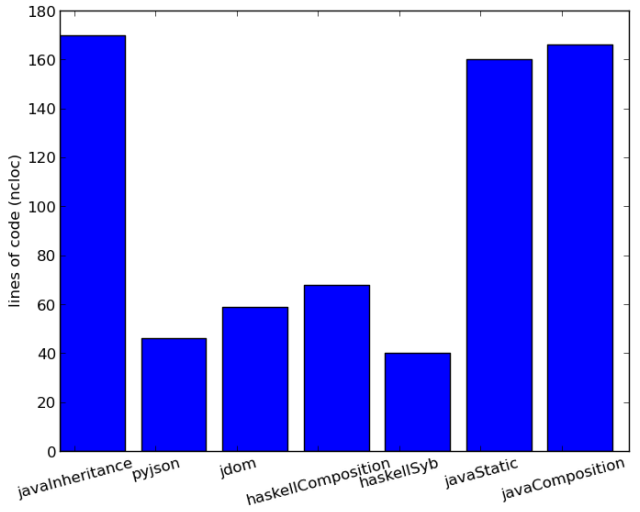

Figure 5. Comparison of feature implementations across languages, technologies, and styles (Source: [18])

contributions, there is one, *jdom*, with less LOC than the others. This is because the contribution does not use an explicit object model. Still, one of the Haskell-based contributions, *haskellSyb* (touched upon in §2), has even less LOC, despite an explicit data model. This is a highly simplified discussion, but one can probably see what sort of investigations are made possible by queries over such rich and integrated software data.

4. Layout of the tutorial

The tutorial systematically explores online structured content concerning the subjects of the methods section (§3). In particular, 101wiki, 101explorer, 101data, and 101triples are leveraged for exploration. Concretely, the following topics are covered:

- The use of formal feature models in designing and implementing software systems, as leveraged in 101.
- The design of the *SoLaSoTe* ontology for the classification and characterization of systems, languages, technologies, and concepts; the realization and use of *SoLaSoTe* by means of the formalisms / languages RDF, RDFS, OWL, and SPARQL.
- The rule-based language 101meta and the computational infrastructure 101worker for reverse engineering in the sense of systematic software data extraction on the heterogenous corpus of 101's source code.
- The use of megamodels for technology modeling, as supported by *MegaL*, such that principle artifacts, conceptual entities, data flows, and dependencies are captured, and verification of technology usage in actual systems is achieved.
- The use of *Linked Data* principles, as supported by RDF, HTTP, REST, and SPARQL, for publishing and querying data (knowledge) about software.

References

[1] E. Ahmed. Use of ontologies in software engineering. In *SEDE*, pages 145–150. ISCA, 2008.

[2] D. S. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. of SPLC 2005*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[3] T. Berners-Lee. Linked DataDesign Issues, 2006. `http://www.w3.org/DesignIssues/LinkedData.html`.

[4] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *European MDA Workshops MDAFA 2003 and MDAFA 2004, Revised Selected Papers*, volume 3599 of *LNCS*, pages 33–46. Springer, 2005.

[5] C. Bizer, R. Cyganiak, and T. Heath. How to publish Linked Data on the web, 2007. Online tutorial `http://wifo5-03.informatik.uni-mannheim.de/bizer/pub/LinkedDataTutorial/`.

[6] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[7] J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. *ENTCS*, 127(3), 2004. Proc. of the SETra Workshop.

[8] J.-M. Favre, R. Lämmel, M. Leinberger, T. Schmorleiz, and A. Varanovich. Linking Documentation and Source Code in a Software Chrestomathy. In *Proc. of WCRE 2012*, pages 335–344. IEEE, 2012.

[9] J.-M. Favre, R. Lämmel, T. Schmorleiz, and A. Varanovich. 101companies: A Community Project on Software Technologies and Software Languages. In *Proc. of TOOLS 2012*, volume 7304 of *LNCS*, pages 58–74. Springer, 2012.

[10] J.-M. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In *Proc. of MODELS 2012*, volume 7590 of *LNCS*, pages 151–167. Springer, 2012.

[11] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool, 2011. 1st edition.

[12] M. Krötzsch and D. Vrandecic. Semantic Wikipedia. In *Social Semantic Web*, X.media.press, pages 393–421. Springer, 2009.

[13] I. Kurtev, J. Bézivin, and M. Akşit. Technological Spaces: an Initial Appraisal. In *Proc. of CoopIS, DOA 2002, Industrial track*, 2002.

[14] R. Lämmel. Scrap your boilerplate: prologically! In *Proc. PPDP 2014*, pages 7–12. ACM, 2009.

[15] R. Lämmel. Software chrestomathies. *Sci. Comput. Program.*, 2013. In press.

[16] R. Lämmel and S. L. P. Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proc. of TLDI 2003*, pages 26–37. ACM, 2003.

[17] R. Lämmel and A. Varanovich. Interpretation of Linguistic Architecture. In *Proc. of ECMFA 2014*, volume 8569 of *LNCS*, pages 67–82. Springer, 2014.

[18] R. Lämmel, M. Leinberger, T. Schmorleiz, and A. Varanovich. Comparison of feature implementations across languages, technologies, and styles. In *Proc. of CSMR-WCRE 2014*, pages 333–337. IEEE, 2014.

[19] M. Leinberger. Enhancement of a software chrestomathy for open linked data, 2013. MSc Thesis. Universität Koblenz-Landau. Fachbereich Informatik (FB4). Available online `http://softlang.uni-koblenz.de/LeinbergerMScThesis.pdf`.

[20] L. Naish and L. Sterling. Stepwise Enhancement and Higher-Order Programming in Prolog. *Journal of Functional and Logic Programming*, 2000(4), 2000.

[21] J.-S. Sottet, G. Calvary, J.-M. Favre, and J. Coutaz. Megamodeling and Metamodel-Driven Engineering for Plastic User Interfaces: MEGA-UI. In *Human-Centered Software Engineering*, Springer Human-Computer Interaction Series, pages 173–200, 2009.

[22] V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In *Proc. MODELS'14*, volume 8767 of *LNCS*, pages 50–67. Springer, 2014.