University of Koblenz-Landau
Department of Computer Science

Software Chrestomathy as a Knowledge-Driven Research Infrastructure for Software Engineering

Andrei Varanovich

October 2017

Vom Promotionsausschuss des Fachbereichs 4: Informatik der Universität Koblenz-Landau
zur Verleihung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation

Vorsitzende des Promotionsausschusses:	Prof. Dr. Dietrich Paulus
Berichterstatter:	Prof. Dr. Colin Atkinson
	Prof. Dr. Ralf Lämmel
	Prof. Dr. Alfonso Pierantonio
Datum der wissenschaftlichen Aussprache:	14.07.2017

Abstract

The term "Software Chrestomaty" is defined as a collection of software systems meant to be useful in learning about or gaining insight into software languages, software technologies, software concepts, programming, and software engineering. *101companies* software chrestomathy is a community project with the attributes of a Research 2.0 infrastructure for various stakeholders in software languages and technology communities. The core of *101companies* combines a semantic wiki and confederated open source repositories. We designed and developed an integrated ontology-based knowledge base about software languages and technologies. The knowledge is created by the community of contributors and supported with a running example and structured documentation. The complete ecosystem is exposed by using Linked Data principles and equipped with the additional metadata about individual artifacts. Within the context of software chrestomathy we explored a new type of software architecture – linguistic architecture that is targeted on the language and technology relationships within a software product and based on the megamodels. Our approach to documentation of the software systems is highly structured and makes use of the concepts of the newly developed megamodeling language MegaL. We "connect" an emerging ontology with the megamodeling artifacts to raise the cognitive value of the linguistic architecture.

Zusammenfassung

Der Begriff “Software Chrestomaty” ist als Sammlung von Betriebssystemen definiert, die nützlich sein kann, um über Betriebssprachen, Betriebstechnologien, Konzepts, Programmierung und Software-Engineering zu lernen oder einen Einblick in denen zu gewinnen. *101companies* software chrestomathy ist ein Gemeinschaftsprojekt mit den Merkmalen von Research 2.0 Infrastruktur für die unterschiedlichen Verwahrern in Betriebssprachen und Technologiegemeinschaften. Das Kernstück von *101companies* umfasst ein semantisches Wiki und verbündete Open-Source-Repositories. Wir gestalteten und entwickelten eine integrierte auf Ontologie basierende Informationsbank über Betriebssprachen und Technologien. Die Information ist bei der Gemeinschaft der Beitragsleistenden geschafft und mit einem laufenden Beispiel und strukturiertem Belegmaterial unterstützt. Das ganze Ökosystem wird mit der Verwendung von Link Data Prinzipien aufgedeckt und mit zusätzlichen Metadaten über die individuellen Artefakte ausgerüstet. Im Kontext von Software Chrestomaty untersuchten wir eine neue Art der Softwarearchitektur, bzw. linguistische Architektur, die sich auf das Verhältnis zwischen Sprachen und Technologie in einem Softwareartikel abzielte und sich auf die Megamodelle beruhte. Unser Vorgehen mit der Beschreibung des Betriebssystems ist hoch strukturiert und nutzt die Konzepte der neu entwickelten Sprache MegaL. Wir schließen die entstehende Ontologie mit den Megamodellartefekten an, um den kognitiven Wert der linguistischen Architektur zu erhöhen.

Acknowledgements

This research was done within the Software Language Team of the University of Koblenz-Landau led by Prof. Dr. Ralf Lämmel – my supervisor. His vision and professionalism played a big part in the success of this research. The energy and inspiration of Jean-Marie Favre from the University of Grenoble (France) helped enormously at the beginning of this research. His broad outlook helped to set up a research agenda. I am grateful to GTTSE 2011, and SoTeSoLa 2013 summer school participants and organizers. Vadim Zaytzev from CWI (Amsterdam, the Netherlands) was the early adopter of our megamodeling approach, which was quite inspiring and led to a series of productive visits. It would never be possible to implement so complex technical infrastructure without the substantial contributions of Thomas Schmorleiz, and Martin Leinberger who are fully committed to the project. Finally, my thanks to the long list of students of the University of Koblenz-Landau who contributed to the project in various roles.

A number of events helped to validate the early research ideas. SATToSE seminar series enabled valuable discussion with the broader yet very friendly community. I am thankful to Paul Klint from CWI (Amsterdam, the Netherlands) for his critical retrospective on the matters of science and Erwann Wernli from University of Bern (Switzerland) for his tips as an "experienced" Ph.D. student.

Additionally I would like to thank my two colleagues at the University of Koblenz-Landau, Sabine Hülstrunk and Ekaterina Pek.

This journey would never be possible without my wife Tatsiana.

Dedication

In memory of Anatoly Alexeevich Minkovsky (1945 - 2014), my teacher and friend.

"The game of science is, in principle, without end. He who decides one day that scientific statements do not call for any further test, and that they can be regarded as finally verified, retires from the game."

Sadia Malik

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter we identify the scope of the research, introduce the key definitions and contributions, outline the research methodology.

Research Context

Today's developers face a myriad of software technologies and software languages. The IT industry demands technology-savvy, polyglot developers with strong knowledge of entire development ecosystems. Any project of significant size can involve a dozen different technologies and languages, each one with specific concepts and terminology, possibly obfuscated by buzzwords. Types of languages include programming languages, modeling languages, and technology-specific languages for configuration and metadata. In 2006, Neal Ford, a software architect at ThoughtWorks, commented on the state of the IT industry, specifically the growing popularity of .NET platforms, Java platforms, and web development: "Now, increasingly, we're expanding our horizons. More and more, applications are written with Ajax frameworks (i.e., JavaScript). If you consider the embedded languages we use, it's even broader: XML is used as

This chapter is an original part of the thesis. The key definitions were introduced in two conference papers: [FLSV12] Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz and Andrei Varanovich. 101companies: A community project on software technologies and software languages. In TOOLS (50), pages 58–74, 2012. [FLV12] Jean-Marie Favre, Ralf Lämmel and Andrei Varanovich. Modeling the linguistic architecture of software products. In MoDELS, pages 151–167, 2012.

an embedded configuration language widely in both the Java and .NET worlds."[1] In 2008, the adoption of polyglot persistence began with the new generation of database platforms: "Polyglot persistence, like polyglot programming, is all about choosing the right persistence option for the task at hand."[2]

We believe this industrial trend brings new challenges for programming-language and software-engineering research in the form of new models of development, documentation, evolution, adoption, comprehension, comparison, and education, where such heterogeneous, multilanguage and multitechnological setups are considered a key requirement.

Technological Spaces

In the modeling, metamodeling, and software language communities, the notion of a technological space (TS)[3] helps in identifying and communicating commonalities and differences in grammarware, XMLware, modelware, objectware, and tupleware [KBA02a, DGD06]. A TS is defined as a "working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities" [KBA02a]. It is often associated with a given user community with shared know-how, educational support, common literature, and even regular workshops and conferences. In the model transformation context, a TS can be defined as the metametamodel that is used [MVG06], but this definition cannot be applied generally.

All TSs have different levels of maturity. For some TSs, the scope of research and best practices are not established. For instance, in the grammarware context, the divide between having best practices and not having them is called "hacking versus engineering," where *hacking* refers to accomplishing tasks with ad hoc approaches for a long time [KLV05].

As suggested by the definition of TS, inventions are often made in the context of a single engineering field, typically associated with a specific conference series. Consequently, TSs are often more like silos, but there are efforts to build bridges between them (i.e., to build interoperability

[1] `http://memeagora.blogspot.de/2006/12/polyglot-programming.html`

[2] `https://www.altamiracorp.com/blog/employee-posts/polyglot-persistence`

[3] The term *technological space* is often referred to as a *technical space*. In this thesis, we use the former, original variant.

between technological platforms). In many cases, the impedance mismatch is revealed, and the bridging technologies are developed [LM06b, Mei06]. The process of bridging TSs shows the need for a proper level of abstraction over them to be able to reuse the artifacts from one TS in other TSs. Not surprisingly, semantic web ontologies and Model Driven Engineering (MDE)—in particular model transformations—became promising research directions [DGD06, BDJ+03]. Based on sound foundations, they aim to orchestrate different TSs. Engineers benefit from them by understanding the diversity of things that can be modeled. On a practical level, it is still difficult to see a big picture due to the variety of involved technologies and limited tool support.

An open, visionary question underlies the general agenda for research on TSs: is there a unifying theory of TSs? Answering this question requires a joint community effort. Such an effort should be cross-disciplinary and requires coordinated effort from different TS communities. One promising example is the Software Language Engineering (SLE) community: "SLE's mission is to fuse several communities that have traditionally looked at software languages from different and yet complementary perspectives."[4]

Data Mining from Community Knowledge Resources

With the growth of the Internet and social media, community knowledge has become an integral part of the development process. Many developers invest their time in answering questions on popular forums, such as StackOverflow.[5] Their contributions on such forums, in turn, increase their development performance [VFS13]. In 2008, GitHub launched a social-coding platform with rich community functionalities (such as wikis, followers, and network graphs). By the end of 2013, GitHub had reached 10 million source-code repositories.[6]

Such knowledge sharing ecosystems became an important source of information for many research communities. For instance, the International Working Conference on Mining Software Repositories (MSR) has hosted a mining challenge since 2006.[7] In 2013 and 2014,

[4] http://www.sleconf.org/2014/
[5] http://stackoverflow.com
[6] https://github.com/blog/1724-10-million-repositories
[7] http://msrconf.org/challenge.php

the datasets for its mining challenge were created from StackOverflow and GitHub, respectively [Bac13, Gou13].

Recommendation systems and integrated development environments (IDEs) are additional contexts that employ community knowledge. Recent research on bug-tracking systems has shown that bug-fixing times are lower when reports about bugs come with additional Twitter, *Wikipedia*, or StackOverflow links [CS13]. This boost in productivity also occurs in IDEs where developers do not need to change the context while facing a question about an unknown API or an exception's stack trace [RYR13].

Today's polyglot developers are a part of the global, collaborative knowledge-creation ecosystem. This knowledge-creation ecosystem has complex attributes, such as quality of knowledge (i.e., based on voting and individual-expertise score) and problem context (i.e., based on tagging). On the other side, many books are available online and provide more classical, expert views on different TSs; these books are often more systematic and tend to cover a well-defined scope. Such a rich and diverse knowledge ecosystem contains state-of-the-art information about many domains of software engineering. However, the lack of a uniform model of the knowledge leaves the question of mining the data for research unanswered.

Linked Data and Open Data for Software Engineering

An important aspect of software engineering research and education is that resources be open and free to access, which enables researchers to develop, reuse, compare, and evaluate different techniques.

Collaborative research is joint work by different partners where everyone contributes new knowledge and effort in response to a research challenge. An example of collaborative research is e-research, which further stimulates the use of advanced technologies to solve research tasks that are data intensive and require cross-disciplinary effort. E-research is built around information-centric research capabilities and technologies that help researchers collect, manage, share, process, analyze, store, find, understand, and reuse information.

In software engineering, Linked Data[8] and Open Data [ABK+07b] are two notable concepts that support technical collaboration. Linked Data principles help researchers expose the most popular software-engineering artifacts. Software repositories exposed in this way are ready for software-repository mining [KFH+12c, KFRC11]. Heterogeneity of documentation (e.g., via wikis and schemas) in different repositories can be properly orchestrated [How08].

In the context of software-engineering education, a Massive Open Online Course (MOOC) is a new tool that leverages Linked Data and Open Data principles. In addition to traditional course materials, MOOCs comprise a community of students participating in the learning process.

On the conceptual side, MOOCs often promote Open Data compliance, for example, through open and free-to-use content and supporting resources. However, MOOCs became just a different delivery model from the ones used in traditional curricula (i.e., university courses). Most of the visible MOOCs use closed licenses and limit access rights. Their learning resources are only loosely integrated into a global knowledge space.

Viewpoints in Software Architecture

IT professionals need to be equipped with various knowledge tools to deal with the complexity of real-world software systems. Knowledge about actual systems is often abstracted and represented in whatever form is most useful for stakeholders. The *viewpoint-oriented approach* aims to integrate multiple perspectives in system development [FFK+92]. In the context of enterprise architecture, this approach has already been popular for two decades [SAtDL04] and covers requirements for engineering and aligns viewpoints with modeling languages, such as Unified Modeling Language (UML) [KS96, SE93]. In practice, viewpoints—perspectives on the software project—are used on the basis of the Reference Model of Open Distributed Processing (RM-ODP)[9] and the Institute of Electrical and Electronics Engineers (IEEE) 1471 standard.[10] Each viewpoint is typically associated with one or more designated modeling languages [DQPvS03] and is subject to different metaware, that is, metamodels and model-

[8] `http://linkeddata.org/`
[9] `http://www.rm-odp.net/`
[10] `http://standards.ieee.org/findstds/standard/1471-2000.html`

driven software technology [Fav04a]. Software architecture typically comprises the following four views [HNS99]:

1. The *conceptual* view describes the architecture in terms of domain elements (i.e., the functional features of the system).

2. The *module* view describes the decomposition of the software and its organization into layers.

3. The *execution* view is the run-time view of the system. It is the mapping of modules to run-time items, defining the communication among them, and assigning them to physical resources.

4. The *code* view captures how modules and interfaces in the module view are mapped to source files and how they are further organized into directories. This information also affects the build process of the system.

The viewpoint-oriented approach is also used to support design decisions, which are cross cutting and intertwined. During the software maintenance process, the design rules and constraints are often violated [Bos04]. The traditional software-architecture viewpoints span all stages of the software-development process and offer different levels of problem-specific abstractions.

The viewpoints are not limited to software, but also extended to learners and other stakeholders. Zachmann framework [Zac02] provides a matrix of perspectives and aspects without prescribing *how* the enterprise architecture description should look like. However, there is no standard way of using the various viewpoints and diagram types identified in general model-driven development approaches [AS08]. One approach is to have a conceptual model for defining and navigating around different views [AT14].

Problem Statement

Organizing Software Languages and Technologies

The following quote from the website of the International Conference on Software Language Engineering provides an approximate definition of *software language*:

"The term 'software language' comprises all sorts of artificial languages used in software development including general-purpose programming languages, domain-specific languages, modeling and meta-modeling languages, data models, and ontologies. Used in its broadest sense, examples include modeling languages such as UML-based and domain-specific modeling languages, business process modeling languages, and web application modeling languages. The term 'software language,' in contrast to a 'programming language,' also comprises APIs and collections of design patterns that are implicitly defined languages."[11]

This definition emphasizes the complexity of languages used in different TSs and goes far beyond the notion of a programming language. Modern polyglot developers are concerned with the applicability of software languages to a particular domain problem. There still exists a well-known notion that developers "think" in a given software language such that developers are framed within a certain context of software language and technology.

The lack of a classification scheme and the diversity of software languages are two factors that prevent effective communication between communities. Developing such a classification requires a proper abstraction level that authors and users would understand. Because doing so is clearly not a one-person effort, collaboration should be supported. Technical communities often use wikis as collaboration platforms. Most advanced wikis provide some limited classification idioms (e.g., categories of pages). *Wikipedia* is a well-known example. It is a knowledge resource for many software-engineering topics and explicitly declares the lack of a classification scheme for programming languages: "There is no overarching classification scheme for programming languages. A given programming language does not usually have a single ancestor language. Languages commonly arise by combining the elements of several predecessor languages with new

[11] `http://planet-sl.org/sle2011/`

ideas in circulation at the time. Ideas that originate in one language will diffuse throughout a family of related languages, and then leap suddenly across familial gaps to appear in an entirely different family."[12]

Ontology-based Knowledge Management

The etymology of the term *knowledge* breaks it into two elements: (1) *know-* means to have learned from experience and implies having gained an understanding of a subject, and (2) *-ledge* means an organized body of facts or teachings and implies a need for knowledge organization and reuse.[13] In this thesis, we use these two attributes as a definition of knowledge.

Ontology is often considered a means to capture domain knowledge [GPFI+]. It defines domain vocabulary and relationships between terms.[14] There exist many methods and tools to build, maintain, and reuse ontologies. Most of them rely on the deep analysis and understanding the underlying domain. The most significant challenge to conducting research in the field of SLE is the rapid pace of change in the field. SLE is constantly evolving because new languages and new paradigms for using them constantly emerge. For example, JavaScript existed for a long time as a language for client-side programming for the web. However, with the evolution of server-side technologies, JavaScript has successfully entered the server-side context. Today, without server-side JavaScript experience, one cannot be considered a JavaScript expert, whereas that would not have been the case several years ago.

Several approaches should be used to manage knowledge about SLE. Because there is no single domain expert, managing knowledge about SLE needs to rely on contributions from many experts. The most appropriate modeling language(s) should be selected such that the level of abstraction is raised. An ontology specification language(s) might be considered. Documentation of knowledge should include examples that apply to the technologies of interest. Documentation should be structured so that its diverse audience (e.g., computer science students, professional developers, and SLE researchers) can understand and use it.

[12] `http://en.wikipedia.org/wiki/Programming_language`
[13] `http://www.etymonline.com/index.php?term=knowledge`
[14] We do not analyze philosophical aspects of the ontology in this thesis.

Knowledge Integration

With the rise of online communities, more knowledge about software engineering becomes available as so-called user-generated content. Examples include *Wikipedia*, more domain-specific wikis, possibly open and online textbooks, forums like StackOverflow, the Apple Knowledge Base, and support forums by Microsoft. The organizational- and process-management research community recognized the value of and potential for integrating organizational knowledge two decades ago. In these communities, there is agreement that "the primary role of the firm, and the essence of organizational capability, is the integration of knowledge" [Gra96]. In that context, the definition of knowledge integration is "an ongoing collective process of constructing, articulating and redefining shared beliefs through the social interaction of organizational members" [HN03].

As discussed in section 1.1, software developers extensively participate in communities that are outside their organizational boundaries. Community resources (such as wikis, forums, and open online textbooks) are important knowledge resources, each using a particular vocabulary and knowledge model. To understand how developers use such resources is an open research problem [WLJ13]. We believe the knowledge- and education-centric view on such resources is also of potential value. Such a view is concerned with aggregating, organizing, and maintaining knowledge in the programming domain to be useful specifically for learning [LSV14].

Knowledge integration, in fact, can also contribute to ontology development and knowledge management in general. Assuming there is a process for creating topic models and vocabularies on top of the integrated online resources, the process could support an ontology for software languages and technologies. The resulting vocabularies and relationships between the resources help teach how to program and how to document programs.

Discovery Learning

Discovery learning is a method where students are free to work in a learning environment with little or no guidance [May04]. Maintaining knowledge via heterogeneous community resources

and software repository goes beyond e-learning delivery models. Individuals can access these resources any time on their own. Certain organization and guidance should be embedded in the content, as pure discovery learning rarely efficient [May04]. Additional structure required to facilitate class-room instructor-led trainings (so-called synchronous model). This needs to be efficiently combined with the self-education capabilities, where the learners are exposed to the broad set of contributions in various forms (source code implementations, textual content and ontology).

Understanding Modern Software Products

Understanding modern software products is difficult. The discipline of software architecture is about mastering the complexity of software using the view that the essential entities and relationships of software constitute the model (i.e., the architecture) of a software product. Different kinds of models serve different points of view (say, different stakeholders of a software project). Most existing approaches to software architecture focus on logical architecture, where entities of interest are coarse grained (e.g., modules, interfaces, classes, features, or aspects) and relationships are imports, calls, and other dependencies. Established approaches also exist for dealing with physical architecture (i.e., build systems), such as Ant, Maven, and other implicit approaches that are integrated in IDEs, such as Eclipse. In the case of physical architecture, entities of interest are coarse-grained entities (such as files), and relationships are build dependencies and invocations of tools (such as compilers and linkers) [FLV12].

The MDE discipline tries to raise the level of abstraction at which software engineers write code. It aims to improve developers' short-term productivity by increasing the value of the functionality in software artifacts, and it aims to improve long-term productivity by preventing software artifacts from becoming obsolete too quickly [AK03]. But MDE has several practical limitations [LV14]:

- Not all software projects are parts of model management systems.
- Not all software projects are within Technological Spaces that are sufficiently covered by MDE.

- Metamodels or metamodel-like artifacts, such as schemas, are often unavailable or of limited relevance outside clean-room MDE. That is, people often refer to languages instead of metamodels (i.e., to conceptual entities rather than artifacts).

Developers use several practical approaches to understand all non-trivial relationships among the elements of a software project. Test-driven development (TDD) is one frequently used methodology that helps developers understand the design of existing software and develop new software. According to Agile Alliance, TDD is "the craft of producing automated tests for production code, and using that process to drive design and programming."[15] Similar low-level approaches can rely on build systems or scripts that check project integrity. A build system might contain build-related data as well as various filtering techniques to define views of the build architecture [ATSM07].

The spectrum of approaches is diverse, ranging from the low-level, source-code–based to different kinds of software architecture. MDE brings its own stack of tools to manage the complexity of software projects across various dimensions. But we are not aware of an approach that treats languages (as opposed to their descriptions via metamodels or grammars) as first-class citizens that provide a linguistic view of software projects. Such a viewpoint can be useful as a cognitive model for the benefit of software language engineers and possibly software engineers.

Research Method

Traditional scientific methods use the hypothetico-deductive model of research [GS09], which can be roughly summarized as the following sequence of steps:

1. Formulate a question.
2. Hypothesize.
3. Study.

[15] `http://agilealliance.org/programs/roadmaps/Roadmap/tdd/tdd_index.htm`

4. Analyze.

5. Evaluate.

The scope of this thesis includes several research areas (see figure 1.2 for a summary) and targets a broad audience of software engineering researchers and educators with different motivations. The research questions are the following:

- Can the notion of software chrestomathy be adopted in software-engineering research and education?
- How can a software chrestomathy be instantiated as a Research 2.0 platform?
- How can a software chrestomathy be made useful in understanding software languages and technologies?
- How can a software chrestomathy be made useful in software-engineering education?

To answer these questions while coping with their complexity, we adopt the design research methodology: "Design science refers to an explicitly organized, rational and wholly systematic approach to design; not just the utilization of scientific knowledge of artifacts, but design being in some sense a scientific activity itself" [Cro07]. Our goal is to obtain evidence that answers the research questions as unambiguously as possible. Doing design research requires the creation of an innovative, purposeful artifact for a special problem domain [PTRC07]. For this research project, the artifact is a *101companies* software chrestomathy and the underlying infrastructure; We evaluate it using a case-study design under the action-research paradigm to show its applicability. Chapter 3 contains an in-depth discussion (section 4.3) of how this artifact would answer the research questions stated in this thesis. Figure 1.1 illustrates the steps as they apply to this work.

To identify the scope and requirements of this research, we follow a phased system-analysis method. As it further follows from the structure of the thesis (section 1.6), chapter 3 and chapter 4 mainly correspond to the following phases: scope definition, problem analysis, and

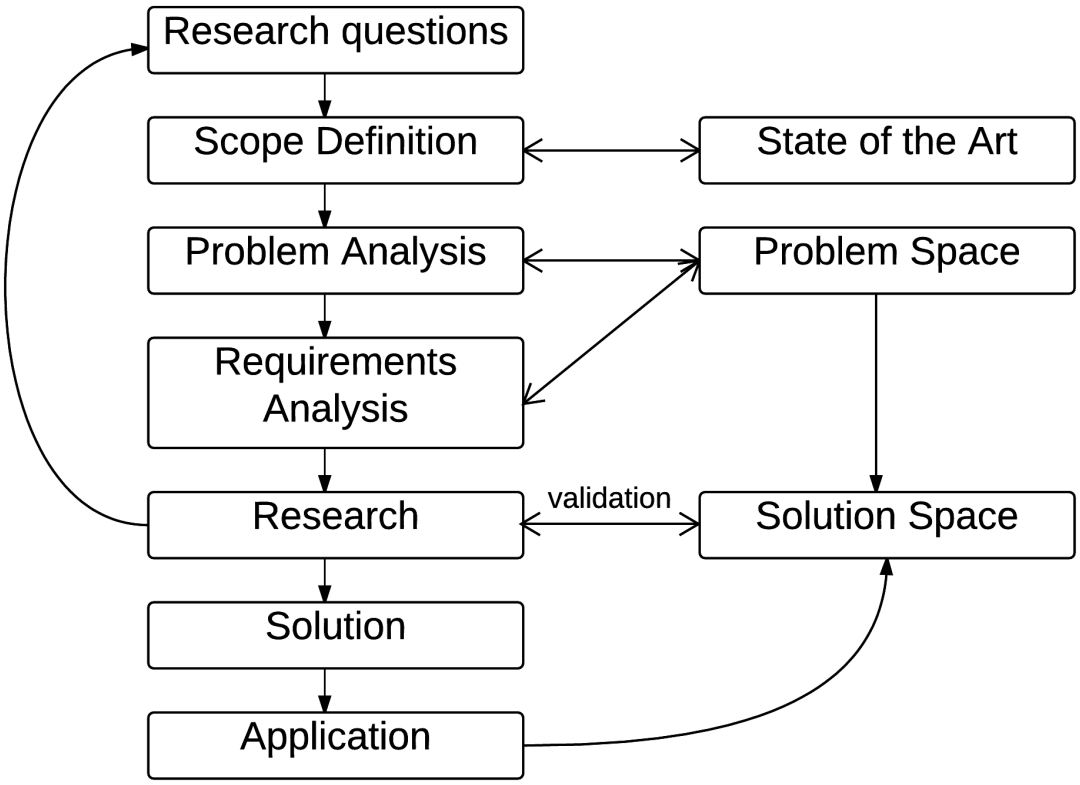


Figure 1.1: Research method

requirements analysis. After the problem is identified in chapter 3, the rest of the research is organized in such a way that each section establishes a link between a problem space (chapter 4) and a solution space – that is, each section has self-contained objectives and contributions. In that manner, the validation also becomes an integral part of the thesis. The applications of the solution conclude the study – that is, we show the usefulness of the *101companies* software chrestomathy as a solid artifact in several research contexts.

Contributions

The original contributions of this thesis are the following:

- development and validation of a software chrestomathy as an integral part of the design research methodology;
- ontology-based knowledge management for software chrestomathy;

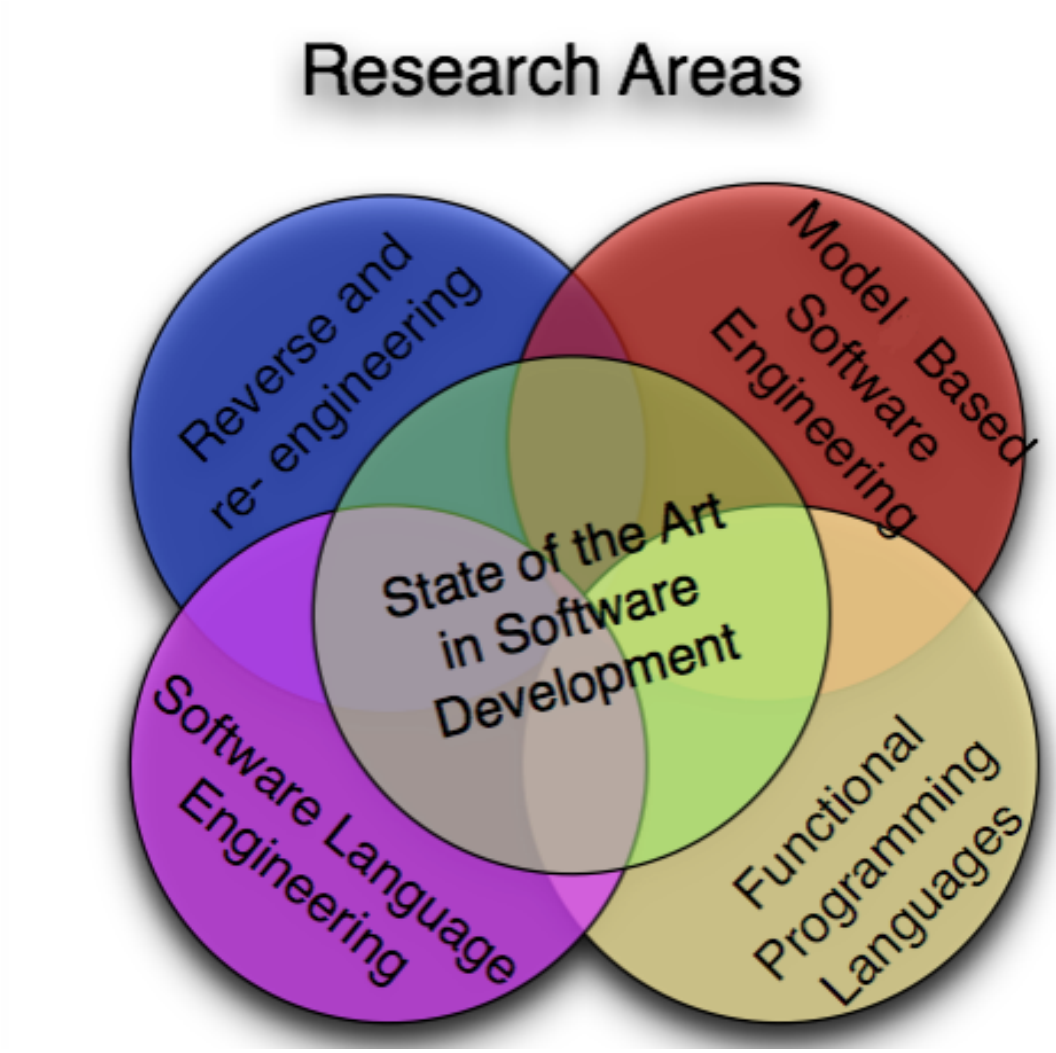


Figure 1.2: Research areas of the thesis

- validation of the various aspects of the interpretable linguistic architecture.

Software Chrestomathy

We adopted the linguistic notion of chrestomathy for the SLE context; a software chrestomathy is a collection of software systems meant to be useful in learning about or gaining insight into software languages, software technologies, software concepts, programming, and software engineering [FLSV12, Lae13]. We aimed to develop a free, structured, wiki-style knowledge resource that included an open-source repository for different stakeholders with interests in software technologies, software languages, and TSs (notably teachers, software developers, software technologists, ontologists, and learners of software engineering and software languages). Based on that objective, we designed and implemented the *101companies* platform (also called

101project or just *101*). The software chrestomathy we developed serves a Research 2.0 platform for software-engineering research and education.

The core of *101companies* combines a semantic wiki and confederated open-source repositories. We developed an approach to establish links between source code and documentation using a rule-based language that invokes file processors for validation, fact extraction, fragment location, and other functions. It can handle many languages and technologies using techniques that can be language specific or language agnostic. Additionally, the refined *101companies* chrestomathy as well as the underlying techniques and derived information resources are available in the open-source ecosystem. As such, it supports Research 2.0, in particular, Open Science [Nie11] and Linked Data [HB11a]. Resources on the platform adhere to Linked Data principles and are available to stakeholders in the way that is most useful to them. The relevant formats and the underlying ontology are accessible and documented; the platform generally supports both programmatic and interactive access. In chapter 9, we argue the case for the applicability of these resources for reverse engineering and data mining. The chrestomathy-based course for computer-science students is another case of the evaluation.

On the knowledge-management side, we implemented a textbook-driven knowledge-integration mechanism that establishes a consolidated vocabulary for chrestomathy, wikis, and textbooks. The degree of knowledge integration in terms of vocabulary usage can be monitored. We further enriched all entities (e.g., implemented features, languages, technologies, concepts, and external resources) with metadata for classifying and identifying relationships among entities. To prove the usefulnesses of such knowledge organization in classical undergraduate education, an introductory functional programming course was layered directly on top of *101companies*. Validation boils down to the argument that the available content (semantic wiki, interlinked source code, and interlinked external resources) intrinsically provides the foundation for a viable course. We further discuss the educational perspective of *101* in chapter 9.

101companies' role model is a *Wikipedia* for the community of polyglot developers and TS travelers. We expect *101companies* to serve as Research 2.0 infrastructure.

Technology Modeling

In the context of the *101companies* project, we developed a megamodeling approach for technology modeling supported by the MegaL language and associated tools for editing and exploring megamodels—a new form of modeling for software technologies that addresses the linguistic architecture of software products. Megamodeling has MDE foundations [BJV04a]; however, our approach to modeling software technologies was not tailored to MDE. Our approach addresses the *linguistic architecture* of software products: the relationships between software artifacts (e.g., files), software languages (e.g., programming languages), and software technologies (e.g., code generators). This new type of software architecture complements other, more established dimensions: logical architecture (the subject of classical software architecture); specific paradigms (such as component-, feature-, or aspect-oriented software development); and physical architecture (typically concerned with building, packaging, and deploying software and, hence, with entities such as files and servers).

The value of our megamodels is a cognitive one. They facilitate understanding technologies and their usage. We strongly improve such cognitive value by enabling a form of linked megamodels such that entities and relationships are linked to resources (e.g., in the *101companies* repository) so that megamodels can be explored and validated. This process is instantiated in a form of interpretation: resolution of megamodel elements to resources (e.g., system artifacts) and evaluation of relationships subject to designated programs (tools) available through a plug-in framework. Interpretation reduces concerns about the adequacy and meaning of megamodels because it helps to apply the megamodels to actual systems. We leverage Linked Data principles for the results of resolution (e.g., links to GitHub repositories or DBpedia resources). We implemented MegaL with an object-oriented (OO) framework with dynamically loaded plug-ins. The implementation is also supported by an executable specification of interpreted megamodels that are formalized in a deductive system. Without this enhancement, megamodeling does not provide sufficient validated insight into actual systems.

Through this contribution, we have equipped the megamodeling notion for the linguistic architecture of software systems with a language mechanism for resolving entities, capturing

traceability between them, and evaluating relationships.

Basic Terminology

Chrestomathy is one of the central terms in this work. The word is formed from the Greek words *chresto*, "useful," and *mathein*, "to learn." In philology and linguistics, chrestomathy refers to a collection of sample texts (literary passages), usually in one language, possibly from different authors, designed to be useful in learning a language, specifically its grammar. For instance and quoting from [Läm13], the *Coptic Gnostic Chrestomathy* collects texts in the Coptic language; these texts were systematically edited to include annotations for grammatical analysis, such as relationships between prepositions, verbs, and nouns [Lay04].

E-science is a broadly interpreted term, mainly concerned with the application of IT throughout the scientific process [Boh13]. E-science "promotes innovation in collaborative, computationally- or data-intensive research across all disciplines, throughout the research lifecycle."[16]

Linked Data is "a set of best practices for publishing structured data on the Web." It is based on five principles [BCH07, HB11b]:

1. Use URIs as names for "things."
2. Use HTTP URIs so that people can look up names.
3. Provide useful information in the HTTP response.
4. Include links to other "things" to make them discoverable.
5. Use standards for response formats and query languages (e.g., RDF and SPARQL).

Open Data is a concept implying that a given set of data is available to everyone without any copyright restrictions or other means of protection [ABK+07a].

[16] `https://escience-conference.org/`

Open Science was defined by Michael Nielsen in his TED talk as "the idea that scientific knowledge of all kinds should be openly shared as early as is practical in the discovery process."[17] The six principles of Open Science are open methodology, open source, open data, open access, open peer review, and open educational resources [KLRB11].

Program or *programming chrestomathy*[18] has been a concept in the wild in the programming community for several years [Läm13]. A program chrestomathy is a collection of sample programs in one or more programming languages designed to be useful in learning about programming and programming languages. Such a chrestomathy may focus on specific language aspects, such as comparison of programming style, expressiveness, and applicable programming techniques in one language or across several languages (e.g., [Rue01, Wei13, Man13, RJJ+08]). Importantly, the collected programs are expected to implement certain features (such as tasks or requirements) as prescribed by the chrestomathy.

Reproducible research is research that can be redone by researchers other than those who conducted the original research. In the context of SLE, research is mainly made reproducible by making the code and data associated with research publications available.[19]

Science 2.0 or *Research 2.0* is a version of e-science that includes a shift from publishing final results by well-defined collaborative groups toward a more open approach that includes publicly sharing raw data, preliminary experimental results, and related information [FH09]. E-science motivates the development of data-intensive scientific applications based on semantic methodologies and technologies. It has no particular connection to computer science, and we use it more as an umbrella term, emphasizing the following characteristics of information sharing and collaboration: e-research, networked science, e-science, computational science, big science, and linked science. Research 2.0 contrasts with the traditional ways of doing research and involves following modern Web 2.0 techniques oriented toward collaboration. This thesis contributes to Research 2.0 efforts in computer science.

[17] `http://www.ted.com/talks/michael_nielsen_open_science_now`

[18] `http://c2.com/cgi/wiki?ProgrammingChrestomathy`

[19] Reproducible Research (`http://reproducibleresearch.net/`) is a portal that collects the bibliography of reproducible papers.

Software chrestomathy is a program chrestomathy that has also collected (build-able, runnable, modularized) systems rather than just programs in the sense of single source-code units. When moving from programs to systems, a number of software-engineering–related aspects may be covered (e.g., build management, testing, documentation, and modeling features to be implemented by the systems) [Läm13].

A *technical* or *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 [*sic*] 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" [KBA02b]. This concept is one of the motivating items for the work presented in the thesis.

The *101companies project* is a community project aimed at collecting systems that implement certain features of a small, hypothetical information system—the *101system* for human resource management—in different ways [FLSV12]. These systems are referred to as *contributions*; they are maintained in a confederated GitHub-based repository, the *101repo*.[20] All contributions are documented on a wiki for the project, – the *101wiki*.[21]

Structure of the Thesis

In chapter 3, "State of the Art," we perform a systematic and critical analysis of the state of the art in software engineering, as it concernes the scope of our research on a software chrestomathy. In chapter 4, "Problem Space," we set up the problem space and identify the scope of challenges associated with an open-research software chrestomathy. In chapter 5, "*101companies* Software Chrestomathy," we define key concepts associated with software chrestomathy. We also describe *101companies* and illustrate its usefulness for different shareholders. In chapter 6, "Chrestomathic Knowledge Integration," we describe the knowledge-integration framework that we used to create a consolidated vocabulary for making linkages among chrestomathy, wikis,

[20] `https://github.com/101companies/101repo`
[21] `http://101companies.org`

and online textbooks. In chapter 7, "A Chrestomathic Ontology," we discuss the ontology-based approach to knowledge organization in the *101companies* software chrestomathy. We also discuss principles for modeling knowledge about software technologies and languages. In chapter 8, "Technology Modeling," we describe a megamodeling approach to linguistic architecture. Chapter 9, "Evaluation of the *101companies* Software Chrestomathy," demonstrates the validity of our research. We analyze *101companies* using action research to show its applicability in practice. The final chapter concludes the thesis and discusses the broad potential for further improvement of the *101companies* software chrestomathy and its applicability.

Related Publications

This section lists publications that I co-authored and contributed to as part of the research for this thesis.

[FLSV12] Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz and **Andrei Varanovich**. 101companies: A community project on software technologies and software languages. In TOOLS (50), pages 58–74, 2012.

[FLL+12b] Jean-Marie Favre, Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz and **Andrei Varanovich**. Linking documentation and source code in a software chrestomathy. In WCRE, pages 335–344, 2012.

[FLV12] Jean-Marie Favre, Ralf Lämmel and **Andrei Varanovich**. Modeling the linguistic architecture of software products. In MoDELS, pages 151–167, 2012.

[LMV13] Ralf Lämmel, Dominik Mosen and **Andrei Varanovich**. Method and tool support for classifying software languages with wikipedia. In SLE, pages 249–259, 2013.

[LLSV14b] Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz and **Andrei Varanovich**. Comparison of Feature Implementations across Languages, Technologies, and Styles. In Proc. of IEEE CSMR-WCRE 2014. IEEE, 2014

[LSV14] Ralf Lämmel, Thomas Schmorleiz and **Andrei Varanovich**. The 101haskell Chrestomathy

– A Whole Bunch of Learnable Lambdas. In Postproceedings of IFL 2013, 2014.

[LV14] Ralf Lämmel and **Andrei Varanovich**. Interpretation of Linguistic Architecture. In European Conference on Modelling Foundations and Applications (pp. 67-82). Springer International Publishing.

[LVL+14] Ralf Lämmel, **Andrei Varanovich**, Martin Leinberger, Thomas Schmorleiz and Jean-Marie Favre. Declarative Software Development (Distilled Tutorial). In Proc. of PPDP 2014.

Below we quote a number of the reviews, as a sign of appreciation for the novelty of the research has been represented in our publications.

- TOOLS 2012: "Being an unconventional paper, it is written in an unconventional manner. I would have not written the paper in this way for introducing a project like that. Anyway, I definitely prefer to respect the way it is written because it reflects an original style and an original attitude. Both things are needed to science and technology to progress. Some of the ideas are arguable, and many projects like this have failed in the past due to lack of support by the community or, alternatively, due to excessive and too tight control by it. Authors should stick to their ideas and should try to build the project in their way."

- WCRE 2012: "Points in favour: 101companies project: http://101companies.org; this is certainly a worthwhile repository of the techniques and results by the parsing (programming language) and reverse engineering communities; a very useful teaching tool; it would have been useful to have had this resource for the Y2K conversion process."

- IFL 2013: "Both the paper and its topic are far from the usual kind of research papers that we normally see. That does not mean that the paper without merit. In fact I liked this paper quite a bit. The paper made me search out a few of the citations to get a better understanding of the larger context. Which further whetted my appetite for what the paper contained."

The publications mentioned above contribute to the significant part of the thesis with the

substantial revisions and additions. In particular, with the relation to the scope and the consistency of the research method applied in this work. The thesis is written in the style of a monograph, thus, none of the publications have one-to-one correspondence to the chapter. The degree of contribution is emphasized in the footnotes at the beginning of each chapter.

Chapter 2

Background

This chapter provides the background on two key notions developed in this thesis.

Program Chrestomathies

There is very little reusability in research generally (see figure 2.1 from [Smi11]). In fields such as physics and chemistry, experimental papers provide a lot of technical details to enable others to reproduce experiments and validate findings. In software engineering research, many tools are built to support publications. However, there are not many means to distribute the tools together with the papers, so reusability remains in general low.

Using open source is an easy yet efficient way to distribute such tools [MVDBK14a], though not the only one. The Executable Paper Grand Challenge is a contest created to improve the way scientific information is communicated and used.[1] Three winners of this contest illustrate the variety of approaches to enable the reusability of publications. Nowakowski et al. introduce the concept of *executable papers*, which are static papers supplemented with interactivity [NCH+11]. SHARE provides an operation management for virtual machines that can be cited from research papers [GM11]. Such virtual machines contain all the necessary software and data to reproduce results presented in the paper, making the research papers fully reproducible. Verifiable Result

This chapter is an original part of the thesis based on the analysis of the existing research work.

[1] `http://www.executablepapers.com/`

Identifier (VRI) [GD11], once included in the publication, "can be used by any reader with a web browser to locate, browse and, where appropriate, re-execute the computation that produced the result."

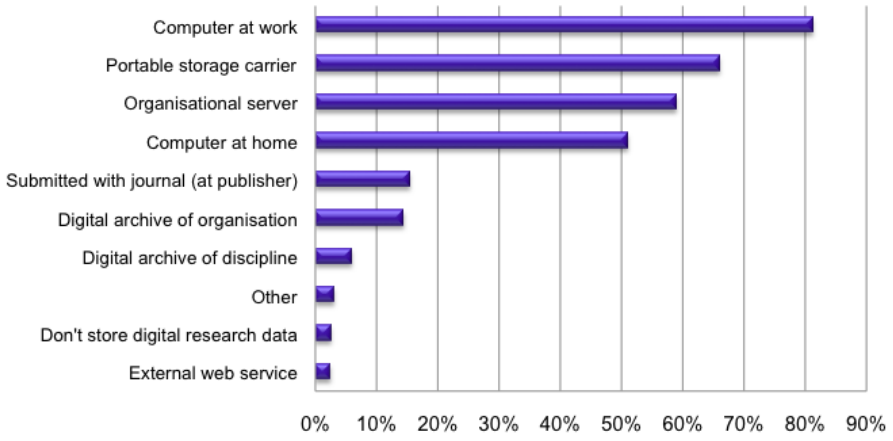


Figure 2.1: Where do you currently store your research data? (researchers/multiple answers, N=1202)

In software engineering, three well-known cases support reusability:

1. Natural open source and social software ecosystems. These are collaborative-oriented environments, with the minimal structure and maximum flexibility oriented on collaborative software creation.

2. Program chrestomathies. Their primary focus on learning makes them useful in different contexts. However, lack of standards and reusability leads to proprietary vendor or task specific instances.

3. Linked data. The most structuraly and technicaly sophisticated approach which is often enriched with the ontology and mostly focuses on the data to be exposed. Has limited capabilities, mainly inspired by the underlying technologies. As a result, a lot of useful data is not explored.

Program chrestomathies are collections of program examples, possibly in different languages, which are used to demonstrate the specifics of programming languages, their implementations, paradigms, and platforms. In the simplest case, a chrestomathy may contain the "Hello World!"

example. In more advanced cases, chrestomathies demonstrate stacks or even product portfolios by vendors (e.g., see chrestomathies for Microsoft[2] or Oracle/Sun.)[3] Table 2.1 shows a concise comparison of programming chrestomathies.

Suites of performance benchmarks account for a specific category of programming chrestomathies. Such suites consist of source code from programs that is suitable for challenging the performance of language implementations systematically. The results can be compared across different implementations of a language or across different languages. For instance, the Computer Language Benchmarks Game (CLBG) is a widely used repository of such comparisons across a wide range of programming languages.[4]

Table 2.1: Comparison of programming chrestomathies

Aspect	**Java Pet Store**[a]	**99 Bottles of Beer**[b]	**Rosetta Code**[c]	**CLBG**[d]
Focus	Java platform	Single Task	Tasks	Algorithms
Scope of comparison	-	Languages	Languages	Languages
Technological spaces	Limited	Ignored	Limited	Ignored
Ontology-driven	No	No	No	No
Classroom-tested	Unknown	No	No	No
Source code	Zip archive	Website	Website	Website

[a] `http://java.sun.com/developer/releases/petstore`

[b] `http://www.99-bottles-of-beer.net/`

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

[d] `http://shootout.alioth.debian.org/`

As a conslusion which followed from the comparison, we anticipate the need for a collection of related software products (i.e., possibly small, actually running systems) using various combinations of software languages and technologies, for the purpose of demonstrating languages and technologies as well as software engineering concepts such as architecture, modeling, deployment, documentation, testing, and reverse engineering. The *101companies* software chrestomathy introduced in this thesis, is meant to provide another useful knowledge resource to the software development and engineering community.

[2] `http://archive.msdn.microsoft.com/ContosoAutoOBA`
[3] `https://wikis.oracle.com/display/code/Home`
[4] `http://shootout.alioth.debian.org/`

In [ON08], two open source contexts are distinguished: software and content, and we analyze to what extent the entry barriers for contributors are different in the two contexts. While software contribution requires a certain threshold of expertise in order to pass the review process, content contribution has virtually no entry barriers other than basic computer literacy. To some extent, both roles are separated by the proposed contributor roles in the *101companies* project. However, ideally, a contributor of an implementation is also supposed to contribute content (so that the implementation is strongly documented).

Linguistic Architecture

Several abstraction levels and views are used in software development. For instance, structural, behavioral, component-based, and architectural models are commonly used. Most models use different kinds of artifacts. However, there are types of elements (namely, languages and technologies) that do not have a physical representation. Nevertheless, they play a key role in software engineering. For example, when modeling the programming language Java, the whole Java ecosystem is often considered. A person who is a *Java developer* implicitly has broad knowledge of Java technologies, often accompanied by a certain way of thinking (e.g., Java-thinking versus Prolog- or Haskell-thinking). Linguistic architecture considers such entities and their relations.

Instances of Linguistic Architecture

In this section we analyze several domains of software engineering. We seek traces of entities and relations between them, and those that are the main concern in each specific domain (community). Communities can vary in how explicitly they treat the artifacts in a model-driven way. We highlight the discovered notions in *italic*.

In Software Maintenance

Karus and Gall study the coevolution of languages and other project artifacts, such as documentation, binaries, and graphics files [KG11]. The ultimate goal of the research is to find a

coevolution profile of languages, that is, pairs of languages that are coevolving in open-source software (OSS) projects. The results show that beside programming languages like Java or C, markup languages (e.g., XML, Web Services Description Language [WSDL]) and transformation languages (e.g., extensible stylesheet language [XSLT]) constitute such coevolution profiles.

Feilkas, Ratiu, and Jurgens are concerned with the loss of architectural knowledge, namely that documentation and source code are not kept in sync [FRJ09]. They analyze the degree of conformity between documentation and source code. For architecture *conformance analysis*, they represent a software system as a set of types with a set of dependencies between them. Such representation allows applying the *conformance checking*. The results reveal that more structured documentation in a machine-readable form and automatic *dependency analysis* create bigger architecture-awareness in the development team.

Eichberg et al. present an approach to expressing *constraints on structural dependencies* between elements of software [EKKM08]. Such elements consist of architectural-, design-, and implementation-level decisions. Further, there are design-level constraints, such as stating that only factory classes can access constructors of product classes when the factory pattern is employed. Finally, there are implementation-level constraints, such as stating that the fields of a certain class can only be accessed via getter and setter methods of the same class. The approach uses *declarative queries to group source elements* across programming-language module boundaries into overlapping ensembles. The paper presents ensembles, which are continuously enforced as the software evolves.

Harrison et al. write that the concern manipulation environment (CME) provides a way to represent concerns across different types of software-engineering artifacts in the context of aspect-oriented software development [HOJT05]. One of the mechanisms to assign concerns is a fixed set of *queries over relations, such as extends, implements, refersTo, or referredToBy.* The querying capabilities of a CME rely on a predefined set of predicates.

In the context of software maintenance, it is useful to know how developers *understand a program* and fix bugs. Soh et al. *represent program entities as a resource* (XML, MANIFEST.MF,

properties, HTML files, etc.) or composite parts of a Java program (i.e., project, package, file, class, attribute, or method) [SKG+13]. By surfacing such program entities from interaction history, the authors were able to identify exploration strategies for developers.

In Antipattern Detection

Palma et al. analyze static and dynamic properties of REST services to detect REST antipatterns, such as forgetting hypermedia (see [Til08]), where *the links between web resources* are not present in the response body or header [PDMG14]. HTTP responses are considered *structural entities*, and certain queries are executed on top of them for pattern and antipattern detection in an automated manner.

Palma et al. apply a similar method to detect service-oriented architecture (SOA) antipatterns in web services [PMTG14]. The relevant properties of web-service–specific antipatterns are encoded in DSL. For instance, names are analyzed to identify Remote Procedure Call (RPC)-like behavior, exposing create, read, update, and delete (CRUD)-type operations (e.g, read_ and create_), which is are CRUD interface antipatterns.

A number of similar approaches exist for OO antipatterns, where *structural program elements (e.g., classes) are queried* [KKS+11, SMSB11]. Settas et al. go a step further and put antipattern definitions into *semantic relationships* in the form of the OWL ontology SPARSE [SMSB11]. SPARSE is represented through 31 OWL ontology antipattern instances. Linskey and Prud'hommeaux analyze field-level access and mutation for an object-relational (O/R) mapper to accelerate data access through caching [LP07].

In Software- and Language-Modeling Foundations

Wang et al. attempt to establish an identity criterion for software-related artifacts through a requirements-engineering perspective [WGGM14]. A notion of the *program as an artifact* with internal behavior is complemented by notions of the software system as an interface to the environment and the software product that determines the specific effect in the environment based on certain conditions. A program, however, cannot be identified either with a code, a

process, or an algorithm. Thus, the need for the technical artifact is motivated. The *identity of the artifact* is further connected to its proper function, that is, the function the artifact is intended to perform [Bak04].

In the context of the evolution of languages, Meyers and Vangheluwe study the two main aspects of a language model: its syntax (how it is represented) and its semantics (what it means) [MV11]. An abstract language model is represented by multiple concrete syntaxes. The parsing function maps between concrete and abstract syntaxes. A metamodel describes the abstract syntax and static semantics of a language. Dynamic semantics are not covered by the metamodel. The abstract syntax of a model can be represented as a graph with nodes that are *elements of the language* and edges that are relations between these elements and elements of the language. *Instance models of the language* are said to conform to the metamodel of the language. Kühne refers to this conformance as linguistic *instance of* [Küh06a].

In Traceability Recovery

Traceability is a necessary system characteristic for software management, software evolution, and validation [Nas05]. A survey on tracing approaches in traditional software engineering and the elaboration of a traceability taxonomy is presented in [VKP02]. *Entities and relations* constitute a conceptual trace model [RJ01]. We consider such a model from two perspectives. The first perspective is a focus of the traceability-recovery field; it is actual recoverability in which the accuracy of extracted dependencies plays the critical role.

In Egyed's study [Egy03], trace dependencies characterize the relationships between the following software elements: test scenarios, data flow, use case, class diagrams, and implementation classes. Different types of trace dependencies provide a set of high-level relationships between software artifacts. The approach relies on an observable and executable *software system, a list of development artifacts*, and scenarios describing test cases or usage scenarios for those development artifacts. Analyzing traceability between classes and requirements in an OO system is a promising way of improving the accuracy (precision and recall) of traditional information retrieval (IR) approaches [AGA13].

The second perspective of the trace model is concerned with the actual modeling of *traceability links*, which make the model even more strongly connected to the notion of linguistic architecture. The research approaches using this perspective still take into consideration a somewhat basic model of the program, mainly focusing on the novel methods of information retrieval [ACC00]. However, the application-domain knowledge that programmers express in identifiers [ACDLM99] can also represent the essential properties of an OO system [FA98]. A separate model for traceability links themselves also facilitates the loose coupling between models and traceability information [GG07].

Concept location is another instance of exploring linguistic architecture [MSRM04a]. In fact, software-engineering concepts in a broader sense are central in the chrestomathic ontology presented in the chapter 7 of this thesis. The concepts themselves are worth analyzing as soon as their types are also a concern in our analysis of linguistic architecture. Wilde et al. represent a program as a set of "program-components–implement-functionality pairs, where a test case establishes a relation between a program component and a functionality" [WGGS92]. Traces of test-case execution are used to identify *elements of the source code* that implement that feature. Barbero, Jouault, and Bézivin use domain-specific conceptual entities, such as bug reports (which contain definitions of several bugs, define relationships between them, and connect them to code patches) or Java projects (models with files, dependencies, and bug trackers) [BJB08].

In Model-driven Engineering

MDE has provided a direct inspiration for the notion of linguistic architecture, which is not limited to MDE-only, however. Bézivin uses megamodels to illustrate the relations between a model, a metamodel, and a meta-metamodel [Béz04]. Other entities are not explicitly captured. Two relationships (representedBy and conformantTo) are used; they correspond to instanceOf and inherits in the OO paradigm. Favre considers languages in a set-theoretic sense next to models. Additionally, Favre considers programs, functions, interpreters, and recognizers [Fav04b]. Favre and Nguyen propose a notation for megamodels supported by several examples [FN04a]. In their notation, everything is a system (a real system, an abstract system, a

model, a metamodel, and so forth) or a set (in practical terms, a language). Such entities are bound by five kinds of well-defined relationships: δ (DecomposedIn), μ (RepresentationOf), ε (ElementOf), χ (ConformsTo), and τ (IsTransformedIn).

Klint, Lämmel, and Verhoef use conceptual entities (such as schemas, grammars, and specifications), functions (transformation and processor), technology (tooling), and processes (coding, generation, customization, recovery, evolution, implementation) to define a grammarware TS [KLV05].

Chapter 3

State of the Art

In Jim Gray's last talk to the Computer Science and Telecommunications Board on January 11, 2007, he described his vision for the fourth paradigm of scientific research, specifically for data-driven research [HTT]. Data-driven research focuses on tools and methods for data capture, curation, and analysis and for communication and publication infrastructure. In his talk, he uses a data iceberg analogy: "Then comes the publication of the results of your research, and the published literature is just the tip of the data iceberg. By this, I mean that people collect a lot of data and then reduce this down to some number of column inches in *Science* or *Nature*—or 10 pages if it is a computer science person writing. So what I mean by 'data iceberg' is that there is a lot of data that is collected but not curated or published in any systematic way."

Gray's iceberg analogy is also applicable to the research on TSs. *Reproducibility*—the ability to repeat calculations for analyzing data and obtaining computational results [Mes10]—is a fundamental aspect of research. Consider an example in software engineering. Academic tools are essential parts of research, but many tools remain prototypes and hardly become products. One reason is that citation count is the key criterion for researchers' output [MvdBK14b]. For instance, it has become increasingly complex to reproduce methods used in software analysis, and often researchers only reproduce old methods to compare them with new methods. There are positive examples of already available tools that promote collaboration-oriented research

This chapter is an original part of the thesis based on the analysis of the existing research work.

and enable researchers to set up frameworks for comparing items of interest, including tools, benchmarks, and user studies. Such collaboration may advance reproducibility in software engineering research, which in turn may lead to tool platforms where researchers rely on existing tools to build new prototypes. The workshop series at the International Workshop on Advanced/Academic Software Development Tools and Techniques[1] and Elsevier's Science of Computer Programming special issues on Experimental Software and Toolkits (EST) promote this vision.

Reproducible research not only helps "to reproduce figures in the revisions of a paper, [and] to create earlier results again in a later stage of our research"[2] but also serves as a basis for more advanced uses of research data. So reproducibility is closely related to reusability, which is rarely supported by publications. In the software analysis context, the MSR conference has two special types of publications that support reusability. Data papers "should describe data sets curated by their authors and made available to others. They should address the following: description of the data, including its source; methodology used to gather it; description of the schema used to store it, and any limitations and/or challenges of this data." And in a mining challenge, researchers demonstrate the usefulness of their mining tools on preselected software repositories and summarize their findings in a challenge report.[3]

Another dimension of Gray's iceberg is the stakeholder. There are many stakeholders with interests in software technologies, software languages, and TSs, notably software developers, software technologists, ontologists, teachers, and students of software engineering and software languages. And their interests often diverge. As a result, it is difficult to provide clean semantics for the data (what the data means), clearly represent the data, and provide technical means to consume the data using open-web standards. Even within the same use case (e.g., education), diverse stakeholders, including university students and IT professionals, require different levels of insight on the same topics (e.g., learning a new programming language.)

In this work, we are concerned with designing, implementing, and evaluating a software chrestomathy

[1] `http://wasdett.wikispaces.com/`
[2] `http://reproducibleresearch.net/`
[3] `http://2014.msrconf.org/`

as a basis for a knowledge-driven research infrastructure. We define the scope of the research on software chrestomathy and evaluate the state of the art using the following: an analysis of three instances of reusability in software engineering; a discussion of TSs and different means for analyzing and understanding them; the role of ontologies in software engineering; an analysis of linguistic architecture—a concept introduced in this thesis—in different fields of software engineering and its role in technology modeling; and a discussion about issues of knowledge engineering for programming education.

Technical Spaces and Polyglotism

In the communities of modeling, metamodeling, and software languages, the concept of TSs helps in identifying and communicating commonalities and differences for grammarware, XML-ware, modelware, objectware, and tupleware [KBA02a, DGD06]. An open question remains: what technical spaces and programming technologies should be leveraged for a given assignment? There are existing benchmarks in other areas of programming languages, databases, and software engineering, which are usually meant to evaluate technologies and approaches in a more specific domain (e.g., a generic programming benchmark for Haskell libraries [RJJ+08] and STBenchmark, a benchmark for mapping systems for schemas, such as relational-database schemas [ATV08]). A more structural, model-based level of abstraction is promoted by MDE. The top three reasons for using models in MDE are team communication, understanding a problem at an abstract level, and capturing and documenting designs [HWRK11]. Ko, Meyers, and Aung found six learning barriers which users of programming interfaces would encounter: the inherent difficulty of the problem (design), finding available interfaces (selection), combining available interfaces (coordination), using available interfaces (use), evaluating the external behavior of the program that does not match expectations (understanding), and acquiring information about a program's internal behavior (information) [KMA04]. They also found that using examples helped programmers overcome selection and coordination barriers.

Polyglotism is gaining researchers' attention mainly due to practical demand (i.e., the need for new tools for and approaches to multilanguage comprehension in software projects). A

problem of references across artifacts written in different languages—referred to as "semantic cross-language links" [MS12]—is one of the central challenges in this domain. Tomassetti, Torchiano, and Vetro present six categories of semantic interactions as a taxonomy [TTV13]. Tomassetti, Rizzo, and Torchiano introduce a language-agnostic approach to automatically detecting cross-language relations [TRT14]. Tomassetti and Torchiano are concerned with clustering of groups of languages that are used together in the same software project [TT14]. Selected projects hosted on GitHub are used as a corpus for analysis. The authors refer to such clusters as *language cocktails* and introduce several of their aspects, such as the level of polyglotism and the size of the most common clusters. Tomassetti et al. articulate the problem of language integration and present the model-based approach in the context of language workbench [TVT+13].

Education and Knowledge Engineering

Programming Education

Some techniques used in programming education—some of which may also be adopted in the context of software chrestomathy—include the following: collaborative learning [HZMK08]; designated search methods [YYN+07]; teaching-oriented, domain-specific development environments [RDL08]; systematic and semi-automated means of involving peers [YYN+07]; and the use of traceability to connect implementations with designs or specifications for the sake of understanding and validation [Gas08].

Various related accounts of programming education exist that clearly emphasize the importance of examples. For instance, Zhang and Nguyen describe the methodology of a Java tutorial, which fundamentally leverages simple, short, correct, and interactive examples [ZN04]. However, a software chrestomathy should not simply be considered a collection of examples; organization and annotation of examples are crucial. Finally, there are course designs that share the goal of covering programming broadly across programming domains and TSs (e.g., [Jac04]).

Ontologies and Linked Data in Education

Dietze et al. apply Linked Data principles to address the challenge of integrating the fragmented landscape of technology-enhanced learning (TEL) repositories to provide "rich and well-interlinked data for the educational domain" [DYG+12]. The project operates on existing TEL resources, addressing the heterogeneities of APIs, schemas, and response message formats. Borges, Maldonado, and Barbosa use an existing ontology of software testing, ontoTest [BNRM08], to exercise a conceptual modeling of the educational content [BMB11]. Such an approach relies on ontologies to provide better comprehension of the knowledge domain and support for knowledge sharing and reuse.

Sosnovsky and Gavrilova describe an ontology for teaching and learning C programming [SG05]. More generally, Kasai et al. describe an ontology of fundamental concepts in IT education and their relationships [KYNM06]. The dimension of ontology-based support for education still needs to be explored in the *101companies* project; related courses have used simple, less structured education methods.

Collaborative Knowledge Creation

In the e-learning context, some authors propose a combination of collaborative forums and wiki technologies to motivate deeper student engagement, better coordination, and progress monitoring [GLM+08]. Students are encouraged to produce new material based on discussion. Empirical results indicate better performance among active participants. Such a collaborative discussion-based model is quite typical for online e-learning, which by definition is lacking in-person context. In contrast, *101companies* stimulates the creation and aggregation of reference-knowledge artifacts by integrating trusted and authoritative knowledge resources and wiki content created by domain experts. Forte and Bruckman discuss a collaborative knowledge construction platform, Science Online [FB07]. This approach is called constructionism and advocates learning by working on personally meaningful projects. Under this approach, collective models of knowledge production and learning are connected. Because in MediaWiki, references are typically associated with one article, the authors built a bibliographic extension

to systematically handle references to external resources that automatically created a page for the reference and motivated a centralized discussion of linked resources.

E-Learning in Software Engineering

Despite of the broad application of the e-learning in general, there is very limited empirical evidence of its applicability for software engineering in particular. Teaching Software Engineering is challenging,as the benefits of Software Engineering concepts only become visible and understandable for students if they are applied to realistic problem scenarios of appropriate size and complexity. Wikis have gained the strongest attention lightweight platforms for exchanging reusable artifacts between and within software projects [DRR+05]. Some evaluations [MT07] have also confirmed that the strength of a wiki, as a collaborative authoring tool, can facilitate the learning of course concepts. Along the pedagogical dimension, the e-learning collaboration tools enable the process of an active construction of the new knowledge, so-called "knowledge-as-construction", opposite to the "knowledge-as-transmission" in the traditional class learning with the lecturer [Had03]. MOODLE, a course management system, supports a range of different resource types that allow including almost any kind of digital content into the courses [ATN04]. Further approaches include specially designed modeling tools support an easy access to specific Software Engineering concepts and ease their understanding by students [DEH+05]. Integrated virtual learning [Ham08] and game-based [CSH07] environments is another instance, still lacking a solid empirical evidence of their impact.

Ontologies for Software Engineering

According to [Smi01], three areas are creating a demand for the application of ontologies in computer science: database and information systems, software engineering (in particular, domain engineering), and artificial intelligence. Guizzardi discusses their in-depth analysis as well as the philosophical and historical foundations of ontologies in general [Gui05]. We analyze ontologies and their applications from specific domains of software engineering.

Ruiz and Hilera distinguish two categories of ontology for software engineering and technology:

domain knowledge ontologies and ontologies as software artifacts, typically used in software processes [RH06].

Souza, Falbo, and Vijaykumar provide a systematic literature review of ontologies in software testing [SdAFV13]. They point out there is a lack of uniformity in vocabularies and limited domain coverage. The Web Ontology Language (OWL) is used as an implementation level for ontologies in all studies. They also find that all ontologies are presented as UML class-diagrams. And they conclude the following: most ontologies have limited coverage; ontology evaluation is not discussed; none of the analyzed testing ontologies is truly a reference ontology; and none of them is based on a foundational ontology. A need exists for a well-established reference software testing ontology. A reference ontology must be a heavyweight ontology; therefore, it must comprise conceptual models that include concepts, several kinds of relations, and axioms that describe constraints and allow information to be derived from domain models.

Henderson-Sellers et al. combine conceptual modeling and ontology engineering approaches to normalize the semantics of the terms used in the software engineering ISO standards developed by SC7, a subgroup responsible for software engineering standards [HSGPML14]. The authors aimed to create unambiguous definitions of all terminology, increase conformity of standards to the ontological descriptions of terminology, and categorize the standards themselves.

Ratiu, Feilkas, and Jurjens are concerned with the extraction of domain ontologies from domain-specific APIs [RFJ08]. They analyze the similarity between APIs based on their graph representations. They use the following relations to represent API usage: isA, the taxonomic relation; hasProperty, the relation between a concept and its properties; actsOn, the relation between an action and its patients (the parameter of a method); and isDoer, the relation between an agent and its actions (the method of a class). Such a simple yet flexible model enabled a high degree of abstraction from different APIs.

Barcellos and Falbo developed a Software Measurement Task Ontology (SMTO) to support semantic interoperability between different measurement-related standards and to support semantic integration of software applications that support measurement [BdAF13]. The authors acknowledge the limitation of domain ontologies. They note that it is important to achieve a

common understanding regarding both domain- and task-related aspects of software measurement.

De Carvalho, Almeida, and Guizzardi propose using ontologies to define semantics for declarative, domain-specific languages (DSLs) [dCAG14]. Their approach uses a reference domain ontology to define the admissible states of the world. They use a representation of the valid expressions of a DSL to determine the abstract syntax and the real-world semantics of the language. The elements are are axiomatized in three corresponding logic theories, enabling a systematic treatment of real-world semantics, including formal tooling to support language design and assessment.

A software evolution ontology is introduced in [TKB10]. Different aspects of object-oriented (OO) systems are captured in three models: the software ontology model (som), the bug ontology model (bom), and the version ontology model (vom). The software ontology model is based on FAMOOS Information Exchange Model (FAMIX), a programming language-independent model for representing OO source code [DDT99]. FAMIX and other metamodels abstract OO concepts in a similar way. The version ontology model specifies the relations between files, releases, and revisions of software projects and the projects themselves, and the bug ontology model is inspired by the bug-tracking system Bugzilla. Nardi, Falbo, and Almeida study an adoption of foundational ontologies for dealing with semantic conflicts in enterprise application integration (EAI) initiatives [NdAFA13].

In [TKB10], the conceptual and digital entities are referred to as the “global cloud of software source code” and enriched with related information (versions, releases and bug reports). The usage of semantic technologies, such as OWL, RDF, and SPARQL Protocol and RDF Query Language (SPARQL), is proposed to represent a software project semantically with the ability to interlink other projects.

Linked Data for Software-Engineering Research

Linked Data provides a structural approach to exposing different software artifacts from source code repositories [KFH+12c, KFRC11]. MSRs create the context for exposing additional analysis and source-code–processing tools. The diversity of repositories is supported with the various documentation techniques used across them. Linked Data techniques are used to link documentation across different repositories [How08]. Source-code traceability is another promising application of Linked Data [IH12]. More ambitious approaches, such as Linked Data driven software development methodology, aim to expose all data from source code, version control systems, and bug tracking tools as Linked Data [IUHT09].

In addition to just surfacing the artifacts, organizing a common vocabulary is an important means for increasing reusability. Asset Description Metadata Schema (ADMS) describes "semantic assets, defined as highly reusable metadata and reference data" [Dek13]. Peristeras applied ADMS in the context of e-government to tackle the semantic interoperability of systems [Per11]. There is also ADMS for Software (ADMS.SW) [Goe11], which Berger applied to the Debian Package Tracking system [Ber12]. Berger's goal was to "generate RDF [Resource Description Framework] meta-data documenting the source packages, their releases and links to other packaging artifacts" and to link the packages to other open-source software and derivative distributions for traceability. Another use case for Linked Data is simplifying the integration of data from multiple code forges in open-source repositories, specifically to interlink the identities of a developer across different data sources on the web [ICH12].

In a broader context, the need to combine and expose data is called *mash up architecture* [MG08]. An architecturally similar approach has been applied in software engineering to improve software-engineering tools by enriching them with information from different sources such as web-based APIs and information repositories [GTS10]. Overall, we must admit there is limited adoption of data-driven approaches to surface software artifacts in software-engineering research.

Open-Source and Social-Software Ecosystems

We consider two types of ecosystems. One is the "traditional" open source software ecosystem. This is an instance of the ecosystem from [MS05] "a collection of software products that have some given degree of symbiotic relationships", which is consistent with the alternative definition by [Lun08] "a collection of software projects which are developed and evolve together in the same environment." In open source collaboration is the important aspect of such environment. Such ecosystems are considered similar to natural ones [MCG14], [MBM13]. The health [Jan14] and quality [FACF14] models are important characteristics of such ecosystems.

Another complementary type of ecosystem is the social-programmer ecosystem, which stems from modern notions of social coding and developer communities [SFFC+13]. The primary goal of such an ecosystem is improving and expanding people's knowledge adoption rather than software development. In addition to building relationships, collaborative tools (such as Twitter) help developers learn new technologies and adopt new practices [SFS14]. With the rise of social-computing platforms (such as StackOverflow), the importance of socio-cultural context also increases [MMM+11]. Software development is "increasingly less characterized by writing code and more by the need to aggregate, compose, and debug a diverse set of languages, components, services, and code snippets into functioning systems" [GCS13].

Chapter 4

Problem Space

The research agenda on software chrestomathies [Lae13] is concluded with the following statement: "research on software chrestomathies challenges software engineering and computer science in various respects." In this chapter we bring these challenges to the research context of the thesis in the following ways: we list the relevant challenges and then we define the requirements, that scope the problem space of our research, according to the design research methodology, discussed in chapter 1.

The majority of this chapter is an original part of the thesis, with some requirements derived from the conference publication [FLL+12b] Jean-Marie Fayre, Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz and Andrei Varanovich. Linking documentation and source code in a software chrestomathy. In WCRE, pages 335–344, 2012.

Challenges

Open Science Challenge

Open science and generally Research 2.0 promotes a development of the data-intensive collaborative scientific application. Software chrestomathy is one instance of such an application. It might be considered within a broader scope of the eScience agenda [FH09]. More specifically, it can serve as an infrastructure to evaluate the languages, technologies and tools across technological spaces. Software chrestomathy is an advanced effort to illustrate the variation points and technology options that are linked to the notion of technical spaces. The usage of the software chrestomathy goes beyond the research community. Professional developers may be framed in a specific technological space acquiring "silos of knowledge", laboriously. However, developers are expected to travel technological spaces and adapt to new technologies and languages rapidly and continuously. This will only be possible once developers obtain convenient access to sufficiently organized, abstract, and connected knowledge resources.

Knowledge Engineering Challenge

With all the many features, contributions, languages, technologies, and concepts in scope, one can easily understand that some degree of knowledge engineering is needed for maintaining the quality and usefulness of a chrestomathy. This is the central research challenge comprised of the following research questions:

- What is the *ontology* administering a chrestomathy so that all involved entities (such as code artifacts, languages, technologies, and concepts) are organized (classified, associated)?
- What is the *vocabulary* of such an ontology, e.g., the concrete programming concepts to be covered by the feature model or to be mentioned in the documentation of contributions?
- How can existing *knowledge resources*, such as textbooks or *Wikipedia*, be usefully integrated in the scope of a chrestomathy for the immediate benefit of users?

Knowledge Integration Challenge

We assume the following definition of knowledge integration: "Knowledge integration is the dynamic process of linking, connecting, distinguishing, organizing and, structuring models of scientific phenomena."[Lin00] A key knowledge integration challenge for software chrestomathy is the heterogeneity and complexity of the involved sources, which require methodological and algorithmic precautions to actually obtain a vocabulary that is useful, say, for teaching and documentation. To this end, the mining thresholds should be controlled, the queries (visualizations) on vocabulary data for decision making and quality assurance should be used, and non-automated validation steps along the way should be applied.

To develop a process for an integration of such community resources, certain questions have to be addressed:

- What are the "useful" resources to be integrated?
- What is the balance between knowledge integration and knowledge creation?
- How should an integrated knowledge base be connected with the actual software engineering best practices?

Reverse Engineering Challenge

A software chrestomathy, as opposed to traditional software products, collects source-code samples that exercise many software languages, technologies, and concepts. Chrestomathies imply specific forms of complexity that challenge reverse engineering techniques: heterogeneity in terms of the "many" languages and technologies used by the collected samples and variability in terms of tasks or features implemented by the samples. Linking the expected architecture described by documentation with the actual architecture extracted from source code is a well-known reverse engineering problem [MNS95, MNS01]. There exist methodologies for *software architecture reconstruction* (SAR) including Symphony [vDHK+04], *CacOphoNy*[Fav04b], and others [PFGJ02, KLL09, LOV02]. However, previous research on SAR addressed traditional

architecture (as opposed to linguistic architecture; see [DP09] for a survey) and traditional products (as opposed to software chrestomathies). Our approach is specifically concerned with the *linguistic architecture* of software products [FLV12] with links, for example, from source-code artifacts to software languages, software technologies, and software concepts. (Traceability for product features in source code is also of interest.)

Linked Data Challenge

Surfacing software chrestomathy data and vocabulary so that it is useful for the stakeholders, such as software analysts and consumers of software knowledge is another original challenge in the context of software chrestomathy. We are aware of a single effort concerning the provision of facts about source-code repositories using *Linked Data*. That is, *SeCold* is an open and collaborative platform for sharing software datasets, as introduced in [KFH+12a]. In its first release, the dataset contains about two billion facts such as source-code statements, software licenses, and code clones from 18,000 software projects exposed using *Linked Data* principles. Overall, our research community is just at the beginning of handling such repositories and derived artifacts as *Linked Data*. When applied to a software chrestomathy, *Linked Data* principles imply that all wiki pages, all source code units, all derived resources including metadata, and all ontological entities (software concepts, languages, technologies) must be referable through HTTP URIs with responses that reveal content, metadata, and semantic links to other resources, also including external resources.

Ontology Engineering Challenge

Ruiz an Hilera distinguish two categories of ontology for Software Engineering and Technology[RH06]: domain knowledge ontologies and ontologies as software artifacts, typically used in software processes. Software chrestomathy, by definition, relies on the community authored knowledge, and has an open model of contributions, that provide insights into technologies and software engineering best practices. From this perspective, the ontology engineering challenge for software chrestomathy is twofold. The ontology is meant to help managing knowledge

about programming technologies and relates in this regard to other applications of ontologies to knowledge management [CJB99, SSSS01]. For instance, the work of [RFD+08] describes the semi-automatic derivation of an ontology for domain-specific programming concepts—as they are supported, for example, by APIs for XML or GUI programming. Such ontologies may feed into a more comprehensive ontology of programming technologies. An ontology for software languages, technologies, and concepts as an important abstraction level for software engineers and programmers meant to be useful in understanding, comparing, or learning about such entities. Such a general ontology has not been delivered before.

Technology Modeling Challenge

Technology models describe important characteristics of a software technology in relation to relevant software artifacts, software languages, and other entities. Technology models are "declarative entity-relationship models with entities for languages, technologies, concepts, and artifacts and with relationships to express data flow, dependencies, conformance, and others." [LVL+14] Such technology modeling is a form of megamodeling focusing on the linguistic architecture of software projects. The notion of megamodeling has received much recent interest, specifically in the MDE community with diverse application areas such as model management [BJRV05], software architecture [HMMP10], and models at runtime [SNG10]. Different definitions of "megamodel" are in use, see, for example, [DKM13] for a more recent proposal. Usually, it is assumed that a megamodel is a model whose model elements are again models by themselves while the term "model" is interpreted in a broad sense to include metamodels, conformant models, and transformation models. We analyzed the instances of the linguistic architecture in chapter 2. As the analysis suggests, technology models can also benefit from ontological knowledge. The latter focuses on classification and characteristics overall. The former focuses on the use of the technology in terms of the involved artifacts, their characteristics, and related data flows. The challenge at hand is to investigate the potential for a general-purpose technology modeling language that captures the essential linguistic relations and is not tailored to a particular TS or field of MDE.

Educational Challenge

Software chrestomathy, by definition, is supposed to be useful in learning. We foresee diverse profiles of learners. The aspects of software chrestomathy, discussed in chapter 3, are of different relevance. Professionals require a more abstract view of different technologies involved, while university students may benefit from open chrestomathy-based courses on programming technologies. An ontology for teaching and learning C programming is described in [SG05]. More generally, an ontology of fundamental concepts in IT-education and their relationships is described in [KYNM06]. The dimension of ontology-based support for education still needs to be explored in the software chrestomathy; related courses used simple, less structured education methods so far. Since a software chresomathy aims at representing and conveying knowledge, an effort must be made to effectively serve knowledge consumers.

Requirements

Core Properties of Software Chrestomathy (R1)

In chapter 3 we defined a software chrestomathy as an advanced conceptualization of the program chrestomathy notion. To generalize this notion, the following requirements from Table 2.1 should be implemented:

1. Focus on software technologies and software languages.
2. Scope of the comparison is implementations of the same software system.
3. Various technological spaces should be covered.
4. Ontology-driven knowledge organization.
5. Classroom tested.

All code and data should be made available following the precepts of Research 2.0, and specifically Open Science, and Linked Data. The software chresomathy project should always favor simplicity and incrementality over sophistication and completeness, thereby catering to community involvement and continuity of the project.

Ontology-driven Classification (R2)

A chrestomathy breaks down into physical entities (contributions, individual source files, fragments thereof, and source code illustrations other than contributions) and conceptual entities (languages, technologies, concepts, features, and others). These entities engage in certain ontological relationships.

Despite some disagreement on what an ontology is [PGS02], we do not aim to review ontological definitions. Our goal is to demonstrate the usefulness of an ontological approach to knowledge organization in a software chrestomathy. The following requirements should be realized:

1. A wiki should be used to document all involved entities, with an explicit semantics and taxonomy. The ultimate taxonomy of software languages should subsume and integrate existing, fragmented classifications in a transparent manner.

2. Semantic web-like properties should serve as an explicit linkage between concepts, their relationships, and generic theories.

3. A context of modularization should be clear, such as classification of languages, technologies or features.

4. A minimal axiomatization to detail the difference between similar concepts should be supported.

5. A good naming policy should be realized.

6. A wiki should provide the means for rich documentation, including software engineering metadata, associated with a given contribution, the ability to link to existing knowledge resources for languages and technologies or express the dependencies between languages and technologies.

Linking Documentation and Source Code (R3)

A software chrestomathy is concerned with these kinds of links:

Actual links. These links reside in the source code. Reverse engineering techniques may recover these links. The graph of the links and the underlying entities is referred to as the *actual* (or "as-implemented") architecture.

Expected links. These links reside in higher-level models or documentation. The corresponding graph is referred to as the *expected* (or "as-designed") architecture.

Links to establish. In order to ensure consistency between source code and documentation, links should be established between source-code and documentation entities. These links may

be documented within the code, within the documentation, or elsewhere; they may also be discovered by appropriate analyses (e.g., based on name conventions).

To support the co-evolution of documentation and source code in the highly diverse chrestomathic ecosystem we add the following requirements for linking documentation and source code:

Generality. The approach must work for most, if not all, software languages and software technologies.

Scalability. The approach must support trading off accuracy of fact extraction against development effort for fact extractors. (That is, simple textual or lexical fact extractors should provide enough basic information, while more language-specific and possibly costly syntactical or semantical extractors provide optional information, e.g., for more advanced links and more rigorous checks.)

Declarativeness. The approach should rely on declarative rules, as opposed to any low-level encoding.

Scoping. Rules must be controllable in terms of the specific directory-, file-, or fragment-level scope.

Evolvability. Manual and automated addition and removal of rules should be straightforward and traceable.

Assistance. The analysis of existing metadata should be used in generating recommendations for metadata revisions.

Reuse. Existing language-technology-aware analysis tools, libraries or web services should be reused.

Vocabulary Engineering Through Knowledge Integration (R4)

In addition to collaborative knowledge creation, a software chrestomathy knowledge management process relies on knowledge integration. The goal is to establish a consolidated vocabulary

for inter-linkage across chrestomathy, wikis, and textbooks. We aim at the informed (hence, semi-automated) consultation of multiple technical, readily indexed textbooks for the sake of deriving a consolidated and manageable vocabulary with confirmed links to key sources such as *Wikipedia*. Vocabulary mapping should be established in the context of ontology matching [ES07, Ome02] with vocabularies of substantial size that may need to be matched largely automatically. In our context, the size needs to be limited to allow for human intervention. The underlying framework as well as data related should be publicly available.

Linked Data Enabled Infrastructure (R5)

To address the key *Linked Data* challenge, we need to surface a software chrestomathy in a way that confederates all resources in a useful and sufficiently efficient manner. In particular, the resources should be conveniently explorable; navigation should be discoverable and feasible for all relationships (links) between resources; the relevant formats and the underlying ontology should be accessible and documented; both programmatic and interactive access should be generally supported.

The following scenarios enable turning different parts of software chrestomathy into a linkable and navigatable cloud of resources:

Navigate from wiki to repository. Contributions rely on a distributed repository, that is, more then one physical repository for all contributions. With this setup, the naive naming-convention based approach is not sufficient to maintain a clickable link on contribution pages on the wiki and the associated folder in the physical repo. A plugin to enable navigation from the wiki to the repo should be developed; the plugin has to interpret the registry of the confederated repo.

Navigate from repository to wiki. The source code resides on the arbitrary source control system, such as GitHub. Often, the contributors own the repositories in a way, that any external infrastructure can only read the content. With that in mind, we favor an approach of having an explorable view on the repo from which one can navigate from the files and folders to the associated wiki pages.

Reference fragments on wiki pages. Such referencing of fragments requires design and development of the concepts of fragment descriptor and fragment location. They should have an unambiguous identifiers via URL and the underlying content.

Associate derived resources with primary resources Derived resources should be linked to their primary counterparts, i.e. the source code from which the metadata was produced.

Operate on the wiki like a graph. A wiki-based knowledge base has a graph structure and should provide the means to represent the chromatography as a traversable graph, with the typed edges and the links between them.

Operate on the repo like a tree. Federated source code repository has a tree structure and should be possible to browse through a repository, with the underlying folders and files.

A Chrestomathic Ontology (R6)

Ontologies are increasingly useful in software engineering for analysis, design, implementation, documentation, testing, and maintenance of software systems. The chrestomathic ontology *SoLaSoTe* should serve for knowledge representation, management and integration in the broad context of software languages and software technologies – as opposed to more specific categories of ontologies (such as domain ontologies, task ontologies, application ontologies, or (very) high-level ontologies). It should handle 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 should be maintained on the chrestomathic wiki.

Linguistic Architecture of Software Products (R7)

Linguistic architecture is another original term developed in this thesis. It is a new, unexplored, viewpoint on the software architecture and considered as a fundament for technology modeling. Its relevance has to be shown and further connected to a software chrestomathy and should help in managing diversity and heterogeneity of software technologies.

General-purpose Language for Technology Models (R8)

To address the technology modeling challenge stated in the previous section, we need:

- To develop a megamodeling approach that is useful for understanding the linguistic architecture of software products in terms of the involved languages, technologies, and linguistic relationships. The approach should be supported by the language and an associated tool suite.

- To demonstrate technology modeling in the challenging context. For instance, Object/Relational/XML [Tho03, LM06a, MAB07] mapping (or O/R/X mapping).

- To improve the cognitive value of technology models by enabling a form of *linked megamodels* such that entities and relationships are linked to resources (e.g., in the repository of the software chrestomathy) so that technology models can be explored and validated.

Validation

In compliance with the design research methodology discussed in chapter 1, the evaluation should demonstrate how software chrestomathy contributes to the state of the art analyzed in this chapter. The contribution spans across several research contexts that address the different aspects of the problem statement.

According to subsection 1.2.1, "Organizing Software Languages and Technologies," the project should serve a more general scope in software development when compared to other efforts on program collections and domain-specific software development challenges or benchmarks.

According to subsection 1.2.3, "Knowledge Integration," the project should support comparison and cross-referencing for diverse software technologies and software languages across TSs. From an education perspective, the project should provide content and structure for different forms of classical education, self-learning, or e-learning.

According to subsection 1.2.2, "Ontology-based Knowledge Management," a large-scale effort on organizing knowledge on programming languages and software engineering requires organization principles for vocabulary, documentation, and code. The project should provide an infrastructure applying ontologies to software engineering [Ahm08]. In the context of software chrestomathy, an ontology for software languages, technologies, and concepts that is meant to be useful in understanding, comparing, or learning about such entities is an important abstraction level for software engineers and programmers.

According to subsection 1.2.5, "Understanding Modern Software Products," the project should use new forms of models for software technologies, such as megamodels [BJV04b, SCFC09], and for comparative (and other) forms of software linguistics [FGLP11]. As a result, the project should provide a macroscopic view of software projects that operates at a high level of abstraction such a way that it identifies the essential entities of a software project and the fundamental relationships between them.

Chapter 5

101companies Software Chrestomathy

In this chapter we provide a comprehensive introduction into a software chrestomathy and develop its key properties together with the underlying infrastructure. The conent of this chapter contributes to the following requirements: *R1. Core Properties of software chrestomathy*, *R2. Ontology-driven Classification*, *R3. Linking Documentation and Source Code*.

Introduction

The software chrestomathy of the *101companies* [1] community project demonstrates "many" software languages and software technologies by implementing "many" variants of a human resources management system; each implementation selects from "many" optional features. All implementations are available through a source-code repository and they are documented on a wiki. Source code and documentation encode references to software languages, software technologies, software concepts, and product features, which, by themselves, are also documented

This chapter is solely based on two conference papers: [FLSV12] Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz and Andrei Varanovich. 101companies: A community project on software technologies and software languages. In TOOLS (50), pages 58–74, 2012. [FLL+12b] Jean-Marie Favre, Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz and Andrei Varanovich. Linking documentation and source code in a software chrestomathy. In WCRE, pages 335–344, 2012.

[1] The "companies" postfix in "101companies" refers to the kind of system that is built time again in this project: a system that models companies in terms of some human resources aspects: department structure, employees, salaries. The "101" prefix in "101companies" refers to "101 ways" of building said system. Indeed, there are more than "101 ways" of building a human resource management system with different software technologies and software languages.

and linked on the wiki. This setup implies the challenges of establishing links between source code and documentation as well as verifying that source code and documentation are in agreement. We describe an approach that addresses these challenges: it relies on a rule-based system that extracts relevant information from source-code artifacts (e.g., information about language and technology usage) and assigns metadata to the artifacts (e.g., methods for validation and fact extraction). The linked source-code repository and wiki as well as various derived information resources are available through the *101ecosystem* for the benefit of the reverse engineering community.

Welcome *101companies*

Objective of *101companies*

101companies is a community project in computer science (or software science) with the objective of developing a free, structured, wiki-accessible knowledge resource including an open-source repository for different stakeholders with interests in software technologies, software languages, and technological spaces; notably, teachers and learners in software engineering or software languages as well as software developers, software technologists, and ontologists.

The Notion of Contribution

The project relies on the aggregation, organization, annotation, and analysis of an open-source corpus of contributions to an imaginary Human Resource Management System: the so-called *101companies* system, which is prescribed by a set of optional features. Contributions may be implementations of system variations and specifications thereof. Each contribution should pick a suitable, typically small set of features and demonstrate original and noteworthy aspects of software technologies and software languages in a focused manner. Contributions are grouped in themes to better apply to varying stakeholders and objectives. The project also relies on contributions in the broader sense of resources for software technologies and software languages, or components of an emerging ontology.

Illustration

The following illustrations are meant to clarify the nature of the project and the scale that has been reached. These illustrations should not be confused with any sort of (scientific) validation of the project or the existing contributions.

The *101companies* system (or just "the system") is an imaginary Human Resource Management System (HRMS) that serves as the "running example" in the *101companies* project. That is, contributions to the project implement or specify or otherwise address a HRMS system for a conceived company as a client. A company consists of (top-level) departments, which in turn may break down hierarchically into further sub-departments. Departments have a manager and other employees. The imaginary system may be used by conceived employees and managers within the conceived company. Employees have a name, an address, and a salary. The system may support various features. For instance, the system could support operations for totaling all salaries of all employees and cutting them in half; it could provide a user interface of different kinds; and it could address concerns such as scalability or security. Features of the system are discussed in section 5.4. Figure 5.1 specifies the basic data model of the system in UML.

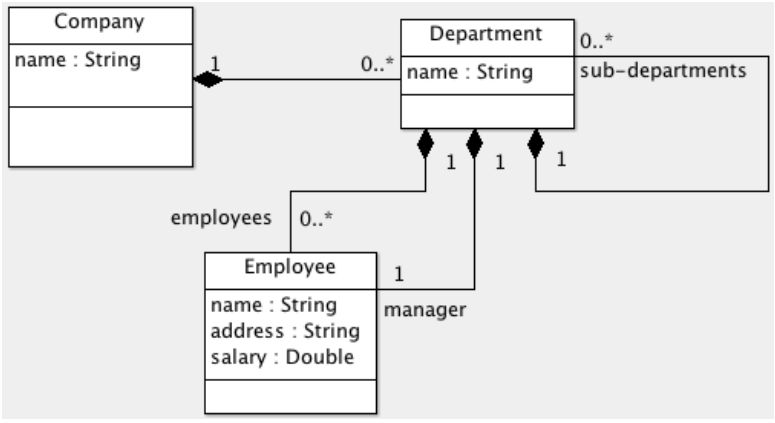
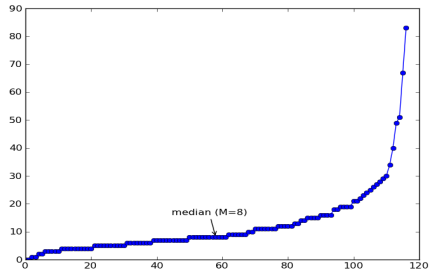


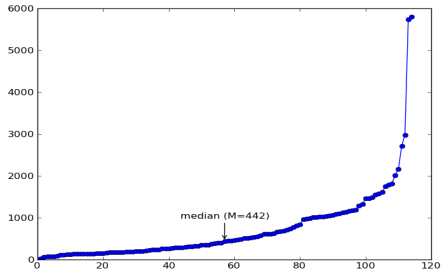
Figure 5.1: A UML class diagram serving as an illustrative data model

Figure 5.2 gives an idea of the varying code-level complexity for the existing implementations of the *101companies* system. (Other forms of contributions are not considered here.) Based on systematic tagging, we count only developer-authored code-like units as opposed to generated code or IDE support files. The plots show the number of files and the lines of code for such units.

At the time of writing, there are 192 implementations in the repository.



Implementations ordered by number of files



Implementations ordered by lines of code

Figure 5.2: Illustrative code-level complexity indicators for *101companies* implementations

(Lines of code may include comments.) The number of files hints at "file-level modularity" of the implementations. As the medians suggest, most implementations are in the range of a few hundred lines of code and less than ten files. The distribution of these numbers is a result of different programming languages, different technologies, different feature sets that are implemented as well as subjective factors due to developer choices. We should emphasize that the plots serve for illustration; they cannot be expected to hint at any proper metric.

Here are some metrics for the *101* software chrestomathy:

- 25 *features* that can be implemented, subject to a feature model [Bat05], e.g.: *Feature* `Hierarchical company` for the data model of a company to break down into departments recursively with employees and a manager per department; *Feature* `Total` for computing the salary total for all employees in a company; *Feature* `Logging` for logging all salary changes. About 10 features are popular. The remaining features are either more specific or perhaps premature or obsolete.

- There are 193 *contributions*, i.e., implementations of the *101system*. (Several of these contributions are premature, unmaintained, insufficiently documented, or otherwise sub-optimal in terms of software engineering best practices.)

- 40 *languages* (mostly programming languages) are exercised.

- 96 *technologies* (e.g., libraries) are exercised.

- 477 *concepts* are referenced by the wiki-based documentation. The term concept is used here in the broad sense to include programming techniques, programming domains, classifiers (for programming languages, technologies, and features), and so forth.

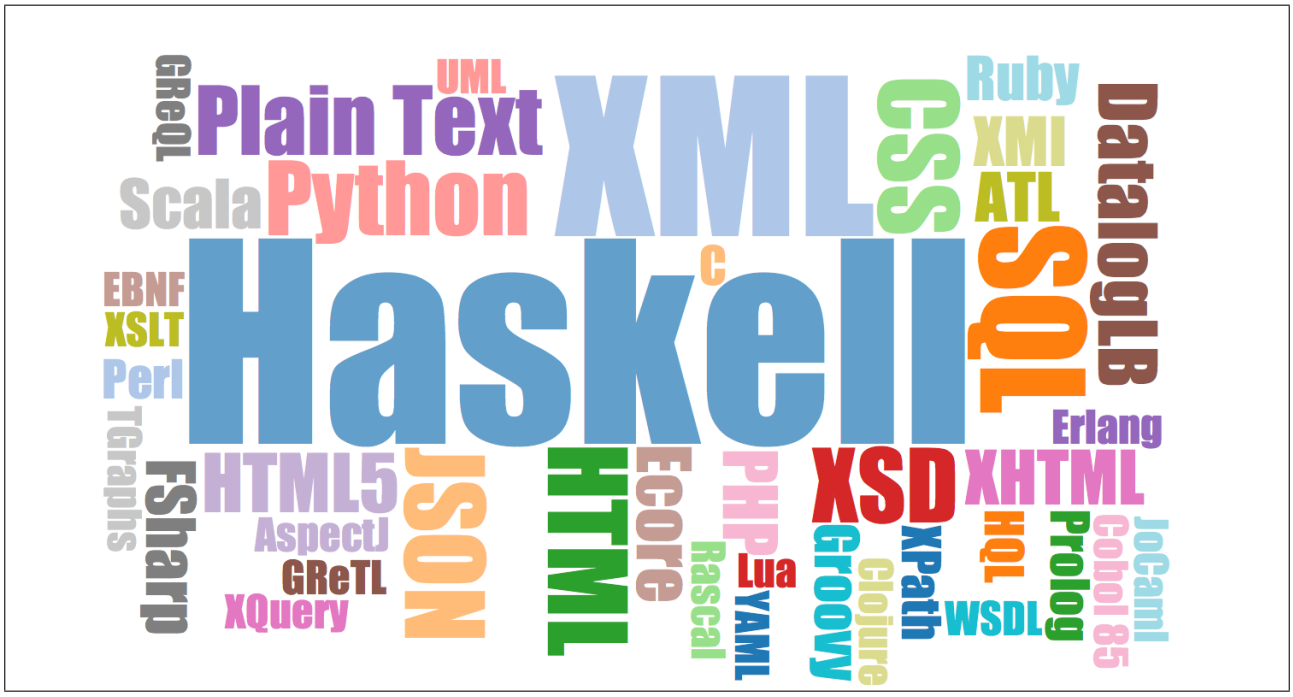


Figure 5.3: Illustrative tag clouds regarding usage of languages

Figure 5.4: Illustrative tag clouds regarding usage of technologies

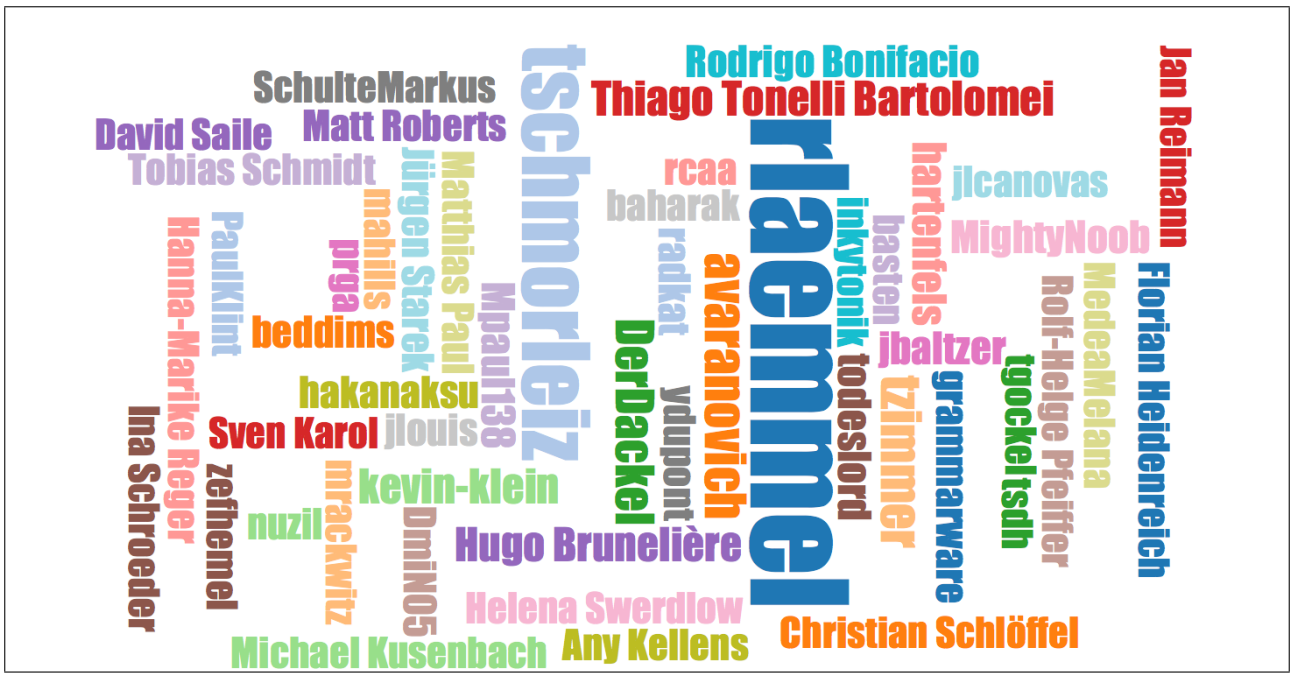


Figure 5.5: Illustrative tag clouds regarding contributors

Features of the *101companies* System

The term feature should be understood broadly as referring to optional requirements for the imaginary system (e.g., as in the established sense of functional and non-functional requirements). The optionality of features encourages smaller contributions that demonstrate specific aspects of software technologies and software languages. The feature set is not driven by concepts of the human resources domain. Instead, the important consideration for each new feature is that it should be helpful in demonstrating particularities of software technologies and software languages. The feature set is constantly under revision since each additional technology in scope, each new implementation (planned or completed) may trigger additional features or a revision of the classification tree. Some features intentionally are left underspecified so that contributions can apply suitable refinements for the benefit of useful demonstrations. (For instance, there is a feature Client-server to constrain design such that an implementation uses a client-server architecture without though specifying precisely client and server.) Ideally, each feature would be demonstrated by several contributions. Some features may not be demonstrated yet at all because they are at the stage of calling for contributions.

The classification of features is shown in Table 5.1.

Table 5.1: Classification of features in the *101companies system*

Alternative feature	An alternative feature in feature modeling. *Feature* `Serialization` – serialization for company data and *Feature* `Company` of the *101system* are alternative features in that they provide fundamentally different (mutually exclusive) options for serialization and data modeling.
Mandatory feature	A mandatory feature in feature modeling. *Feature* `Company` of the *101system* is a mandatory feature because it is pretty much impossible to have any system without a data model.
Optional feature	An optional feature in feature modeling. Pretty much all features of the *101system* are optional (except for the mandatory feature *Feature* `Company` with its subfeatures) because an actual implementation of the *101system* may freely choose to implement or not to implement certain data, functional, non-functional, and UI requirements.
Or feature	An "or" feature in feature modeling. *Feature* `Parallelism` of the *101system* is an or feature because one could use both *Feature* `Task` parallelism and *Feature* `Data parallelism`.

Features are represented as requirements for a new or altered software system or component. Consider the following functional requirement for the 101system: "The system must be able to total the salaries of all employees of the company and to report the total to the user." *Feature* `Total` describes this requirement in more detail.

Table 5.2: Requirements of the *101companies* system

Requirements types	
Functional	A required IO behavior of a software system or a component
Non-functional	A required quality of a software system or a component. For example Development-time quality or Run-time quality
Data	A constraint on the data model of a software system or a component
UI	A requirement regarding the user interface of a software system

In the context of the *101companies* project, it is important that "features" are directly related to requirements and are loosely related to feature-oriented software development in that one expects features to be actually or potentially implemented according to this paradigm. Table 5.3 summarizes the features and their relations to the particular requirement.

Figure 5.6 gives an idea of the distribution of feature coverage by existing contributions (in fact, by implementations). The size of a feature name in the tag cloud expresses the frequency

Table 5.3: Features of the *101companies* system

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

of demonstration across all the implementations: the bigger, the more implementations declare to demonstrate the feature. (The popularity of Company, Total, and Cut is a consequence of the fact that most implementations pick these functional requirements as a lower bar for demonstration.) In fact, these basic structural and behavioral features are interesting enough to demonstrate already many programming techniques, mapping concerns, and overall capabilities of software technologies.

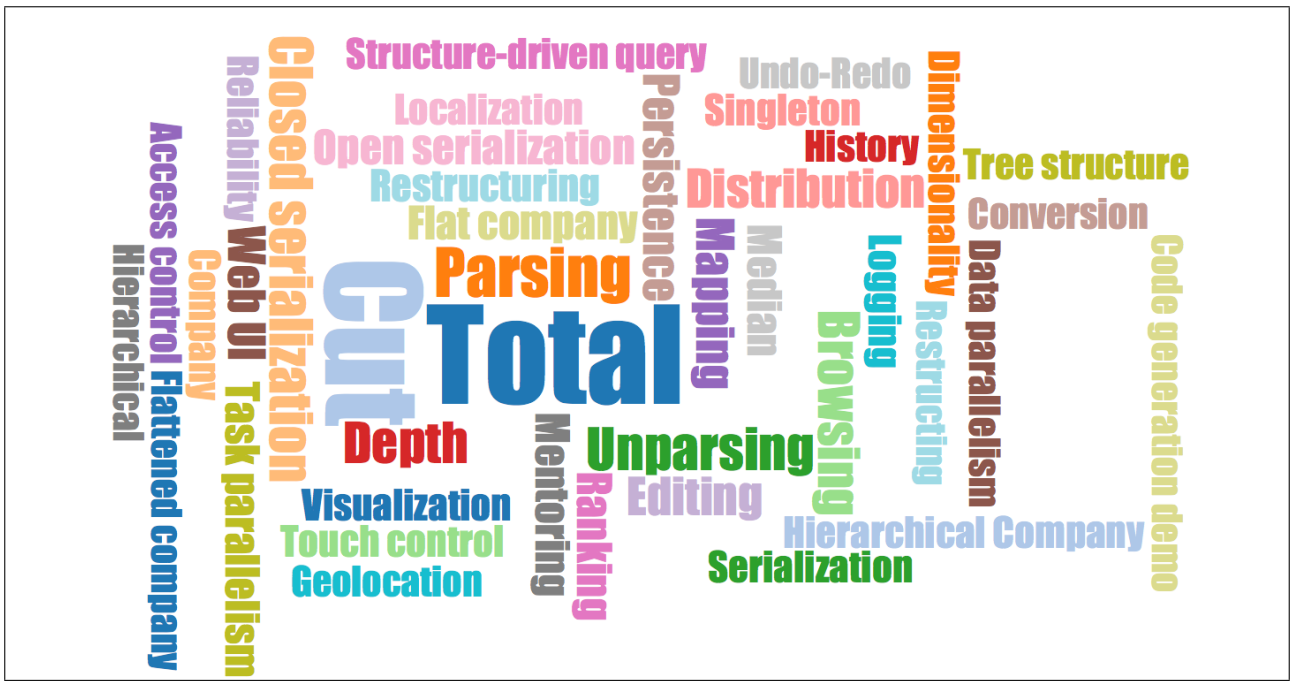


Figure 5.6: Illustrative tag cloud regarding feature frequency for implementations

The *101haskell* Chrestomathy

101haskell may be understood simply as the Haskell-specific cut through *101* and it thus demonstrates different functional programming techniques and technologies. There are 35 Haskell-based contributions.

It turns out that *101haskell* served as the driving force in advancing the chrestomathy notion assumed in *101* over the last 18 months. As a result, *101haskell* is best suited to demonstrate some aspects of an advanced chrestomathy:

- Software engineering best practices (subsection 5.4.1):
  - The contributions are hosted on a sub-repo of *101repo*.
  - The Haskell code is modularized.
  - Haskell's Cabal is used for build management.
  - Haskell's HUnit is used for unit testing.
- The documentation is semantically enriched (section 5.6).
- Haskell textbooks are integrated with the *101wiki* (chapter 6).

An Excerpt of *101haskell*

We present the excerpt in a feature-driven manner: one feature per subsection, with the following format for the subsections:

Requirement Specification. The feature is explained at the level of a requirement specification for an information system. For instance, we may encounter a functional requirement for a specific functionality expected by a user of the *101system*.

Chrestomathic Purpose. The feature is motivated in terms of its value for a software chrestomathy. That is, we mention relevant programming domains, concepts, techniques, or classes of technologies that are naturally associated with the feature.

Haskell Illustration. One or more Haskell-based contributions ("implementations") are briefly sketched and explained. The emphasis is on pointing out the involved programming techniques, technologies (libraries), and concepts.

Feature `Flat company`

Requirement Specification

A data model of companies is to be supported as follows. A company has a name and aggregates employees. Each employee has a name, an address, and a salary. Names of companies and employees are strings; addresses are strings, too; salaries are floating-point numbers. For now, companies are "flat" in that they aggregate employees without any hierarchical company structure, but see "non-flat" companies in subsection 5.4.2.

Chrestomathic Purpose

The data model exercises very *basic data modeling facets*: primitive types, tuples, lists, and non-recursive type declarations.

Haskell Illustration (*Contribution* `haskellStarter`)

The data model is defined in terms of a system of type synonyms. Company and employee terms are composed as tuples. Employees are aggregated as lists. Salaries are represented as single-precision floating-point numbers. Thus:

```
type Company = (Name, [Employee])
type Employee = (Name, Address, Salary)
type Name = String
type Address = String
type Salary = Float
```

Consider the following sample instance:

```
sampleCompany :: Company
sampleCompany =
  ( "Acme Corporation",
    [
      ("Craig", "Redmond", 123456),
      ("Erik", "Utrecht", 12345),
      ("Ralf", "Koblenz", 1234),
      ("Ray", "Redmond", 234567),
      ("Klaus", "Boston", 23456),
      ("Karl", "Riga", 2345),
      ("Joe", "Wifi City", 2344)
    ]
  )
```

Feature `Hierarchical company`

Requirement Specification

Departments are added to the data model of *Feature* `Flat company`. Companies aggregate (top-level) departments. Each department has a name, a manager, and aggregates both sub-departments as well as employees – other than the manager.

Chrestomathic Purpose

The nesting of departments necessitates the data modeling facet of *recursive types*. Also, the fact that a department aggregates both sub-departments and employees implies that a choice must be made as to whether *homogeneous or heterogeneous collections* are used. In the first case, a department has two separate (homogeneous) containers – one for employees, another for sub-departments. In the second case, employees and departments end up in one (heterogeneous) container.

Haskell Illustration (*Contribution* `haskellComposition`)

The homogeneous option is exercised here. The data model is defined mainly in terms of *type synonyms*, as before, but an *algebraic data type* is used for recursive departments since type synonyms cannot be recursive.

```
data Department
  = Department Name Manager [Department] [Employee]
```

Consider the following sample instance (with an elision):

```
sampleCompany :: Company
sampleCompany =
  ( "Acme Corporation",
    [ Department "Research"
        ("Craig", "Redmond", 123456)
        []
        [ ("Erik", "Utrecht", 12345),
          ("Ralf", "Koblenz", 1234) ],
      ...
```

Feature `Total`

Requirement Specification

Given a company, the salaries of all its employees are to be totaled. An implementation of this functional requirement is to be demonstrated for a sample company (e.g., by means of a unit

test).

Chrestomathic Purpose

Conceptually, the required functionality corresponds to a *query* over the company structure such that all employees are projected to their salaries, which are then summed up. Depending on the data modeling and programming paradigm at hand, it may be straightforward to quantify all employees (e.g., in a relational database with a table for employees) or it may require some sort of *walking over the tree-like structure* of a company (e.g., in an object-oriented or functional program). In the latter case, various programming techniques for traversal could be exercised (e.g., visitors in OO programming or generic functions in functional programming [LJ03a, RJJ+08]).

Haskell Illustration (*Contribution* `haskellList`)

The following implementation assumes the most basic data model (as of subsection 5.4.1). Some non-basic bits of functional programming come to play in this contribution. That is, we use the *map* combinator for *list processing* and *local scope* for arranging helper functions in the scope of `total`. Also, *function composition* is used explicitly, thereby providing an example of *point-free style*. Thus:

```
total :: Company → Float
total = sum . salaries
  where
    -- Extract all salaries in a company
    salaries :: Company → [Salary]
    salaries (n, es) = map getSalary es
      where
        -- Extract the salary from an employee
        getSalary :: Employee → Float
        getSalary (_, _, s) = s
```

Haskell's *HUnit* framework for unit testing is leveraged for exercising the query. That is, a comparison of the expected result of `total` with the actual result is declared as follows:

```
totalTest :: Test
totalTest = 399747.0 ¬ =? total sampleCompany
```

Feature `Cut`

Requirement Specification

Given a company, the salaries of all employees of the company are to be cut in half. An implementation of this functional requirement is to be demonstrated for a sample company (e.g., by means of a unit test).

Chrestomathic Purpose

Conceptually, the required functionality corresponds to a *transformation* over the company structure such that each employee is updated in terms of the salary, while preserving the structure and the data of the company otherwise. The chrestomathic purpose can be compared to the one of *Feature* `Total` (subsection 5.4.3), except that queries and transformations are fundamentally different. That is, different programming idioms may be needed (e.g., update rather than read). Additional decisions may be necessitated: a choice between *immutable data* and *destructive update*.

1st Haskell illustration (*Contribution* `haskellComposition`)

The following implementation assumes the data model with nested departments (as of subsection 5.4.2). The transformation is modeled as a *pure function*. Local scope and `map` is used for clarity and conciseness.

```
cut :: Company → Company
cut (n, ds) = (n, (map cutD ds))
  where
    -- Cut all salaries in a department
    cutD :: Department → Department
    cutD (Department n m ds es)
      = Department n (cutE m) (map cutD ds) (map cutE es)
```

```
  where
    -- Cut the salary of an employee in half
    cutE :: Employee → Employee
    cutE (n, a, s) = (n, a, s/2)
```

The transformation is exercised by the following *HUnit* test, which simply verifies that the individual cuts of the employees equate to cutting in half the total:

```
cutTest :: Test
cutTest
  = total sampleCompany / 2 ¬ =? total (cut sampleCompany)
```

2nd Haskell Illustration (*Contribution* `haskellSyb`)

The following implementation leverages *generic functional programming* according to the "Scrap your boilerplate" style (SYB [LJ03a]):

```
cut :: Company → Company
cut = everywhere (extT id (/(2::Float)))
```

The generic function is clearly more concise than the previous attempt. The function is applicable to any data model for companies for as long as floating-point numbers are used for salaries only, but the function could also be revised slightly to apply instead more carefully to salaries. For SYB to be applicable, the involved algebraic data types must instantiate the typeclasses `Typeable` and `Data`. The corresponding instances can be derived in a regular manner by the compiler (GHC) on the grounds of suitable deriving clauses. The following module (fragment) even demonstrates that such derivation can be requested even after the fact: in a module that "stands alone" from the module with the actual data types:

```
{-# LANGUAGE DeriveDataTypeable, StandaloneDeriving #-}

import Company.Data
import Data.Data
import Data.Typeable
```

```
deriving instance Data Department
deriving instance Typeable Department
```

Various other generic programming approaches could be illustrated as well, also including variations on SYB; see, for example, a comparison of approaches in [RJJ+08]. In fact, *101haskell* already covers a few approaches.

All *101haskell* contributions rely on Cabal for build management. In this manner, they are easy to build including dependency chasing on Hackage; they are easy to test, too. Here is the Cabal spec for *Contribution* `haskellSyb` (with an elision):

```
name: haskellSyb
version: 0.1.0.0
synopsis: Generic Programming in Haskell with SyB
homepage: http://101companies.org/wiki/Contribution:haskellSyb
build-type: Simple
cabal-version: >=1.9.2
library
  exposed-modules:
    Company.Data
    Company.Sample
    Company.Generics
    Company.Total
    Company.Cut
    ...
  build-depends: base >= 4.4 ∩< 5.0, syb >= 0.3
...
```

Feature `Parsing`

Requirement Specification

Company data is to be represented in a format that is suitable for human consumption and editing (e.g., in some well-defined textual concrete syntax, a schema-defined XML-based format,

or CVS). This representation is to be parsed for the purpose of carrying out computations. For brevity, we omit here any discussion of demonstration (testing).

Chrestomathic Purpose

Parsing brings the programming domain of *language processing* into the scope of the chrestomathy. In this manner, different kinds of *parsers* (e.g., context-free grammar-based ones or *XML* parsers), different parser technologies, and different styles of parsing can be exercised and possibly compared.

Haskell Illustration (*Contribution* `haskellAcceptor`)

Let us use Parsec [LM01] – a popular, *monadic combinator library* for parsing in Haskell. All parser functions are of the following type:

```
type Acceptor = Parsec String () ()
```

Thus, we use the identity monad as the base monad, the stream type for parsing is set to `String`, the type for state along parsing is set to '()', and the result type is also set to '()' – as we will be concerned here with acceptance only. (There is the more complete *Contribution* `haskellParsec` that also constructs abstract-syntax trees according to the algebraic data types of subsection 5.4.2.)

Let us implement this sort of concrete syntax of departments:

```
department = "department" literal "{"
               manager
               subunit*
             "}"
```

The corresponding parsing function follows:

```
parseDepartment :: Parser
parseDepartment = do
```

```
parseString "department"
parseLiteral
parseString "{"
parseManager
many parseSubUnit
parseString "}"
```

Feature `Logging`

Requirement Specification

Updates of salaries are to be logged so that salary changes can be reviewed afterwards. Each log entry identifies the relevant employee by name and lists salary before and after update. In particular, such logging is to be supported for *Feature* `Cut`. Here is an example of a log in CSV format:

```
"Craig", 123456.0, 61728.0
"Erik", 12345.0, 6172.5
"Ralf", 1234.0, 617.0
...
```

The log is to be analyzed in a statistical manner to determine the median and the mean of all salary deltas. Such demonstration (testing) is omitted for the sake of brevity.

Chrestomathic Purpose

Logging updates require some level of *program instrumentation* so that a plain computation involving updates does indeed log those updates. Different programming techniques may be used for this purpose. Logging calls for *separation of concerns* such that logging should not affect parts of the program that are conceptually unrelated to logging. Indeed, logging is a favorite *aspect-oriented programming* scenario [KLM+97] in that aspects (advanced modules) may facilitate separation of concerns.

1st Haskell illustration (*Contribution* `haskellLogging`)

Logs may be modeled in Haskell as lists of entries as follows:

```
data LogEntry =
    LogEntry {
      name :: String,
      oldSalary :: Float,
      newSalary :: Float
    }
```

A knowledgeable Haskell programmer may immediately suggest the use of the *writer monad* for logging. However, let us start with a beginner's implementation. When learning about monads, a non-monadic implementation is definitely helpful. Consider, the following function for cutting salaries at the department level, with logging integrated:

```
cutD :: Department → (Department, Log)
cutD (Department n m ds es)
  = (Department n m' ds' es', log)
  where
    -- Cut the manager's salary
    (m',log1) = cutE m
    -- Cut all salaries in the sub-departments
    (ds', logs2) = unzip (map cutD ds)
    -- Cut all salaries of all immediate employees
    (es', logs3) = unzip (map cutE es)
    -- Compose intermediate logs
    log = concat ([log1]++logs2++logs3)
```

Unfortunately, the need for logging affects the overall processing of departments; see the occurrences of the idioms for grouping company data with a log, also implying *zipping* and *unzipping* in the case of lists.

2nd Haskell Illustration (*Contribution* `haskellWriter`)

A particular *monad* [Wad92], the *writer monad*, can be used to encapsulate logging. Functionality involving salary updates is to be converted to monadic style. We revise *Contribution* `haskellLogging` as follows:

```
cutD (Department n m ds es) =
  do
    m' ←cutE m
    ds' ←mapM cutD ds
    es' ←mapM cutE es
    return (Department n m' ds' es')
```

The function is oblivious to logging; it is prepared for any effect.

Stakeholders of the *101companies* Project

A stakeholder of the *101companies* project is someone who affects or is affected by the project or could be expected to do so. There are *users* of the project: *learners* subject to self-learning, professional training, etc. who use the project to learn about software technologies and languages as well as *teachers* in university or professional education who use the project to prepare their courses, lectures, etc. Further, there are contributors to the project: *developers* of implementations or specifications of the *101companies* system, *authors* of wiki content including classifications of software technologies and languages, *community engineers* who manage the project from a Research 2.0 perspective, and yet other kinds of contributors. There are also stakeholders who may be interested in the project more broadly because they are *researchers* in a relevant context (such as ontology engineering or software linguistics) or *technologists* (such as owners of a software technology). Stakeholder roles are non-disjoint. For instance, a technologist might be expected to also serve as an educator (a teacher) as well as a contributor.

The classification tree of stakeholders is shown in figure 5.7.

101stakeholder	a stakeholder of the *101companies* project
- ***101contributor***	anyone who contributes to the *101companies* project
– ***101advisor***	anyone who serves on the advisory board of the project
– ***101author***	anyone who authors content for the wiki of the *101companies* project
– ***101developer***	anyone who develops a contribution to the *101companies* project
– ***101engineer***	anyone who contributes to the infrastructure of the *101companies* project
– ***101gatekeeper***	anyone administering wiki and repository of the *101companies* project
– ***101research20er***	anyone who contributes as a community engineer to the *101companies* project
– ***101reviewer***	anyone who reviews a *contribution* to the *101companies* project
- ***101researcher***	anyone interested in research on software technologies and languages
– ***101linguist***	anyone researching software linguistics
– ***101ontologist***	anyone researching *ontologies* for software technologies and languages
- ***101technologist***	anyone seeking technology adoption through the *101companies* project
- ***101user***	anyone who uses the *101companies* project
– ***101learner***	anyone who leverages the *101companies* project for learning
– ***101teacher***	anyone who leverages the *101companies* project for teaching

Figure 5.7: Stakeholders of the *101companies* project

Key Categories of the *101companies* Ontology

The ontology classifies all entities that are relevant for the *101companies* project. The categories (or classes or concepts) are organized in two dimensions. In the first dimension, we distinguish between general entities that can be said to exist regardless of the project (such as technologies) versus project-specific entities (such as the features of the 101companies system). In the second dimension, we distinguish between primary entities versus subordinated entities. While the first dimension is profound, the second dimension is only introduced for convenience of consuming the classification. The categories for primary, general entities are *Technology* for the deep classification of actual software technologies, *Capability* for the deep classification of capabilities of technologies, *Language* for the deep classification of actual software languages and *Space* for the enumeration of actual technological spaces. When adding a new technology to the ontology, then classifiers from the *Technology* tree are to be applied and the technology may also be associated with capabilities, languages and technological spaces. When documenting a *contribution* to the *101companies* project, the contribution is to be associated with technologies and languages. Classification and association help the *users* of the *101companies* project to navigate between software technologies, capabilities thereof, software languages, technological spaces, and contributions of the project. In figure 5.8, the categories of technologies and languages are broken down into (only the immediate) subcategories for the classification of

such entities; also, actual technological spaces are revealed as members of category *Space*. Specific technologies and languages may be members of multiple subcategories, and they may be associated with multiple technological spaces.

Technologies may be subdivided into *development* or *application technologies* depending on whether they target the developer by providing some kind of tool support or the application by providing some kind of reusable components. For instance, *IDEs* or *tools* count as development technologies whereas *libraries* or *frameworks* count as application technologies. Classification of technologies may also apply to their possible status of being a *programming technology* in the sense that they serve specific programming domains; consider, for example, *web technology* or *data technology*. Given the central role of *technological spaces*, classification of technologies may also apply to their possible status of being a *mapping technology* across spaces. Finally, some technologies specifically support some software language, giving rise to further classification according to *language technology*: consider, for example, *compilers* or *program generators*. All these categories of technologies may be broken down further into subcategories. Some technologies may be naturally instances of multiple categories. Technologies are further characterized by their *capabilities*.

Technologies are seen as providing capabilities to the developers or the systems that use the technology. Examples of capabilities include *logging*, *serialization*, *data parallelism*, and *mapping*. Thus, each specific technology is not just classified according to technology subcategories, but it is also to be associated with capabilities. For instance, the *mapping* capability further breaks down into *Object/XML mapping*, *Object/Relational mapping*, etc. Each specific technology can be indeed associated with several capabilities. For instance, *JAXB* provides the capabilities of both *Object/XML mapping* for Java and (XML-based, *open*) serialization.

Themes of *101companies* Contributions

A theme is an explicitly declared group of contributions to the *101companies* project. Themes are meant to help users of the *101companies* project efficiently consume knowledge about contributions, software technologies, and capabilities, software languages, and technological spaces.

Programming technologies	
Technology	a software technology
- ***Application technology***	a technology that is reusable in software applications
- ***Development technology***	a technology that is used in software development
- ***Language technology***	a technology that is dedicated to one or more *software languages*
- ***Mapping technology***	a technology for mapping between *technological spaces*
- ***Programming technology***	a technology that is dedicated to a certain *programming domain*

Software languages	
Language	a software language
- ***Domain-specific language***	a software language that addresses a specific domain
- ***Format language***	a software language that defines a representation format
- ***Markup language***	a software language that facilitates the annotation of text
- ***Metadata language***	a software language that facilitates the addition of metadata to artifacts
- ***Metalanguage***	a software language to define software languages
- ***Modeling language***	a software language to express information or knowledge or systems
- ***Programming language***	a software language for implementing programs
- ***Query language***	a software language for executable queries
- ***Scripting language***	a software language that is used to control applications
- ***Style sheet language***	a software language for presenting structured documents
- ***Tool-defined language***	a software language that is effectively defined by a tool
- ***Transformation language***	a software language for executable transformations
- ***XML language***	a software language that uses XML for representation

Technological spaces	
Space	a community and technology context
- ***Fileware***	a technological space focused on sequential and indexed files
- ***Grammarware***	a technological space focused on (textual) language processing
- ***Lambdaware***	a technological space focused on functions and *functional programming*
- ***Modelware***	a technological space focused on modeling and model-driven engineering
- ***Objectware***	a technological space focused on objects and OO *programming*
- ***Ontoware***	a technological space focused on ontologies and knowledge engineering
- ***Relationalware***	a technological space focused on relational databases
- ***XMLware***	a technological space focused on XML representation and XML processing

Figure 5.8: The key categories of the *101companies* ontology

To this end, themes are tailored towards interests of specific stakeholders. Such specificity translates into focus on a certain technological space, a category of technologies, or a certain programming language. For instance, the Haskell theme addresses interests of those who want to approach Haskell through the *101companies* setup as well as those who want to approach the *101companies* setup on the grounds of Haskell knowledge.

Themes should be of a manageable size: 4-10 contributions per theme. Accordingly, the composition of a theme needs to be selective in identifying theme members. For instance, the XML theme covers presumably all fundamental approaches to XML processing, but it leaves out variations in terms of APIs and languages. Such variations can still be discovered easily by users because contributions are richly tagged and cross-referenced. Appendix A contains the

comprehensive list of themes.

Java mapping: *Java* theme of implementations that travel technological spaces	
Description	
Subject to appropriate bridges, i.e., subject to *mapping* facilities, any programming language can be made to access and process *models*, *XML*, relational database *tables*, and *text* (concrete syntax) in a type-based (say, schema-aware or metamodel-aware or grammar-aware) manner. The present theme collects corresponding implementations for the programming language *Java*.	
Members	
- antlr4Acceptor	An ANTLR4-based acceptor for textual syntax
- antlr4Lexer	Lexer-based processing with *ANTLR4*
- antlr4Objects	Object/Test mapping for Java with *ANTLR4* for parsing
- antlr4ParseTreeListener	Parsing text to trees and walk them with *ANTLR4* Listeners
- antlr4ParseTreeVisitor	Parsing text to trees and walk them with *ANTLR4* Visitors
- antlr4Parser	Processing textual syntax with semantic actions of *ANTLR4*
- antlrAcceptor	An ANTLR-based acceptor for textual syntax.
- antlrLexer	Lexer-based processing with *ANTLR*
- antlrObjects	Object/Text mapping for *Java* with *ANTLR* for parsing
- antlrParser	Processing textual syntax with semantic actions of *ANTLR*
- antlrTrees	Parsing text to trees and walk them with *ANTLR*
- emfGenerative	Model-Object mapping for *Ecore* and *Java* with *EMF*
- hibernate	Object-Relational mapping for *Java* with *Hibernate*
- jaxbChoice	Object-XML mapping for *Java* and *XSD* with *JAXB*
- jaxbComposition	Object-XML mapping for *Java* and *XSD* with *JAXB*
- jaxbExtension	Object-XML mapping for *Java* and *XSD* with *JAXB*
- jaxbSubstitution	Object-XML mapping for *Java* and *XSD* with *JAXB*
- jdom	Process *XML* data with Java's *JDOM* API
- sax	Push-based *XML* parsing in *Java* with *SAX*
- xom	Exercise in-memory *XML* processing with *XOM* in *Java*
- xpathAPI	Exercise *XML* processing with *XPath* embedded in *Java*

Figure 5.9: Java theme of 101implementations

Linking Documentation and Source Code

101companies Chrestomathy – Inventory

Separation between documentation and source code

Programming chrestomathies may easily embed source code fragments directly into the documentation repository (such as a wiki). In contrast, software chrestomathies have to use a source code repository to organize, persist, and maintain all software artifacts. Thus, an additional documentation repository (which may also be based on a wiki) is used on top. Likewise,

(This is not a UML diagram.) This picture illustrates informally the differences between the constituents of traditional software products and those of software chrestomathies. The model will be further refined in this chapter, leading progressively to an accurate UML class diagram in figure 5.13 specifying precisely the problem to be solved and a UML deployment diagram in figure 5.14. The different kinds of links are drawn in different colors and modes (rectilinear, dotted, oblique) to emphasize their fundamentally different nature.

Legend

Family of *actual* associations (coded knowledge), e.g., calls, imports, and other inter/intra-artifact references.
Family of *expected* associations (documented knowledge), e.g., *subsystemImplemlentsFeature*.
Family of associations to *establish*, e.g., *classPertainsToSubsystem* and *functionImplementsFeature*.

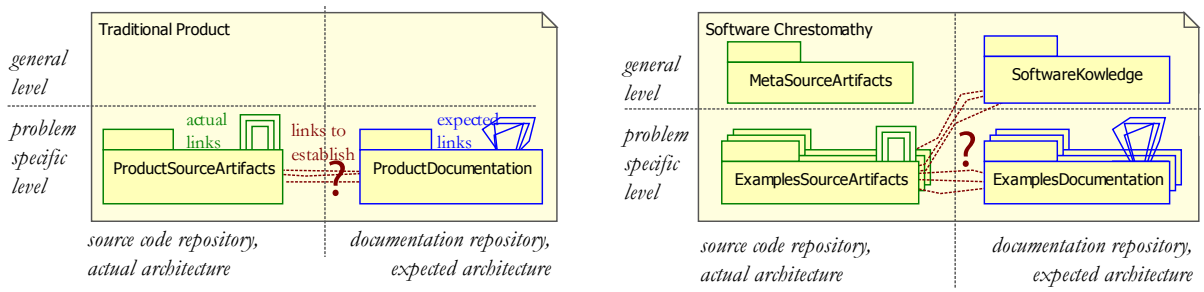


Figure 5.10: Linking in the context of traditional software products vs. software chrestomathies.

source code and some forms of documentation are commonly separated for traditional software products; see figure 5.10.

Product-specific versus general level

Software chrestomathies differ from traditional software products in an important way as depicted in figure 5.10 by the *general level* with the explicit materialization of *software knowledge* and *meta-source artifacts* that help with managing such knowledge. In the case of *101companies*, software knowledge is represented as wiki pages containing information about software languages, software technologies, and software concepts. Gathering and organizing such knowledge is an integral, non-trivial objective of the chrestomathy. As suggested by the dotted associations on the right of figure 5.10, the problem of establishing links is no longer limited to links between source code and its documentation, but links between source code and software knowledge have to be established as well. For instance, one may want to establish that a particular file is valid according to a given language, that a particular file fragment uses a particular technology, or that a particular set of files constitutes an instance of a software concept, such as the MVC pattern. Meta-source artifacts help with establishing such links and with processing

the samples of the chrestomathy otherwise. For instance, one kind of meta-source artifact may validate files to pertain to certain languages.

Metamodels for the 101companies Chrestomathy

On the left, an informal metamodel of *101repo*, the GitHub-based source-code repository, is presented. On the right, an informal metamodel of *101wiki*, the MediaWiki-based wiki, is presented. The association corresponding to the *links to establish* will be specified in figure 5.13. Elements drawn in green and in rectilinear mode correspond to the actual architecture. Elements drawn in blue and in oblique mode correspond to the expected architecture. Elements in red and marked with '?' are links to establish.

Legend

Family of *actual* associations corresponding to links buried in source artifacts.
Individual *actual* UML-style associations corresponding to available facts.
Individual *expected* UML-style associations corresponding, for example, to links on the wiki.
Family of associations to *establish* to be made precise in the next UML diagram.

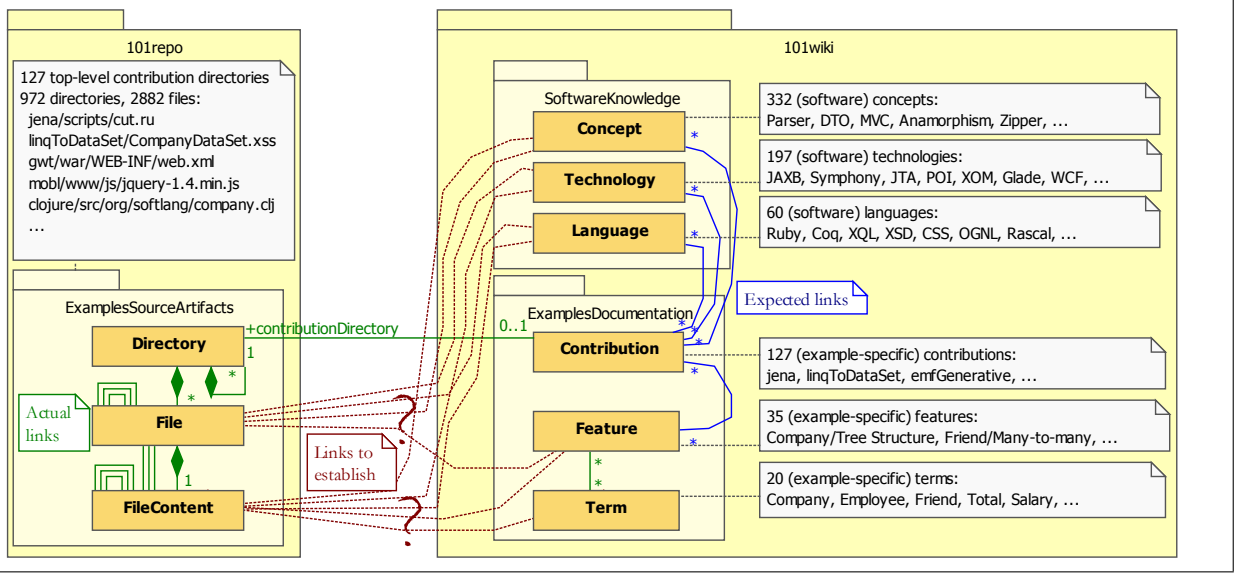


Figure 5.11: Informal megamodel of *101repo* and *101wiki* with links.

In figure 5.11, the repositories and constituents of the *101companies* project are described by simplified and informal metamodels. We take the view that the instances of the classes on the left are materialized by actual directories and files in the *101repo* source-code repository, while the instances of the classes on the right are materialized by wiki pages on the *101wiki*. Examples of names of such instances are provided as an illustration. The total number of instances (at the time of writing) is also given for each class.

Consider the top-level directories in the middle of the figure: 'jena', 'linqToDataSet', etc.

correspond to different variants of the *101companies* system, called *contributions* (or *implementations*).

Consider the metamodel for the source-code repository on the left of figure 5.11. The hierarchical nature of the file system is modeled in a straightforward manner. Files are also decomposed into content (or fragments), which is important if, for example, the expected architecture makes claims at the fragment level.

Consider the metamodel for the wiki on the right of figure 5.11 rooted in the *Contribution* class whose instances document contributions. Physically, instances correspond to wiki pages decomposed into sections that can be represented in the metamodel by attributes such as "heading", "description", or "issues". The details of this metamodel are out of the scope of the present paper, but it is important to note that contributors are required to document all software languages, software technologies, and software concepts used in their contributions as well as the implemented product features. This leads to the *expected* links on the right of figure 5.11. There is no guarantee, though, that these "claims" comply with the actual architecture.

Opportunities for Links in the 101companies Chrestomathy

The problem to be solved is therefore to compare the actual architecture buried in *101repo* with the expected architecture documented in *101wiki*. Such reconciliation begins at the roots with the classes *Directory* and *Contribution*. The corresponding links are trivially established because contributions have the same name in *101repo* and *101wiki*. At the component level, one has to dive into the file system hierarchy and possibly into file contents to establish the remaining links. Some links may be established generally on the grounds of rules that capture rules of usage or, at least, best practices for languages and technologies. For instance, there is a rule for files with a file extension '.rb' to imply that the file uses the *Ruby* language. Other links may require rules that are specific to a contribution and possibly software artifacts thereof. For instance, a rule may express that a specific class of a specific contribution associates with the software concept *Parser*.

The Exploration Use Case

Before we discuss what information needs to be prepared, we should consider the different stakeholders related to the enriched chrestomathy (developers, teachers, learners, and so sorth) and their needs. For brevity, we focus on the central use case of *exploration*, which is of interest to all stakeholders. That is, consumers of and contributors to the *101companies* chrestomathy want to navigate contributions in the hierarchical sense of the repository (i.e., directories, files, and fragments) while also observing relevant knowledge about software languages, technologies, and concepts, thereby also taking advantage of additional navigation paths.

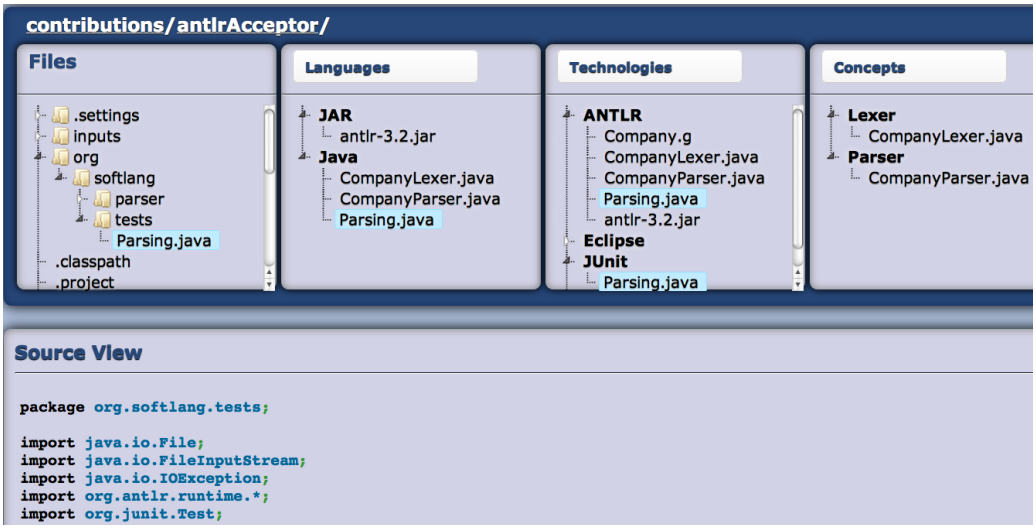


Figure 5.12: Exploration of the *101companies* implementation *antlrAcceptor*.

Figure 5.12 shows a snapshot of the *101explorer* tool, which provides designated support for the exploration use case. The figure snapshots some state of exploration for a specific contribution *antlrAcceptor*, which was designed to demonstrate the *ANTLR* parser generator in a *Java* setup. The four panels show the files, languages, technologies, and concepts for the contribution. Each listed language, technology, and concept is also associated with the files that justify the listing. For instance, the file `Company.g` is associated with the technology *ANTLR* because this file is an input of *ANTLR* as hinted at by the '.g' extension. Likewise, some '.java' files are associated with the software concepts *Parser* and *Lexer*.

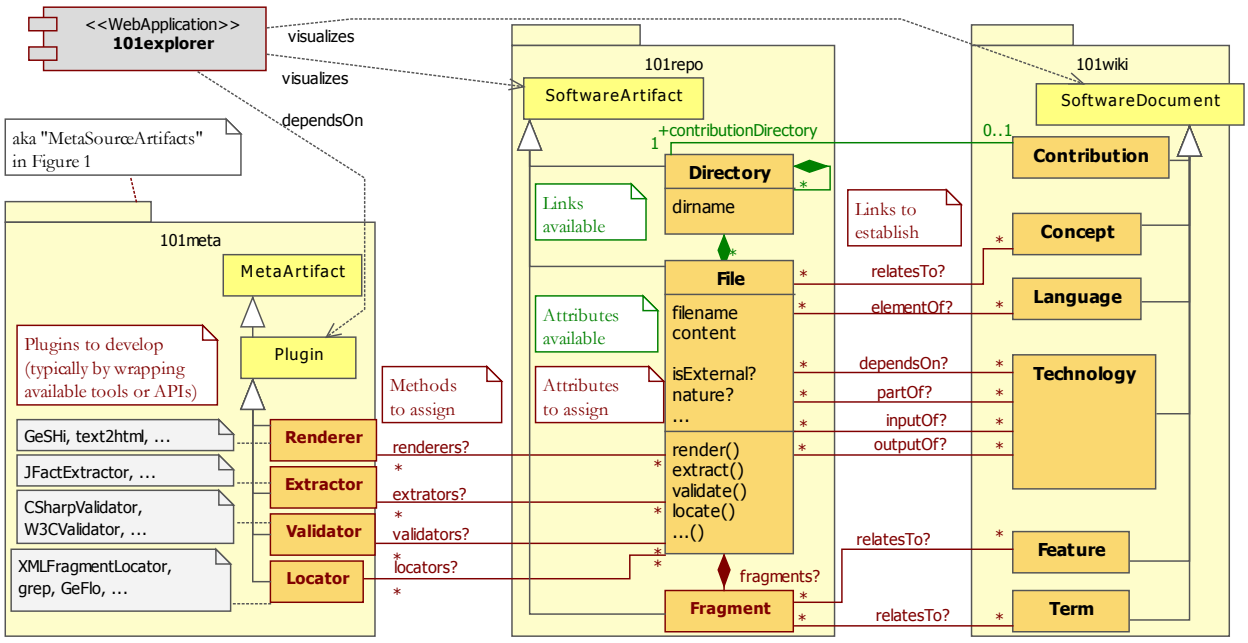


Figure 5.13: Information of interest. (Some attributes and associations were omitted for brevity and clarity.)

Specification of the Information of Interest

We need to specify in more detail what information to extract, to assign, and otherwise to prepare in support of the exploration use case specifically; see figure 5.13. One can view all such information as *metadata* to be associated with directories, files, and file fragments. Association of metadata may rely on automated, generic rules (e.g., based on matching suffixes of filenames) or it may require contribution- or file-specific rules or declarations.

Classification of Metadata

- **Links to establish.** The links define, for instance, which languages and technologies are used (in the sense of megamodeling or linguistic architecture [FLV12]), and also which software concepts, features and (domain) terms are involved; see the associations 'elementOf', 'inputOf', and others on the right-hand side of figure 5.13.

- **Attributes to assign.** The attributes help with processing source-code artifacts: see the middle of figure 5.13. For instance, the "nature" of each file (e.g., binary, source file, or archive) is used by the *101explorer* to determine whether the content of a file should be displayed or not.

- **Methods to assign.** Files are associated with methods (e.g., for 'rendering' [i.e., producing output that can be presented to users], 'validation' [i.e., establishing that a file is valid according to some language], 'fact extraction', or 'fragment location'). These methods are plugged into the framework as shown on the left-hand side of figure 5.13.

Rule-based Metadata Assignment

Our approach relies on a rule-based system to assign metadata to software artifacts so that links are established and further processing of the artifacts is instructed. The rule-based system is effectively supported by the domain-specific language *101meta*[2]. In the current implementation of *101meta*, rules are represented in JSON, but we use a simple concrete syntax in this paper for clarity's sake. The language is evolving, as new features are requested, but the assumed main elements are presented here, first using a grammar, then through a series of examples.

The *101meta* Language

Rules can be collected in sets. Each rule consists of a condition (on the left-hand side), an optional scope (to deal with fragment-level metadata), and a set of metadata units to be assigned (on the right-hand side). Thus:

RuleSet ::= *Rule*∗ .
Rule ::= *Condition* → [*AssignmentScope*] *Assignment*∗ .
AssignmentScope ::= ***fragment*** *FragmentDesignator* .

Operationally, rules are to be checked on files (and directories). Whenever a file meets the condition, then the corresponding metadata units are assigned to it. In fact, when a fragment designator is specified, then the metadata units are effectively assigned to the specified fragment instead.

Conditions can be formed from Boolean connectors and basic constraint forms:

Condition ::= *Constraint* | ¬ *Constraint* | *Constraint* ∧ *Constraint* | ... .
Constraint ::=

[2] `http://101companies.org/Language:101meta`

```
  filename (String | RegExp)
| basename (String | RegExp)
| dirname (String | RegExp)
| suffix String
| content (String | RegExp)
| predicate Command .
```

The **filename** constraint allows matching on a given filename either via a simple string or a regular expression. The constraints **basename** and **dirname** correspond respectively to a constraint on the filename without any directory part or on the directory name. The **suffix** constraint is essentially a shorthand for a pattern to constrain only the suffix (typically, the extension) of a filename. The **content** constraint is used to express conditions on the content of files by regular expression matching. Finally, the **predicate** constraint makes it possible to perform arbitrary computations for conditions by applying an executable command to the file under investigation, subject to an interpretation of the exit code. (We think of all such commands as being plugged into the framework.)

In the *101companies* project, so far, the following forms of metadata assignments have been found useful, but the approach is obviously extensible in that new forms of metadata could be added easily:

```
Assignment ::=
    LinkAssignment
  | AttributeAssignment
  | MethodAssignment .
LinkAssignment ::=
    elementOf LanguageName
  | dependsOn TechnologyName
  | partOf TechnologyName
  | inputOf TechnologyName
  | outputOf TechnologyName
  | concept ConceptName // association relatesTo
  | feature FeatureName // association relatesTo
  | term TermName . // association relatesTo
```

```
AttributeAssignment ::=
    external // attribute isExternal
  | nature String . // attribute nature
MethodAssignment ::=
    renderer Command
  | extractor Command
  | validator Command
  | locator Command .
```

These forms directly correspond to the specification of subsection 5.8.3 and figure 5.13 specifically.

Language Links

Here are a few, very simple examples:

```
suffix ".jar" → nature "archive" .
suffix ".xq" → nature "source" .
suffix ".xq" → elementOf "XQuery" .
```

The rules states that '.jar' files are archives, while files with suffix '.xq' are source files pertaining to the *XQuery* language. Such information can be used by tools such as the *101explorer* or metrics tool as only source files should be rendered in the browser and count in metric calculations.

There is an important conceptual difference between attribute assignment (first two rules) and link assignment (third rule): in the first case 'archive' and 'source' are just scalar values, but 'XQuery' is actually representing a reference to a conceptual entity. Since the class *Language* is the target of the 'elementOf' association (see figure 5.13), the string 'XQuery' must constitute a valid language name in the ontology of the *101companies* project. In particular, language names can be completed in URLs on *101wiki*.[3] In this manner, tools like *101explorer* can interpret metadata units for the purpose of navigation. We mention in passing that metadata assignments can be represented as RDF triples, subject to an appropriate megamodeling ontology [FLV12].

[3] *XQuery* maps to `http://101companies.org/index.php/Language:XQuery`.

Technology Links

Let us consider also links to technologies as opposed to languages. The parser generator *ANTLR* is used here for illustration. When *ANTLR* is used with *Java*, the technology is packaged as a '.jar' archive. Hence, let us associate, for example, the (version-specific) file 'antlr-3.2.jar' with the technology *ANTLR*.

```
basename "antlr−3.2.jar" partOf "ANTLR" .
```

The **basename** constraint implies that we do not care about the directory of the matched file here. We use **partOf** here in the sense that a software artifact, such as a '.jar' archive, can be considered part of a technology, which is a conceptual (abstract) entity [FLV12]. We may also perform regular expression matching on the **basename** to cover all possible versions of the '.jar' file:

```
basename "#^antlr−.*\.jar$#" → partOf "ANTLR" .
```

Let us also consider indicators of technology usage. The **suffix** '.g' is an indicator of *ANTLR* usage because this extension is used for grammar files.

```
suffix ".g" → inputOf "ANTLR" .
```

Link assignment expresses here that the matched files serve as input for the parser generator *ANTLR*. The use of *ANTLR* may also be inferred on the grounds of generated files. When *ANTLR* is used in a common manner, then generated code for parser and lexer are to be found in files with specific names as follows:

```
basename "#^.*(Parser|Lexer)\.java$#" → outputOf "ANTLR" .
```

Actually, the use of 'Parser' or 'Lexer' in filenames does not generally imply usage of *ANTLR*. Thus, we need to further constrain the rule in a way that the content of the files can be checked to support the assumption about *ANTLR* usage. Specifically, looking at files actually generated by *ANTLR*, a simple signature stands out in the first line:

```
// $ANTLR 3.2 Sep 23, 2009 12:02:23 Company.g .
```

This is indeed enough here to help with decision making. We would like to 'grep' the file to search both for the '$ANTLR' string and the distinguished extension '.g' in the same line.

```
basename "#^.*(Parser|Lexer)\\.java$#"
∧ content "#// \$ANTLR.*\\.g#"
  → outputOf "ANTLR" .
```

Another kind of evidence for *ANTLR* usage concerns imports of its runtime API 'org.antlr.runtime'.

```
suffix ".java"
∧ content "#^[ \t]*import[ \t]*org.antlr.runtime\.#"
  → dependsOn "ANTLR" .
```

Clearly, such import matching could be useful for many other technologies, in fact, APIs. Thus, we may also factor such matching into a more general purpose predicate:

```
#!/bin/sh
# usage: javaImportPredicate.sh <package> <javaFile>
grep -q "^[ \t]*import[ \t]*$1\." $2
```

Thus, we revise the rule to invoke the predicate instead of using a content constraint:

```
suffix ".java"
∧ predicate "javaImportPredicate.sh org.antlr.runtime"
  → dependsOn "ANTLR" .
```

We may later decide to re-implement the predicate at a syntax-aware level as opposed to regular expression matching.

Concept Links

We may also want to annotate files with any software concepts of the *101companies* chrestomathy. For instance, we may want to express that certain files define a parser, a GUI, or contribute to a MVC architecture. Here is a concrete example where files of a specific contribution, *antlrObjects*, are tagged with the **concept**s *Parser* and *Lexer*:

```
filename "antlrObjects/org/softlang/parser/CompanyParser.java"
```

```
  → concept "Parser" .
filename "antlrObjects/org/softlang/parser/CompanyLexer.java"
  → concept "Lexer" .
```

More general rules may also be conceivable here.

Links Related to the *101companies* Domain

We can also assign **feature**s of the *101companies* system as well as **term**s of the domain to files and fragments. For instance:

```
filename "javaStatic/org/softlang/behavior/Total.java" →
    feature "Type−driven query"
    term "Total" .
```

Method Assignments

The preparation of information, as needed for the exploration use case requires several methods, as discussed in subsection 5.8.3. The assigned methods may be invoked by functionality such as the *101explorer*, which operates on the matched file system. There are rules for method assignment for many suffixes and 'special' filenames. For instance:

```
suffix ".wsdl" → renderer "php geshiRenderer.php xml" .
basename "Makefile" → renderer "php geshiRenderer.php text" .
```

That is, renderer methods are assigned to '.wsdl' files and makefiles. The Generic Syntax Highlighter, *GeSHi*[4] is leveraged here; this is a PHP package supporting HTML generation for source files for more than 200 languages with awareness of keywords, strings, comments, and others. WSDL files are rendered as XML files; see the 'xml' argument being passed; makefiles are rendered as plain text files; see 'text'.

Eventually, we may also want to use more advanced renderers than *GeSHI*. For instance, we could use a more WSDL-aware renderer that supports navigation for some of the WSDL elements.

[4] `http://qbnz.com/highlighter/`

In the following example, we register Python programs for validating *Java* source code and extracting facts from the code.

```
suffix ".java" →
  extractor "JFactExtractor.py"
  validator "JValidator.py" .
```

Extractors extract facts from source code (e.g., the declared classes or the imported packages in the case of *Java*). The facts can be used in various ways (e.g., for the purpose of establishing technology-related links). Validators are meant to validate assumptions about files (e.g., to pertain to certain languages). Validators can also be remotely invoked. For instance, '.html' files may be validated by a script which wraps an online service provided by the W3C consortium.

Validation may seem very similar to the predicate form of conditions; see subsection 5.8.6. However, predicates serve for matching conditions whereas validators serve for the validation of committed matches. Unsuccessful matching (based on predicates) is to be expected; unsuccessful validation pinpoints an issue with software artifacts or rules, and hence, user intervention may be required.

Fragment Scope

In all examples, so far, we really meant to annotate complete files. In general, it may be necessary to limit the scope of metadata to apply only to file fragments. To this end, a fragment designator has to be used as a scope of assignments. Consider, for example, the data model for companies, as defined by a trivial *Haskell*-based contribution; one file contains all the data types for companies, departments, and employees:

```
module Company where
data Company = Company Name [Department]
data Department = Department Name Manager [SubUnit]
data Employee = Employee Name Address Salary
```

We would like to point to all the specific domain terms 'Company', 'Department', and 'Employee'. The following rule involves fragment designation to link the appropriate **fragment**

(i.e., the data type 'Company') to the **term** 'Company':

```
filename "haskell/Company.hs"
 → fragment { "data" : "Company" } term "Company" .
```

We rely on language-aware support for fragment location. In the example, we rely on a *Haskell*-specific locator, which is known due to the following method assignment:

```
suffix ".hs"
 → locator "HsFragmentLocator.py" .
```

There is also a more lexical and generic approach to fragment selection based on GeFLo,[5] a *101companies*-specific technology for generic fragment location, which, in turn, is based on GeSHi.

Summary of *101meta* Usage

Various derived resources are available online [6]; some of them also directly illustrate and measure the use of *101meta* in the chrestomathy. Here are some illustrative numbers, recorded at the time of writing:

- Examined 2910 repository files.
- Gathered 307 *101meta* rules.
- Assigned metadata to 2030 files.
- Performed 6778 metadata assignments.
- Mapped 41 suffixes (pertaining to languages).
- Mapped 57 *Java* packages (accounting for APIs).

Rule execution for matching and subsequent phases for validation, rendering, etc. are performed continuously on a worker machine of the *101companies* project.

[5] `http://101companies.org/index.php/Technology:GeFLo`
[6] `http://worker.101companies.org/data/resources/`

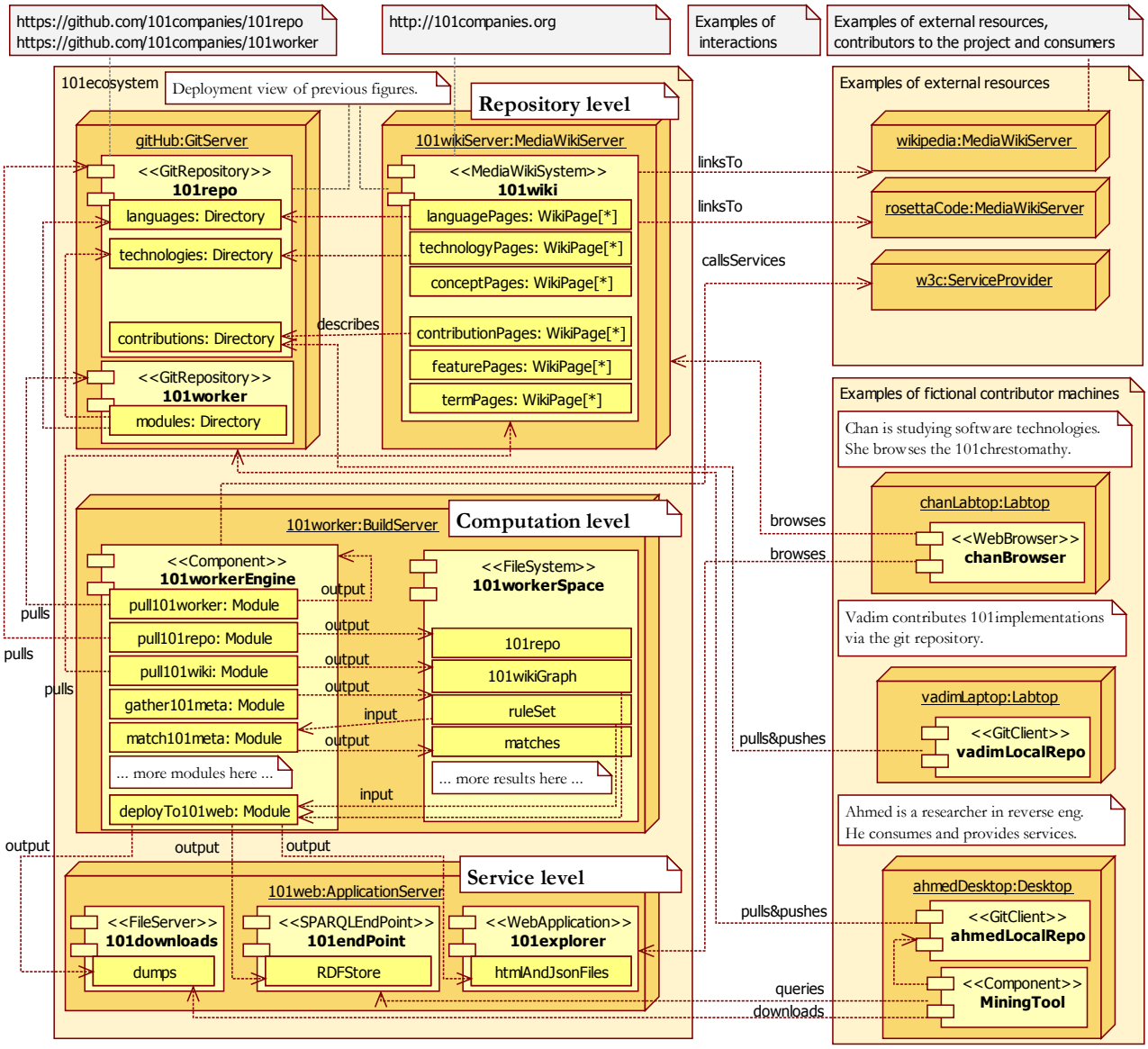


Figure 5.14: Architecture of the *101ecosystem* (on the left) with examples for providers, consumers, and resources (on the right).

The *101ecosystem*

It remains to integrate the analysis technique of the previous section into a framework that supports the different stakeholders and objectives of a software chrestomathy. The UML diagram of figure 5.14 describes the resulting ecosystem of the *101companies* project (see on the left) and examples of external resources, consumers or producers machines (see on the right). The ecosystem is an instance of the notion of metaware environment with meta-usecases according to [Fav04b]. Each node (3D box) in the diagram corresponds to a different machine (virtual or not).

The *101ecosystem* consists of three levels corresponding to the flow of data from top to bottom. At the *repository level*, the *101repo* and *101wiki* repositories of the chrestomathy are located. The *101meta* rules reside in *101repo* directories for specific languages, technologies, or contributions.

At the *computation level*, a build server (the so-called *101worker* machine) continuously executes modules to process the repositories so that information is extracted, computed, validated, and prepared for presentation. For instance, the module *match101meta* executes all gathered *101meta* rules and performs the assignment of metadata for *101repo*. The use of a designated build machine like this is a common architectural pattern.

At the *service level*, the results of the computations are served through endpoints of different kinds. For instance, the repository is surfaced in a form that is ready for browsing; results of matching are surfaced in a form that allows for human validation and provides assistance in producing additional rules; the repositories are surfaced in different fact-extraction formats to facilitate query techniques (e.g., based on SPARQL for RDF).

Let us consider the scenarios (on the right) of the figure; they are based on fictional characters playing different roles.

Chan takes advantage of the knowledge contained in the chrestomathy and uses the *101explorer* to visualize information.

Vadim actively contributes to the project by providing a new implementation of the *101companies* system, demonstrating how the languages and technologies he has designed could be used to solve software development problems.

Ahmed is interested in reverse engineering technologies. He is actively developing his own analysis tools and tests them on the chrestomathy taking advantage of direct access to extracted facts by either downloading dumps in different formats or querying through the SPARQL endpoint. His tools (or parts thereof) could later be reused by the *101ecosystem* for the benefit of the community.

Related Work

Ontologies. Ontologies are widely used for knowledge management [CJB99, SSSS01], and we have started to develop a *101companies* ontology for organizing technologies, languages, technical spaces, implementations of the *101companies* system, the system's specification, and all documentation in the *101companies* project. A few related uses are the following. The semi-automatic derivation of an ontology for domain-specific programming concepts—as they are supported, for example, by APIs for XML or GUI programming—is described in [RFD+08]. Clearly, such domain-specific ontologies are needed as modules of a broader ontology of programming technologies, which we develop in chapter 7.

Wikis. In [VKV+06], semantic enhancements to *Wikipedia* are described: an extension to the wiki-link syntax and an enhancement of the article view to cover visualization of semantic data. Instead of such a powerful approach, we currently leverage more standard capabilities of MediaWiki. That is, we basically use "categories" for classification, and we provide a number of customized views and synthesized article elements that help in understanding and navigating the *101companies* wiki.

An even more advanced approach is described as OntyWiki in [ADR06]. In this approach, "informational maps" visualize semantic content; each node at the information map is represented visually and interlinked with related digital resources. Users are further enabled to change the

knowledge schema and to contribute instance data. Our approach essentially relies on the much simpler notion of a category tree (in fact, a direct acyclic graph) for classification, which can also be modified and richly navigated.

Open-source contributors. Our proposal of contributor roles for the open-source *101companies* project is based on role sets that have been used in open-source projects, or that have been inferred or identified after the fact by studying open-source projects (e.g., [And11, KUM11, LPT06]).

In [ON08], two open source contexts are distinguished: software and content, and it is analyzed to what extent the entry barriers for contributors are different in the two contexts. While software contribution requires a certain threshold of expertise in order to pass the review process, content contribution has virtually no entry barriers other than basic computer literacy. To some extent, both roles are separated by the proposed contributor roles in the *101companies* project. However, ideally, a contributor of an implementation is also supposed to contribute content (so that the implementation is strongly documented).

Derivation of abstractions. Program comprehension and reverse engineering routinely involve the (semi-) automated derivation of models as abstractions over the source code. The Rigi system [TWSM94, TMWW93] serves here as a seminal example: the system implements automated techniques to compute and maintain architectural documentation from source code using the Rigi Command Language (RCL).

Our approach mainly aims at establishing links between documented, conceptual entities (i.e., languages, technologies, software concepts, domain terms, and product features) and implemented, physical entities (i.e., files and fragments thereof). Such link establishment can be seen as a sort of source-code summarization in the sense of [HAMM10].

Consistency between models and source code. Software reflexion models [MNS95, MNS01] also help software engineers to establish links between (high-level or design) models

and source code. The reflexion approach is concerned with traditional (as opposed to linguistic) architecture and focuses on summarizing differences and making the models work as a lens on the source code.

Multi-language analyses. Previous research has also aimed at tools and methods that apply to multiple languages without, however, addressing linguistic architecture. [MWHH06] describes a fact extractor and an analysis to understand cross-language dependencies in projects using scripting languages. [MSC+01] provides a web-based experience for analyzing C, C++, and Java programs. [KWDE98] targets multi-language systems by using a common, graph-based conceptual model for the involved languages. [KLW06] focuses on the formalization, management, exploration, and presentation of multi-language program dependencies. Moose [NL12] serves versatile exploration of source code for all languages with an import path to its FAMIX metamodel.

File extension usage in OSS. [KG11] examines the use of popular file extensions to analyze language usage along the evolution history of some open-source projects. The approach of this paper enables more diverse analyses of projects (e.g., in terms of different aspects of technology usage), and it assigns methods to files, thereby supporting functionality such as exploration or fact extraction.

Embedding approaches. Arguably, the link extraction and consistency problem would not exist, if source code and documentation were embedded into the same artifacts either based on *literate programming* [Knu84] (such that source-code fragments are embedded into the program documentation and subject to extraction for compilation) or the opposite approach of *documentation embedding* (such that documentation fragments are embedded into the source-code artifacts, subject to extraction for building the documentation). The latter approach is used by such popular technologies such as Javadoc[7] and Doxygen[8].

[7] `http://java.sun.com/j2se/javadoc/`
[8] `http://www.doxygen.org/`

These two embedding approaches are essentially program-centric in the sense that the integrated artifacts are aligned with program structure. A multi-language, multi-technology software product involves additional abstractions (e.g., ontologies of software development, domain glossaries, feature models, and functional and non-functional requirements). Embedding all such information in a single kind of artifact appears to be impractical in terms of both logistics (as different stakeholders are involved) and concise representation (as n:m relationships would imply duplication during embedding). In fact, some embedding approaches also integrate the notion of link (see `@see X` in Javadoc). Our work shows how to deal with a highly heterogeneous situation in terms of kinds of artifacts, languages, technologies, concepts, and kinds of links.

Metadata assignment. Metadata, specifically in the sense of annotations, is often used to support program comprehension and software maintenance [SCBR06, BGGN08] potentially even enabling sharing among developers. Popular tools such as the TagSEA system[9] or the Eclipse Resource Tagger[10] support tagging for the benefit of navigation and location finding. Our work is inspired by metadata approaches that describe metadata external to the addressed artifacts [SDdOV08, TS10].

Conclusion

The *101companies* chrestomathy collects contributions, which implement certain features of a feature model for an imaginary human-resources management system. The feature model is designed to touch upon many issues in programming and the use of software technologies. In particular we highlight the *101repo* – a federated repository to maintain all source code of the contribution with a version control system and *101wiki* – a 101-specific semantic wiki to maintain semi structured documentation of the contributions and related entities. A software chrestomathy involves entities that go beyond "conservative" source-code and documentation artifacts (i.e., languages, technologies, system features, software concepts, etc). Hence, a chrestomathy should be enriched "semantically" so that all the artifacts are linked to the

[9] `http://tagsea.sourceforge.net/`
[10] `http://taggerplugin.sourceforge.net/`

relevant concepts and rich exploration is enabled and the linked chrestomathy is amenable to some limited consistency checks. We have described a corresponding approach for enriching the *101companies* chrestomathy; it relies on the rule-based language *101meta* for metadata assignment. Also, the approach enriches the repositories of the chrestomathy with derived information resources and services as they are of interest for consumers of and contributors to the chrestomathy, culminating in the *101ecosystem*.

Chapter 6

Chrestomathic Knowledge Integration

In this chapter we describe the process of the software knowledge integration as a way to enrich the vocabulary of the *101companies* software chrestomathy. This chapter contributes to the requirements *R4. Vocabulary Engineering Through Knowledge Integration* and *R5. A Chrestomathic Ontology.*

Introduction

Software knowledge is available from many resources. We are specifically interested in (open online) community resources; think of *Wikipedia*, more domain-specific wikis, possibly open and online textbooks, forums like StackOverflow, the Apple Knowledge Base, or the support forums by Microsoft.

Each resource uses its specific vocabulary, in fact, its specific knowledge model. This hampers effective use of such distributed, complementary resources. For instance, consider the situation of a student who learns Haskell by consulting a textbook, *HaskellWiki*, and *Wikipedia*. Those three resources use different terms and different means of organization; these resources lack effective integration.

This chapter is based on the conference publication [LSV14] Ralf Lämmel, Thomas Schmorleiz and Andrei Varanovich. The 101haskell Chrestomathy – A Whole Bunch of Learnable Lambdas. In Postproceedings of IFL 2013, 2014.

We speak of knowledge integration here in the sense of the process that leads to the consistent interlinkage of the chrestomathy's documentation (on the *101wiki*) and appropriate external resources such as pages on external wiki pages or textbook paragraphs.

Knowledge integration is meant to be continuous in that, for example, more and more resources may be integrated over time and monitoring is applied so that the effective use of the integrated vocabulary is measured.

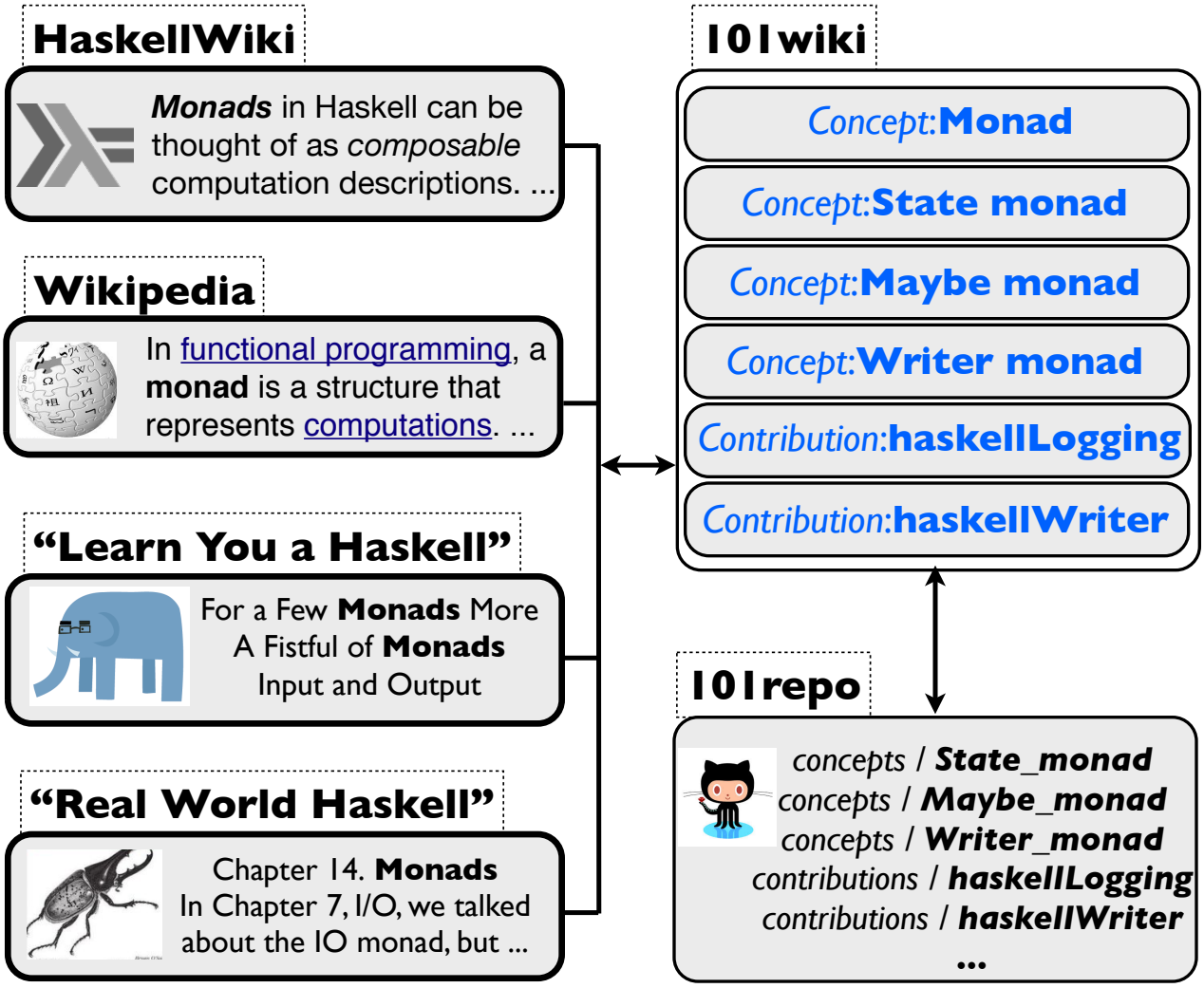


Figure 6.1: Illustration of knowledge integration for the '*Monad*' concept according to different knowledge resources.

Consider figure 6.1 for an illustration based on the programming concept *Monad* [Wad92]. The concept is reified on the *101wiki*.[1] The chrestomathy contains contributions that exercise monads (see chapter 5); there are also additional small examples in the *101repo*. The *101wiki*

[1] `http://101companies.org/wiki/Monad`

links to *Wikipedia*, *HaskellWiki*, and two online textbooks [Lip11, OSG08]. The *101wiki* page for monads is not meant to be comprehensive on the subject, but it links to textbook chapters for additional information about monads. For instance, several chapters of [OSG08] are linked to: *Monads. Monad transformers. Programming with monads. Error handling. The Parsec parsing library.*

In the present chapter, we explain the semi-automatic, primarily textbook-driven process for knowledge integration, as it was applied to *101haskell*. This process yields terms to be covered on the *101wiki* and links to wikis and textbooks, subject to text mining and summarization techniques. The process is supported by the designated *101integrate* framework[2] which helps with processing (textbook) resources for the raw vocabulary extraction, mapping "raw" terms to "consilidated" terms used on the wiki, and linking terms back to a resource's paragraphs in the interactive experience. A deliberate limitation of our approach is its assumption of a vocabulary with a manageable number of terms – amenable to human-based validation and content enrichment.

Selection of Textbooks

We begin with the selection of the resources. Selection was influenced by the assumed objective to cover Haskell in the *101kb* specifically in the context of teaching functional programming at the introductory level. Some of the more powerful programming techniques such as functors or monads should also be covered.

We determined that two popular textbooks were available online with open access: [OSG08, Lip11]. (We are not aware of any other Haskell textbooks with this profile.) So we favored these two books as we intended to produce a good experience in properly connecting to textbooks from the *101wiki*, especially for the benefit of students in the corresponding programming course.[3]

We decided to select more resources to increase diversity and to actually study the contributions

[2] `https://github.com/101companies/101integrate`
[3] `http://101companies.org/wiki/Course:Lambdas_in_Koblenz`

of different resources in a meaningful way. Thus, we selected the offline resource [Hut07] because it is an established introductory text with which we were also familiar through teaching. Further, we selected the offline resource [Tho11] because it is also an established text on Haskell; its mathematical or logical approach was thought of as being complementary to the other texts. In both cases, we were able to negotiate agreements with the authors so that we could indeed access and analyze offline sources for the books and publish the extracted vocabularies including some additional reporting information.

We apply continuous knowledge integration to these popular textbooks on functional programming in Haskell:

CRAFT [Tho11] "Haskell: The Craft of Functional Programming"

PIH [Hut07] "Programming in Haskell"

RWH [OSG08] "Real World Haskell"

LYAH [Lip11] "Learn You a Haskell"

Term Extraction

Index and content normalization. The textbooks are normalized (cleaned up) upfront. The rest of the extraction process is completely uniform for all books. The index and the raw content are extracted from each book, while performing resource-specific data cleaning steps so that formatting markup is eliminated. Source code was also excluded to eliminate related mining challenges.

The raw index terms and the raw content are subsequently stemmed [4]. As a result, one obtains a normalized set of raw terms that is free of trivial redundancy. Subsequently, the list of raw terms is reduced automatically by applying a threshold rank for "common English"[5]. The threshold was defined manually by inspection of the raw terms sorted by rank. This process

[4] We use the Natural Language Toolkit (NLTK [BLK09]).
[5] We use `http://www.wordcount.org` for "common English"

led to approximately 2000 index entries for the four books combined. Common English was also removed from the raw content.

Mining of candidate terms. All index entries are matched with all the content of all books. Match counts for the index entries are broken down for the book chapters. Based on these counts, candidate terms are determined by text mining and summarization [RBB97, AZ12], as explained shortly. Candidate terms are either *chapter terms*, *popular terms*, or *title terms*, as explained below.

Chapter terms are those terms per book and per chapter that are matched "most frequently" in a chapter (say, the most frequent five terms per chapter), that also appear "often enough" (say, at least three times) in one or more chapters of the book, but only in 'few' chapters (say, in 25 % of the chapters or less). This is a variation on inverse document frequency [Jon72] in that globally frequent terms are not selected; instead, locally frequent terms without many scattered occurrences are selected. Figure 6.2 shows a good part of the chapter terms for one of the Haskell textbooks. There is one row per term and one column per chapter. A bullet in a cell associates a term with a chapter. The size of the bullet represents matching frequency relative to other chapter terms.

Popular terms are those frequently matched technical terms that are actually scattered over many chapters; think of "function", for example. Such terms cannot be identified by inverse document frequency. Instead, we pick popular terms per book by ordering matched terms by frequency, excluding terms that are already covered by the chapter terms, and selecting all the more popular terms up to a threshold (such as the terms with >10 % of topmost term's frequency). It turns out that the different books agree on the popular terms to a good extent. For instance, all four books have "function" and "list" among the top-3 popular terms. Only very few terms were added in this manner; see Table 6.3.

Title terms are determined by manual inspection of chapter titles. That is, the central terms of a chapter's topic were added, if they were still missing. For instance, the chapter "Reasoning about programs" of [Tho11] has "induction", "proof", and "testing" as chapter terms, but

another central term, "equational reasoning", is missing. Only very few terms were added in this manner; see Table 6.3.

Term validation. Chapter terms and popular terms are validated to exclude "uninteresting" terms. The idea is here to focus on terms that concern functional programming, Haskell programming, or generally programming. A term could end up being uninteresting because it relates to a specific example in the book, which is not worth interlinking. For instance, the term "picture" appears frequently in [Tho11] for the sake of a specific example. Also, the application of a stop list for "common English" may have missed terms that are "common" in the specific technical text at hand. For instance, the term "model" is used frequently in [Tho11] in the style of "We can model a tournament by this type definition."

Table 6.1: Numbers of candidate terms

Book	Chapter terms	Title terms	Popular terms
CRAFT	52/55	12	2
PIH	25/31	6	3
RWH	65/72	8	4
LYAH	34/38	4	0

The discussed thresholds can clearly be used to control the number of candidate terms. As we mentioned before, it is important to aim at a manageable number because of the subsequent steps involving human intervention. Table 6.3 summarizes the numbers for the Haskell textbooks; the first column also shows the difference between chapter terms before and after validation.

Vocabulary Consolidation

At this stage, the overall objective is to map candidate terms from the books to a consolidated vocabulary on the *101wiki*.

Term mapping and reification. We systematically process each candidate term as follows. If the term is already present on the *101wiki*, then we proceed right to the next term. If the

term goes by a different name on the *101wiki*, then we record this correspondence so that an accordingly aggregated *mapping* can be used for interlinking wiki terms and resources. For instance, the term "class" appears as a chapter term of [Tho11]'s chapter "Overloading type classes and type checking." The term "class" would be overly ambiguous in a broader context of programming, which is the context assumed by *101.* Thus, we map "class" to "type class" on the wiki.

The remaining case is when we face a candidate term that is not yet present on the wiki. (Most of the textbook terms were initially indeed missing.) In this case, we need to reify ("make existent") the term on the wiki. To this end, we locate the term on *Wikipedia* or *HaskellWiki*. In this manner, each term on the *101wiki* is linked to yet another relatively standardized and stable URL (URI). We aim at locating a *Wikipedia* and/or *HaskellWiki* page for a concept that is the 'same as' the emerging concept on the *101wiki*.

Organization of sub-vocabularies. All terms are to be organized in a simple taxonomy of sub-vocabularies, thereby helping to understand the consolidated vocabulary. These are some of the sub-vocabularies used for the Haskell textbooks:

- *Haskell*: Concepts that are effectively Haskell-specific (e.g., *TMVar* and *Haskell package*).
- *functional programming*: Concepts broadly associated with functional programming (e.g., *map function* or *infinite lists*.
- *Programming*: Concepts associated with programming in general (e.g., *process* and *error*).
- *Data*: Concepts focused on data structures, data types, data management, et al. (e.g., *queue* and *list*).
- *Programming theory*: Concepts associated with mathematical or formal treatment of programs (e.g., *induction*).
- *Software engineering*: Concepts associated with software engineering (e.g., *Programming* and *Testing*(.

- *Software architecture*: Concepts associated with the high-level structure of software systems (e.g., *User interface* or *Module*).

- *Mathematics*: Concepts effectively rooted in mathematics (e.g., *Identity element* and *Induction*).

- *Computing*: Concepts broadly associate with computing (networking, operating systems, and so forth.) (e.g., *TCP* or *UDP*).

As for the textbooks at hand, the most popular vocabularies are (in decreasing order) *Programming*, *Data*, and *Functional programming*. The remaining vocabularies are considerably less frequent. In particular, the *Haskell* vocabulary was only populated by a few concepts; see figure 6.3. This means that the books do not operate at a Haskell-specific abstraction level.

Comparison of the resources. At this stage, with a consolidated vocabulary, we can compare the textbooks in terms of the contributing vocabularies. In particular, we review the terms uniquely contributed by each textbook; see figure 6.4. At the bottom of the figure, all remaining ("non-unique") terms are listed. We observe that each book makes a contribution to the consolidated vocabulary. [Tho11] contributes terms related profoundly to formal or mathematical areas of functional programming such as "Proof" and "Calculation". [Hut07] contributes the fewest terms and much of them are concerned with basic functional programming concepts such as "Function application" and "Function definition". [OSG08] contributes the most terms, overall, and it mentions several technologies; the other books do not. [Lip11] contributes terms related to advanced functional programming concepts, such as zippers and applicative functors.

Monitoring Vocabulary Usage

Subject to appropriate metrics, all terms should be "reasonably" referenced by the *101wiki*. In particular, the documentation of contributions and the description of features should make use of the vocabulary. In this manner, we would be able to claim that the chrestomathy covers the

textbooks and is interlinked with these knowledge resources. Therefore, we monitor coverage and interlinkage, as described below.

Figure 6.5 illustrates monitoring of vocabulary usage. Such tables are computed from a given state of the wiki. Derived terms (shown here for one book only) are listed vertically and ordered by the number of referring contributions. Contributions are listed horizontally and ordered by the number of referenced terms. The big bullets indicate direct references, whereas the small bullets report on indirect references. For each term, the counts of directly and indirectly referring contributions are shown. For each contribution, the counts of directly and indirectly referenced terms as well as uniquely (directly) referenced terms are shown. For brevity, the table is cut off horizontally and vertically so as not to show more terms and contributions without any direct references.

Such monitoring tables can be used to drive improvement of coverage/interlinkage. We use guidelines as follows. *Each reified textbook term should be referenced directly by some number of contributions (e.g., 1). Each contribution should refer directly to some number of terms (e.g., 3). Each contribution should refer uniquely to some number of terms (e.g., 1).*

Clearly, the figure shows the situation at a point in time, when not yet many textbook terms are directly referenced. Thus, the figure suggests that the cut-off terms need to be referenced from existing contributions, or perhaps suitable contributions are missing. Likewise, the cut-off contributions need to be better linked to the vocabulary, unless they address concepts out of the scope for the book at hand.

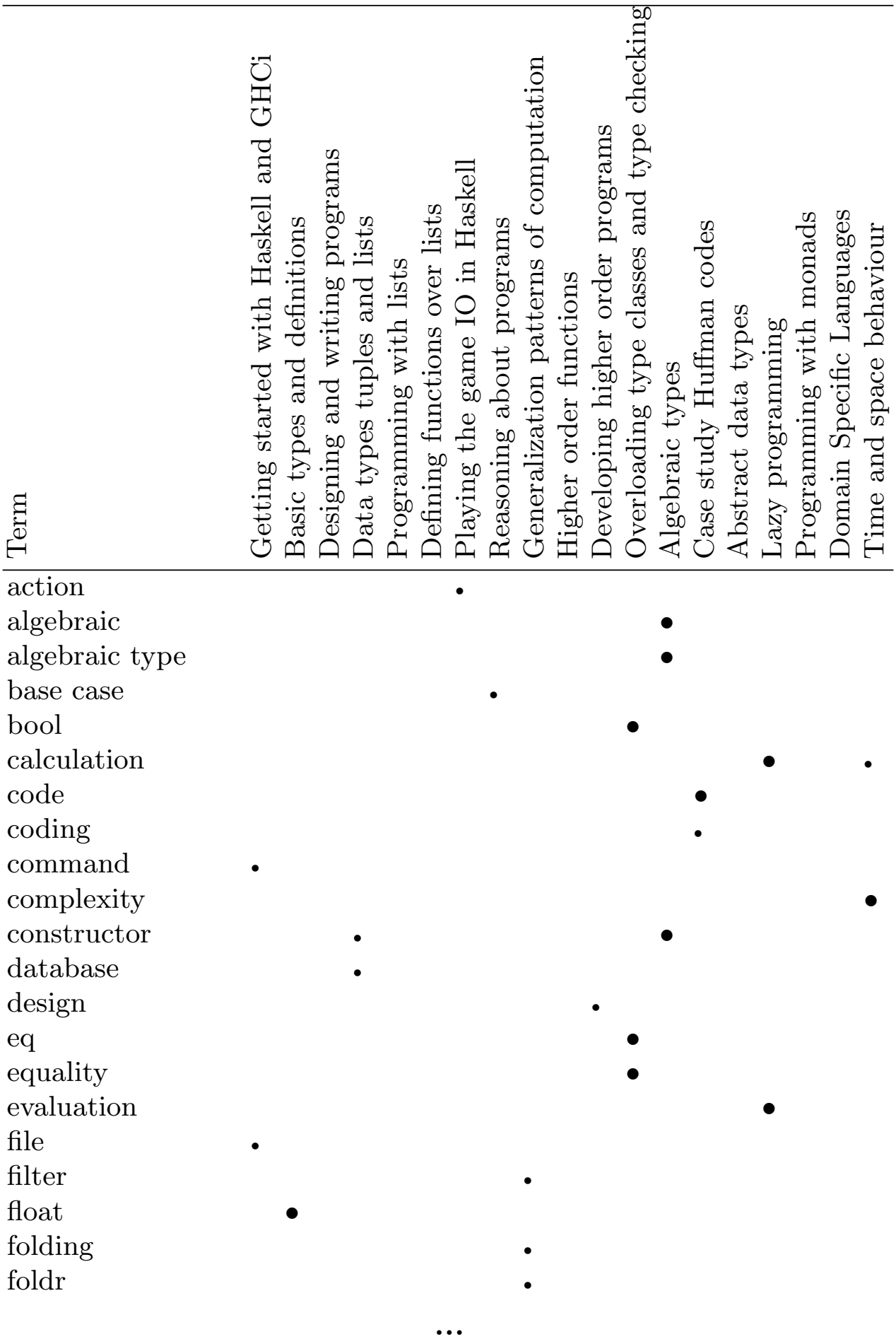


Figure 6.2: Chapter terms for [Tho11]

Name	Headline
Haskell package	A distribution unit for Language:Haskell
Haskell script	A file with Haskell code
MVar	A thread synchronization variable in Language:Haskell
Maybe type	A polymorphic type for handling optional values and errors
Prelude	The standard library of Language:Haskell
TMVar	A transactional MVar of Language:Haskell's STM monad
Type class	An abstraction mechanism for ad-hoc polymorphism
Type-class instance	Type-specific function definitions according to a type class

Figure 6.3: The derived Haskell vocabulary

Terms in [Tho11] only: *Local scope*, *Value*, *Complexity*, *Proof*, *Calculation*, *Equational reasoning*, *Head*, *Equality*, *Programming*, *Queue*, *Argument*, *Result*, *Base case*, *Partial application*, *Program*, *Tuple*, *Set*, *Program design*, *Type checking*, *Higher-order function*, *Name*, *Algebraic data type*, *Infinite list*, *Float*
Terms in [Hut07] only: *Haskell script*, *too generic term*, *Equation*, *Function application*, *Parser combinator*, *Identity element*, *Declaration*, *Function definition*, *Product function*, *Lambda abstraction*
Terms in [OSG08] only: *Foreign function interface*, *Predicate*, *Operator precedence*, *Polymorphism*, *Thread*, *Performance*, *MVar*, *Profiling*, *TCP*, *Directory*, *Property*, *Loop*, *Technology:Parsec*, *Parsing*, *Monad transformer*, *Pointer*, *Technology:HPC*, *Type system*, *User interface*, *Language:XML*, *Core*, *Technology:Glade*, *Exception*, *Error*, *Process*, *Type signature*, *Type definition*, *Program optimization*, *Data type*, *Technology:GHC*, *Pure function*, *Association list*, *Query*, *Output*, *UDP*, *Table*
Terms in [Lip11] only: *Fmap function*, *Accumulator*, *type-class instance*, *Functor*, *Data structure*, *Monadic value*, *Import*, *Factorial*, *Zipper*, *Condition*, *Expression*, *Sum function*, *Applicative functor*
Terms in more than one book: *Monoid*, *Character*, *Type-class instance*, *Bit*, *List comprehension*, *Testing*, *Fold function*, *Operator*, *Lazy evaluation*, *Recursion*, *I/O system*, *Number*, *State*, *Input*, *Haskell package*, *Type*, *String*, *Type class*, *Random number*, *Tree*, *Command*, *Parser*, *Filter function*, *Code*, *Data constructor*, *Pattern*, *Integer*, *Database*, *Catamorphism*, *Evaluation strategy*, *Action*, *Technology:GHCi*, *Text*, *Tail*, *Regular expression*, *Map function*, *Language:Haskell*, *Induction*, *Function*, *Pattern matching*, *Prelude*, *Stack*, *Eager evaluation*, *List*, *Maybe type*, *Monad*, *Module*, *Guard*, *Boolean*, *File*

Figure 6.4: Comparison of the different Haskell textbooks

Contribution/Term	haskellList (4/1/0)	haskellStarter (3/2/2)	haskellLambda (2/3/1)	haskellMonoid (2/2/0)	haskellParsec (2/2/0)	haskellAcceptor (2/0/0)	hxtPickler (2/0/2)	haskellEngineer (0/5/0)
Anonymous function (2/10)	●	·	●	·	·			·
Map function (2/10)	●	·	·	●	·			·
Parsing (2/8)					●	●		
Recursion (2/2)	●	●	·					·
Fold function (2/1)	●		·	●				
Parser combinator (2/1)					●	●		
Function application (1/1)		●						·
Lambda abstraction (1/1)	·		●					
String (1/1)		●						·
Type class (1/1)							●	
Type-class instance (1/0)							●	
Function (0/5)				·				

Figure 6.5: Vocabulary usage for [Hut07] at a given point in time.

Conclusion

This chapter described the foundations of a software chrestomathy with a customized, semi-automatic process for knowledge integration, in fact, vocabulary engineering tailored towards producing a manageable number of interesting terms to be used for metadata in documentation. The foundations are realized for 101haskell – the Haskell-specific sub-chrestomathy of the more general *101* project. A functional programming course has been developed on top of the chrestomathy (see chapter 9). Our experiences substantiate that an appropriately organized and enriched software chrestomathy can be very useful for learning (and teaching).

Chapter 7

A Chrestomathic Ontology

In this chapter we describe *SoLaSoTe* – an ontology for software technologies, languages, and concepts. It contributes to the requirement *R6. A Chrestomathic Ontology*.

Introduction

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*. An ontology targets software languages, technologies, and concepts as an important abstraction level for software engineers and programmers meant to be useful in understanding, comparing, or learning about such entities. In this chapter we provide an advanced case study on ontology development and ecosystem provision, thereby addressing the issue of how to exactly edit, maintain and inquire an ontology so that its authors and potential users can effectively carry out their activities. We also demonstrate the feasibility of illustrating, assessing, and driving the continuous advancement of an ontology by means of a systematic collection of

This chapter is based on the two publications with the substantial revisions and additions: [LMV13] Ralf Lämmel, Dominik Mosen and Andrei Varanovich. Method and tool support for classifying software languages with wikipedia. In SLE, pages 249–259, 2013. [LVL+14] Ralf Lämmel, Andrei Varanovich, Martin Leinberger, Thomas Schmorleiz and Jean-Marie Favre. Declarative Software Development (Distilled Tutorial). In Proc. of PPDP 2014.

artifacts – in this case: a software chrestomathy. Full details, including all schemas, RDFS queries and their results can be found in Appendix A.2.

Basic Principles of *SoLaSoTe*

An ontology is a special kind of information object that allows for formally representing the relevant concepts and relations of a considered domain in a machine readable format [Obe06]). Ontologies are a means to explicitly specify conceptual models with logic-based semantics. An important step is the ontology design is a precise classification of the ontology under development.

Classification criteria

Staab et. al [SSFS11] provide a concise yet solid analysis and definition of the nature of foundational ontologies, core ontologies, and domain ontologies, and their relations to each other. Such classification follows the three-layered architecture of ontology libraries [GFK$^+$04] and discriminate between foundational ontologies, core ontologies, and domain ontologies [Obe06]. A design approach for building core ontologies is further provided. Let us quickly review several important points that are taken into consideration while classifying two ontologies presented in this paper.

Ontologies can be classified across several dimensions ([Obe06], [GGM$^+$02]).

- **Foundational ontologies** *span across many fields* and serve reference purposes [Obe06] and describe concepts independently of a particular problem or domain. Therefore, a foundational ontology is used to build an ontology library relying on ontological choices known from philosophy, linguistics and mathematics.

- **Domain ontologies** represent knowledge that is *specific for a particular domain.* Domain ontologies use terms in a sense that is relevant only to the considered domain and which is not related to similar concepts in other domains [ES13].

- **Core ontologies** provide a detailed abstract definition of structured knowledge in one of these fields. Core ontologies can be based on foundational ontologies and provide a

refinement to foundational ontologies by adding detailed concepts and relations in their specific field. Core ontologies *span across a set of domains in a specific field.*

- **Task ontologies** describe the conceptualization related to a generic task.
- **Application ontologies** describe concepts dependent on a particular domain and a task.

We classify *SoLaSoTe* as a domain ontology for software technologies and languages.

Design principles

We include four additional design principles into our research methodology for a chrestomathic ontology:

- **Content-oriented**. We rely on a *101companies* software chrestomathy as a knowledge accumulation platform. In fact, we lift up the scope of such software engineering knowledge to the level of the social coding domain. Conceptual entities are enriched with the structured documentation authored by the community of experts. *101companies* domain specific wiki supports the authoring. Such separation between the data and conceptual entities also enables us to specify integrity constraints [Gru95].
- **Cognitive value over formal inference**. The expectation is that the ontology helps developers understanding the languages and technologies and concepts they are dealing with. The RDFS infrastructure is leveraged to primarily facilitate advanced querying capabilities to explore ontological entities and relations.
- **Continuity**. A software chrestomathy is a special instance of a social coding as it is concerned the code-centric collaboration perspective [KDSG14]. In particular, it deals with the major part of social coding scenarios [KGB+14], that is about software engineering and collaboration-enabled. A chrestomathic ontology should leveverage the existing vocabulary integration, structured documentation, and automatic medatada infereence capabilities of the *101infrastructure* (see chapter 5).

- **User adoption**. Solid metamodeling foundations. Use tools and techniques that are familiar to people in SE/MDE domain. In this matter, we developed a three-staged approach that involves a DSL for conceptual-level design. As an implementation level we use RDFS and SPARQL that are generated from DSL. In this way we ensure a first-class experience from various semantic web tools, such as triple store or inference engines.

Ontology authoring with *101wiki*

In the terminology of knowledge representation and integration, the software chrestomathy *101* (chapter 5) and specifically its wiki (i.e., *101wiki*) can be viewed as a knowledge base; we also use the name *101kb*, thus. That is, *101kb* contains for *general software knowledge* with categories for software *concepts* (e.g., "object composition" or "unit testing"), software *languages* (e.g., Haskell, XML, and SQL), and software *technologies* (e.g., "javac", "hibernate", or "ant"). Further, *101kb* contains more *specific, illustrative softwareknowledge* in terms of documentation for many implementations of the *101system* – a Human Resources Management System. These implementations are also referred to as *contributions* because they constitute the central means of contributing to the community resource *101*.

Consider figure 7.1 for some illustration of knowledge available through *101kb*. Knowledge about the software concept 'Zipper' is shown. The *Headline* section explains the concept in informal terms. The *Metadata* section contains classification-related or ontological knowledge. In particular:

- 'this' (i.e., "Zipper") is an instance of the namespace for software concepts.
- 'this' is a member of the vocabularies "data" and "functional programming".
- 'this' is classified as a data structure.

The *Backlinks* section readily reports on other wiki pages mentioning the "Zipper" concept. Specifically, three contribution pages are listed – these are documentations of small software systems in the chrestomathy that make use of a zipper or are inspired by the concept. Another mention arises from the page for the software concept "Zipper monad" which essentially provides a specific implementation of the concept.

Ultimately, the *Resources* section links to external resources. The "Learn You a Haskell" resource is the online available textbook of the same name [Lip11]; the link takes one right to the book's chapter on zippers. Further, there is a DOI-based reference for the seminal

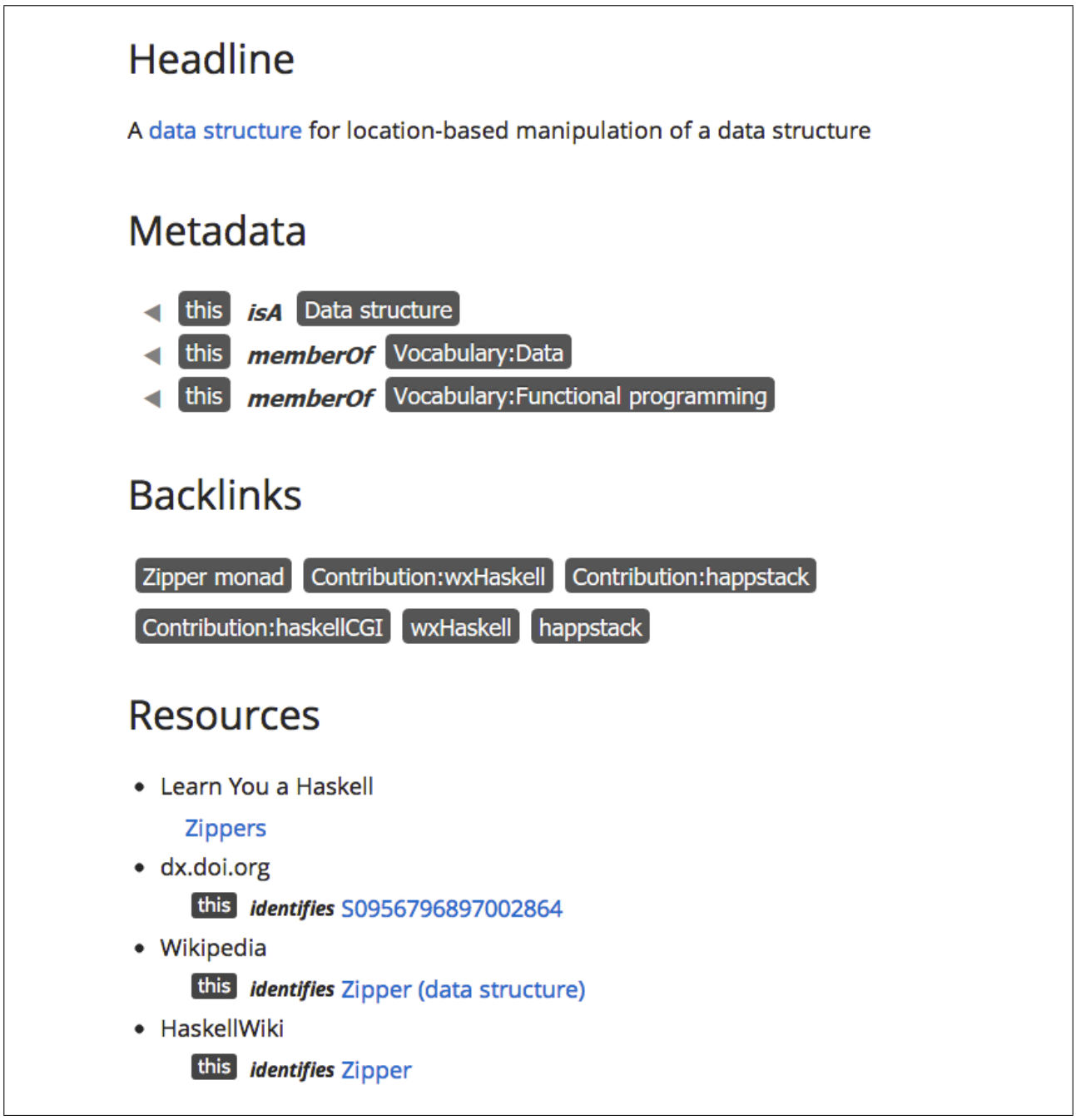


Figure 7.1: The software concept 'Zipper' as rendered on the *101wiki*.

paper on zippers [Hue97]. Finally, there are also links to zipper-related pages on *Wikipedia* and *HaskellWiki*. Except for the textbook link, all the other resource links are marked with the predicate "identifies", which, in the ontology of *101*, implies that these resources are judged as being dedicated to the relevant concept as opposed to merely providing relevant information, in which case a weaker predicate 'linksTo' would be used. Textbook links are currently not classified in this manner.

Stakeholders of *101* are defined in chapter 5. *101* uses GitHub for hosting a source code of the

contributions. Every contributor has a GitHub account. In fact, single-sign-on with GitHub is a default authentication process of *101wiki*. In this way the user profile fof *101contributor* is complemented by the corresponding GitHub profile. Figure 7.2 illustrates this concept: a list of *101contributions developedBy* enriched with the public information about the contributor, including the contact details and a broadened performance metrics of the contributions in open source projects.

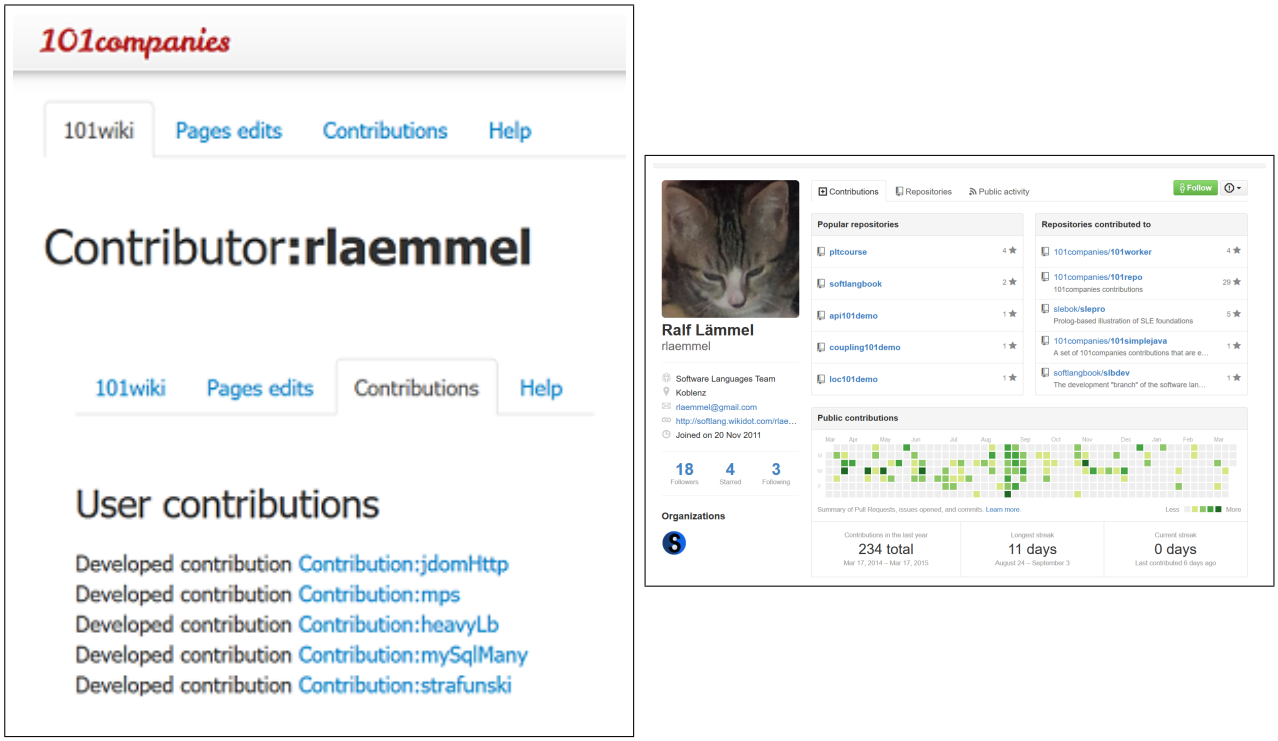


Figure 7.2: 101wiki and GitHub user profile pages

SoLaSoTe

A chrestomathy breaks down into physical entities (contributions, individual source files, fragments thereof, and source code illustrations other than contributions) and conceptual entities (languages, technologies, concepts, features, and others). These entities engage in certain ontological relationships, as we discuss in this section. In the case of *101,* all documentation is manifested on the *101wiki*. Thus, the ontology is modeled and maintained through a so-called Semantic Wiki [Bou09]. The ontology can also be accessed programmatically through an RDF-based triple store.

Figure 7.3 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.

SoLaSoTe and 101 are intertwined in so far that the semistructured documentation of contributions on 101wiki refers to *SoLaSoTe* entities and uses designated properties; see the schema's block with "Contribution" as subject in figure 7.3. 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, figure 7.4 lists the 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 figure 7.3 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

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 7.3: Sketch of *SoLaSoTe* schema

- 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 7.4: Properties for a Java-based contribution.

"popular concepts" (i.e., it sorts concepts referred to in the documentation of contributions by the number of referring contributions).

Entity Types

101 aggregates knowledge about entities of the following types:

Contribution. Chrestomathy members as discussed before.

Contributor. People who submit and maintain contributions.

Feature. Tasks to be implemented by contributions.

Language. Software languages used by the contributions.

Technology. Software technologies used by the contributions.

Concept. Software concepts exercised by the contributions.

Theme. Sets of contributions covering specific topics.

External sources. See later.

The types serve as namespaces on the wiki, while they correspond to proper types in the underlying RDF representation.

Metadata

Entities (say, resources in the sense of RDF) can be associated through triples with appropriate predicates. We also refer to such triples as (semantic) metadata.

Metadata for contributions

- **Contribution *developedBy* Contributor**: a contribution, e.g., *Contribution* `haskellHxt`, was developed by a contributor, e.g., *Contributor* `Thomas Schmorleiz`.
- **Contribution *implements* Feature**: a contribution, e.g., *Contribution* `haskellSyb`, implements a feature, e.g., *Feature* `Cut`.
- **Contribution *uses* Language**: a contribution, e.g., *Contribution* `haskellHxt`, uses a language, e.g., *Language* `XML`. All *101haskell* contributions use *Language* `Haskell`.

- **Contribution *uses* Technology**: a contribution, e.g., *Contribution* `haskellAcceptor`, uses a technology, e.g., *Technology* `Parsec`. All *101haskell* contributions use *Technology* `GHC`, *Technology* `Cabal`, and *Technology* `HUnit`.

- **Contribution *emphasizes* Concept**: a contribution, emphasizes the relevance of some concept. For instance, *Contribution* `haskellWriter` emphasizes *Concept* `Writer monad`. Such triples are *synthesized* from mentions of concepts in the section motivating the chrestomathic purpose.

- **Contribution *relatesTo* Contribution**: a contribution, relates to another contribution in terms of reuse. For instance, *Contribution* `haskellWriter` was derived from *Contribution* `haskellLogging` by setting up monadic style. Such triples are *synthesized* from mentions of contributions in the section explaining relationships between contributions.

General metadata

- **Concept *isA* Concept**: a concept, such as *Concept* `Functional programming language`, is a specialization of a concept, such as *Concept* `Programming language`.

- **Entity *instanceOf* Entity**: an entity, such as *Language* `Haskell`, is an instance of an entity, such as *Concept* `Functional programming language`. Also, *Feature* `Cut` is an instance of *Concept* `Functional requirement`.

- **Entity *partOf* Entity**: an entity, such as *Technology* `GHC`, is part of another entity, such as *Technology* `Haskell Platform`.

- **Entity *mentions* Entity**: an entity, such as *Contribution* `haskellWriter` mentions another entity, such as *Concept* `Writer monad`, which in turn mentions *Concept* `Output`, which in turn is been mentioned by *Concept* `IO`. Such triples are synthesized from ordinary links on the wiki pages.

Consider figure 7.5 for an illustration: declared (but not synthesized) metadata is shown for *Contribution* `haskellWriter`.

- this *developedBy* Contributor:Ralf LÃďmmel
- this *developedBy* Contributor:Thomas Schmorleiz
- this *implements* Feature:Cut
- this *implements* Feature:Hierarchical company
- this *implements* Feature:Logging
- this *instanceOf* Namespace:Contribution
- this *instanceOf* Theme:Haskell introduction
- this *instanceOf* Theme:Haskell potpourri
- this *uses* Language:Haskell
- this *uses* Technology:Cabal
- this *uses* Technology:GHC
- this *uses* Technology:HUnit

Figure 7.5: Metadata triples with *Contribution* `haskellWriter` as the subject (abbreviated as 'this') and other entities as *objects*.

- *haskellStarter*: Basics of functional programming
- *haskellEngineer*: Basics of software engineering
- *haskellList*: List processing with map and friends
- *haskellProfessional*: Idiomatic code for many features
- *haskellLambda*: Anonymous functions
- *haskellComposition*: Recursive algebraic data types
- *haskellVariation*: Multiple constructors per type
- *haskellMonoid*: Queries in monoidal style
- *haskellLogging*: Logging in non-monadic style
- *haskellWriter*: Logging in monadic style
- *haskellParsec*: Parsing with the Parsec library
- *haskellSyb*: Generic programming à la SYB style

Figure 7.6: Instances of the *Theme* `Haskell introduction`.

Themes of contributions. The aforementioned predicates are helpful in imposing additional structure onto the chrestomathy. For instance, the 'instance-of' relation is used for assembling themes, i.e., sets of contributions. Haskell-centric themes collect *101haskell* contributions that address a specific topic (say, stakeholder), e.g., introduction to Haskell or generic functional programming; see figure 7.6 for a theme. *101haskell* contributions also participate in themes that are not Haskell-specific, thereby demonstrating the "Haskell way" of addressing some

problem (e.g., parsing or web programming).

References to External Resources

The *101wiki* also links to external resources such as *Wikipedia*, *DBpedia*, or *HaskellWiki*. The idea is here to make good use of external knowledge resources rather than reproducing large amounts of content on the *101wiki*. Such references are authored again as triples in the metadata section of an entity's page. The references are to be qualified by an appropriate predicate to express the degree of semantical similarity between the internal and external resources.

- **Entity *sameAs* URI**: The *101wiki* and the other resource are assumed to be semantically identical. For instance, the *101wiki* uses "Zipper" in the same sense as *Wikipedia*.[1]
- **Entity *similarTo* URI**: The two resources are similar with some degree of semantical mismatch though [HHT11]. For instance, the *101wiki* uses "equality" in close reference to programming, whereas *Wikipedia* associates equality primarily with mathematics.[2]
- **Entity *linksTo* URI**: Any relationship weaker than 'similarTo'.

References to Code Fragments

The illustration section of a contribution tends to include code fragments from the actual contributions. Copying and pasting fragments into wiki pages would imply two drawbacks. First, the code on the page may go out sync with the code in the *101repo*. Second, the wiki user would be insufficiently supported in navigating from the wiki to the repo. Thus, we extend a resource-centric approach such that the wiki may refer to fragments in the repo. Consider figure 7.7 for an illustration.

These references rely on a URI scheme (as illustrated in the figure), a language-specific fragment locator (to look up the relevant code fragment), and a language-specific renderer. In this

[1] `http://en.wikipedia.org/wiki/Zipper_(data_structure)`
[2] `http://en.wikipedia.org/wiki/Equality(mathematics)`

A *101wiki* paragraph from *Contribution* `haskellSyb`

Several of the operations on companies can be implemented in a very concise manner based on the SYB style of generic programming. For instance, the operation for totaling salaries simply extracts all floats from the given term and reduces them by addition:

Explore

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

Underlying markup for the explorable code fragment

```
<fragment url="src/Company/Total.hs/pattern/total"/>
```

Figure 7.7: The wiki paragraph contains a code fragment from which one can navigate right away to the relevant code in the repository; see the "Explore" button. The underlying markup specifies the location of the file 'src/Company/Total.hs', the syntactic category "pattern" of the fragment, and the name 'total'.

manner, source code (repo) and documentation (wiki) may evolve in a loosely coupled manner with possibly different authors for both artifacts, while being still effectively connected, thereby supporting navigation by wiki users. We note that the illustration on the wiki must not be confused with regular code documentation. That is, the illustration serves for highlighting a contribution's characteristics with regard to the chrestomathy, as opposed to the comprehensive documentation of a given contribution as an individual software system. Thus, classic techniques, such as literate programming [Knu84], are not applicable. We need to enable loose coupling for wiki and repo.

Workflow of ontology processing

The concept of *101worker* as a part of *101wiki* and refers to a computational infrastructure that executes modules to process the repositories so that information is extracted, computed, validated, and prepared for presentation [FLL+12a]. Its usage scenario was later illustrated in the context of cross-language and technology feature detection and metric computation [LLSV14a]. It is organized as an executable pipeline, consists of modules – units of execution. *101worker* provides an API to feed an output of one module as an input to another one. The pipelines are usually scenario specific, and encoded in a JSON configuration files. The worfkow of has 3 modules [**"wiki2json", "onto2ttl", "wiki2triples"**].

wiki2json Produces a so-called wikidump – a JSON dump of the all *101wiki* pages. In this dump the semantic properties are separated from the content of the page sections. They uniformly represent the relations between wiki page and subresources within the page, that have *partOf* relation to it.

```
{
 // always an one−sentence summary of the page
 "headline": "A data structure for location−based manipulation of a data structure"
 // the page is a member of 2 vocabularies
 "memberOf": [{"p": "Vocabulary", "n": "Data"},
              {"p": "Vocabulary", "n": "Functional programming"}],
 "n": "Zipper", // name of the page
 "p": "Concept", // prefix of the page
 "subresources": {}, // the page is not a container
 "mentions": [{"p": null, "n": "Data structure"}],
 "isA": [{"p": null, "n": "Data structure"}],
 "identifies": ["http://en.wikipedia.org/wiki/Zipper (data structure)", ....]
}
```

Figure 7.8: Wikidump of the Zipper page from section 7.3

wiki2triples Turns wikidump into RDFS tripples and populates the triple store. Not the whole wiki is turned into tripples. The blacklist is used to filter out namespaces, that are not a part of the ontology. For every page from the remaining list the mapping to the SSN entity type is established. The first state of the validation is performed: check of the allowed properties. The goal is to report on the semantic properties used on the page, that are not defined in SSN

entity declarations and therefore cannot be put into an ontology. *Validation* process for every page is implemented as follows. A list of allowed relationships is commuted using SSN models as an input. Overloading table is built to support base properties. The models define which properties to expect in the different types. Models can inherit properties from other models (i.e. subtyping is allowed). For every SSN model, a list of properties directly defined in the model is created. After this the inherited properties from the base type are appended to the list. Once the mapping between the page and the list of allowed properties are identified, we check whether the properties used on the wiki page are contained in the list of allowed properties. Figure 7.9 summarizes the input-output of the module.

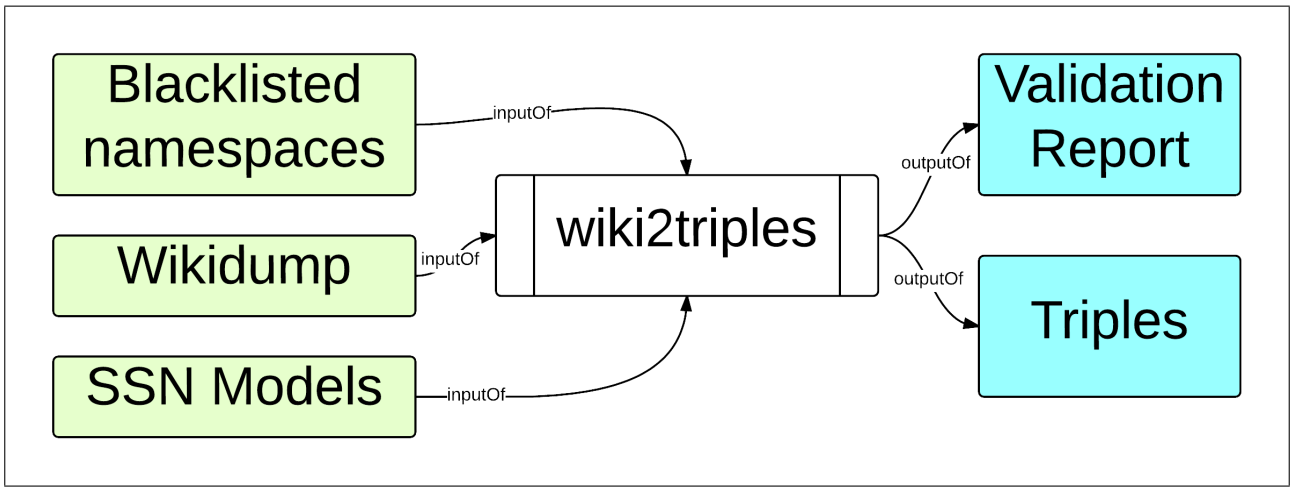


Figure 7.9: Wiki2triples I/O

To further illustrate the process, let us use an example.

```
"@id": "Language",
"@type": ["Entity","Instrument"],
 ...
 "properties": [
  {
    "property": "langDesignedBy",
    "super": "designedBy",
    "range": "foaf:Person",
    "minCardinality": "0",
    "comment": "Designers of languages"
   }
]
```

```
"@id": "Entity",
"@type": "Resource",
 ....
 "properties": [
 {
   "property": "designedBy",
    "range": "foaf:Person",
    "minCardinality": "0",
    "comment": "Designers of entities"
  }
```

Figure 7.10: Property inheritance for Language

If we look at the *Language* model (figure 7.10), there is a property 'langDesignedBy' which is

a sub-property for ‘designedBy’. Inherited properties have unique names, to avoid ambiguity in OWL reasoning. However, on the corresponding wiki page, the super-property is to be expected. The page for Language:Java will have the property ‘designedBy’, which is a natural name to use for the page author. All subtypes of ‘designedBy’ propery which have the domain ‘Language’ are checked. If the range of the sub-property matches the type of the object of the ‘designedBy’ relation on the language page (foaf:Person in our example), then the property is used correctly.

onto2ttl Produces a RDFS (turlte syntax) representation from *SSN* representation of the conceptual model of the ontology. The triple store is populated by using RDF(s) data.

Evaluation criteria

To deal with the ontology validation, we use the following approach. We pick two sources with the quality criteria for a well designed ontology. These sources cover the broad scope on ontology design scenarios, with the different relevance and applicability to *SoLaSoTe*. Below we create an evaluation matrix, address the criteria one by one, reflecting on a particular applicability.

[OLG+06] provides a list of problems that should be avoided when a carefully engineered foundational ontology is used as a modeling basis. We provide a list below in a form, so that the problems are already negated and turned into requirements for our ontology design. Additionally, we use the selected criteria for "beatiful ontologies" [dG11].

Table 7.1: Evaluation criteria

Conceptual unambiguity [OLG+06] It should be straightforward for users to understand the intended meaning of concepts, the accociations between the concepts, and their relations to the modeled entities.	This aspect is fully instantiated in *SoLaSoTe* with the support of *101wiki* and the underlying infastructure (see chapter 5)
Good axiomatization [OLG+06] An ontology should not contain modeling artifacts, i.e those concepts and accociations that do not carry ontological meaning. The clarity of concept definitions should be supported by the axiomatization of the ontological entities.	*SoLaSoTe* does not include a formal axiomatization yet. However, it has a partial instantiation in technology modeling, with the underlying interpretation and resolution engines (see chapter 8)
Broad scope [OLG+06] The distinction between the objects and events within an information system (regarding data and the manipulation of data) and the real-word objects. As an example consider the distinction between a user account and its corresponding natural person(s).	We do not see this point as obviously applicable, as *SoLaSoTe* mainly talks about digital entities. Physical artifacts, such as files, are unambiguously distinguished.
Structure [dG11] Reusing foundational ontologies; being designed in a principled way; being formally rigorous; implementing also non-taxonomic relations; following strictly an evaluation methodology; being modular, or embedded in a modular framework.	*SoLaSoTe* is a content and community driven effort, which is, by the nature, opposite to 'structure-first' approaches. This is also facilitated by the *101wiki*. There is an effort needed, to investigate the possibilities for integration of foundational ontologies. Right now some basics are supported, such as FOAF [3] for contributors.
Conceptual coverage [dG11] Providing important reusable distinctions; having a good domain coverage.	*SoLaSoTe* covers a number of technological spaces, primarily motivated by various types of contributions to the *101companies* software chrestomathy.
Conceptual task [dG11] Being oriented at an explicit task; having spelled out requirements from scenarios.	*SoLaSoTe* supports a broader community effort and thus this requirement is weakly applicable.
Social sustainability [dG11] Being the result of an evolution (many revisions); implementing scientific knowledge.	Being first introduced at GTTSE 2011 [4] with more then 40 external contributors, we believe this requirement fully instantiated, also via underlying evolution of *101companies*
Pragmatic sustainability [dG11] Having applications built on top of it; designed for efficient query answering; maintaining original expressivity of data, and improving or enriching it; able to get rid of clunky constructs or to overcome expressivity limitations; being well documented; solving other technical aspects.	At this point the Linked Data part of *101companies* is mainly leveraged in a number of scenarios (see chapter 9). The biggest potential we see in a joint development of the core ontology for linguistic architecture, supported by *SoLaSoTe* as a domain ontology, and its applications.

Conclusion

In this chapter we provided an advanced case study on ontology development and ecosystem provision, thereby addressing the issue of how to exactly edit, maintain and inquire an ontology so that its authors and potential users can effectively carry out their activities. We also demonstrated the feasibility of illustrating, assessing, and driving the continuous advancement of an ontology by means a systematic collection of artifacts – in this case: software chrestomathy (i.e., a collection of small software systems).

Chapter 8

Technology Modeling

In this chapter we focus on technology modeling as a mean to describe the linguistic architecture of software technologies, supported by the modeling language *MegaL*. This chapter contributes to the requirements *R7. Linguistic Architecture of Software Products* and *R8. General-purpose Language for Technology Models*

Introduction

101companies software chrestomathy collects implementations (software systems) in different technologies spanning across many technological spaces. Together with the produced artifacts they are engaged in a certain type of relations; in fact, any software systems is an entity-relationship model with entities for languages, technologies, concepts and artifacts and with relationships to express data flow, conformance, and others. Such types of models are known as megamodels in MDE. A software chrestomathy contributions are essentially non-instantiated technology models.

This chapter is based on two publications without major additions: [FLV12] Jean-Marie Favre, Ralf Lämmel and Andrei Varanovich. Modeling the linguistic architecture of software products. In MoDELS, pages 151–167, 2012. [LV14] Ralf Lämmel and Andrei Varanovich. Interpretation of Linguistic Architecture. In European Conference on Modelling Foundations and Applications (pp. 67-82). Springer International Publishing.

The upper frame uses the MegaL/yEd visual notation for megamodeling.
The lower frame shows *linked artifacts* of the product explained later in the paper.

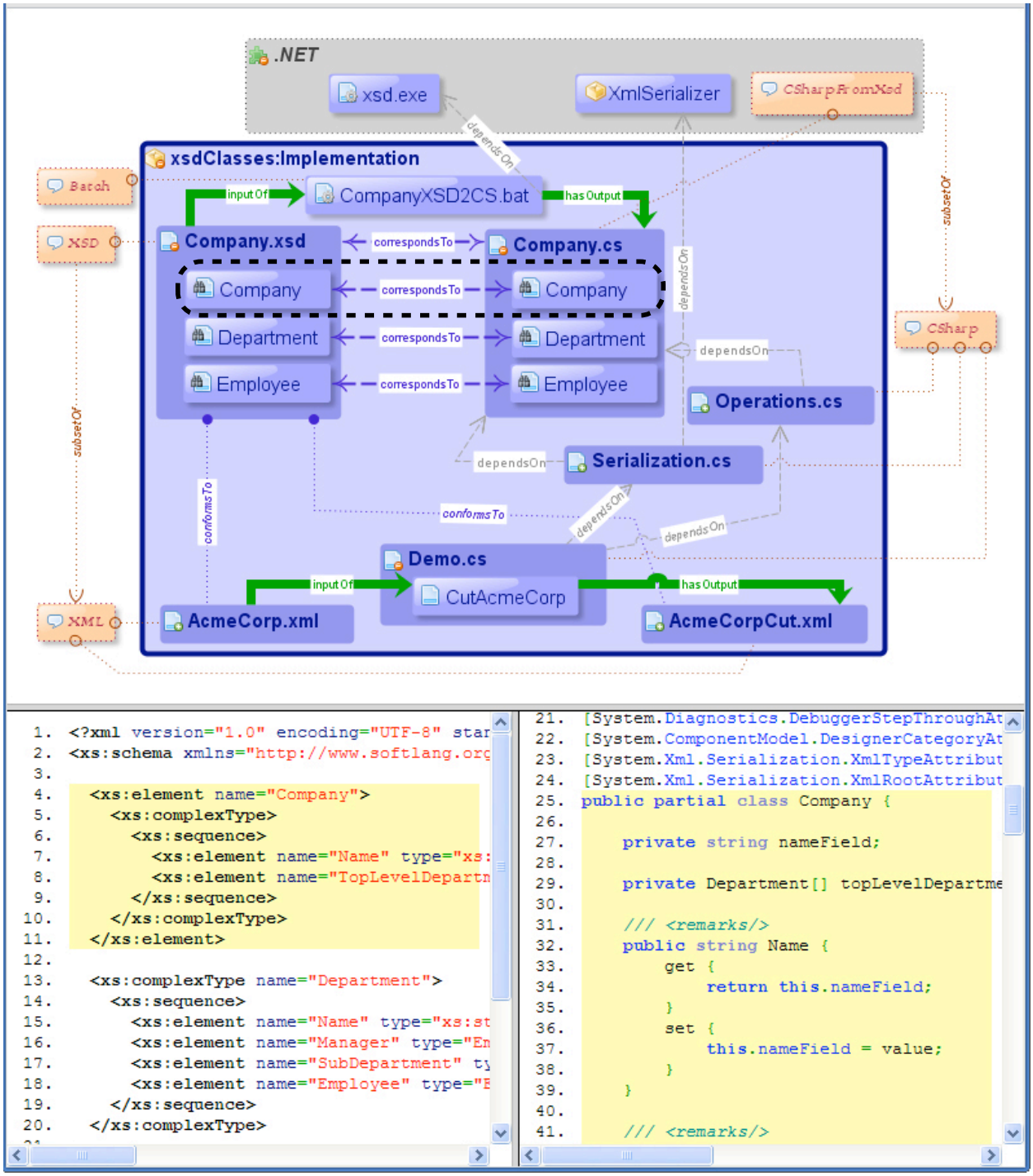


Figure 8.1: The linguistic architecture of a software product when displayed with the MegaL/Explorer tool.

Illustration of Linguistic Architecture

Consider the upper frame in figure 8.1. (The lower frame will be discussed in section 8.6.) The linguistic architecture of a software product is described in the MegaL/yEd visual notation.[1]

[1]MegaL is currently a combination of an ontology and a set of concrete syntaxes; there exist these flavors of the language: MegaL/yEd—a visual notation, MegaL/TXT—a textual notation and MegaL/RDF—an RDF

The product is a *C#*-based application which makes use of .NET's Object/XML mapping technology.[2] In fact, the product is a *101companies* implementation, which is named *xsdClasses* and available online. Hence, the application deals with companies (as in the human resources domain) including operations for totaling and cutting salaries (symbolized by the model element *Operations.cs*) as well as XML-related functionality for de-/serialization (see *Serialization.cs*). There are model elements for XML types according to the XSD language for XML schemas (see file *Company.xsd*) and *C#* classes (see file *Company.cs*) with fragments (see *Company*, *Department*, and *Employee*). There are correspondence relationships between the XML and object types to show that instances of these types can be (roughly) converted into each other (modulo the X/O impedance mismatch [LM07]). Class generation is automated with a batch file (see *CompanyXSD2CS.bat*), which essentially invokes the .NET tool `xsd.exe` (see *dependsOn*). Ultimately, the operation for cutting companies is invoked by demo functionality (see *Demo.cs*) and applied to a specific company – the *Acme Corporation*.[3]

The displayed linguistic architecture describes artifacts as they arise during development time and runtime together with the relationships regarding dataflow, language membership, schema/-type conformance, and correspondence. Characteristics of the .NET technology for Object/XML mapping are clearly identifiable. Consider, for example, the fact that the class generator is not described as generating "arbitrary" *C#*. Instead, the subset *CSharpFromXsd* is introduced for referring to regular *C#* as produced by the generator. The identification of such "hidden" languages is fundamental to the understanding of software technologies.

Entity and Relationship Types for Megamodels

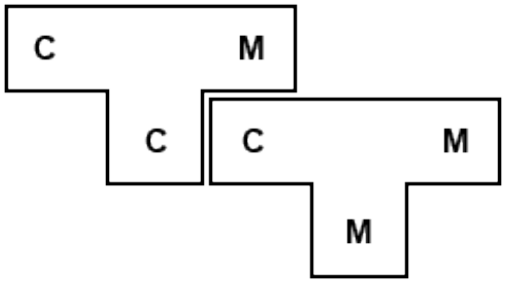
The proposed form of megamodeling essentially involves the identification and classification of entities and relationships that make up the linguistic architecture of software products or the underlying software technologies. In this section, we gather a set of entity and relationship *types* that may be used in megamodels.

version of MegaL. The correspondance between these notations is rather straightforward and it will be introduced by means of illustration in the course of the paper.

[2] `http://msdn.microsoft.com/en-us/library/x6c1kb0s(v=vs.71).aspx`

[3] `http://en.wikipedia.org/wiki/Acme_Corporation`

A tombstone diagram for bootstrapping a C compiler[4]



Mechanics of an ATL-based model transformation[5]

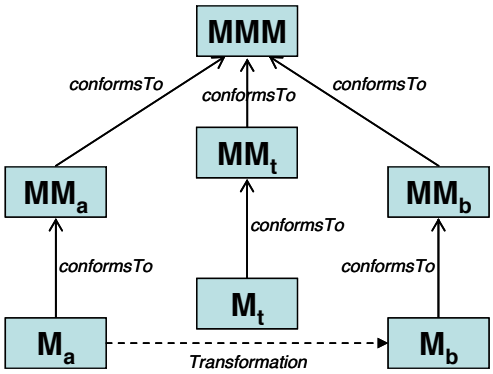


Figure 8.2: Megamodels in different areas of computer science.

Background

Megamodel-like models exist in different areas of computer science. Linguistic relations have been of interest since the early days of computing as *tombstone diagrams* testify. In figure 8.2, on the left, we show a tombstone diagram, as it is used in compiler construction to describe the bootstrapping process for a C compiler, also written in the programming language C and compiling to M (the machine language) such that initially another C compiler is needed – this time written (or executable) in M. Hence, *languages* and *compilers* serve as entities while relationships are concerned with *dataflow* or *function application* and *membership*.

On the right, we show a much more recent diagram, as it appears in the documentation of the *ATL* transformation language; the diagram shows the mechanics of a model transformation in terms of entities for the involved *models* and *metamodels* as well as relationships for *conformance* and *dataflow*.

Further inspiration, specifically regarding linguistically relevant relationships, can be drawn from fundamental research on modeling and model management. The 'conformsTo' relationship is established in modeling for relating models and metamodels [Fav05a, Küh06b]. We also rely on yet other basic modeling relationships (as in UML)—in particular 'partOf' and 'dependsOn'. The 'elementOf' and 'subsetOf' relationships are hardly used directly in regular modeling,

[4]Source: `http://en.wikipedia.org/wiki/Tombstone_diagram`
[5]Source: `http://wiki.eclipse.org/ATL/Concepts#Model_Transformation`

but it appears in fundamental discussions, when *the usage of languages* is taken into account as opposed to sole restriction to metamodel-based conformance [Fav05a, Küh06b]. The 'modelOf' or 'representationOf' relationship [Küh06b, MFB09, MFBC11] is important for capturing the roles of descriptions, definitions, specifications, programs, or more generally models in megamodels. Ideally bidirectional intermodel mappings [DXC11, DMC12], with interpretations at both the schema (metamodel) and the instance (model) level, give rise to the 'correspondsTo' relationship in our terminology.

Based on this background, the MegaL ontology defines a set of entity and relationship types as discussed below.

Entity Types of MegaL

We distinguish three kind of entities: *abstract* entities, which appear at the mathematical level of thinking; *conceptual* entities, which are cognitive elements such as languages or technologies; *digital* entities, which correspond to artifacts that reside in and are processed by computers.

We use these types of *abstract entities*: Entity, Set, Pair, Relation, Function, FunctionApplication (i.e., pairs pertaining to a function). For instance, functions are needed to model the meaning of tools or programs. Further, we use these types of *conceptual entities*: Language and Technology. Languages can be viewed (in a simplified manner) as sets. Technologies can be viewed as compound entities with components for tools, languages, and others. Finally, we use these types of *digital entities*: Artifact (the base type for the following types), File, Fragment (of a file), Program, Library, ObjectGraph.

The aforementioned entity types are just sufficient for the examples in this paper. The megamodel ontology can be extended to cover different domains, technological spaces, or engineering activities [Fav04a]. For instance, a megamodel in the context of model-driven engineering may benefit in clarity from additional digital entity types for models, metamodels, and model transformations.

Relationship Types of MegaL

Based on the fundamental relationships and the types of entities, as identified above, the following relationship types can be derived. Again, the list is trimmed down for the scope of this paper. We apply a UML-like convention to use ':Type' for a concrete (anonymous) entity of the given type.

- :Language **subsetOf** :Language
- :Artifact **elementOf** :Language
- :Language **domainOf** :Function
- :Function **hasRange** :Language
- :FunctionApplication **elementOf** :Function
- :Artifact **inputOf** :FunctionApplication
- :FunctionApplication **hasOutput** :Artifact
- :Artifact **conformsTo** :Artifact
- :Artifact **partOf** :Artifact
- :Artifact **correspondsTo** :Artifact
- :Artifact **dependsOn** :Artifact
- :Artifact **dependsOn** :Language
- :Artifact **realizationOf** :Function
- :Artifact **definitionOf** :Language
- :Program **partOf** :Technology
- :Library **partOf** :Technology

Megamodels initially just *declare* entities and relationships. Eventually, megamodels may be *linked* so that both entities and relationships are meaningfully demonstrated by actual artifacts of specific software products. This will be discussed in section 8.6.

An Initial Megamodel for O/X Mapping

Megamodeling is demonstrated in this section for O/X mapping. In (schema-first) O/X mapping [Ron12, LM06a], one is concerned with marrying object-oriented programming with XML-based data representation in such a way that an object model for data representation is generated from an XML schema and library functionality is responsible for mediating between XML documents ("files") and objects back and forth. The population of objects from XML data is also called de-serialization, while the other direction is referred to as serialization. The notion of O/X mapping is also known as XML data binding. In the context of the .NET platform, the term XML serialization is used as well.

Stepwise Development of the Megamodel

Let us develop an initial megamodel for O/X mapping, step by step. We use MegaL/TXT–this simple textual notation can express the same concepts as the visual notation MegaL/yEd that we used earlier. The textual notation comes with straightforward syntactic shorthands for recurring patterns [Fav04a] such as '$\rightarrow$' and '$\mapsto$' (instead of combinations of 'domainOf', 'hasRange', 'inputOf', 'hasOutput').

We begin with the *languages* involved in O/X mapping:

Languages *XSD, CSharp, XML, ClrObjectGraphs .*

The *C#* (or *CSharp*) language is mentioned because it is assumed here that schema-derived object models are represented in *C#*. We could make the object-oriented programming language a parameter of the megamodel, but we commit to *C#* here for the sake of concreteness. *XSD* is the language of XML schemas. *XML* is the language of XML trees (or XML documents), (i.e., the primary ["on file"]) representation format for data. Finally, *ClrObjectGraphs* is the language of object graphs. Again, we could make the in-memory representation of objects a parameter of the megamodel, but we commit to .NET's CLR representation here for sake of concreteness.

In fact, another language should be identified:

Language *CSharpFromXsd* ***subsetOf*** *CSharp* .

That is, *CSharpFromXsd* proxies for the C# subset that is used by the class generator of the O/X mapping technology. In conservative discussions of O/X mapping, this language is never articulated. However, awareness of this language and its characteristics helps to understand O/X mapping.

The characteristics of schema-derived object models vary indeed for each O/X mapping technology. In the case of .NET's O/X mapping technology, we can state the following characteristics for all $x \in$ *CSharpFromXsd*: (i) x declares classes only—as opposed to interfaces, enumerations, etc. (ii) The classes of x declare fields and properties as members, but no methods. (iii) x use attributes controlling XML serialization.

Let us now consider the major artifacts involved in O/X mapping. There are two type-level artifacts involved in such O/X mapping: an XML schema and an object model. There are also two instance-level artifacts involved: an actual XML document and an actual object graph:

File *xmlTypes* ***elementOf*** *XSD* .
File *ooTypes* ***elementOf*** *CSharpFromXsd* .
File *xmlDoc* ***elementOf*** *XML* .
ObjectGraph *clrObj* ***elementOf*** *ClrObjectGraphs* .

We also need to impose 'conformsTo' relationships as constraints on the instance-level artifacts: an arbitrary XML document would not be suitable; it must conform to the XML schema at hand; likewise for the object graph. Thus:

xmlDoc ***conformsTo*** *xmlTypes* .
clrObj ***conformsTo*** *ooTypes* .

Ultimately, we expect an O/X mapping technology to provide functionality for class generation and for deserialization (as well as serialization, which we skip here though). To this end, we introduce the following conceptual entities, in fact, functions, and we apply them in the expected manner to relate the artifacts at the type and instance levels. Thus:

Function *classgen : XSD → CSharpFromXsd .*

Function *deserialize : XML → ClrObjectGraphs .*

classgen(xmlTypes) ↦ ooTypes .

deserialize(xmlDoc) ↦ clrObj .

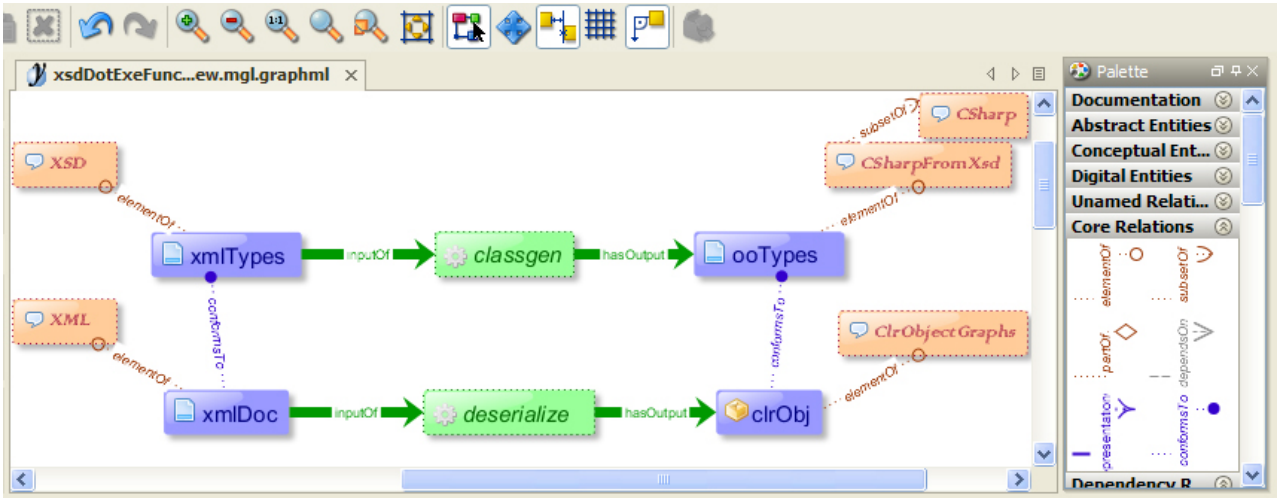


Figure 8.3: An initial megamodel for O/X mapping drawn with the MegaL/yEd editor.

Summary of the Megamodel

Figure 8.3 summarizes the megamodel in the form of a diagram drawn with the MegaL/yEd editor.[6] The visual and the textual notation convey the same information. Note that icons and colors are bound to entity types in the diagram. Some megamodel elements can be mapped to different visual elements. For instance, 'partOf' relationships are represented by node embedding in the upper frame of figure 8.1, but a regular 'partOf' edge could also be used.

Discussion

The initial megamodel of this section was deliberately kept simple. This intermediate state also allows us to reflect on methodological questions of megamodeling:

- Do we model all important aspects of O/X mapping overall?
- What specifics of .NET's O/X mapping technology should be modeled?

[6]Our implementation uses yEd for megamodel editing `http://www.yworks.com/en/products_yed_about.html` with GraphML `http://graphml.graphdrawing.org/` for the representation.

Without focus on O/X mapping, these questions take the following form:

- Do we model all general aspects of the kind of technology at hand?
- What specifics of a concrete technology should be modeled?

It is relatively easy to observe that the megamodel could be enhanced to incorporate additional aspects of O/X mapping, overall. For instance, we did not yet model the fact that O/X mapping is carried out 'for a purpose': some OO program is meant to use the generated object model to implement data-processing functionality. As to the question of technology-specific aspects, we did not yet model the components of .NET's technology for O/X mapping. These and additional aspects are addressed in the following section.

A Megamodel for O/X Mapping with .NET

We advance the megamodel of the previous section to cover generally more aspects of O/X mapping and to also apply more directly to the situation for the .NET platform.

The Use of Schema-Derived Object Models

The value proposition of O/X mapping depends on the fact that it enables essentially OO programming on XML data. We capture this aspect in the megamodel by introducing a problem-specific program that is said to depend on the schema-derived object model. This is another placeholder for an entity that does not belong to the technology itself, but instead to the software product that uses the technology. Thus:

```
File problemProgram elementOf CSharp .
problemProgram dependsOn ooTypes .
```

Technology Components for .NET

The technology consists of a code-generation tool, `xsd.exe`, a library, hosted by the namespace *System.Xml.Serialization*, and custom attributes (annotations) for metadata.[7] We declare corresponding entities:

```
Program xsdDotExe . -- the "xsd.exe'' tool
Library XmlSerializer . -- namespace "System.Xml.Serialization"
Language XsdMetadata .
```

We can model now the fact that the `xsd.exe` tool realizes the class generation functionality for O/X mapping. In fact, the tool also realizes additional functionality (e.g., related to O/R mapping). To this end, the tool can be used in different modes controlled through the command line or an API, but we do not model such variability here. Thus:

```
xsdDotExe realizationOf classgen .
```

Previously, we simply assumed a function, *deserialize*, for deserializing XML into objects, without however clarifying the origin of the function. It is the problem-specific program that essentially performs de-serialization. In fact, we assume that some part of the program realizes serialization by making appropriate use of .NET's library for XML serialization. Thus:

```
Fragment deserialization partOf problemProgram .
deserialization dependsOn XmlSerializer .
deserialization realizationOf deserialize .
```

We can also clarify the role of metadata in O/X mapping. We assume that, subject to an appropriate interpretation of 'partOf' for languages, the *C#* language indeed comprises a part for metadata such that metadata for O/X mapping is a subset of general metadata.

```
Language CSharpMetadata .
CSharpMetadata partOf CSharp .
XsdMetadata subsetOf CSharpMetadata .
```

[7] `http://msdn.microsoft.com/en-us/library/ms950721.aspx`

Also, we can capture the characteristics of schema-derived classes to depend on metadata for O/X mapping. We do not formalize other characteristics of *CSharpFromXsd*. Thus:

ooTypes ***dependsOn*** *XsdMetadata* .

Additional Linguistic Details

Let us call *problemLanguage* a problem-specific language underlying the involved type-level artifacts. We think of this language as being abstract, rather than concretely represented by XML trees or object graphs. This language can be viewed as a proxy for the domain that is covered with an Object/XML mapping effort.

Language *problemLanguage* .
xmlTypes ***definitionOf*** *problemLanguage* .
ooTypes ***definitionOf*** *problemLanguage* .

It remains to establish a correspondence relationship between XML and object types as well as the involved instances:

xmlTypes ***correspondsTo*** *ooTypes* .
xmlDoc ***correspondsTo*** *clrObj* .

At the instance level, the object graph, which is obtained by de-serialization, is expected to be a *representation of* the original XML document and *vice versa* such that the original document could be re-obtained by serialization from which we abstract here for simplicity.

At the type level, correspondence means that (ideally) XML schema and object model are related by bidirectional intermodel mappings (say, 'structure-preserving' bijections) modulo difficulties due to the O/X impedance mismatch [LM07]. The couple of de-serialization and serialization functionalities should be considered the concrete interpretation of these mappings at the instance level, but this view is not developed in detail here. More intuitively, we could say that there is 1:1 mapping of types driven by name equality or similarity, and for each couple of associated types there is also a correspondence at the 'member' level.

Discussion

We conclude with a discussion of potential directions for enhancing the megamodel. We have focused here on de-serialization, but serialization could also be of interest, if XML transformation or generation is to be modeled. Further, we have not modeled any variability or configurability admitted the mapping technology, as needed for advanced usage scenarios of the technology.

We claim originality for analyzing O/X mapping by megamodeling. For comparison, the arguably most comprehensive catalog of O/X mapping technologies [Ron12] uses an informal metamodel to compare technologies (tools) on the grounds of capabilities and limitations—linguistically relevant entities and relationships are not considered.

Linked Megamodels

A difficulty with metamodeling and even more with megamodeling approaches resides in the high level of abstraction involved. This difficulty is even exacerbated by megamodels that deal with technologies, as in the previous two sections, because of the gap between the abstract notation and the very concrete artifacts a software engineer deals with, e.g., some files or objects. As a result it may be hard to convince anyone that any given statement in the megamodel holds.

Linked megamodels close the gap between abstraction and concreteness by linking each entity in the megamodel to a web resource. Thus, an entity is no longer represented merely as an identifier, leaving considerable room for misunderstanding and misinterpretation; instead, the identifier is linked to a unique resource that can be browsed and examined at will. Relationships can also be linked. As a result, it becomes much easier to understand and to validate megamodels.

Binding Placeholder Entities

Note that in the megamodel discussed in the previous two sections, artifact placeholders were used for some entities (e.g., *xmlDoc* and *clrObj*). When the goal is to validate or illustrate the

megamodel, then it is useful to "bind" placeholders to actual artifacts. This has been done in figure 8.1 with the concrete artifacts being part of a particular software product. That is, X/O mapping is illustrated thanks to the *xsdClasses* implementation of the *101companies* project. For instance, the placeholder *xmlDoc* is bound to *Company.xsd* – an XML schema file of the *xsdClasses* implementation.

Exploring Linked Megamodels

From the end-user perspective, linked megamodels are seen as hypertext documents that can be explored. Figure 8.1 shows a screenshot of the MegaL/Explorer tool. The upper frame corresponds to a clickable image that is produced with MegaL/Editor. Within the context of the explorer, a click on an entity displays the corresponding resource in the lower frame. For instance, clicking on the *CSharp* node leads to a wiki page for C# according to the *101companies* project; clicking on `xsd.exe` node also leads to a page for the tool; clicking on a file (e.g., *Company.xsd*), displays the content of the file extracted from the *101companies* repository. Relationships (i.e., graph edges) are also clickable. In figure 8.1, the user has selected the (circled) correspondence link between the *Company* fragments respectively in *Company.xsd* and *Company.cs*. As a result, the source fragments are shown side by side in the lower frame—clearly showing what the `xsd.exe` tool actually generates for a given example.

MegaL/RDF, Linked Megamodels and Linked Data

Technically, linked megamodels are represented in RDF by following Linked Data principles. Figure 8.4 and figure 8.5 show fragments of two megamodels expressed in MegaL/RDF as sets of triples while using RDF/turtle syntax. The first figure is concerned with the general megamodel for O/X mapping. It contains therefore placeholders with generic names (e.g. *xmlDoc* and *xmlTypes*). By contrast, the second figure is concerned with the bound megamodel for the *101companies* implementation *xsdClasses*. It contains product-specific names (e.g. , *CompanyDotXSD*), but also, and this is a very important aspect, links to concrete software artifacts, which should be considered as *resources* according to RDF principles.

```
_:xmlTypes rdf:type mgl:File .
_:xmlTypes rdfs:label "xmlTypes" .
_:xmlTypes mgl:elementOf lang:XSD .
_:xmlTypes mgl:inputOf _:classgen .

_:xmlDoc rdf:type mgl:File .
_:xmlDoc rdfs:label "xmlDoc" .
_:xmlDoc mgl:elementOf lang:XML .
_:xmlDoc mgl:conformsTo _:xmlTypes .
_:xmlDoc mgl:inputOf _:classgen .

_:classgen_app_1 rdf:type mgl:FunctionApplication .
_:classgen_app_1 rdfs:label "classgen" .
_:classgen_app_1 rdf:elementOf _:classgen .
_:classgen_app_1 rdf:hasOutput _:ooTypes .

... etc. ...
```

Entities are associated with a prefix, e.g., `rdf`, corresponding to a unique URI (not shown here). This means that each entity is now associated with a URL where the corresponding resource can be found. Only 'blank nodes', i.e., those with the `_` prefix, are local identifiers. The `rdf` and `rdfs` prefixes refers to RDF and RDFS definitions respectively. The prefix `mgl` refers to the MegaL ontology which contains definitions for both entity types (represented as OWL classes) and relationships types (represented as OWL properties).

Figure 8.4: Figure 8.3 expressed in MegaL/RDF.

```
_:CompanyDotXSD rdf:type mgl:File .
_:CompanyDotXSD rdfs:label "Company.xsd" .
_:CompanyDotXSD mgl:elementOf lang:XSD .
_:CompanyDotXSD mgl:inputOf _:CompanyXSD2CSDotBat .
_:CompanyElement mgl:partOf _:CompanyDotXSD .
_:CompanyElement rdf:type mgl:FileFragment .
_:CompanyElement rdfs:label "Company" .
... other fragments omitted ...

_:CompanyDotXSD mgl:partOf impl:xsdClasses .
_:CompanyDotXSD mgl:filename "./Company.xsd" .
_:CompanyElement mgl:xpathLocation
                  "//*[@name=\"Company\"]" .
... etc
```

The first block of triples shows some properties of the file `Company.xsd` including its decomposition into fragments.

The second block models links to online software artifacts. For instance the `impl` prefix refers to *101companies* implementations, `./Company.xsd` refers to a file name, and the property `mgl:xpathLocation` refers to a fragment of the schema file.

Figure 8.5: RDF-based links for the megamodel of figure 8.1.

Since all information in the *101companies* project is represented as RDF triples, links between *101companies* resources and external ones such as Wikipedia pages (i.e., Dbpedia[8]) resources in terms of RDF, this approach therefore enables the integration of megamodels and various other resources in the *Linked Data* global data space.

Interpretation of Linguistic Architecture

The megamodeling language *MegaL* is designed to model the linguistic architecture of software systems: the relationships between software artifacts (e.g., files), software languages (e.g., programming languages), and software technologies (e.g., code generators) used in a system. The present chapter delivers a form of interpretation for such megamodels: resolution of megamodel elements to resources (e.g., system artifacts) and evaluation of relationships, subject

[8] `http://dbpedia.org`

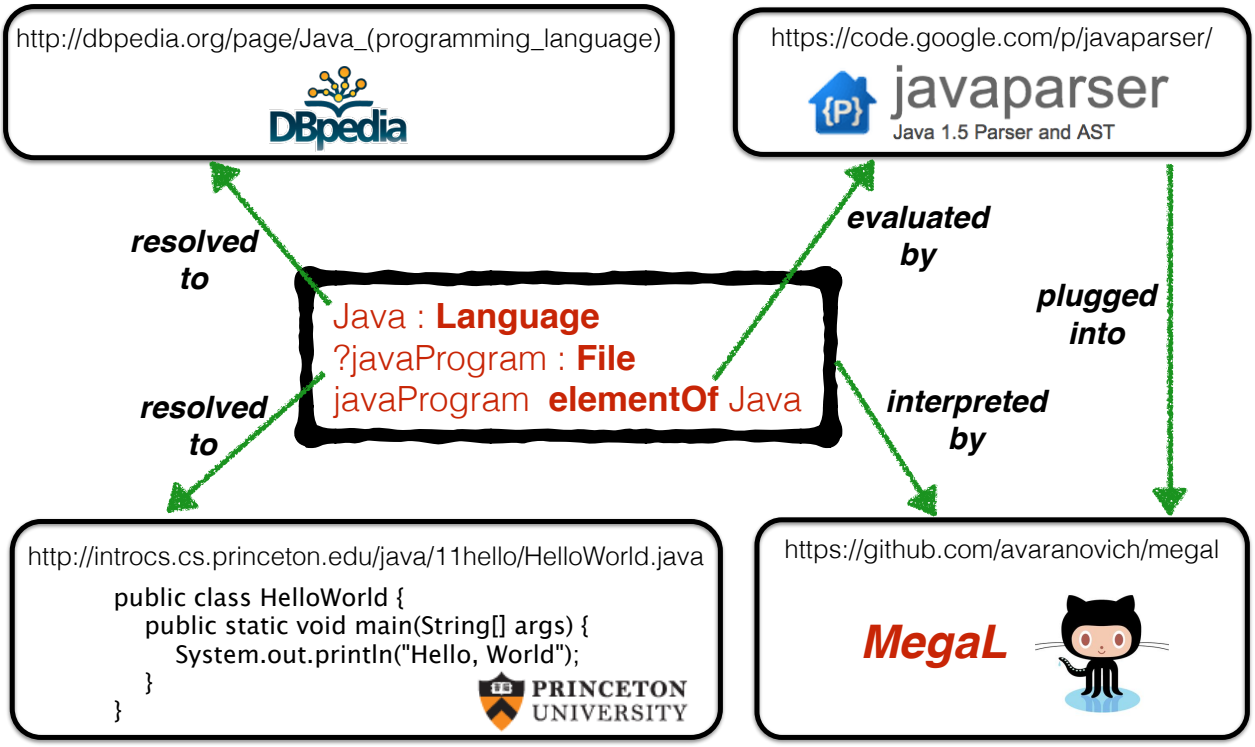


Figure 8.6: Interpretation of a megamodel

to designated programs (such as pluggable "tools" for checking). Interpretation reduces concerns about the adequacy and meaning of megamodels, as it helps to apply the megamodels to actual systems. We leverage *Linked Data* principles for surfacing resolved megamodels by linking, for example, artifacts to *GitHub* repositories or concepts to *DBpedia* resources. We provide an executable specification (i.e., semantics) of interpreted megamodels and we discuss an implementation in terms of an object-oriented framework with dynamically loaded plugins.

The present section fills in the notion of interpretation of megamodels. In this manner, we provide a general facility to apply megamodels to actual systems and to validate the claims that are made by megamodels.

Consider figure 8.6 as an illustration. The megamodel in the center of the figure declares a language Entity 'Java', a file entity parameter 'javaProgram', and a relationship between these entities such that the latter is an element of the former. Thus, the megamodel essentially describes a trivial Java-based system. The *MegaL* model can be interpreted as indicated in the figure, subject to a configuration and suitable plugins not shown here in detail. The interpretation entails these aspects:

- The language 'Java' is resolved in terms of the corresponding resource (page) according to the ontology provided by *DBpedia*.
- The parameter 'javaProgram' is resolved to the on-line version of a 'hello world' program on a web server at the *Princeton University*.
- The 'elementOf' relationship is evaluated by the Java parser of the *javaparser* project hosted on *Google Code*.

Characteristics of the Approach

We begin with characteristics of the basic *MegaL* approach, essentially inherited from [FLV12].

- *Extra models on top of systems*: A megamodel is seen as an abstraction over an existing system, added "after the fact", as opposed to forming a part of a system or expressing its composition, as in the case of model management.
- *Flexibility in terms of technological spaces*: Software technologies and systems may involve different technological spaces (such as grammarware or Javaware) without preference for a specific one such as MDE.
- *Decreased relevance of metamodels*: Metamodels or metamodel-like artifacts (e.g., schemas) are often unavailable or of limited relevance outside clean-room MDE. That is, we often refer to languages instead of metamodels (i.e., to conceptual entities rather than artifacts).

We continue with characteristics of interpretation.

- *Resource-based resolution of entities*: The entities in a megamodel may be resolved to resources that can be addressed with URIs, thereby enabling transparent reuse of existing ontologies (e.g., *DBpedia*) and repositories (e.g., *GitHub* repos). We leverage *Linked Data* principles.
- *Flexibility in terms of ontologies*: A comprehensive ontology for software engineering does not exist. Thus, different ontologies, subject to a plugin infrastructure, may be combined to assign meaning to the entity types and the conceptual entities in a megamodel.

- *Tool-based interpretation of relationships*: Relationships may be interpreted by designated programs ("tools") (e.g., a program implementing the membership test for a given language). This is supported by a plugin infrastructure, without favoring any particular semantics formalism.

- *Traceability recovery*: The actual semantics of transformation relationships is often unaccessible, as it is buried in software technologies. Thus, it may be preferable to construct a simplified and accessible variant of the actual semantics that provides insight due to its simplicity and through recovered traceability links for the involved artifacts.

Megamodeling with *MegaL*

This section describes the language elements of *MegaL*. We develop a relatively simple, illustrative megamodel, which will serve as the running example of the paper. All original aspects of interpretation are deferred to the next two sections.

MegaL Entities

All *entities* in a megamodel must get assigned an *entity type*. These types are also defined in *MegaL*. Entity types are declared as subtypes of the root entity type *Entity* or subtypes thereof. In this manner, a classification hierarchy (i.e., a taxonomy or ontology of entity types) is described. Here are some reusable entity types, as declared in actual *MegaL* syntax:

```
Set < Entity // Sets such as languages; see below
Language < Set // Languages as sets, e.g., sets of strings
Technology < Entity // Technologies in the sense of conceptual entities
Artifact < Entity // Artifacts as entities with a physical manifestation
File < Artifact // Files as a common kind of artifact
Function < Set // A function such as the meaning of a program
FunctionApplication < Entity // A particular application of a function
```

Entity types are used in entity declarations as those of figure 8.6:

```
Java : Language // Entity Java is of type Language
?javaProgram : File // Entity (parameter) javaProgram is of type File
```

We defer the discussion of the exact difference between entities and entity parameters (see the prefix '?') until we deal with resolution in subsection 8.7.3.

MegaL Relationships

All relationships between entities are instances of appropriate relationship types. Again, these types are defined in *MegaL*. Here are some reusable relationship types, as declared in actual *MegaL* syntax:

```
elementOf < Entity * Set // Membership in the set−theoretic sense
conformsTo < Artifact * Artifact // Conformance in the sense of metamodeling
defines < Artifact * Entity // Such as a grammar defining a language
domainOf < Set * Function // The domain of a function
rangeOf < Set * Function // The range of a function
inputOf < Entity * FunctionApplication // The input of a function application
outputOf < Entity * FunctionApplication // The output of a function application
partOf < Entity * Entity // A physical or conceptual containment relationship
```

Relationship types are used in declarations as this one in figure 8.6:

```
javaProgram elementOf Java
```

An Illustrative Megamodel

Let us capture key aspects of ANTLR usage in a software system. ANTLR[9] is (among other things) a parser generator that targets, for example, Java. Thus, ANTLR can be used to generate Java code for a parser for some language from a grammar given in ANTLR's grammar notation.

Entities

We declare the essential entities of ANTLR usage for parser generation:

[9] `http://www.antlr.org/`

```
ANTLR : Technology // The technology as a conceptual entity
Java : Language // The language targeted by the parser generator
ANTLR.Notation : Language // The language of parser specifications
ANTLR.Generator : Function ( ANTLR.Notation → Java )
?aLanguage : Language // Some language being modeled with ANTLR
?aGrammar : File // Some grammar defining the language at hand
?aParser : File // The generated parser for the language at hand
?anInput : File // Some sample input for the parser at hand
```

We leverage a notation for compound entities; see the names *ANTLR.Notation* and *ANTLR.Generator*. That is, ANTLR's notation for grammars is a conceptual constituent of the ANTLR technology as such. ANTLR's generation semantics is also such a constituent. The dot notation implies part-of relationships as follows:

```
ANTLR.Notation partOf ANTLR // Notation is conceptual part of technology
ANTLR.Generator partOf ANTLR // Generator semantics as well
```

We also leverage special notation for function entities; see the declaration of *ANTLR.Generator*. The arrow notation is desugared as follows:

```
ANTLR.Notation domainOf ANTLR.Generator
Java rangeOf ANTLR.Generator
```

Relationships

The previously declared entities engage in relationships as follows:

```
aGrammar elementOf ANTLR.Notation // The grammar is given in ANTLR notation
aGrammar defines aLanguage // The grammar defines some language
aParser elementOf Java // Java is used for the generated parser
ANTLR.Generator(aGrammar) ↦ aParser // Generate parser from grammar
anInput elementOf aLanguage // Wanted! An element of the language
anInput conformsTo aGrammar // Conform also to the grammar
```

The declaration of the '↦' relationship is actually a shorthand. We need a designated entity for the function application. Thus, desugaring yields this:

ANTLR.GeneratorApp1 : FunctionApplication

ANTLR.GeneratorApp1 ***elementOf*** *ANTLR.Generator*

aGrammar inputOf ANTLR.GeneratorApp1

aParser outputOf ANTLR.GeneratorApp1

Interpretation of Megamodels

Interpretation entails resolution of megamodel entities and evaluation of megamodel relationships. Resolution of entity parameters commences in a "pointwise" manner in that the parameters are mapped to specific URIs. Resolution of entities (as opposed to parameters) commences in a schematic manner, subject to "resolvers" (i.e., programs) for mapping entity names to URIs. Evaluation relies on "evaluators" (again, programs) for checking the relevant relationships and possibly producing traceability evidence. Pointwise mappings, resolvers, and evaluators are identified in a configuration that goes with a megamodel.

Megamodel Processing

The *MegaL* processor is a Java-based object-oriented framework. Given a megamodel and a configuration, the *MegaL* processor performs the steps summarized in figure 8.7.

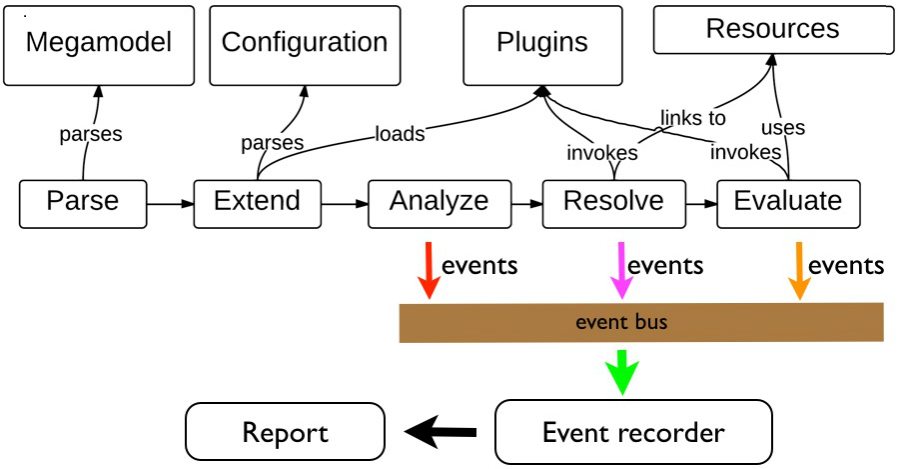


Figure 8.7: *MegaL* processing pipeline

That is, the megamodel is parsed into an abstract syntax tree based on a suitable object model. In the next step, the configuration file is processed and the corresponding plugins are dynamically loaded and associated with the appropriate AST nodes for entity and relationship types. In the next step, the megamodel and the plugins are analyzed for well-formedness and mutual compliance; see subsection 8.7.5 for a precise, formal account. Eventually, resolvers and evaluators are invoked. Resolution determines entity URIs and pings them for availability. Evaluation applies evaluators to the resources (the underlying content) of entities.

Along this pipeline, events are triggered and reported, making the process fully transparent. Any resolution and evaluation problems would also be reported along the way. For instance, the resolution of the 'Java' entity of figure 8.6 is reported as follows:

> Looking up entity type *Language*.

< Looked up entity type *Language* successfully.

> Linking entity *Java*.

- URI located via configuration.

< Linked entity *Java* successfully.

Ideally, all entities of a megamodel should be resolved (successfully) and relationships should be evaluated (successfully). However, this is not always feasible. That is, one may be missing resolvers or evaluators for some of the used entities and relationships. In this sense, interpretation may be incomplete, but this would be evident from the event report generated by megamodel processing.

Configuration of the Interpretation

Configuration relies on a simple JSON-based DSL with language elements for URI mapping and registration of mapping resolvers as well as evaluators.

Figure 8.8 shows the configuration for the introductory Java example. In the "links" section, the parameter 'javaProgram' is resolved in a pointwise manner so that it links to the 'hello

```
{
  "links" : [ {
       "name": "javaProgram",
       "resource" : "http://introcs.cs.princeton.edu/java/11hello/HelloWorld.java"
    } ],
  "resolvers" : [ { "plugin" : "megal.resolvers.dbpedia" } ],
  "evaluators" : [ {
       "plugin" : "megal.evaluators.FileElementOfLanguage"
       "checkers" : [ { "plugin" : "megal.checkers.languages.Java" } ]
    } ]
}
```

Figure 8.8: The configuration for the megamodel in figure 8.6

world' program on Princeton University's web server. In the 'resolvers' section, we register a *DBpedia* resolver which is prepared to resolve entity names of the language type to resource URIs on *DBpedia*. In particular, this resolver handles the 'Java' entity of the megamodel. In the "evaluators" section, we register an evaluator '....FileElementOfLanguage', which can evaluate 'elementOf' relationships when the left operand is a file resource and the right operand is a language. The 'elementOf' plugin relies on second-level plugins, 'checkers', for individual languages. In the configuration file, we register indeed a checker (i.e., a membership test) for 'Java'. This checker is a wrapper around the Java parser of the javaparser project. In the *MegaL* project, we aim at collecting all such plugins as consolidated and reusable interpretations of well-defined resources identified through *Linked Data* principles.

Application to the Running Example

Let us consider the interpretation of the megamodel for ANTLR, as introduced in subsection 8.7.2. To begin with, we should pick some software system that exercises ANTLR. Clearly, there is no shortage of such systems. As it happens, the *MegaL* implementation itself also uses ANTLR. Thus, let us apply the *MegaL* model for ANTLR to *MegaL*'s parser.

Entity Parameters

They are resolved as follows:

aLanguage The language at hand is fixed to be *MegaL*. A link is needed. We choose to link to the language's *GitHub* project.[10]

[10] `https://github.com/avaranovich/megal/`

aGrammar The grammar at hand is the ANTLR-based parser specification of *MegaL*. Thus, we need to link to a specific file *.../MegaL.g4* in said repository.

aParser The parser at hand is a Java source-code file *.../MegaLParser.java* that was generated by ANTLR—again, a file in said repository.

anInput Any *MegaL* source could be linked here. We choose to link to *MegaL*'s prelude with the predefined types, as discussed in subsection 8.7.1—again, a file in said repository.

Entities

They are resolved as follows:

Java A *DBpedia* resolver is used as explained in figure 8.7.3.

ANTLR The *DBpedia* resolver may not be used here because we rely on the fact that *ANTLR* is a compound entity with constituents, as listed below. In the 101companies project [FLSV12], software technologies, languages, and concepts are organized in an ontological manner. There is a suitable composition-aware '101companies' resolver for technologies, which links *ANTLR* to a resource.[11]

ANTLR.Notation Use the same resolver as for *ANTLR*.

ANTLR.Generator Use the same resolver as for *ANTLR*.

ANTLR.GeneratorApp1 An application is a pair of the input and output entities. Thus, an application entity is resolved, at a basic level, once input and output are resolved. A more advanced resolution entails the identification of a system artifact's fragment that expresses the application. More specifically, the application of ANTLR's generator could be pinpointed in a build script.

Relationships

They are evaluated as follows:

[11] `http://101companies.org/resources/technologies/ANTLR`

elementOf The evaluator *....FileElementOfLanguage* of figure 8.7.3 is enriched by additional second-level plugins (i.e., "checkers") to serve *aLanguage* (thus, *MegaL*) and *ANTLR.Notation* – in addition to just *Java* previously.

conformsTo Another evaluator *....FileConformsToFile* is needed. It is the language of the right operand that defines the applicable conformance semantics. The result of a conformance test can be richer than just a Boolean value; it may be a set of traceability links between the operands; see subsection 8.7.4.

defines An evaluator *....Triangle* is used which simply checks that a megamodel with the relationship '*x defines y*' also contains the relationships '*z* ***elementOf*** *y*' and '*z* ***conformsTo*** *x*'. This is Favre's triangle [Fav05a].

'↦' In fact, we evaluate *ANTLR.GeneratorApp1* ***elementOf*** *ANTLR.Generator* after desugaring. That is, we need to check that *aParser* is the output generated by *ANTLR.Generator* from *aGrammar*. There are several options for checking function applications. As suggested earlier, we may pinpoint the actual application (e.g., in a build script). We could also pinpoint traces of the application (e.g., the Java comment included by ANTLR into the generated source file). We could also apply the function (i.e., run the generator) and compare the result with the existing output artifact. Ultimately, we may analyze input and output and establish problem-specific traceability links based on our understanding of the mapping, thereby also sharing our understanding with others. This is illustrated below.

Traceability Recovery

Traceability links may be recovered, for example, for conformance relationships and function application relationships (i.e., "transformations"). This is illustrated for the application of *ANTLR.Generator*. The input, *aGrammar*, is essentially a list of ANTLR rules with unique nonterminals on the left-hand sides. The output, *aParser*, is essentially a Java file exercising certain code patterns. In particular, for each nonterminal n, there is a corresponding method that implements the rule:

```
public final nContext n() throws RecognitionException { ... }
```

```
// Get methods of interest
val methods = aParser.getMembers()
    .filter(x => x.isInstanceOf[MethodDeclaration])
    .filter(x => ((x.getThrows().map(y => y.getName()).
        contains("RecognitionException"))))
// Get grammar rules
val rules = aGrammar.rules
// Check 1:1 correspondence of names including the same order
val isAlligned = methods.zip(rules).forall(x => x._1.getName().equals(x._2))
```

Figure 8.9: Scala-based traceability check for ANTLR's generator

Thus, a suitable approach to traceability recovery is to retrieve nonterminals from the grammar and all relevant methods from the generated Java source and to check for a one-to-one correspondence; see figure 8.9 for illustration. For brevity, we show simplified evaluator code that checks only for correspondence, while the actual evaluator collects traceability links (i.e., pairs of URIs) of the following form:

$$\langle \text{ "http://.../MegaLParser.java/class/MegaLParser/method/megamodel/1" , "http://.../MegaL.g4/grammar/megal/rule/megamodel/1" } \rangle$$

The URIs describe the relevant fragments in a language-parametric manner. That is, the URIs start with the actual resource URI for the underlying artifact. The rest of the URI, which is underlined for clarity, describes the access path to the relevant fragment. To this end, syntactical categories of the artifact's language (see "class" and "method" versus "rule") and names of abstractions (see "megamodel") are used. (We note that "megamodel" is the first nonterminal, in fact, the startsymbol of the grammar for *MegaL*.)

Executable Specification of *MegaL*

The following specification of *MegaL* clarifies the meaning of entity resolution and relationship evaluation. The specification assumes an abstract *MegaL* syntax—without convenience notation for functions and function applications and without consideration of compound entities. The specification does also not cover traceability recovery (subsection 8.7.4).

Specification Style

The specification is a deductive system, as commonplace for type systems and operational semantics. The specification is executable – directly as a logic program in Prolog.[12] *MegaL* is not a regular programming language. Thus, it requires some insight to identify counterparts for what is usually referred to as static versus dynamic semantics.

We assume that interpreted megamodels consist of two parts: the actual megamodel and (the description of) the interpretation—the latter as an abstraction of the configuration, resolvers, and evaluators used in the actual implementation of subsection 8.7.3. Given a megamodel *MM* and an interpretation *Interp*, the informal process of figure 8.7 is formally described as follows:

```
process(MM, Interp) ⟹
  megamodel(MM), % Inductive syntax definition of megamodels
  okMegamodel(MM), % Well−formedness relation for megamodels
  interp(Interp), % Inductive syntax definition of interpretations
  okInterp(Interp), % (Trivial) well−formedness of interpretations
  correct(MM, Interp), % Correctness of interpretation w.r.t megamodel
  complete(MM, Interp), % Completeness of interpretation w.r.t. megamodel
  evaluate(MM, Interp). % Evaluation of relationships
```

We discuss the contributing judgments in turn.

Abstract Syntax of Megamodels

A megamodel is a list of *declarations*. There are declarations for entity-types (***etdecl***s), relationship types (***rtdecl***s), entities (***edecl***s), entity parameters (***pdecl***s), and relationships (***rdecl***s). The declared names are atoms ("ids") and so are all the references to the names. Thus:

```
megamodel(MM) ⟹ map(decl, MM).
decl(etdecl(SubT, SuperT)) ⟹ atom(SubT), atom(SuperT).
```

[12]The specification is available online `http://softlang.uni-koblenz.de/megal-interpretation/` Basic logic programming is used, except for higher-order predicates [NS00] for list processing: *map* (for applying a predicate to the elements of a list), *filter* (for returning the elements that satisfy a predicate), and *zip* (for building a list of pairs from two lists).

```
decl(rtdecl(R, T1, T2)) ⟹ atom(R), atom(T1), atom(T2).
decl(edecl(E, T)) ⟹ atom(E), atom(T).
decl(pdecl(E, T)) ⟹ atom(E), atom(T).
decl(rdecl(R, E1, E2)) ⟹ atom(R), atom(E1), atom(E2).
```

Well-formedness of Megamodels

Well-formedness is defined as a family of relations, as usual, on the syntactical domains. Well-formedness ensures that all referenced names of entity types, relationship types, and entities (or parameters) are actually declared. (This is part of what we call "Analyze" in figure 8.7.) We omit most of these routine definitions; a more insightful detail is well-formedness of relationship declarations:

```
okRDecl(MM, rdecl(R, E1, E2)) ⟹
  member(rtdecl(R, Tl1, Tr1), MM), % RType exists
  getEntityType(MM, E1, Tl2), % Type of left entity
  getEntityType(MM, E2, Tr2), % Type of right entity
  subtypeOf(MM, Tl2, Tl1), % Left type Ok
  subtypeOf(MM, Tr2, Tr1). % Right type Ok
```

That is, any declared relationship between two entities E_1 and E_2 must be based on a relationship-type declaration for the same relationship symbol R with entity types Tl_1 and Tr_1 in such a way that the actual entity types Tl_2 and Tr_2 are subtypes of the declared types Tl_1 and Tr_1. Subtyping is defined in terms of the type hierarchy defined by entity-type declarations. This is subtyping like in a single-inheritance OO programming language.

Abstract Syntax of Interpretations

We invent a representation of interpretations (say, definitions) of parameters (***pdef***s), entity types (***etdef***s), and relationship types (***rtdef***s). In this manner, we abstract from the plugins of the OO framework and the configuration as discussed in subsection 8.7.3. Thus:

interp(Interp) $\Longrightarrow$ *map(def, Interp).*

*def(**pdef**(E, U))* $\Longrightarrow$ *atom(E), uri(U).*

*def(**etdef**(T, F))* $\Longrightarrow$ *atom(T), function(F, [atom], [uri]).*

*def(**rtdef**(R, T_1, T_2, P))* $\Longrightarrow$ *atom(R), atom(T_1), atom(T_2), predicate(P, [uri, uri]).*

That is, a parameter definition (***pdef***) associates an entity parameter E with a URI U; an entity-type definition (***etdef***) associates an entity type T with a function F mapping entity names to URIs; a relationship-type definition (***rtdef***) associates a relationship type $\langle R, T_1, T_2 \rangle$ with a predicate P on entity URIs. Thus, ***etdef***s and ***rtdef***s model resolvers and evaluators, respectively. We view the aforementioned predicates and functions here as being defined by their extension (i.e., a suitable set of tuples). Thus:

predicate(Tuples, Types) $\Longrightarrow$ *set(Tuples), map(tuple(Types), Tuples).*

function(Tuples, Domain, Range) $\Longrightarrow$ *... % likewise for functions*

tuple(Types, Tuple) $\Longrightarrow$ *zip(Types, Tuple, TT), map(apply, TT).*

In the actual implementation, resolvers and evaluators are of course programs that may retrieve resources via the URIs over the Internet.

Correctness and Completeness

We present correctness and completeness as two aspects of well-formedness of the megamodel-interpretation couple. (We do not discuss well-formedness of interpretations by themselves, as there are only a few trivial constraints.)

Correctness means that an interpretation does not provide any definitions that are not used anymore by the associated megamodel. Provision of superficial definitions may be acceptable, though, in practice.

Completeness means that an interpretation suffices to resolve all entities or parameters and to evaluate all relationships for a given megamodel. As discussed, in practice, we do not necessarily require completeness, as we may be unable to resolve certain entities or to evaluate certain

relationships, at a given point. However, ambiguities regarding resolution or interpretation should be reported.

Correctness and completeness are again specified as families of relations. For example, here is the judgment for establishing correctness of relationship-type definitions w.r.t. a megamodel.

```
correctRTDef(MM, rtdef(R, Tl1, Tr1, _)) ⟹
  okT(MM, Tl1), % Left entity type exists
  okT(MM, Tr1), % Right entity type exists
  member(rtdecl(R, Tl2, Tr2), MM), % Relationship type exists
  subtypeOf(MM, Tl1, Tl2), % Definition vs. declaration (left)
  subtypeOf(MM, Tr1, Tr2). % Definition vs. declaration (right)
```

That is, for each relationship-type definition of the interpretation, we can find a corresponding declaration of the megamodel that uses the same or more general entity types.

Let us also consider the counterpart from the family of relations for completeness (i.e., the relation for establishing that a given relationship can be evaluated unambiguously by a definition). This judgment is involved – it is comparable to resolution of names in a non-trivial programming language.

```
% Relationship−type definition unambiguous
completeDecl(MM, Interp, rdecl(R, El, Er)) ⟹
  getRTDef(MM, Interp, R, El, Er, _).

% Determine suitable relationship−type definition
getRTDef(MM, Interp, R, El, Er, RTDef) ⟹
  getEntityType(MM, El, Tl), % Look up left entity type
  getEntityType(MM, Er, Tr), % Look up right entity type
  filter(applicableRTDef(MM, R, Tl, Tr), Interp, RTDefs),
  reduceRTDefs(MM, RTDefs, RTDef).

% Applicability of a relationship−type definition
applicableRTDef(MM, R, Tl1, Tr1, rtdef(R, Tl2, Tr2)) ⟹
```

```
  subtypeOf(MM, Tl1, Tl2),
  subtypeOf(MM, Tr1, Tr2).


% Eliminate more general relationship−type definition
reduceRTDefs(_, [RTDef], RTDef). % One rtdef left
reduceRTDefs(MM, RTDefs1, RTDef) ⟹
  member(RTDef1, RTDefs1), % Pick some rtdef
  member(RTDef2, RTDefs1), % Pick some rtdef
  RTDef1 ≠ RTDef2, % Two different rtdefs
  RTDef1 = rtdef(R, Tl1, Tr1, _),
  RTDef2 = rtdef(R, Tl2, Tr2, _),
  subtypeOf(MM, Tl1, Tl2),
  subtypeOf(MM, Tr1, Tr2),
  delete(RTDefs1, RTDef2, RTDefs2), % Remove the more general rtdef
  reduceRTDefs(MM, RTDefs2, RTDef).
```

This approach is similar to instance resolution in Haskell [HHJW96], the one for multi-parameter type classes with overlapping instances specifically [SSS06]. That is, definitions ("instances" in Haskell terms) are not proactively rejected by themselves – just because they are overlapping in some sense. Instead, any given relationship is considered as to whether it can be associated uniquely with a definition that is more specific than all other applicable definitions.

Evaluation of Relationships

Evaluation is straightforward at this stage, as all preconditions have been established. That is, entities or parameters thereof can be replaced by URIs and relationships can be evaluated on the URIs for the arguments. Thus:

```
evaluateDecl(MM, Config, rdecl(R, El, Er)) ⟹
  getRTDef(MM, Config, R, El, Er, rtdef(_, _, _, P)),
  getEUri(MM, Config, El, Ul),
  getEUri(MM, Config, Er, Ur),
```

```
  applyPredicate(P, [Ul, Ur]).

% Get URI for entity via definition
getEUri(MM, Config, E, U) ⟹
  getEntityType(MM, E, T), % Look up entity type
  member(etdef(T, F), Config), % Look up definition
  applyFunction(F, [E], [U]). % 'Resolve'

% Application of extension−based predicates and functions
applyPredicate(Tuples, X) ⟹ member(X, Tuples).
applyFunction(Tuples, Arg, Res) ⟹ append(Arg, Res, X), member(X, Tuples).
```

Soundness (i.e., alignment between "type system" and "semantics") follows trivially in this approach – as the completeness judgment immediately ensures that all instances of entity resolution and relationship evaluation can be attempted. Thus, the only remaining option for *evaluateDecl* to fail is that a resolution was not successful or a specific relationship failed.

Related work

Megamodeling. Megamodeling is somewhat established in the communities of modeling and model-driven engineering. Existing forms of megamodels do not cover the range of linguistic relationships of MegaL (such as 'elementOf', 'subsetOf', and 'correspondsTo'); they have not been used in a manner to understand software technologies across technological spaces. We look at representative examples. In [SCFC09], megamodeling is applied to the human-computer interaction domain. In [Fav05b], a UML/OCL-based megamodel of MDA/MDE is provided, thereby supporting reasoning about MDA/MDE. In [VSG11], megamodeling is used for organizing and utilizing runtime models and relations in a model-driven manner while also supporting a high level of automation. In [JVB+10], megamodeling is used to coordinate "heterogeneous" models in the sense of conforming to a multiplicity of metamodels expressed in different DSLs. In [Gra07], megamodeling is applied to model transformation with the objective of supporting

the evolution of software architectures. In [HSG10], some forms of megamodels and associated applications are surveyed.

Some model transformation approaches involve explicitly chains or compositions of transformations, perhaps even involving different model transformation languages and dealing with different "modeling spaces". Such compositions can be viewed as a form of executable megamodels. In [LR11], the authors motivate the need for a precise semantics for model-to-model transformations, thereby enabling verification of correctness for compositions, thereby, in turn, encouraging reusability.

Foundations of Modeling. Our work is substantially inspired by recent efforts on the foundation of modeling from which we derive basic idioms of megamodeling. We rely on established relationships such as 'conforms to' and 'element of' [Fav05a, Küh06b]. Further, there is the multi-faceted 'represents/models' relation [MFB09, MFBC11]. We derive the correspondence relation from the field of model management. In [DXC11, DMC12], a categorical approach to intermodel mappings including heterogeneous (meta)model correspondences is developed.

Viewpoints. We compare *MegaL* with several approaches to megamodeling. The Atlas MegaModel Management approach (AM3) conveys the idea of modeling in the large, establishing and using general relationships, such as conformance, and metadata on basic macroscopic entities (mainly models and metamodels) [BJRV05]. Based on the assumption that all managed artifacts are models conforming to precise metamodels, a solution for typing megamodeling artifacts is proposed in [VJBB11]. Model typing is based on the conformance relationship; metamodels are used as types. *MegaL* is clearly not restricted to modeling resources and does not require the existence of metamodels. Also, *MegaL*'s approach to megamodel interpretation provides an open, heterogeneous type system.

A formal, graph-oriented view on megamodels is considered in [DKM13]; entities are vertices and relations are edges between them. It is argued that the semantics of relations are hidden in the type name and are not presented in the megamodel. To fill this gap, the authors zoom into

nodes and edges and disassemble them into more elementary building blocks. In the case of *MegaL*, such a formal analysis of the relationships is less relevant, as it is not directly applicable to actual software projects and technologies. Instead, as shown in subsection 8.7.4, we leverage tool-based relationship evaluators with optional traceability recovery. *MegaL* is also influenced by existing megamodeling patterns and idioms discovered in theoretical work [FN04b, Fav05a, DKM13].

In a comprehensive survey [WvP10] of traceability in MDE, the authors conclude, that traceability practices are still emerging, specifically in the MDE context. *MegaL*'s interpreted megamodels may associate entities in relationships with traceability links, as it was shown in subsection 8.7.4. This approach is again heterogeneous in terms of the technological spaces; it assumes a language-parametric approach to fragment location. Traceability is also used in megamodeling for models at runtime [SNG10], where high-level relationships between models are derived from observable low-level traceability between model elements.

A type system and a type inference algorithm for declarative languages with constraints for MDE are presented in [JSB12]. Elsewhere [BJ06], OCL [Gro06] constraints and ATL rules [JK06] are used to implement consistency and conformance checking.

Megamodels of metamodels and model transformations are organized into an architectural framework [FM06], which promotes re-usability of architectural elements and realizes architectural descriptions [HMMP10]. We plan to re-implement such descriptions in *MegaL*, thereby providing evidence of its usefulness as an architecture description language.

MegaL relies on the resources being exposed via HTTP and uniquely identifiable. Such resources can be directly exposed via web servers and web-accessible source control systems. Another promising direction is to apply *Linked Data* [BHBL09] principles, which allows attaching rich metadata. *MegaL* already applies such principles (e.g., in the sense of the *DBpedia* and 101companies resolvers). *Linked Data* principles are also leveraged in [KFH+12b] in a related manner for the purpose of exposing facts about artifacts in software repositories.

Architectural frameworks and notations. Formal notations and models can be used to characterize and reason about a system design, thus a part of software architecture discipline [SG96]. While the intuition might suggest a deeper relation between *MegaL* and the existing architectural notations, *MegaL* primary goal is to operationalize the technology modeling concepts developed in this thesis. [AG97] distinguishes the implementation and the interaction relationships between part of the system being described. *MegaL* primarily focuses on the interaction part and consistency checking, and does not cover the breath and depth in terms of other possible aspects of the software system. For instance, Zachman architecture framework [Zac87] explicitly states, that entity-relationship diagrams are not able to express all the constraints and ignore the operations performed by and on the entities [SZ92]. *MegaL* considers some relations executable, i.e. in a form of a function application, capturing input and output and leaving out the details of the operation itself.

Conclusion

We have developed a form of technology modeling that targets the linguistic architecture of software technologies and software products. Megamodels serve as cognitive models for the benefit of software engineers, software linguists, and others.

We have equipped the megamodeling notion for the linguistic architecture of software systems with a language mechanism for resolving entities, capturing traceability between them, and evaluating relationships. Our approach is not tailored to MDE. We applied the approach to a megamodeling scenario that indeed involves elements of Javaware and grammarware. We formalized the key ideas of interpreted *MegaL* models in a deductive system and described an open-source implementation. Without this enhancement, megamodeling does not provide enough validated insight into actual systems.

The types of megamodeling relationships with the underlying entity types represent patterns of the linguistic architecture of software systems. *MegaL* has already been applied to some typical scenarios of technology usage, as they are demonstrated by software systems in the *101companies* chrestomathy, thereby capturing important entity and relationship types. It

remains to complete a megamodeling ontology in a systematic manner so that we can be confident that all such major types have been discovered.

Chapter 9

Evaluation of *101companies* Software Chrestomathy

This chapter evaluates the *101companies* software chrestomathy, according to a design research methodology selected in this work.

Introduction

According to the design research methodology, we need to validate the usefulness of the artifact, created in this work – *101companies* software chrestomathy. It should facilitate a certain number of research directions in an open and reproducible manner, based on the underlying infrastructure. In this chapter, we present three research scenarios, from the research agenda on software chrestomathies [Lae13].

Polyglot analysis and transformation. In section 9.2 we describe and validate a method for comparing programming languages or technologies or programming styles in the context of implementing certain programming tasks. To this end, we analyze a number of "little software

This chapter is mainly based on two publications with some additional related work: [LLSV14b] Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz and Andrei Varanovich. Comparison of Feature Implementations across Languages, Technologies, and Styles. In Proc. of IEEE CSMR-WCRE 2014. IEEE, 2014. [LVL+14] Ralf Lämmel, Andrei Varanovich, Martin Leinberger, Thomas Schmorleiz and Jean-Marie Favre. Declarative Software Development (Distilled Tutorial). In Proc. of PPDP 2014.

systems" readily implementing a common feature set. We analyze source code, structured documentation, derived metadata, and other computed data. More specifically, we compare these systems on the grounds of the NCLOC metric while delegating more advanced metrics to future work. To reason about feature implementations in such a multi-language and multi-technological setup, we rely on an infrastructure that enriches traditional software artifacts (i.e., files in a repository) with additional metadata for implemented features as well as used languages and technologies.

Integration with teaching. In section 9.3 an introductory functional programming course is layered directly on top of *101haskell*. The validation boils down to the argument that the available content, in the intrinsic form (semantic wiki, interlinked source code, interlinked external resources) provides the foundation of a viable course.

Similarity management. In section 9.4 we summarize the work done in the context of code-sharing management and product line engineering: a software chrestomathy contains "little systems" (contributions) that are similar by design, and hence, one can expect to detect clones. An approach to use clone detection is to actually manage the similarity of contributions to help with understanding and evolution. The referenced work was done by several members of the *101companies* collaboration with the external research group. *101companies* and its infrastructure enabled the research with the corpus and some software analysis capabilities. The author of this work was not a part of the collaboration, and it is used in this section for one purpose: to provide an additional proof of the applicability of software chrestomathy across research domains.

Comparison of Feature Implementations across Languages, Technologies, and Styles

Consider the following research question: "*Which programming language or technology or style is most suited for implementing a certain programming task*"

For instance, let us pick the task of "totaling salaries of employees in a company", which is a possible functional requirement for an information system. This task could be implemented in any given language in many different styles, making use of many different programming technologies. The task could be implemented in *Java* in such a manner that the company structure is represented in either plain objects or *DOM*-based objects for *XML*. The latter option is illustrated by using the *JDOM*[1] API:

```
// doc represents an input XML document
Iterator<?> iterator =
  doc.getDescendants(new ElementFilter("salary"));

// Iterate over all salary elements
while (iterator.hasNext()) {
  Element elem = (Element)iterator.next();
  Double salary = Double.valueOf(elem.getText());
  total += salary;
}
```

Consider another implementation of the task, this time in *Haskell* while assuming a designated data model for the company structure and the use of a generic programming style ("Scrap your boilerplate" (SYB) [LJ03b]) for processing the data, also subject to a designated library.[2] Thus:

```
-- Traverse "everything" to aggregate all floats (salaries)
total :: Company → Float
total = everything (+) (extQ (const 0) id)
```

We want to compare many such different implementations. In principle, feature implementations can be discovered using static text retrieval techniques [MM03], dynamic program analysis [EKS03], or combinations of the two [PGM+07]. For the shown samples, it may be straightforward to locate feature "Total". Consider another feature regarding the hierarchical organization of the company structure in terms of top-level departments breaking down hierarchically into sub-departments. We refer to this feature as "Hierarchical company". At the code level, this feature may be associated with a recursively defined type for departments. The

[1] http://www.jdom.org/
[2] http://hackage.haskell.org/package/base-4.6.0.1/docs/Data-Data.html

Haskell implementation does indeed define such a data model (not listed here). The feature's implementation is *implicit* though in the *JDOM*-based *Java* implementation because of the lack of an explicit data model.

We describe and execute a methodology for the comparison of feature implementations across languages, technologies, and styles. We leverage the *101companies* project, as it provides a suitable chrestomathy [Läm13] (i.e., a collection of systems exercising different languages, technologies, and styles), while being comparable in terms of implemented features. The comparison relies on feature location and metrics calculation. *101*'s infrastructure is readily leveraged for all required analysis. All resources including available metadata and computed information are organized and exposed according to *Linked Data* principles [BCH07, HB11b] so that they are conveniently explorable; both programmatic and interactive access is possible. The relevant formats and the underlying ontology are openly accessible and documented. The methodology is "context-aware" in that different viewpoints and interactions are considered for feature location. (We adopt this meaning of context awareness from [PXT+11].) For instance, we distinguish "as intended" features versus "as implemented" (i.e., features specified by the system documentation versus features located by the analysis of source code).

Eventually, we compare the same feature set in different systems (i.e., implementations or "contributions" according to *101)* in terms of the NCLOC metric. We postpone more advanced metrics to future work. The methodology includes non-automated aspects for the sake of selecting targets for comparison, the validation of results, and their interpretation.

The Underlying Infrastructure

The infrastructure for creating, analyzing and accessing resources is briefly summarized in figure 9.1. We expose and access all involved resources according to *Linked Data* principles.

System documentation is provided by a wiki platform—the *101wiki*. A textual description is enriched with a metadata section for semantic properties; see the upper left corner of figure 9.1. Authoring such properties is part of the structured documentation process that goes beyond plain source-code documentation. For instance, some properties specify the features thought to

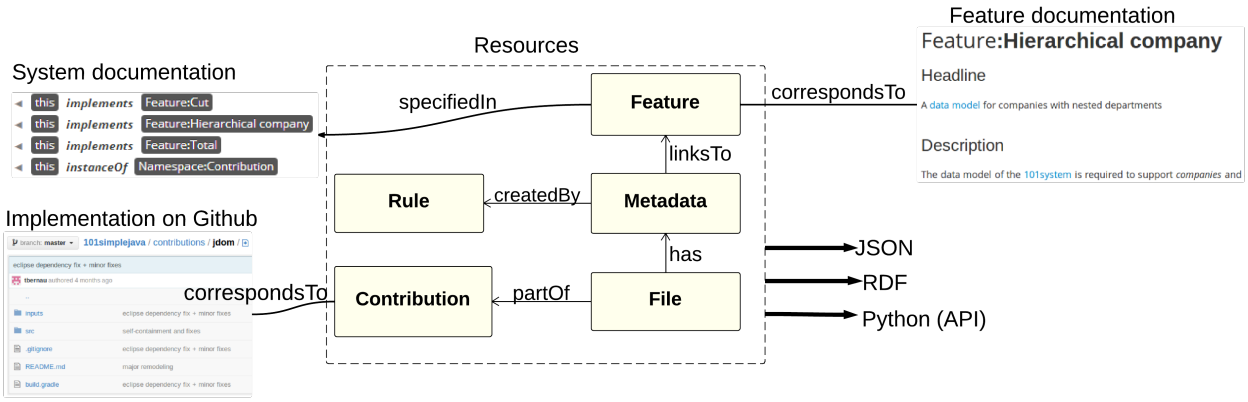


Figure 9.1: Overview of the underlying infrastructure hinting also at 'links' in the sense of *Linked Data*

be implemented by the system. Such knowledge is exposed through "links" – some of them are shown in figure 9.1. Features are also documented on the *101wiki*; see the upper right corner of figure 9.1.

Source code is maintained in a (GitHub-based) repository – the *101repo*; see the lower left corner of figure 9.1. We use a computational infrastructure, the *101worker*, to analyze source files for the sake of deriving resources and dumps, which are again published. In the present paper, we are specifically interested in computed metadata as follows: *a)* the features implemented by source files, subject to feature location; *b)* the languages and technologies used by the source files, subject to simple heuristics; *c)* data for source-code metrics, in fact, NCLOC.

To produce *a)* and *b)* we use simple, automated rules expressed in *101meta* – a language for associating metadata units with files. We introduced this language in chapter 5. For instance, the Java code from section 9.2 is associated with these units, rendered here in JSON notation:

```
[ { "language" : "Java" },
  { "technology" : "JDOM" },
  { "feature" : "Total" } ]
```

To compute derived resources such as metrics data in the sense of *c)* we plug designated modules into the *101worker*. A metrics module associates the source file f (i.e., a "primary" resource) with a "derived" resource f.metrics.json with metrics data formatted again in JSON.

Methodology

We use the following methodology for comparison of feature implementations across languages, technologies, and styles in terms of the diversity present in *101:*

Feature selection. Determine a set of features that are often enough implemented to be promising in terms of a comparison. Some features in *101* are indeed very popular. The feature set needs to be small enough to make it relatively easy to validate the correctness of all results, as manual validation and interpretation is necessary; see below.

Implementation selection. Determine a set of implementations (contributions in *101'*s terminology) that are promising in terms of comparison. We can hardly assume that feature location and other algorithmic aspects of the methodology are completely robust across the diversity at hand. Thus, the set of selected implementations must be small enough to allow validation. Here we note that *101'*s contributions often modularize feature implementations with one source file per feature, thereby enabling straightforward consideration of implementations even with supersets of the selected features.

Language, technology, and style detection. Perform language and technology detection. In this paper, we only care about "programming style" in so far as it is hinted at by the use of technologies. When interpreting comparison results, we may very well take knowledge of styles into account.

Feature location. Locate the selected features in the selected implementations. Some features may be missed because of their implicit implementation and thus require manual tagging. More generally, feature location must be validated. For simplicity, we do not admit implementations where any source file mixes the feature selected with additional features, as this would require a degree of feature location that is not available to us across many languages.

```
# Walk over all files of all contributions and build a mapping from features to files
featureIndex = {}
for folders, files in walk(Namespace('contributions')):
    for file in files:
        for feature in file.features:
            featureIndex.setdefault(feature, []).append(file)

# Validation step
validateAutomaticTagging(featureIndex)
for contribution in config.selectedImpls:
    for f in (set(contribution.implements) & config.selectedFeatures):
        if not any(file.member =contribution for file in featureIndex.get(f, [])) and
            not f in config.implicit.get(contribution.name, []):
                ... # Feature not found, error handling kicks in

# Count NCLOC for every file that belongs to a selected implementation and is concerned with a selected feature
contributionIndex = {}
for feature in config.selectedFeatures:
    for file in featureIndex.get(feature, []):
        member = file.member
        if member in config.selectedImpls and file.relevance ='system':
            contributionIndex[member.name] =
                contributionIndex.get(member.name, 0) + file.metrics.ncloc
```

Figure 9.2: Idealized *Python* code for the comparison. The code operates on *Linked Data*.

Metric computation. In this work, we limit ourselves to NCLOC as metric, as there is hardly any other metric available at this point for many different languages, but see the discussion in subsection 9.2.5. The metric is computed for source files identified by feature location or manual tagging. The metric values are summed up for all the files of an implementation.

Interpretation. The different NCLOC sums are interpreted by an expert who consults the selected implementations.

Execution

We facilitate feature and implementation selection by building a mapping from all features to the files implementing a feature; see the first code block in figure 9.2. To this end, we use the *Linked Data* access path to the *101repo* holding (all source-code files of) all implementations of *101*. Feature location has been applied to the *101repo* upfront. We note in passing that *101repo* is a confederated repository with many distributed, physical repositories, but the *Linked Data* access path shields the programmer from such complexity.

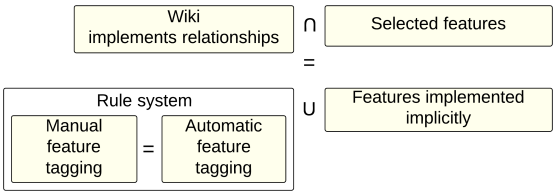


Figure 9.3: Objective of the validation of feature location

Based on inspection of the mapping, we should pick features that stand out as being popular and modularly implemented; see the parameter `config.selectedFeatures` in figure 9.2. In this paper, we pick *101*'s features for totaling and cutting salaries on top of the data model for hierarchical companies. Eventually, we also pick a few implementations; see the parameter `config.selectedImpls` in figure 9.2. One selection is discussed in subsection 9.2.4.

Eventually, we compute a mapping from *101*'s contributions to NCLOC; see the last code block in figure 9.2. To this end, we iterate over the selected features and, in turn, over all files concerned with each feature. If the file belongs to a selected implementation and is also tagged as being "system" relevant (as opposed to generated code or included third-party code), then the file's NCLOC value is counted towards the general NCLOC value for the associated implementation ("contribution").

Feature location is validated semi-automatically; see the middle code block in figure 9.2 and see figure 9.3 for a summary of the validation objective. In particular, the results of rule-based feature location are compared with the documented ("specified") features, thereby revealing potential discrepancies between feature location and documentation. (As an aside, for several contributions, feature location was also performed manually, thereby enabling the validation of the identified source files).

Feature location may miss "implicit" implementations, as discussed in the introduction. This situation may be confirmed by code inspection. In those cases, the discrepancies between feature location and documentation are manually silenced by declaring the implicit status; see the parameter `config.implicit` in figure 9.2.

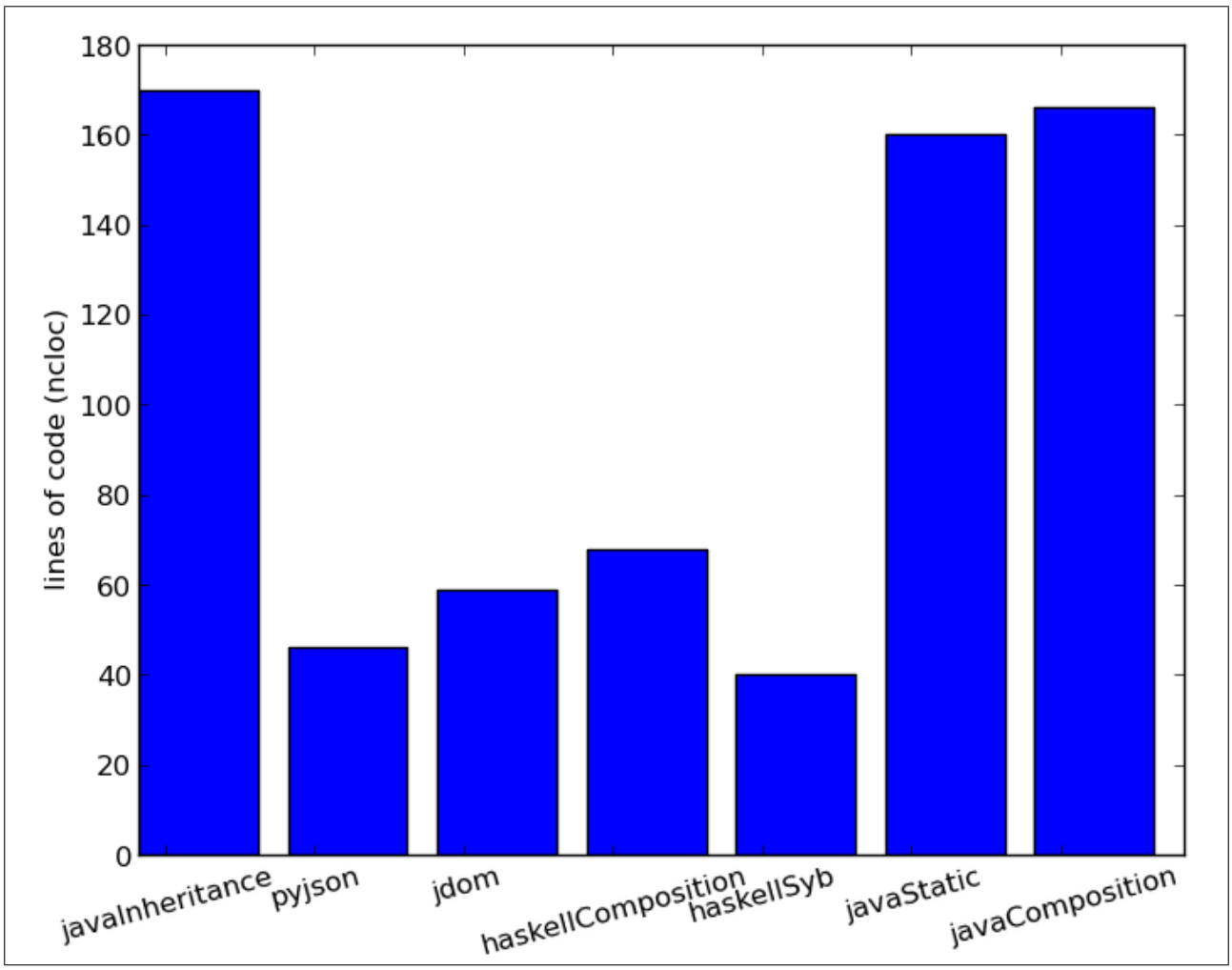


Figure 9.4: An NCLOC-based comparison of implementations of features "Total", "Cut", and "Hierarchical company"

Results

Within the scope of an early research achievement, presented in this section, we aim to address exemplarily the research question as to what language, technology, or style is more suited for implementing a certain programming task.

Figure 9.4 shows the chart for the execution of subsection 9.2.3 for seven selected implementations with a coverage of three languages—*Java*, *Python*, and *Haskell*.

There are four Java-based implementations. Three of them (see the names java...) show very similar metrics, which is a consequence of the fact that they are implemented in different but basic styles of OO programming (class inheritance versus object composition versus families of static methods). The fourth Java-based implementation, *jdom*, is strikingly more concise. This

is the case because an implicit data model and a query API for features such as "Total" are used.

The *Haskell*-based implementation *haskellComposition* uses an explicit data model and it uses no special API. Nevertheless, it is nearly as small as the *Java*-based implementation *jdom*. The *Python*-based implementation *pyjson* is even more concise; it does not leverage an explicit data model. It turns out that *haskellSyb* is the most concise implementation, despite its explicit implementation of a data model. Code inspection and expertise suggests the following arguments. Haskell is generally very concise, compared to Java. Python may be similarly concise and it may benefit from the omission of explicit data models. However, Haskell's SYB style [LJ03b] makes the implementation of query and transformation features such as "Total" and "Cut" highly concise; such a style is not established in Python or Java.

Related Work

The comparison aspect of the presented research is original. However, there is related work, along different dimensions, that could suggest improvements of the methodology for comparison and suggest more interesting experiments.

Feature location. Feature (or concept) location [RW02] is a common maintenance activity [DRGP13] performed by developers. A systematic survey is provided in [DRGP13]. In the absence of external documentation, advanced information retrieval methods, such as latent semantic indexing (LSI), are applied [MSRM04b]. So far, we use only a very basic, text-based location approach operating on source text including program identifiers and comments, while only taking advantage of knowledge of the lexical structure; see some of the rules in [FLL+12b].

Novel context-aware approaches to feature location [PXT+11] suggest using both a requirement model (e.g., a feature model) and a program model as input. However, deeper investigation and adoption of "just enough" requirement model is an open issue. An interactive exploration approach is proposed in [WPXZ13] to support the human-oriented and information-intensive process of feature location. The results show an increase of developer productivity while using

multi-faceted search.

Program comprehension & API analysis. Creating a conceptual model of the source code is used in various program comprehension activities such as multi-language cross-referencing [KWDE98] or business-rules extraction [CCA+13]. As we demonstrated, API usage might encapsulate most of the code of a more traditional feature implementation. We should bring models of APIs (e.g., classifiers for APIs, API domains, and API facets [DRLP13]), into the scope of the rule-based system for metadata computation, thereby imposing more structure on comparison.

Software metrics. We use NCLOC as a starting point for comparing implementations. Empirical evidence exists about correlation of the size of a software system with its fault-proneness [EBGR01] and maintainability [DJ03]. In the case of object-oriented systems, more advanced size metrics, such as NIM (Number of Instance Methods) or TNOS (Total Number Of Statements) [LK94] can be considered. However, the methodology clearly requires metrics that can be used across various languages and paradigms. Furthermore, any selected metrics should also agree with complexity as perceived by programmers [KK12]. More research is needed on metrics suitable for comparison.

A Chrestomathy-based Course

101haskell, as described in the present section, was used in an introductory functional programming course during summer semester 2013 at the University of Koblenz-Landau.[3] (Two-thirds of the students were in the second semester and already had basic Java programming skills. The remaining students were in the first semester.) The present section describes the underlying teaching concept, motivates designated course content on top of *101haskell*, and discusses a limited course evaluation.

[3] `http://101companies.org/wiki/Course:Lambdas_in_Koblenz`

Teaching Concept

We highlight aspects that set the present teaching concept apart from common practice. These aspects relate to the use of the *101haskell* and infrastructure of *101.*

"Favor live programming." Most of the lecture time is dedicated to live programming, where all relevant concepts are systematically illustrated. The list of concepts for each lecture is published on the *101wiki*. The illustrations given during live programming are essentially variations on the illustrations readily available on the wiki. More complex examples, such as non-trivial *101haskell* contributions, are not developed from scratch but are readily demonstrated as available from the *101repo*. Slides are not used. Some amount of wiki content may be projected, however. Also, *101wiki* pages may contain embedded media.

"Embrace multiple external resources." Past teaching experience has suggested that our students are rarely willing to follow given textbook recommendations; instead, unstructured search is popular. In this course, we respond to this attitude by helping students leverage available online resources more systematically. In particular, *Wikipedia*, *HaskellWiki*, and Haskell textbooks are readily linked from the course material, as discussed in chapter 6.

"Complement the running example." The lectures spend considerable time on explaining all concepts with the help of diverse, basic examples that are unrelated to the *101system*, but even these examples are available through the *101repo*. Implementations of the *101system* serve typically as less basic illustrations. The homework assignments are not necessarily tied to the *101system*. Occasionally, an assignment could be concerned with the modification of a given contribution.

"Open source and open linked data." Absolutely all course material is open. Reuse in courses and collaborative advancement is appreciated and straightforward. In particular, reuse

does not cause any copyright issues whatsoever because lecturers may reuse wiki content and repo content simply by linking to it, without copy-and-paste as needed for slide-based reuse.

Course Content

Most of the content is readily available via the wiki pages for contributions, concepts, and others. The only course-specific content is the lineup of all lectures and per-lecture scripts for the itemized and linked content of the lectures.

- Lecture First steps
- Lecture Basic software engineering
- Lecture Searching and sorting
- Lecture Basic data modeling
- Lecture Higher-order functions
- Lecture Type-class polymorphism
- Lecture Functors and friends
- Lecture Monads
- Lecture Parsing and unparsing
- Dry run for final
- Lecture Generic functions
- Final

Figure 9.5: Lectures in the functional programming course.

Figure 9.5 shows the lineup of the lectures for the course. Two lecture slots are repurposed for the final exam and the exam's dry-run. In the next edition of the course, we expect to make space for an extra lecture slot, in which case we plan to cover functional data structures as an additional topic.

Figure 9.6 shows a particular lecture script, as it is rendered on the *101wiki*. Thus, each lecture comes with a headline (a title), a summary, a longer list of concepts, and a shorter list of *101haskell* contributions covered by the lecture. This also clarifies the modus operandi of the lecturer: the listed concepts are illustrated in some order; the listed contributions are eventually explored. The exact order is unspecified but it may be influenced by the dynamics of the lecture. If time turns out to be insufficient, some concepts or contributions may also be

Headline

Lecture "Higher-order functions in Haskell" as part of Course:Lambdas in Koblenz

Summary

Higher-order functions are functions that take functions as arguments or return functions as results. Much of the expressiveness and convenience of functional programming is a consequence of the status of functions to be first-class citizens. In this lecture, we focus on higher-order functions for list processing, e.g., the map function. We also look at important related concepts such as partial application of functions or anonymous functions.

Concepts

- Polymorphism
- Parametric polymorphism
- Partial application
- Higher-order function
- Currying
- Uncurrying
- Map function
- Fold function
- Filter function
- Zip function
- List comprehension
- Anonymous function
- Lambda abstraction

Languages

- Language:Haskell

Contributions

- haskellEngineer: No higher-order functions
- haskellList: Leverage map and sum
- haskellLambda: Leverage anonymous functions
- haskellProfessional: Richer demonstration

Figure 9.6: The script for a lecture on higher-order functions.

delegated to the lab.

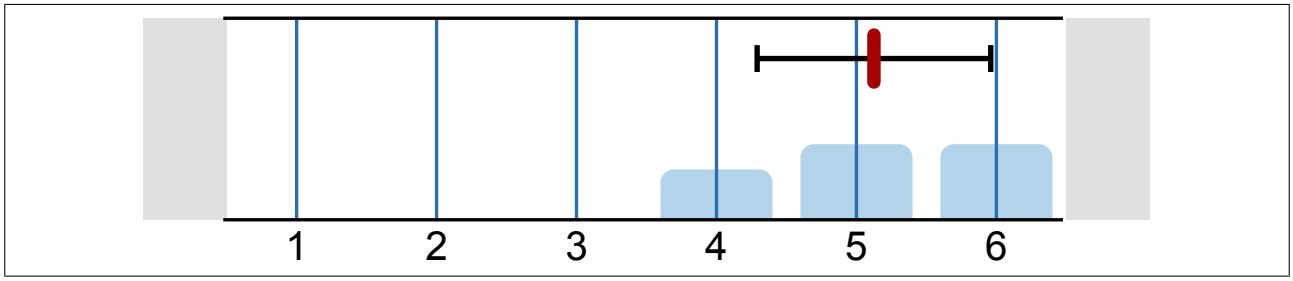


Figure 9.7: Course evaluation: satisfaction of the students with practical illustrations on 1-6 scale (higher is better).

Course Evaluation

Our university runs evaluations for all courses. However, student participation in the polls is voluntary. The questionnaires are relatively complex, which may add to the low turnout. Ten

out of 73 enrolled students submitted their scores for the functional programming course. All results are available online. [4] Our experience with other introductory courses (first or second semester) suggests that these courses tend to be less well received. (A significant percentage of students cancel their studies during this period; there is no "numerus clausus" for computer science.) The present course received mostly favorable scores. The course received an overall score of 2.3 ("good") on a 1-5 (very good to insufficient) scale.

In figure 9.7, we show poll results for a question related to the use of practical examples. We take the results to mean that the balanced use of the *101system* as the running example was appreciated.

The written final contained basic tasks for the first seven lecture topics of figure 9.5 and almost all the students succeeded in the exam. (This is rather surprising for a first/second semester course.) As we have not conducted the course previously, we cannot compare learning results.

Code-sharing Management

In [AJB+14] a new approach to a product line engineering (PLE) is presented. PLT adoption strategy is incremental and minimally invasive. It is called a "Virtual platform" and provides a number of cloning strategies, between ad-hoc clone and own-and, supported by six governance levels. *101companies* practice is classified as ad-hoc clone-and-own, supported by a feature model, where each contribution is described as a set of features. PLE governance levels are illustrated by using *101haskell* chrestomathy described in chapter 5.

In [SL16] the clone-and-own approach by a combination of variability and clone management (which we refer to as similarity management) is presented. *101companies* is used in the case study, where similarity for clowned-and-owned Haskell-based variants of a simple human-resources management system is managed. Clone detection is used to manage the similarity of contributions to help with understanding and evolution. The Linked Data exposed for the *101repo* is traversed in the tree-like structure and the file content is mapped to file, URIs. In this manner groups of perfect clones are determined.

[4] `http://softlang.uni-koblenz.de/101haskell/`

Threats to validity

The main threat *external validity* is how adequate and realistic the context of the applicability of the software chrestomathy as a design research artifact, to support usage scenarios described in this chapter. To provide a sound answer, the replication and cross-validation at various conditions is certainly required. The results of our evaluation should not be seen as directly applicable to other or even similar contexts. Rather then the *possibility* of running such experiments enabled by the *101companies* makes it a relevant artifact – a result of the design research carried out in this thesis. We consider the maturity of the *101companies* software chrestomathy as the main threat to internal validity, as the results of the evaluation methodology heavily relies on the existing *101companies* infrastructure.

Conclusion

We presented three cases for utilizing a software chrestomathy as a design research artifact aligned with the research agenda on software chrestomathies. Our evaluation is not meant to be an empirical one but to provide a solid argument towards the need for validation of usefulness. The importance of the collaborative and learning aspects of software chrestomathy stand out rather clearly in the evaluations presented, supporting the central goal of this work – a software chrestomathy facilitating a knowledge-driven research infrastructure.

Chapter 10

Conclusion and Future Work

In this chapter we summarize the results and outline the directions of the future work. We summarize the implemented requirements, then briefly reflect on the original research questions. We offer an additional focus for future work – an extended research agenda – as this is a success factor for *101companies* software chrestomathy as a design research artifact.

Summary of the Thesis Achievements

In this work we instantiated *101* software chrestomathy as a design research artifact that facilitates an organization of software languages and technologies and provides a documentation model to support knowledge organization and integration. Collection of contributions enables a deeper understanding of the linguistic architecture of software products. In fact, the contributions of software chrestomathy, like any other software system, are non-instantiated technology models. Table 10.1 summarizes the original requirements as worked out in several chapters of this work.

Let us also review the original research questions and briefly reflect on how the result of this work has helped to answer them:

- *Can software chrestomathy be adopted in software-engineering research and education?*

This chapter is an original part of the thesis.

Table 10.1: Requirements coverage per chapter

Requirement	Chapter
R1. Core Properties of software chrestomathy	5
R2. Ontology-driven Classification	5
R3. Linking Documentation and Source Code	5
R4. Vocabulary Engineering Through Knowledge Integration	6
R5. Linked Data Enabled Infrastructure	5,7
R6. A Chrestomathic Ontology	7
R7. Linguistic Architecture of Software Products	8
R8. General-purpose Language for Technology Models	8

This is the core question in the research agenda for software chrestomathy. In this work we investigated several application domains for a software chrestomathy *101companies*. Additionally, the evaluation of end-to-end research and educational scenarios is provided in chapter 9.

- *How can software chrestomathy be instantiated as a Research 2.0 platform?* A semantic wiki for documentation linked to source code of the open-source repositories provides an open and extensible platform for software engineering research. An open contribution model promotes a technological space agnostic collaboration model – a key element of Research 2.0. An ontology processing workflow brings the ecosystem into an ontological space. As we highlight in section 10.2, this enables a further research agenda on the chrestomathic ontologies.

- *How can software chrestomathy be made useful in understanding software languages and technologies?* Technology modeling is based on the linguistic architecture, introduced in this work. Entities and relations as core notions in fact are also used by the documentation models of *101companies*. Such shared ontology together with the Linked-Data infrastructure bring a new dimension for exploring details of the complex nature of software languages and technologies.

- *How can software chrestomathy be made useful in software-engineering education?* A structural way of organizing information also facilitates an organization of software engineering courses by using *101wiki*. An extension to the semantic properties allowed us

to organize the course as an ordered list of lectures, fully documented in the wiki. Introduced concepts are strongly linked with the illustrative examples of various complexity. The underlying infrastructure of *101companies*, such as topic vocabularies and linkage to external sources, makes a chrestomathy a useful tool in software engineering education, providing a new perspective on content exploration and understanding.

Future Work

In the previous chapter we summarized how *101companies* software chrestomathy enabled us to answer the original research questions and motivated this work. However, being a useful design research artifact also means that it should open certain directions for further usage. In this section we highlight a number of those directions, most clearly followed by the content of this thesis. In fact, some of them are already carried out in our research group as of time of this writing.

Technology modeling agenda. The role of technology models in software engineering can be further validated. The directions are the following: documentation and specification (cognitive aspect, as it was mainly concerned in this work) can be extended to the automation scenarios(a la make/ant/maven), or for error detection (as a type system for the composition of elements), or other analyses (complexity metrics, etc.) Once the aim is refined, it can be verified more objectively, using established empirical methods.

The notion of linguistic architecture, as developed in this work, was mainly driven by examples. Further axiomatization for the underlying entity types and relations is needed. In this manner, we can answer questions like these: -What does it mean that a system uses a language or a technology? -What does it mean that a technology facilitates a certain concept (e.g., that a technology for web-app development facilitates the model-view controller pattern)? -What does it mean that two artifacts involved in a mapping (e.g., in Object/relational mapping) correspond to each other? Such axiomatization combined with the illustrations will constitute a core ontology for linguistic architecture.

Social coding agenda. A social coding – a new phenomenon in developers' collaboration, significantly raised by GitHub's popularity is a new kind of software ecosystem. A software chrestomathy can provide a complementary view on a social coding, especially from the perspective of the development of domain ontology. Such domain ontology is useful for researchers studying software ecosystems. The ecosystem comprises certain aspects of collaborative authoring, data representation, mapping, metamodeling, and schema-based validation – the same aspects, as presented in the current work on *101companies* software chrestomathy.

Methodological agenda. We do not claim that the techniques used in this work exhaustive or definitive. On the contrary, we believe that a larger community effort is needed to obtain more exhaustive and more definitive results. A chrestomathy-centric empirical method can be developed. This allows the validation of given techniques on a large software corpus. In the present work we focus on the complexity challenges for software chrestomathies. From the corpus perspective, software chrestomathy can be a baseline for certain domains, such as similarity management, as illustrated in chapter 9.

Educational agenda. Software chrestomathy opens a new perspective on learning, as shown in chapter 9.However, a larger effort is needed to make chrestomathy a truly learning environment aligned with the best practices of E-Learning. An empirical study is needed to determine to what extent the chrestomatic ontology and knowledge organization helps students better comprehend knowledge, as compared to "traditional" methods such as lectures and books.

Conclusion.

To conclude this work, we would like to emphasize the importance of the research methodology and the selection of the problem space. This enabled us to set up a scope for the research and a way to evaluate the results. A structured semantic document model for a software chrestomathy together with a collection of contributions enabled us to discover and motivate the linguistic architecture and technology modeling as its application. Further we applied the model to the

teaching context. At the end, we believe this open and expansible artifact can facilitate further research directions, as highlighted earlier in this chapter.

Appendices

Themes of *101companies* Implementations

MDE: Model-driven engineering theme of implementations	
Description	
Several aspects and flavors of developing of model-driven engineering components are exercised. In particular, Model to Model transformations, Model to Text transformations and Text to Model transformations are exercised. Different styles of Model to Model transformation, like the declarative style used by *ATL* or the imperative style of *EMF* in *Java* are considered. For Text to Model transformations, the shown contributions rely on mapping grammar elements on model elements.	
Members	
- atl	Model to Model transformations with *ATL*
- atlCutPlugin	Model to Model transformations with an *ATL* plugin
- atlPluginUsage	Model to Model transformations with *ATL* plugins
- atlTotalPlugin	Model to Model transformations with an *ATL* plugin
- emfGenerative	Model-Object mapping for *Ecore* and *Java* with *EMF*
- emfReflexive	Reflexive *EMF* Model to Model transformation
- gra2mol	ĐŢext to Model transformation with *Gra2Mol*
- jgralab	Use *TGraphs* with *JGraLab* in *Java* for Model to Model transformations
- xtext	IDE Creation with an XText- and Eclipse-based DSL editor

Figure 1: MDE theme of 101implementations

Haskell data: A theme of *Haskell*-based contributions varying data representation	
Description	
Different feature models and design choices are exercised for the *Haskell*-based data model of companies. Thereby, Haskell's data modeling expressiveness and common styles are explored. Any mentioning of "trivial data model" implies *Flat company* as opposed to *Hierarchical company*. The remaining contributions involve data models that deal with *Hierarchical company*. It should be noted that the contributions may serve additional purposes other than just illustrating data modeling options.	
Members	
- haskellComposition	Data composition in *Haskell* with algebraic data types
- haskellData	Use of algebraic data types in *Haskell*
- haskellRecord	Use of record types in *Haskell*
- haskellScott	Exercise Scott encoding in *Haskell*
- haskellStarter	Basic functional programming in *Haskell*
- haskellTermRep	Data processing in *Haskell* with a universal representation
- haskellVariation	Data variation in *Haskell* with algebraic data types

Figure 2: Haskell data theme of 101implementations

Haskell introduction: basics of *Haskell*	
Description	
An introductory collection of *Haskell*-based contributions. This theme collects the following relatively basic *Haskell*-based contributions.	
Members	
- haskellStarter	Basics of functional programming
- haskellEngineer	Basics of software engineering
- haskellList	List processing with map and friends
- haskellProfessional	Idiomatic code for many features
- haskellLambda	Anonymous functions
- haskellComposition	Recursive algebraic data types
- haskellVariation	Multiple constructors per type
- haskellMonoid	Queries in monoidal style
- haskellLogging	Logging in non-monadic style
- haskellWriter	Logging in monadic style
- haskellParsec	Parsing with the Parsec library
- haskellSyb	Generic programming Ãă la SYB style

Figure 3: Haskell introduction theme of 101implementations

Haskell potpourri: A potpourri of Language:Haskell-based contributions	
Description	
This theme demonstrates Language:Haskell's approach to several programming problems: concurrent programming, database programming, generic programming, GUI programming, logging, parsing, unparsing, XML programming, web programming. Some of the contributions nicely demonstrate some strengths and specifics of Haskell. This is true, arguably, for the contributions that illustrate XML programming and generic programming. Some other contributions are mainly included to provide coverage for important programming domains or problems without necessarily arguing that the Haskell-based approach is particularly interesting or attractive. This is true, for example, for the contribution that demonstrates GUI programming. Relatively mature and established technologies are demonstrated as opposed to research experiments.	
Members	
- haskellParsec	Parsing with the Parsec library
- hughesPJ	Unparsing with Text.PrettyPrint.HughesPJ
- wxHaskell	GUI programming with wxHaskell
- happstack	Web programming with Happstack
- haskellDB	Database programming with HaskellDB
- hxt	XML programming with HXT
- writerMonad	Logging with the writer monad
- mvar	Concurrent programming with MVars
- haskellSyb	Generic programming in SYB style

Figure 4: Haskell potpourri of 101implementations

Haskell genericity: different styles of generic functional programming in *Haskell*	
Description	
There are different classes of generic programming. The present theme is concerned with the class of generic programming that involves data type-polymorphic functions such that the functions can be applied to data of different types as, for example, in the case of the "Scrap your boilerplate" style of generic programming. The present theme is focused on different generic programming styles as they exist for Language:Haskell. Certain features of the 101system are particularly relevant for the present theme. These are the features for cutting and totaling salaries as they illustrate the need for data transformations and queries that may need to fully traverse compound data while only some details of such data (i.e., salaries) are conceptually relevant. Thus, Feature:Total and Feature:Cut make up the baseline set of features to be covered by any member contribution of this theme.	
Members	
- haskellSyb	Generic programming in *Haskell* with SYB
- strafunski	Strafunski approach generic programming in *Haskell*
- haskellTree	Data processing in *Haskell* with functors and foldable types
- tabaluga	Dealing with large bananas in *Haskell*

Figure 5: Haskell genericity of 101implementations

NoSQL: Modern database theme of implementations	
Description	
Classically, relational (SQL-based) databases were used to manage large-volume data. These days, additional options have become commonplace. For instance, there are technologies serving the MapReduce programming model on distributed file systems; there are various NoSQL approaches, e.g., document-based databases or BigTable clones. This theme is under construction.	
Members	
- hadoop	Data-parallel processing with *Hadoop*
- gremlin-neo4j	A graph-based implementation using *Neo4J* and the *Gremlin* graph query DSL
- hbase	A NoSQL implementation based on *HBase*
- mongodb	Employ a document-oriented database
- riak	A NoSQL implementation based on *Riak*

Figure 6: NoSQL theme of 101implementations

Python potpourri: Contributions achieving basic coverage of *Python*	
Description	
Contributions achieving basic coverage of *Python*. This theme is under construction.	
Members	
- py3k	a basic implementation of the spec in *Python* 3
- pyjson	Processing *JSON*-based data in *Python*
- pyjamas	Web programming in *Python* with *Pyjamas*

Figure 7: Python potpourri theme of 101implementations

Scrap your boilerplate: Contributions that exercise "Scrap your boilerplate" style of generic programming	
Description	
The "Scrap your boilerplate" (SYB) style of generic programming was originally conceived in a Haskell context, but similar coding styles were subsequently proposed for other programming languages. In fact, even for Haskell alone, variations on SYB style were proposed. Accordingly, the present theme features contributions that exercise SYB style across different host languages. In some cases, these contributions actually include libraries that support SYB style for the host language at hand. Certain features of the 101system are particularly relevant for the present theme. These are the features for cutting and totaling salaries which actually have their origin in the SYB literature. Thus, features *Total* and *Cut* make up the baseline set of features to be covered by any member contribution of this theme. Implementations of yet other features may benefit from SYB style, too.	
Members	
- haskellSyb	Generic programming in *Haskell* with SYB
- jsSyb	Generic programming in *Javascript* with SYB
- pythonSyb	Generic programming in *Python* with SYB
- javaSyb	Exercise SYB-style generic programming with reflection in *Java*

Figure 8: Scrap your boilerplate theme of 101implementations

Starter: A few very simple implementation of the 101system	
Members	
Contribution	**Language**
argoUML	*UML*
haskellStarter	*Haskell*
html5local	*HTML5*
javaComposition	*Java*
prologStarter	*Prolog*
pyjson	*Python*
xslt	*XSLT*

Figure 9: Starter theme of 101implementations

Web programming: Web programming theme of implementations	
Description	
There is a myriad of web-programming frameworks. This theme features implementations that demonstrate arguably the most established frameworks, while at the same time aiming at coverage of different languages and platforms. For instance, *Silverlight* may currently count as a major web-programming approach for the *.NET* platform, and hence, an implementation was included. Also, Language:PHP is generally popular for web development, and hence, an implementation based on the popular *Pyjamas* framework was included. As far as *Java* is concerned, there are again, in turn, many different approaches and frameworks, and the goal was here to select a small number of different and popular approaches. Finally, in the case of *HTML5*, the two options of a local applications versus a client/server architecture is demonstrated.	
Members	
- gwt	Feature:Browsing web programming with *GWT*
- html5local	Web programming based on the *HTML5* ecosystem with local *Web storage*
- happstack	Web programming in *Haskell* with *Happstack*
- html5ajax	Web programming based on the *HTML5* ecosystem using Ajax style
- jsf	Web programming with *JSF*
- pyjamas	Web programming in *Python* with *Pyjamas*
- rubyonrails	Web programming in *Ruby* with *Ruby on Rails*
- seaside	Web programming in *Smalltalk* with *Seaside*
- silverlight	Web programming in *C#* with *Silverlight*
- strutsAnnotation	Web programming in *Java* with *Struts* configuring with annotations
- zend	Web programming in *PHP* with the *Zend framework*

Figure 10: Web programming theme of 101implementations

Web applications in Java: A theme of *Java*-based web applications	
Description	
The theme collects contributions which exercise different kinds of web-application frameworks for *Java*. Different technologies such as *Struts*, *JSF*, and *GWT* are explored. For some technologies, different options of technology usage are explored, e.g., the use of annotations versus XML-based configuration.	
Members	
- gwt	Web programming in *Java* with *GWT*
- gwtTree	*Browsing* web programming with *GWT*
- jsf	Web programming with *JSF*
- seam	Web application development with *Java* and the *Seam framework*
- strutsAnnotation	Web programming in *Java* with *Struts* configuring with annotations
- strutsXml	Web programming in *Java* with *Struts* configuring with *XML*

Figure 11: Web applications in Java theme of 101implementations

The *SoLaSoTe* ontology of software languages, technologies, and concepts

Entities of *SoLaSoTe*

The kinds of entities (say, individuals) in the ontology give rise to what is called *SoLaSoTe*'s set of *entity types*, which are developed in this section. We begin by providing an overview of these types. Next, we describe these types in more detail and illustrate them by means of representative entities. Finally, we discuss classification, as applied to the entities, subject to additional types derived from the entity types, thereby establishing a taxnonomy-like hierarchy with the entity types at the roots of classification. All entity types are modeled as RDFS classes. The types and their instances are exercised by means of SPARQL queries applied to the triplestore of the ontology.

Entity types–Overview

There are these entity types:

Output of query entityTypes.sparql

type	comment
onto:Concept	Software concepts
onto:Contribution	Contributions of the 101 project
onto:Contributor	Contributors to the 101 project
onto:Course	Cources on programming and software engineering
onto:Document	Documents in a broad sense
onto:Feature	Software features
onto:Language	Software languages
onto:Script	Scripts as units of a course
onto:Tag	Tags to be carried by entities
onto:Technology	Software technologies
onto:Theme	Containers of contributions
onto:Vocabulary	Containers of terms

The list of entity types can be retrieved from *SoLaSoTe*'s triplestore as follows:

Query entityTypes.sparql

```
SELECT ?type ?comment {
  ?type rdfs:subClassOf onto:Entity .
  FILTER (?type ≠onto:Entity) .
  FILTER NOT EXISTS { ?type a onto:Classifier } .
  ?type rdfs:comment ?comment
}
ORDER BY ?type
```

That is, the entity types are organized as subclasses of a base type onto:Entity, but we do not include subclasses of the *SoLaSoTe*-specific type onto:Classifier because these are non-root types used for classification, as explained in detail in §A.2.2.2. (In different terms, we only include direct subclasses of onto:Entity.) The following query determines the number of entities (say, individuals or instances) for each type:

Query entities.sparql

```
SELECT ?type (COUNT(?type) AS ?count)
WHERE {
  ?type rdfs:subClassOf onto:Entity .
  FILTER (?type ≠onto:Entity) .
  FILTER NOT EXISTS { ?type a onto:Classifier } .
  ?entity a ?type
}
GROUP BY ?type
ORDER BY ?count
```

Output of query entities.sparql

type	count
Course	5
Tag	6
Document	7
Vocabulary	8
Theme	10
Contributor	14
Script	22
Feature	53
Language	103
Contribution	223
Technology	330
Concept	661

Prefixes used by *SoLaSoTe*

At the RDF level of representation, there is a designated prefix for each entity type. A few additional prefixes are needed so that *SoLaSoTe* can also relate to other ontologies or RDF resources. Here is a complete list of prefix declarations:

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX onto:<http://101companies.org/ontology#>
PREFIX res:<http://101companies.org/resources#>
PREFIX tech:<http://101companies.org/resources/Technology#>
PREFIX lang:<http://101companies.org/resources/Language#>
PREFIX concept:<http://101companies.org/resources/Concept#>
PREFIX voc:<http://101companies.org/resources/Vocabulary#>
```

```
PREFIX doc:<http://101companies.org/resources/Document#>
PREFIX feature:<http://101companies.org/resources/Feature#>
PREFIX contrib:<http://101companies.org/resources/Contribution#>
PREFIX theme:<http://101companies.org/resources/Theme#>
PREFIX contributor:<http://101companies.org/resources/Contributor#>
PREFIX course:<http://101companies.org/resources/Course#>
PREFIX script:<http://101companies.org/resources/Script#>
PREFIX tag:<http://101companies.org/resources/Tag#>
PREFIX sesame:<http://www.openrdf.org/schema/sesame#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
```

The prefixes can be explained as follows:

rdf The RDF data model.[1]

rdfs The schema for RDF.[2]

owl The Web Ontology Language.[3]

xsd XML Schema for data types.[4]

onto Classes of the SoLaSoTe ontology.

res Entities of SoLaSoTe.

lang Language entities of SoLaSoTe.

tech Technology entities of SoLaSoTe.

concept Concept entities of SoLaSoTe.

voc Vocabularies as collections of SoLaSoTe concepts.

doc Document entities of SoLaSoTe.

[1] http://101companies.org/wiki/Language:RDF
[2] http://101companies.org/wiki/Language:RDFS
[3] http://101companies.org/wiki/Language:OWL
[4] http://www.w3.org/2001/XMLSchema

feature Features of 101 as entities of *SoLaSoTe*.

contrib Contributions of 101 as entities of *SoLaSoTe*.

theme Themes as collections of contributions.

contributor Contributors of 101 as entities of *SoLaSoTe*.

script Scripts (parts of courses) as entities of *SoLaSoTe*.

course Courses as collections of scripts.

tag Tags applied to entities of *SoLaSoTe*.

sesame The namespace of the Sesame framework.[5]

foaf The ontology of the 'Friend of a Friend' project.[6]

Entity types–Details

Type 'Language' The following query retrieves all software languages:

Query languages.sparql

```
SELECT ?language ?headline (COUNT(?subject) AS ?count)
WHERE {
  ?language a onto:Language .
  ?language onto:hasHeadline ?headline .
  ?subject ?predicate ?language .
}
GROUP BY ?language ?headline
ORDER BY DESC(?count)
```

[5] http://101companies.org/wiki/Technology:Sesame
[6] http://www.foaf-project.org/

As there are many languages, we order them by 'popularity'. Below, we show only the most popular languages. By popularity we mean the numbers of any sort of subjects referring to the languages—through any sort of predicate. In this manner, we see presumably more well-known, less obscure entities.

Output of query languages.sparql (first few rows)

language	headline
Java	An OO programming language
Haskell	A purely-functional programming language
XML	The extensible markup language
JavaScript	A multi-paradigm programming language for the web et al.
JSON	The JavaScript Object Notation for data exchange
SQL	Data definition and manipulation for relational databases
Python	A multi-paradigm programming language
...	


```
{
    "@id": "Language",
    "@type": [
        "Entity",
        "Instrument"
    ],
    "comment": "Software languages",
    "wikialias": [
        "Software_language"
    ],
    "properties": [
        {
            "property": "langDesignedBy",
            "super": "designedBy",
```

```
            "range": "foaf:Person",
            "minCardinality": "0",
            "comment": "Designers of languages"
        }
    ]
}
```

Type 'Technology' We apply the same kind of query as before:

Query technologies.sparql

```
SELECT ?technology ?headline (COUNT(?subject) AS ?count)
WHERE {
  ?technology a onto:Technology .
  ?technology onto:hasHeadline ?headline .
  ?subject ?predicate ?technology .
}
GROUP BY ?technology ?headline
ORDER BY DESC(?count)
```

Output of query technologies.sparql (first few rows)

technology	headline
Gradle	A build tool inspired by Ant and Maven
JUnit	A framework for unit testing for Java
Eclipse	An IDE for Java with a plug-in system
GHC	A Haskell compiler
.NET	A library and runtime for programming languages on Windows
ANTLR	A parser generator with various language processing capabilities
Cabal	A build automation tool for Language:Haskell
...	


```
{
    "@id": "Technology",
    "@type": [
        "Entity",
        "Instrument",
```

```
        "System"
    ],
    "comment": "Software technologies",
    "wikialias": [
        "Software_technology"
    ],
    "properties": [
        {
            "property": "techUsesLang",
            "super": "uses",
            "range": "Language",
            "minCardinality": "0",
            "comment": "Use of languages by technologies"
        },
        {
            "property": "techUsesTech",
            "super": "uses",
            "range": "Technology",
            "minCardinality": "0",
            "comment": "Use of technologies by technologies"
        },
        {
            "property": "techUsesConcept",
            "super": "uses",
            "range": "Concept",
            "minCardinality": "0",
            "comment": "Use of concepts by technologies"
        },
        {
            "property": "techDependsOnTech",
            "super": "dependsOn",
```

```
        "range": "Technology",
        "minCardinality": "0",
        "comment": "Technologies depending on technologies"
    },
    {
        "property": "techDependsOnLang",
        "super": "dependsOn",
        "range": "Language",
        "minCardinality": "0",
        "comment": "Technologies depending on languages"
    },
    {
        "property": "techDependsOnConcept",
        "super": "dependsOn",
        "range": "Concept",
        "minCardinality": "0",
        "comment": "Technologies depending on concepts"
    },
    {
        "property": "techImplements",
        "super": "implements",
        "range": "Document",
        "minCardinality": "0",
        "comment": "Implementation of a standard or alike"
    },
    {
        "property": "techDesignedBy",
        "super": "designedBy",
        "range": "foaf:Person",
        "minCardinality": "0",
        "comment": "Designers of technologies"
```

```
        }
    ]
}
```

Type 'Concept' We apply the same kind of query as before:

Query concepts.sparql

```
SELECT ?concept ?headline (COUNT(?subject) AS ?count)
WHERE {
  ?concept a onto:Concept .
  ?concept onto:hasHeadline ?headline .
  ?subject ?predicate ?concept .
}
GROUP BY ?concept ?headline
ORDER BY DESC(?count)
```

Output of query concepts.sparql (first few rows)

concept	headline
Web programming	The domain of web application development
Algebraic data type	A type for the construction of terms
OO programming	The object-oriented programming paradigm
Web browser	A system for retrieving and presenting Web resources
Functional programming	The functional programming paradigm
API	An interface for reusable functionality
Software system	A system of intercommunicating software components
...	

```
{
    "@id": "Concept",
    "@type": [
        "Entity",
        "Instrument"
    ],
    "comment": "Software concepts",
    "wikialias": [
        "Software_concept"
    ],
    "properties": [
       {
            "property": "conceptMemberOf",
            "super": "memberOf",
            "range": "Vocabulary",
            "minCardinality": "0",
            "comment": "Concepts collected in vocabularies"
       }
    ]
}
```

Type 'Vocabulary' Concepts can be collected in *vocabularies*. The collected concepts are supposedly used in a certain context of programming or development or by a certain community. There are not yet many vocabularies; we can list them all:

Query vocabularies.sparql

```
SELECT ?vocabulary ?headline
WHERE {
  ?vocabulary a onto:Vocabulary .
```

```
  ?vocabulary onto:hasHeadline ?headline
}
ORDER BY ?vocabulary
```

Output of query vocabularies.sparql

vocabulary	headline
Data structure	Data structure concepts
Functional programming	Functional programming concepts
Haskell	Haskell concepts
Information system	Information system concepts
OO programming	OO programming concepts
Programming	Programming concepts
Programming languages	Programming language concepts
Programming theory	Programming theory concepts

Concepts are included into vocabularies by means of the 'memberOf' predicate; see §A.2.3.3.

```
{
    "@id": "Vocabulary",
    "@type": [
        "Entity",
        "Container"
    ],
    "comment": "Containers of terms"
}
```

Type 'Contribution' *SoLaSoTe* relies on the chrestomathy *101* for evidence in the form of small systems that exercise languages, technologies, and concepts. These systems are called *contributions* (since someone has to 'contribute' them to the chrestomathy). We sort contributions by popularity again:

Query contributions.sparql

```
SELECT ?contribution ?headline (COUNT(?subject) AS ?count)
WHERE {
  ?contribution a onto:Contribution .
  ?contribution onto:hasHeadline ?headline .
  ?subject ?predicate ?contribution
}
GROUP BY ?contribution ?headline
ORDER BY DESC(?count)
```

Output of query contributions.sparql (first few rows)

contribution	headline
haskellEngineer	Basic software engineering for Haskell
haskellComposition	Data composition in Haskell with algebraic data types
mySqlMany	A MySQL database with SQL scripts
javaComposition	Object composition in Java
antlrAcceptor	An ANTLR-based acceptor for textual syntax
javaInheritance	Class inheritance in Java
antlrLexer	Lexer-based text processing with ANTLR
antlrParser	An ANTLR-based parser with semantic actions
jdom	XML processing with JavaâĂŹs JDOM API
antlrObjects	ANTLR-based object-text mapping for Java
antlrTrees	Parsing and tree walking with ANTLR
jaxbComposition	Object-XML mapping with JAXB of the Java platform
...	


```
{
    "@id": "Contribution",
    "@type": [
        "Entity",
```

```
        "System"
    ],
    "comment": "Contributions of the 101project",
    "properties": [
        {
            "property": "contribUsesLang",
            "super": "uses",
            "range": "Language",
            "minCardinality": "1",
            "comment": "Use of languages by contributions"
        },
        {
            "property": "contribUsesTech",
            "super": "uses",
            "range": "Technology",
            "minCardinality": "0",
            "comment": "Use of technologies by contributions"
        },
        {
            "property": "contribUsesConcept",
            "super": "uses",
            "range": "Concept",
            "minCardinality": "0",
            "comment": "Use of concepts by contributions"
        },
        {
            "property": "contribDesignedBy",
            "super": "designedBy",
            "range": "Contributor",
            "minCardinality": "1",
            "comment": "Designers of contributions"
```

```
    },
    {
        "property": "contribDevelopedBy",
        "super": "developedBy",
        "range": "Contributor",
        "minCardinality": "1",
        "comment": "Developers of contributions"
    },
    {
        "property": "contribReviewedBy",
        "super": "reviewedBy",
        "range": "Contributor",
        "minCardinality": "0",
        "comment": "Reviewers of contributions"
    },
    {
        "property": "contribImplements",
        "super": "implements",
        "range": "Feature",
        "minCardinality": "1",
        "comment": "Features implemented by contributions"
    },
    {
        "property": "contribMemberOf",
        "super": "memberOf",
        "range": "Theme",
        "minCardinality": "0",
        "comment": "Contributions collected in themes"
    },
    {
        "property": "contribVaries",
```

```
            "super": "varies",
            "range": "Contribution",
            "minCardinality": "0",
            "comment": "Similarity of contributions"
        },
        {
            "property": "contribBasedOn",
            "super": "basedOn",
            "range": "Contribution",
            "minCardinality": "0",
            "comment": "Reuse of contributions"
        },
        {
            "property": "contribMoreComplexThan",
            "super": "moreComplexThan",
            "range": "Contribution",
            "minCardinality": "0",
            "comment": "Complexity of contributions"
        }
    ]
}
```


Type 'Contributor' Contributions are designed, developed, and reviewed by contributors. *101* required GitHub identities for its contributors. This also helps with identity management and authentication. The most active (most referenced) contributors are listed below:

Query contributors.sparql

```
SELECT ?contributor (COUNT(?subject) AS ?count)
WHERE {
```

```
  ?contributor a onto:Contributor .
  ?subject ?predicate ?contributor
}
GROUP BY ?contributor
ORDER BY DESC(?count)
```

Output of query contributors.sparql (first few rows)

contributor
rlaemmel
tschmorleiz
avaranovich
mpaul138
hartenfels
martinleinberger
todeslord
...

The shown names can be directly used to look up these persons on GitHub.[7] Contributors are associated with contributions by means of the 'developedBy' predicate and friends; see §A.2.3.5.

```
{
    "@id": "Contributor",
    "@type": [
        "foaf:Person",
        "Entity"
    ],
    "comment": "Contributors to the 101project",
    "properties": [
        {
```

[7] For instance, *avaranovich* maps to `https://github.com/avaranovich`.

```
            "property": "profile",
            "range": "rdfs:Literal",
            "minCardinality": "1",
            "comment": "Web page with info about contributor"
        }
    ]
}
```

Type 'Feature' Contributions implement features of *101*'s imaginary human resources management system (the *101system*. We sort the features by popularity again:

Query features.sparql

```
SELECT ?feature ?headline (COUNT(?subject) AS ?count)
WHERE {
  ?feature a onto:Feature .
  ?feature onto:hasHeadline ?headline .
  ?subject ?predicate ?feature
}
GROUP BY ?feature ?headline
ORDER BY DESC(?count)
```

All features (as of writing) are shown here to convey that *101*'s set of features is meant to be manageable. The features at the bottom of the list are potentially obscure, experimental, or outdated.

Output of query features.sparql

feature	headline
Total	Sum up the salaries of all employees
Cut	Cut the salaries of all employees in half
Hierarchical company	Support nested departments in companies
Parsing	Parse an external format for companies
Closed serialization	Serialize companies in a closed manner
Unparsing	Unparse companies to an external format
Browsing	Browse companies in a UI
Depth	Compute the nesting depth of departments
Distribution	Distribute company data and operations
Editing	Edit companies in a UI
Web UI	Support a web-based UI
Mapping	Map companies across technological spaces
Persistence	Persist companies
Flat company	Support companies as plain collections of employees
Open serialization	Serialize companies in an open manner
Ranking	Check salaries to follow ranks in company hierarchy
Company	Model companies
Median	Compute the median of the salaries of all employees
Mentoring	Associate employees in terms of mentoring
Restructuring	UI support for restructuring company data
Task parallelism	Apply task parallelism to total or cut salaries
Logging	Log and analyze salary changes
Data parallelism	Apply data parallelism to total or cut salaries
Serialization	Serialize companies
Access control	Control access for company data and operations
Singleton	Support a single company, not many
History	Maintain and analyze historical company data
Touch control	Support touch control in the UI
Dimensionality	Analyze salary distribution along different dimensions
Visualization	Visualize companies
Offline mode	Continue functioning even with an offline server
Geolocation	Identify the geographic location of the user
Parallelism	Total or cut salaries in parallel
Flattened company	Represent hierarchical companies in a flat manner
Code generator	Develop and demonstrate a code generator
Localization	Support different languages in the UI
Reliability	Make a server reliable
COI	Associate employees in terms of conflicts of interest

Contributions are associated with features by means of the ‘implements’ predicate; see §A.2.3.3.

```
{
    "@id": "Feature",
    "@type": [
```

```
        "Entity",
        "Description"
    ],
    "comment": "Software features",
    "wikialias": [
        "Software_feature"
    ],
    "properties": [
        {
            "property": "featureDependsOn",
            "super": "dependsOn",
            "range": "Feature",
            "minCardinality": "0",
            "comment": "Cross−feature constraint"
        },
        {
            "property": "featureMoreComplexThan",
            "super": "moreComplexThan",
            "range": "Feature",
            "minCardinality": "0",
            "comment": "Comparison of complexity"
        }
    ]
}
```


Type 'Theme' Contributions can be collected in *themes*. The assumption is here that the collected contributions (systems) are of interest to a certain stakeholder, perhaps to persons with a specific learning objective. Let's have a look at the themes:

Query themes.sparql

```
SELECT ?theme ?headline
WHERE {
  ?theme a onto:Theme .
  ?theme onto:hasHeadline ?headline
}
ORDER BY ?theme
```

Output of query themes.sparql (first few rows)

theme	headline
ANTLR	Varying uses of ANTLR
GUI programming	Varying GUI programming approaches
Haskell data	Varying Haskell-based approaches to data modeling
Haskell genericity	Varying generic programming approaches in Haskell
Haskell introduction	Introductory Haskell-based contributions
Haskell potpourri	A potpourri of Haskell-based contributions
MDE	Demonstrations of model-driven engineering
Scrap your boilerplate	Demonstrations of SYB style of generic programming
Starter	Very simple contributions across the board
Web programming	Demonstrations of web programming
...	

Contributions are included into themes by means of the ‘memberOf’ predicate; see §A.2.3.3.

```
{
    "@id": "Theme",
    "@type": [
        "Entity",
        "Container"
    ],
    "comment": "Containers of contributions"
}
```

Type 'Script' Scripts are units of knowledge representation that are most likely outlines for lectures or lab sessions. As of writing, they are used exclusively indeed for lecture scripting in terms of the exercised concepts, technologies, languages, and examples (contributions). Here are some illustrations:

Query scripts.sparql

```
SELECT ?script ?headline
WHERE {
  ?script a onto:Script .
  ?script onto:hasHeadline ?headline
}
ORDER BY ?script
```

Output of query scripts.sparql (first few rows)

script	headline
Aspect-oriented programming	Aspect-oriented programming in AspectJ
Data modeling in Haskell	Basic data modeling techniques in Haskell
Data parallelism	Data parallelism with Hadoop
Database programming	Database access with JDBC and Hibernate
First steps in Haskell	First steps of programming in Haskell
Functional OO programming	Functional OO programming in Java
Functional data structures	Functional data structures in Haskell
...	

```
{
    "@id": "Script",
    "@type": "Entity",
    "comment": "Scripts as units of a course",
```

```
    "properties": [
        {
            "property": "scriptMemberOf",
            "super": "memberOf",
            "range": "Course",
            "minCardinality": "0",
            "comment": "Courses as containers of scripts"
        },
        {
            "property": "scriptDependsOn",
            "super": "dependsOn",
            "range": "Script",
            "minCardinality": "0",
            "comment": "Constraints on script order"
        }
    ]
}
```


Type 'Course' Scripts can be collected in *courses*. The collected scripts (i.e., lectures or alike) are indeed meant to define the modules of an actual course. At this point, there are only two courses that are modeled in this way:

Query courses.sparql

```
SELECT ?course ?headline
WHERE {
  ?course a onto:Course .
  ?course onto:hasHeadline ?headline
}
```

Output of query courses.sparql

course	headline
Lambdas in Koblenz	Introduction to functional programming at the University of Koblenz-Landau
Programming in Koblenz	An advanced BSc course on programming techniques and technologies in Kob
Data technologies for Debeka	Professional training on modern data technologies
HaskellBarchart	
Web programming for Debeka	Professional training on modern web programming

Scripts are included into courses by means of the 'memberOf' predicate; see §A.2.3.3.

```
{
    "@id": "Course",
    "@type": [
        "Entity",
        "Container"
    ],
    "comment": "Courses on programming and software engineering",
    "properties": [
        {
            "property": "courseDesignedBy",
            "super": "designedBy",
            "range": "Contributor",
            "minCardinality": "1",
            "comment": "Designer of a contribution"
        }
    ]
}
```

Type 'Document' Entities (such as languages, technologies, and concepts) are regularly associated with 'external' resources; see the 'sameAs' predicate and friend in §A.2.3.3 for details. In certain situations, it is reasonable though to reify an external resource (a document in a broad sense) as a SoLaSoTe entity. In this manner, it is possible to make the external resource participate in all of SoLaSoTe's properties. We show the 'headlines' of some of the documents reified on SoLaSoTe:

Query documents.sparql

```
SELECT ?headline
WHERE {
  ?doc a onto:Document .
  ?doc onto:hasHeadline ?headline
}
ORDER BY ?headline
```

Output of query documents.sparql (first few rows)

headline
A GPCE 2006 paper on software extension and integration
A report studying object encodings in Haskell
A textbook on information systems
...

We only show the headlines (i.e., short explanations) of the resources, not the names assigned to them in SoLaSoTe, as there is no intuitive, comprehensive scheme for giving names to the documents. There is also no comprehensive style for referring to them: some documents may

be referrable to through DOIs; others may have a manifestation on Wikipedia; yet others may be best resolvable on Amazon; etc. We refer to §A.2.3.3 for predicates that associate *SoLaSoTe* entities with external resources.

```
{
    "@id": "Document",
    "@type": [
        "foaf:Document",
        "Entity",
        "Description"
    ],
    "comment": "Documents in a broad sense"
}
```

Type 'Tag' A simple tagging scheme is used for *SoLaSoTe* so that one can associate 'tags' with entities. As of writing, only very few tags are in use:

Query tags.sparql

```
SELECT ?tag ?headline
WHERE {
  ?tag a onto:Tag .
  ?tag onto:hasHeadline ?headline
}
ORDER BY ?tag
```

For instance, the 'Stub' tag is used to keep track of contributions whose documentation is essentially missing or blatantly incomplete. This idea is inspired by Wikipedia's stub notion. We refer to §A.2.3.7 for the 'carries' predicate which is used to associate entities (such as contributions) with tags.

```
{
    "@id": "Tag",
    "@type": "Entity",
    "comment": "Tags to be carried by entities"
}
```

Classification of entities

SoLaSoTe's entities are roughly classified on the grounds of the entity types, as described previously. For instance, we may obviously ask whether lang:Java is a (software) language:

Query javaOfTypeLanguage.sparql

```
ASK {
  lang:Java a onto:Language
}
```

Evaluation of this query literally returns 'true':

Output of query javaOfTypeLanguage.sparql

```
true
```

A more fine-grained classification is also supported on the grounds of a class hierarchy with the entity types as root types. These extra classes are called 'classifiers'. For instance, lang:Java is of the following (classifier) types:

Output of query typesOfJava.sparql

```
onto:OO programming language
onto:Programming language
```

The (classifier) types of an entity can be retrieved like this:

Query typesOfJava.sparql

```
SELECT ?type
WHERE {
  lang:Java rdf:type ?type .
  ?type a onto:Classifier
}
ORDER BY ?type
```

As the query clarifiers, classifier types can be explicitly selected by testing for the extra type onto:Classifier of those classes, thereby not confusing them with general types of the ontology—such as onto:Language or Entity. The class(ifier) hierarchy can also be queried in itself—without starting from entities. For instance, this is how we ask for the supertypes of the classifier

---

Query supertypesOfOoProgrammingLanguage.sparql

```
SELECT ?type
WHERE {
  onto:OO_programming_language rdfs:subClassOf ?type .
  ?type a onto:Classifier .
  FILTER (?type ≠onto:OO_programming_language )
```

```
}
ORDER BY ?type
```

Output of query supertypesOfOoProgrammingLanguage.sparql

```
onto:Programming language
```

This is how we query for all entities with a certain classifier; in this case, we are interested in all OO programming languages:

Query ooProgrammingLanguage.sparql

```
SELECT ?language
WHERE { ?language rdf:type onto:OO_programming_language }
ORDER BY ?language
```

These are some OO programming languages:

Output of query ooProgrammingLanguage.sparql (first few rows)

```
lang:CSharp
lang:CoffeeScript
lang:FSharp
lang:Java
```

```
lang:JavaScript
lang:Lua
lang:Perl
```

...

SoLaSoTe's approach towards classification makes one specific assumption. For each classifier, there is a corresponding concept (onto:Concept) of the same name modulo different prefixes ('onto' for classifieres, 'concept' for concepts). Whether or not a concept has an associated classifier depends on the fact whether the concept is actually used for classification.

In §A.2.1, we had queried for 'popular' concepts regardless of whether they are (associated with) classifiers. Here is a similar query, which specifically focuses on 'popular' classifiers:

Query classifiers.sparql

```
SELECT ?concept ?headline (COUNT(?subject) AS ?count)
WHERE {
  ?concept a onto:Concept .
  ?concept onto:hasHeadline ?headline .
  ?classifier a onto:Classifier .
  ?classifier onto:classifies ?concept .
  ?subject ?predicate ?concept
}
GROUP BY ?concept ?headline
ORDER BY DESC(?count)
```

In the query, we use a predicate 'classifies' to look up the association between classifier and concept. Here are the first few classifiers returned by the query:

Output of query classifiers.sparql (first few rows)

concept	headline
Algebraic data type	A type for the construction of terms
OO programming	The object-oriented programming paradigm
Web browser	A system for retrieving and presenting Web resources
Functional programming	The functional programming paradigm
API	An interface for reusable functionality
Software system	A system of intercommunicating software components
Client	A component accessing a service provided by a server
...	

Clearly, this list overlaps with the ranking of popular concepts overall; see again §A.2.1. For comparison, here is query for 'popular' non-classifier concepts:

Query nonClassifiers.sparql

```
SELECT ?concept ?headline (COUNT(?subject) AS ?count)
WHERE {
  ?concept a onto:Concept .
  ?concept onto:hasHeadline ?headline .
  FILTER NOT EXISTS {
    ?classifier a onto:Classifier .
    ?classifier onto:classifies ?concept .
  } .
  ?subject ?predicate ?concept
}
GROUP BY ?concept ?headline
ORDER BY DESC(?count)
```

Output of query nonClassifiers.sparql (first few rows)

concept	headline
Web programming	The domain of web application development
Type class	An abstraction mechanism for polymorphism
MVC	Division of an architecture into model, view, and controller
Language	A software language
Fold function	A higher-order function for processing a data structure
GUI	A graphical user interface
Objectware	A technological space focused on OO programming
...	

At this point, we have discussed *SoLaSoTe*'s entity types (rooted in onto:Entity and *SoLaSoTe*'s classification types forming a class hierarchy rooted by the entity types with entities as instances. *SoLaSoTe* uses yet a few extra 'base types' to capture commonalities of entity types in certain contexts. Here is a list of these types and the corresponding subClassOf relationships:

Output of query baseTypes.sparql

type	comment
onto:Container	Base type for vocabularies, scripts, and themes
onto:Description	Base type for features and documents
onto:Instrument	Base type for languages, technologies, and concepts
onto:System	Base type for technologies and contributions

Output of query baseTypeSubClassing.sparql

subtype	supertype
onto:Function	onto:Abstraction mechanism
onto:Instance method	onto:Abstraction mechanism
onto:Method	onto:Abstraction mechanism
onto:Static method	onto:Abstraction mechanism
onto:Newtype	onto:Algebraic data type
onto:Divide and conquer algorithm	onto:Algorithm
onto:Search algorithm	onto:Algorithm
onto:Search problem	onto:Algorithmic problem
onto:Sorting problem	onto:Algorithmic problem
onto:Subtype polymorphism	onto:Bounded polymorphism
onto:Type-class polymorphism	onto:Bounded polymorphism
onto:ADO .NET	onto:Concept
onto:AST	onto:Concept
onto:Abstract data type	onto:Concept
onto:Abstract syntax	onto:Concept
onto:Abstract syntax tree	onto:Concept
onto:Abstraction	onto:Concept
onto:Abstraction mechanism	onto:Concept
onto:Accumulator	onto:Concept
onto:Action	onto:Concept
onto:Active record	onto:Concept
onto:Ad-hoc polymorphism	onto:Concept
onto:Ajax	onto:Concept
onto:Algebraic data type	onto:Concept
onto:Algorithm	onto:Concept
onto:Algorithm design	onto:Concept
onto:Algorithmic problem	onto:Concept
onto:Android Manifest	onto:Concept
onto:Android Menu	onto:Concept
onto:Android Resource	onto:Concept
onto:Android project	onto:Concept
onto:Anonymous class	onto:Concept
onto:Anonymous function	onto:Concept
onto:AppWidget	onto:Concept
onto:Application domain	onto:Concept
onto:Architectural pattern	onto:Concept
onto:Argument	onto:Concept
onto:Aspect-oriented programming	onto:Concept
onto:Assertion	onto:Concept
onto:Association	onto:Concept
onto:Association list	onto:Concept
onto:Backup	onto:Concept
onto:Base case	onto:Concept
onto:Bidirectional transformation	onto:Concept
onto:Bottom-up parsing	onto:Concept
onto:Bounded polymorphism	onto:Concept
onto:Business process	onto:Concept
onto:CRUD	onto:Concept
onto:Calculation	onto:Concept
onto:Catamorphism	onto:Concept
onto:Class	onto:Concept
onto:Closure	onto:Concept
onto:Code	onto:Concept
onto:Combinator	onto:Concept
onto:Command	onto:Concept
onto:Computing	onto:Concept
onto:Concrete data type	onto:Concept
onto:Concrete syntax	onto:Concept
onto:Concrete syntax tree	onto:Concept
onto:Condition	onto:Concept
onto:Constructor component	onto:Concept
onto:Context-free grammar	onto:Concept
onto:Cookie	onto:Concept
onto:Core	onto:Concept
onto:Corecursion	onto:Concept

For instance, both a software technology and a contribution (i.e., an implementation of *101*'s system) can be regarded as a 'software system'. The idea is that these base types are convenient in setting up *SoLaSoTe*'s properties, as discussed in more detail in For instance, both a technology and a contribution may be said to be 'developed by' a person. For completeness' sake, the list of base types and the corresponding subClassOf relationships can be retrieved from *SoLaSoTe*'s triplestore as follows:

Query baseTypes.sparql

```
SELECT ?type ?comment {
  ?type rdfs:subClassOf onto:Resource .
  ?type rdfs:comment ?comment .
  FILTER (?type ≠onto:Classifier) .
  FILTER NOT EXISTS {
    ?type rdfs:subClassOf onto:Entity
  }
}
ORDER BY ?type
```

Query baseTypeSubClassing.sparql

```
SELECT ?supertype ?subtype {
  ?supertype rdfs:subClassOf onto:Resource .
  ?subtype rdfs:subClassOf ?supertype .
  FILTER (?supertype ≠rdfs:Resource) .
  FILTER (?supertype ≠onto:Resource) .
```

```
  FILTER (?supertype ≠onto:Entity) .
  FILTER (?subtype ≠?supertype) .
  FILTER NOT EXISTS { ?supertype a onto:Classifier } .
  FILTER NOT EXISTS { ?subtype a onto:Classifier }
}
ORDER BY ?supertype ?subtype
```

Last but not least, SoLaSoTe also leverages external types, i.e., types of other ontologies. In fact, besides RDF and RDFS, SoLaSoTe currently only uses these FOAF types:

Output of query externalTypes.sparql

subtype	supertype
onto:101companies	onto:rdfs:Class
onto:ADO .NET	onto:rdfs:Class
onto:AST	onto:rdfs:Class
onto:Abstract data type	onto:rdfs:Class
onto:Abstract syntax	onto:rdfs:Class
onto:Abstract syntax tree	onto:rdfs:Class
onto:Abstraction	onto:rdfs:Class
onto:Abstraction mechanism	onto:rdfs:Class
onto:Accumulator	onto:rdfs:Class
onto:Action	onto:rdfs:Class
onto:Active record	onto:rdfs:Class
onto:Ad-hoc polymorphism	onto:rdfs:Class
onto:Ajax	onto:rdfs:Class
onto:Algebraic data type	onto:rdfs:Class
onto:Algorithm	onto:rdfs:Class
onto:Algorithm design	onto:rdfs:Class
onto:Algorithmic problem	onto:rdfs:Class
onto:Android Manifest	onto:rdfs:Class
onto:Android Menu	onto:rdfs:Class
onto:Android Resource	onto:rdfs:Class
onto:Android project	onto:rdfs:Class
onto:Anonymous class	onto:rdfs:Class
onto:Anonymous function	onto:rdfs:Class
onto:AppWidget	onto:rdfs:Class
onto:Application domain	onto:rdfs:Class
onto:Architectural pattern	onto:rdfs:Class
onto:Argument	onto:rdfs:Class
onto:Aspect-oriented programming	onto:rdfs:Class
onto:Assertion	onto:rdfs:Class
onto:Association	onto:rdfs:Class
onto:Association list	onto:rdfs:Class
onto:Backup	onto:rdfs:Class
onto:Base case	onto:rdfs:Class
onto:Bidirectional transformation	onto:rdfs:Class
onto:Bottom-up parsing	onto:rdfs:Class
onto:Bounded polymorphism	onto:rdfs:Class
onto:Business process	onto:rdfs:Class
onto:CRUD	onto:rdfs:Class
onto:Calculation	onto:rdfs:Class
onto:Catamorphism	onto:rdfs:Class
onto:Class	onto:rdfs:Class
onto:Classifier	onto:rdfs:Class
onto:Closure	onto:rdfs:Class
onto:Code	onto:rdfs:Class
onto:Combinator	onto:rdfs:Class
onto:Command	onto:rdfs:Class
onto:Computing	onto:rdfs:Class
onto:Concept	onto:rdfs:Class
onto:Concrete data type	onto:rdfs:Class
onto:Concrete syntax	onto:rdfs:Class
onto:Concrete syntax tree	onto:rdfs:Class
onto:Condition	onto:rdfs:Class
onto:Constructor component	onto:rdfs:Class
onto:Container	onto:rdfs:Class
onto:Context-free grammar	onto:rdfs:Class
onto:Contribution	onto:rdfs:Class
onto:Contributor	onto:foaf:Person
onto:Contributor	onto:rdfs:Class
onto:Cookie	onto:rdfs:Class
onto:Core	onto:rdfs:Class
onto:Corecursion	onto:rdfs:Class
onto:Course	onto:rdfs:Class
onto:Crosscutting concern	onto:rdfs:Class
onto:Currying	onto:rdfs:Class
onto:Data	onto:rdfs:Class

The following query results in the list shown above:

Query externalTypes.sparql

```
SELECT DISTINCT ?subtype ?supertype {
  ?subtype rdfs:subClassOf ?supertype .
  ?subtype rdfs:subClassOf onto:Resource .
  FILTER (?supertype ≠rdfs:Resource) .
  FILTER NOT EXISTS {
    ?supertype rdfs:subClassOf onto:Resource
  }
}
ORDER BY ?subtype ?supertype
```

Relationships of *SoLaSoTe*

SoLaSoTe leverages several relationship types to characterize and relate individuals, as developed in the present section. We begin by providing an overview of the relationships. Next, we describe refinements of these relationships that are often available to better constrain domains and ranges to well-established scenarios. Eventually, we describe all relationships in more detail and illustrate them by means of representative triples.

Relationship types–Overview

There are these overall relationships; ‘properties’ in the terminology of RDFS:

Output of query relationships.sparql

property	comment
basedOn	Reuse of systems
carries	Tagging of entities
dependsOn	Dependence relationships
designedBy	Designers of entities
developedBy	Developers of entities
illustrates	Descriptions serving for illustration
implements	Systems implementing descriptions
linksTo	Non-specific link to external resource
memberOf	Membership relationships
mentions	Nonspecific references to entities
moreComplexThan	Complexity comparison of entities
partOf	Whole-part relationships
profile	Web page with info about contributor
reviewedBy	Reviewer of entities
sameAs	Equivalence relative to external resource
similarTo	Similarity relative to external resource
supports	Instruments supporting instruments
uses	Use of instruments by systems
varies	Similarity of systems

The relationships can be retrieved from *SoLaSoTe*'s triplestore as follows:

Query relationships.sparql

```
SELECT ?property ?comment
WHERE {
  ?property rdfs:domain ?domain .
  ?domain rdfs:subClassOf onto:Resource .
  FILTER NOT EXISTS {
    ?property rdfs:subPropertyOf ?super .
    FILTER (?super ≠ ?property)
  } .
  ?property rdfs:comment ?comment
}
ORDER BY ?property ?comment
```

Before we discuss these predicates in more detail, we should also identify their types in the sense of the assumed domain and range for each predicate:

Output of query domainsAndRanges.sparql

property	**domain**	**range**
basedOn	onto:System	onto:System
carries	onto:Entity	onto:Tag
conceptMemberOf	onto:Concept	onto:Vocabulary
contribBasedOn	onto:Contribution	onto:Contribution
contribDesignedBy	onto:Contribution	onto:Contributor
contribDevelopedBy	onto:Contribution	onto:Contributor
contribImplements	onto:Contribution	onto:Feature
contribMemberOf	onto:Contribution	onto:Theme
contribMoreComplexThan	onto:Contribution	onto:Contribution
contribReviewedBy	onto:Contribution	onto:Contributor
contribUsesConcept	onto:Contribution	onto:Concept
contribUsesLang	onto:Contribution	onto:Language
contribUsesTech	onto:Contribution	onto:Technology
contribVaries	onto:Contribution	onto:Contribution
courseDesignedBy	onto:Course	onto:Contributor
dependsOn	onto:Entity	onto:Entity
designedBy	onto:Entity	onto:foaf:Person
developedBy	onto:Entity	onto:foaf:Person
featureDependsOn	onto:Feature	onto:Feature
featureMoreComplexThan	onto:Feature	onto:Feature
illustrates	onto:Description	onto:Instrument
implements	onto:System	onto:Description
langDesignedBy	onto:Language	onto:foaf:Person
linksTo	onto:Entity	onto:rdfs:Literal
memberOf	onto:Entity	onto:Container
mentions	onto:Entity	onto:Entity
moreComplexThan	onto:Entity	onto:Entity
partOf	onto:Entity	onto:Entity
profile	onto:Contributor	onto:rdfs:Literal
reviewedBy	onto:Entity	onto:foaf:Person
sameAs	onto:Entity	onto:rdfs:Literal
scriptDependsOn	onto:Script	onto:Script
scriptMemberOf	onto:Script	onto:Course
similarTo	onto:Entity	onto:rdfs:Literal

Again, for completeness' sake, the table has been produced by the following query:

Query domainsAndRanges.sparql

```
SELECT ?property ?domain ?range
WHERE {
  ?property rdfs:domain ?domain .
  ?property rdfs:range ?range .
  ?domain rdfs:subClassOf onto:Resource .
}
ORDER BY ?property ?domain ?range
```

Now let's discuss these properties one by one, while also providing typical examples. We pick a particular order that fits convenience of explanation.

Specialized relationships–Overview

Relationships–Details

Links to external resources This is a family of predicates all concerned with linking individuals of SoLaSoTe with external web-based resources, e.g., pages on Wikipedia or resources according to DBpedia. Their meaning and purpose is closely related to owl:sameAs and variations that are discussed by the Semantic Web community [HHT11]. The use of onto:sameAs expresses that the SoLaSoTe individual and the external resource's URL refer to the same thing. For instanceL

Query predicateSameAs1.sparql

```
ASK {
  "http://101companies.org/resources/Document#JLS"
    onto:sameAs "http://docs.oracle.com/javase/specs/"
}
```

That is, there is an *SoLaSoTe* individual for the 'Java Language Specification' (JLS) of entity type onto:Document; we use the onto:sameAs predicate to associate it with Oracle's authorative source for the JLS.

The use of onto:similarTo expresses that the *SoLaSoTe* individual and the external's URL refer to closely related but notably not the same things. An unspecific link to an external resource is enabled by onto:linksTo. We query the links for an illustrative individual, concept:Monad:

Query predicateSameAs2.sparql

```
SELECT ?predicate ?url {
  concept:Monad ?predicate ?url .
  FILTER (
        ?predicate = onto:sameAs
    ∪?predicate = onto:similarTo
    ∪?predicate = onto:linksTo
  )
}
```

Output of query predicateSameAs2.sparql

predicate	url
onto:sameAs	http://www.haskell.org/haskellwiki/Monad
onto:similarTo	http://en.wikipedia.org/wiki/Monad(functionalprogramming)
onto:linksTo	http://en.wikipedia.org/wiki/Monad(categorytheory)
onto:linksTo	http://en.wikibooks.org/wiki/Haskell/Understandingmonads

That is, two pages, one on the Haskell wiki, another on Wikipedia, are linked to in 'sameAs' properties. The idea is here that these two pages describe the notion of monad exactly in the functional programming-centric (perhaps even Haskell-biased) way as intended for SoLaSoTe. Another page on Wikipedia is linked to in a 'similarTo' property because it is concerned with the related notion of monad in category theory. Finally, a page on Wikibooks is linked to in a 'linksTo' property to express that this page is not considered a definitional resource of the notion at hand, but it does provide (pedagogically) valuable information.

Use of instruments by systems Systems (i.e., technologies and *101*'s contributions) can make use of instruments (i.e., software languages, technologies, and concepts). The property of a system to use an instrument expresses that said instrument is used in the design or implementation or execution of said system. This may be more or less observable from the outside; such a property expresses knowledge about 'internals'.

More specifically, use of a language should be understood as 'some artifact of the system being written in said language'. For instance:

Query predicateUses1.sparql

```
ASK {
  contrib:haskellStarter onto:uses lang:Haskell
}
```

Here, onto:haskellStarter is a simple Haskell-based contribution to *101*. Use of a technology should be understood as 'the system being developed or executed with the help of said technology'. Use of a concept should be understood as 'the concept being exercised or taken dependence on in the design or implementation or execution of the system'. For instance:

Query predicateUses2.sparql

```
ASK {
  tech:JAXB onto:uses concept:Java_annotation
}
```

Here, `tech:JAXB`[8] is the Java platform's technology for XML-data binding, which indeed uses 'Java annotations' for controling the mapping.

Instruments supporting instruments A technology can support another technology in that it provides some sort of interface in generalized sense (e.g., an I/O behavior or a plug-in model) so that the supporting technology can be used with the supported technology. For instance:

Query predicateSupports1.sparql

```
ASK {
  tech:CMake onto:supports tech:Make
}
```

---

[8] ... onto:sameAs `http://www.oracle.com/technetwork/articles/javase/index-140168.html`

Here, tech:CMake[9] is a cross-platform, open-source build system which supports tech:Make in that it CMake can generate native makefiles.

The predicate onto:supports generalizes the situation of technologies supporting technologies so that instruments (i.e., languages, technologies, and concepts) support other instruments. Here are the additional situations:

- A technology supporting a language: said interface can be leveraged by using the language.
- A technology supporting a concept: said interface (the use thereof) conforms to the concept.
- A language supporting a concept: the language's characteristics support the concept.
- A concept supporting an instrument: the concept is expected here to denote a class of technologies and languages. Thus, this situation effectively reduces to one mentioned before.

A few illustrative support relationships are queried here:

Query predicateSupports2.sparql

```
SELECT ?subject ?object {
  ?subject onto:supports ?object
}
```

Output of query predicateSupports2.sparql (first few rows)

[9] ... onto:sameAs http://www.cmake.org/

subject	object
tech:Phusion Passenger	lang:Python
tech:HughesPJ	concept:Unparsing
tech:CMake	tech:Make
tech:CGI	concept:Standard
tech:Moops	concept:OO programming
concept:OO programming language	concept:OO programming
concept:Multi-paradigm programming language	concept:OO programming
tech:CMake	tech:Visual Studio
concept:Functional programming language	concept:Functional programming
concept:Multi-paradigm programming language	concept:Functional programming
...	

Descriptions serving for illustration As much as a system may use some instrument, a feature description or any sort of document may be said to illustrate some instrument the point being that a description may not be able to claim 'use' of the instrument, but it may very well stipulate or explain or motivate its use. For instance:

Query predicateIllustrates1.sparql

```
ASK {
  doc:Handbook_of_data_structures_and_applications
    onto:illustrates
      concept:Functional_data_structure
}
```

The listed handbook is claimed to illustrate the concept of functional data structures. The 'illustrates' predicate is specifically helpful in communicating the purpose of software features of *101*'s imaginary software system. Here is a query that looks up concepts illustrated by the features:

Query predicateIllustrates2.sparql

```
SELECT ?feature ?concept ?headline {
  ?feature onto:illustrates ?concept .
  ?feature a onto:Feature .
  ?concept a onto:Concept .
  ?concept onto:hasHeadline ?headline
}
```

Output of query predicateIllustrates2.sparql (first few rows)

feature	concept	headline
Parsing	Parsing	Analysis of text and construction of parse trees
Distribution	Client-server architecture	Division of an architecture into client and server
Hierarchical company	Recursive data structure	A recursively defined data structure
Logging	Separation of concerns	A design principle to modularize concerns
Parsing	Syntax	Rules defining a software language
Flat company	Data modeling	The vocabulary in the context of data modeling
Flat company	Data modeling	The process of creating a data model
...		

Whole-part relationships Whole-part relationships are used in many areas of modeling; they make sense for SoLaSoTe, too. That is, some kinds of SoLaSoTe individuals may be composites of other kinds of SoLaSoTe individuals. For instance:

Query predicatePartOf1.sparql

```
ASK {
```

```
  tech:javac onto:partOf tech:JDK
}
```

That is, the Java compiler, tech:javac,[10] is part of the Java Development Kit, tech:JDK.[11]. Operationally, by installing JDK on a machine, one also gets the executable for the Java compiler. Here is another example exercising another entity type for whole-part relationships:

Query predicatePartOf2.sparql

```
ASK {
  lang:XPath onto:partOf lang:XSLT
}
```

That is, the XPath query language for XML, lang:XPath,[12] is part of the XSLT transformation language for XML Development Kit, lang:XSLT.[13]. The 'part of' relationship must not be confused here with a 'subset off' relationship. That is, by saying XPath is part of XSLT, we refer to the fact that XPath expressions can be used in certain operand position in an XSLT program.

Systems implementing descriptions Systems (i.e., technologies and *101*'s contributions) can implement descriptions (i.e., features or documents). The idea is that the descriptions serve essentially as requirements. For instance:

---

[10]... onto:sameAs `http://en.wikipedia.org/wiki/Javac`
[11]... onto:sameAs `http://en.wikipedia.org/wiki/Java_Development_Kit`
[12]... onto:sameAs `http://www.w3.org/TR/xpath/`
[13]... onto:sameAs `http://www.w3.org/TR/xslt`

Query predicateImplements1.sparql

```
ASK {
  contrib:haskellStarter onto:implements feature:Total
}
```

Here, onto:haskellStarter is again the simple Haskell-based contribution to *101*, which was exercised already earlier on. The triple states that the contribution implements feature:Total (a feature for totaling all salaries in a company of *101*'s system). Here is another example:

Query predicateImplements2.sparql

```
ASK {
  tech:javac onto:implements doc:JLS
}
```

Here, tech:javac refers to the 'standard' Java compiler, as part of JDK, and doc:JLS refers again to the Java Language Specification, as noted earlier. Clearly, the 'standard' is supposed to implement the language 'standard'.

Collections of entities SoLaSoTe organizes individuals in containers. There are the following use cases:

- Vocabularies as containers with terms (typically concepts) as members.
- Courses as containers with scripts (units such as lectures) as members.
- Themes as containers with (*101*'s) contributions as members.

These kinds of containment are illustrated in turn.

Query memberOfVocabulary.sparql

```
SELECT ?concept ?headline {
  ?concept a onto:Concept .
  ?concept onto:memberOf voc:Functional_programming .
  ?concept onto:hasHeadline ?headline
}
ORDER BY ?concept
```

Output of query memberOfVocabulary.sparql (first few rows)

concept	headline
Action	A monad-based computation
Algebraic data type	A type for the construction of terms
Anamorphism	A corecursion scheme for data types dualizing catamorphisms
Anonymous function	A function without a name based on lambda abstraction
Applicative functor	A kind of functor that models some monad-like computations
Arrow	A functional programming idiom for composing computations
Catamorphism	A recursion scheme for data types
...	

Query memberOfCourse.sparql

```
SELECT ?script {
  ?script onto:memberOf course:Lambdas_in_Koblenz .
}
ORDER BY ?script
```

Output of query memberOfCourse.sparql (first few rows)

script
Data modeling in Haskell
First steps in Haskell
Functional data structures
Functors and friends
Generic functions
Higher-order functions in Haskell
Monads
...

SoLaSoTe's individuals may engage in different kinds of collections. (As evident from the queries to follow, such engagement is expressed through the 'memberOf' predicate, which is also discussed again in §A.2.3.3.) We mentioned vocabularies, themes, and courses as forms of collections in §A.2.1. Collection complements classification (§A.2.2.2).

Collection of concepts as vocabularies We refer back to §A.2.2.1 for the entity type of vocabularies. For instance, the 'Haskell vocabulary' collects concepts that are essentially specific to the Haskell style of functional programming or the Haskell community.

Query haskellVocabulary.sparql

```
SELECT ?concept ?headline
WHERE {
  ?concept onto:memberOf voc:Haskell .
  ?concept onto:hasHeadline ?headline
}
ORDER BY ?concept
```

Output of query haskellVocabulary.sparql (first few rows)

concept	headline
Either type	A polymorphic type for disjoint (indexed) sums
Haskell package	A distribution unit for Haskell
Haskell script	A file with Haskell code
IO system	The monadic approach to IO in Haskell
MVar	A thread synchronization variable in Haskell
Maybe type	A polymorphic type for handling optional values and error
Prelude module	The standard library of Haskell
...	

Collection of contributions as themes We refer back to §A.2.2.1 for the entity type of themes. For instance, the 'Starter' theme collects contributions that demonstrate some simple features across the board, i.e., for different software languages without relying on 'advanced' software technologies or software concepts.

Query starterTheme.sparql

```
SELECT ?contribution ?headline
WHERE {
  ?contribution onto:memberOf theme:Starter .
  ?contribution onto:hasHeadline ?headline
}
ORDER BY ?contribution
```

Output of query starterTheme.sparql

contribution	headline
argoUML	Structural modeling in UML with Technology:ArgoUML
haskellStarter	Basic functional programming in Language:Haskell
html5local	Web programming based on the Language:HTML5 ecosystem with local Technology:Web s
javaComposition	Object composition in Java
javaInheritance	Class inheritance in Java
prologStarter	a simple Language:Prolog-based implementation
pyjson	Processing Language:JSON-based data in Language:Python
xslt	XML processing with Language:XSLT

Dependence relationship

SoLaSoTe's individuals may depend on each other in different ways. Scripts, i.e., course units such as lectures or labs, may depend on each other in the sense that one unit builds upon content of another unit. Consider the following dependencies for a course on functional programming:

Query dependsOnScript.sparql

```
SELECT ?earlier ?later {
  ?earlier a onto:Script .
  ?later a onto:Script .
  ?later onto:dependsOn ?earlier .
  ?earlier onto:memberOf course:Lambdas_in_Koblenz .
  ?later onto:memberOf course:Lambdas_in_Koblenz
}
```

Output of query dependsOnScript.sparql

earlier	later
Unparsing and parsing in Haskell	Monads
Functors and friends	Unparsing and parsing in Haskell
Functors and friends	Monads
Functors and friends	Generic functions
Type-class polymorphism	Functors and friends
Higher-order functions in Haskell	Type-class polymorphism
Higher-order functions in Haskell	Functional data structures
Searching and sorting in Haskell	Data modeling in Haskell
First steps in Haskell	Software engineering for Haskell
Software engineering for Haskell	Searching and sorting in Haskell
Data modeling in Haskell	Higher-order functions in Haskell

For instance, the somewhat advanced topic of 'functors' (and friends) depends on prior coverage of 'type-class polymorphism'. Clearly, such dependency relationships can be used to arrange the units in an actual sequential order and it helps self-learners in processing the content in a reasonable order. Here is indeed a query which orders the scripts in such a way; we also list the number of scripts that are prerequistes for each script:

Query prerequisites.sparql

```
SELECT DISTINCT
    ?script
    (COUNT(?prerequisite) AS ?count)
WHERE {
    ?script onto:memberOf course:Lambdas_in_Koblenz .
    OPTIONAL { ?script onto:dependsOn+ ?prerequisite }
}
GROUP BY ?script
ORDER BY ?count
```

Output of query prerequisites.sparql

script	count
First steps in Haskell	0
Software engineering for Haskell	1
Searching and sorting in Haskell	2
Data modeling in Haskell	3
Higher-order functions in Haskell	4
Type-class polymorphism	5
Functional data structures	5
Functors and friends	6
Unparsing and parsing in Haskell	7
Generic functions	7
Monads	8

Another kind of dependency concerns features of *101*'s system. That is, one feature may 'imply' (say, depend on) another feature. For instance, a feature to compute the 'depth' of department nesting implies the feature for 'hierachical companies', as flat companies would not give rise to a meaningful notion of depth. We query all feature dependencies as follows:

Query dependsOnFeature.sparql

```
SELECT ?feature ?implied {
  ?feature a onto:Feature .
  ?implied a onto:Feature .
  ?feature onto:dependsOn ?implied
}
```

Output of query dependsOnFeature.sparql

feature	implied
Editing	Browsing
Restructuring	Browsing
Web UI	Browsing
Restructuring	Browsing
History	Total
Reliability	Distribution
Offline mode	Distribution
Restructuring	Editing
Restructuring	Editing
History	Median
Dimensionality	History
Depth	Hierarchical company

Yet another kind of dependency concerns instruments to depend on other instruments in the sense that one instrument cannot be 'reasonably' used without the other. For illustration, consider the following dependencies for Ruby on Rails:

Query dependsOnInstrument.sparql

```
SELECT ?dependency ?headline {
  tech:Ruby_on_Rails onto:dependsOn ?dependency .
  ?dependency a onto:Instrument .
  ?dependency onto:hasHeadline ?headline
}
```

Output of query dependsOnInstrument.sparql

dependency	headline
MVC	Division of an architecture into model, view, and controller
Ruby	A multi-paradigm programming language
Rake	A build automation tool for Ruby
WEBrick	A web server for Ruby on Rails

For instance, there is no 'reasonable' way of using Rails other than complying with the architectural pattern of MVC. Also, web development with Rails assumes that all functionality etc. is coded in Rails; likewise for the other dependencies.

Designers, developers, reviewers of entities

These properties are concerned with persons and specifically their involvement in the design, development, and reviewing of different kinds of SoLaSoTe's individuals. This is a list of relevant scenarios:

- Persons may design languages, technologies, (*101*'s) contributions, and courses.
- Persons may develop technologies and contributions.
- Persons may review contributions.

SoLaSoTe leverages the entity type onto:Contributor for persons concerned with (*101*'s) contributions. Here is query for the contributions developed by one of the present document's authors:

Query developedBy.sparql

```
SELECT ?contribution ?headline {
  ?contribution onto:developedBy contributor:avaranovich .
  ?contribution a onto:Contribution .
  ?contribution onto:hasHeadline ?headline
}
```

Output of query developedBy.sparql (first few rows)

contribution	headline
xsdDataSet	X/O mapping with .NETś xsd.exe and strongly typed DataSets
xsdClasses	Object/XML mapping with C# and .NETś xsd.exe
wp7	Basics of programming for Windows Phone 7
csharp	Basics of programming in C# and .NET
wcf	A web service based on .NETś WCF
silverlight	Web programming in C# with Silverlight
wpf	GUI programming with .NETś WP
...	

Nonspecific references to entities

The documentation of *SoLaSoTe* on the *101wiki* may use 'semantically weak' references to *SoLaSoTe*'s individuals. We call them 'semantically weak' in that these references would not use any of the specific predicate, but they are essentially plain hyperlinks. These references are represented through 'mentions' properties in *SoLaSoTe*.

For instance, the following query lists all individuals mentioned by a simple Haskell-based contribution to *101*, contrib:haskellStarter:

Query predicateMentions.sparql

```
SELECT ?object ?headline {
  contrib:haskellStarter onto:mentions ?object .
  ?object onto:hasHeadline ?headline
}
ORDER BY ?object ?headline
```

Output of query predicateMentions.sparql

object	headline
concept:Closed serialization	Technology- and platform-dependent serialization
concept:Data composition	Composition of compound data from parts
concept:Float	The primitive data type of floating-point numbers
concept:Function application	Apply a function to an argument
concept:Functional programming	The functional programming paradigm
concept:Pattern matching	Matching values against patterns to bind variables
concept:Pure function	A function with referential transparency
concept:Recursion	The use of self-reference in defining abstractions
concept:String	The primitive data type of strings
concept:Tuple	An indexed collection of component values
concept:Type synonym	Abstraction over type expressions
contrib:haskellEngineer	Basic software engineering for Haskell
feature:Total	Sum up the salaries of all employees
lang:Haskell	A purely-functional programming language

A good number of concepts is mentioned because they are presumably demonstrated ('used') by the contribution. The feature 'Total' is mentioned because the documentation discusses some details of this particular feature; other features are implemented, but not discussed explicitly. The language Haskell is mentioned for obvious reasons.

Ideally, all mentioned individuals should also be linked in a semantically strong way, i.e., by using one of the predicates other than 'mention'. This can be regarded as a quality criterion for the documention on *101wiki*.

Tagging of entities

A simple tagging scheme is used for SoLaSoTe so that one can associate 'tags' with individuals. As of writing, the only noteworthy example of a tag in use is 'Stub', which is used to keep track of contributions whose documentation is essentially missing or blatantly incomplete. This idea is inspired by Wikipedia's stub notion.

Bibliography

[ABK+07a] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. *Dbpedia: A nucleus for a web of open data*. Springer, 2007.

[ABK+07b] SÃűren Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer Berlin Heidelberg, 2007.

[ACC00] Giuliano Antoniol, Gerardo Casazza, and Aniello Cimitile. Traceability recovery by modeling programmer behavior. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 240–247. IEEE, 2000.

[ACDLM99] Giuliano Antoniol, Gerardo Canfora, Andrea De Lucia, and Ettore Merlo. Recovering code to documentation links in OO systems. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 136–144. IEEE, 1999.

[ADR06] Sören Auer, Sebastian Dietzold, and Thomas Riechert. OntoWiki - A Tool for Social, Semantic Collaboration. In *International Semantic Web Conference*, pages 736–749, 2006.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, 1997.

[AGA13] Nasir Ali, Y Gueneuc, and Giuliano Antoniol. Trustrace: Mining software repositories to improve the accuracy of requirement traceability links. *Software Engineering, IEEE Transactions on*, 39(5):725–741, 2013.

[Ahm08] Emdad Ahmed. Use of Ontologies in Software Engineering. In *17th International Conference on Software Engineering and Data Engineering (SEDE-2008), Proceedings*, pages 145–150. ISCA, 2008.

[AJB+14] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. Flexible product line engineering with a virtual platform. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 532–535. ACM, 2014.

[AK03] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, Sept 2003.

[And11] Android.com. People and Roles in Android Open Source Project (AOSP), 2011. `http://source.android.com/source/roles.html`. Last visited April 19, 2011.

[AS08] Colin Atkinson and Dietmar Stoll. Orthographic modeling environment, 2008.

[AT14] Colin Atkinson and Christian Tunjic. Towards orthographic viewpoints for enterprise architecture modeling. In *Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOCW), 2014 IEEE 18th International*, pages 347–355. IEEE, 2014.

[ATN04] Pavlo Antonenko, Serkan Toy, and Dale Niederhauser. Modular object-oriented dynamic learning environment: What open source has to offer. *Association for Educational communications and Technology*, 2004.

[ATSM07] Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. Design recovery and maintenance of build systems. In *ICSM*, pages 114–123, 2007.

[ATV08] Bogdan Alexe, Wang-Chiew Tan, and Yannis Velegrakis. STBenchmark: towards a benchmark for mapping systems. *Proc. VLDB Endow.*, 1:230–244, August 2008.

[AZ12] Charu C. Aggarwal and ChengXiang Zhai, editors. *Mining Text Data*. Springer, 2012.

[Bac13] Alberto Bacchelli. Mining challenge 2013: Stack overflow. In *The 10th Working Conference on Mining Software Repositories*, page to appear, 2013.

[Bak04] Lynne Rudder Baker. The ontology of artifacts. *Philosophical explorations*, 7(2):99–111, 2004.

[Bat05] Don S. Batory. Feature Models, Grammars, and Propositional Formulas. In *Software Product Lines, 9th International Conference, SPLC 2005, Proceedings*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[BCH07] Chris Bizer, Richard Cyganiak, and Tom Heath. How to publish Linked Data on the web, 2007. Online tutorial `http://wifo5-03.informatik.uni-mannheim.de/bizer/pub/LinkedDataTutorial/`.

[BdAF13] Monalessa Perini Barcellos and Ricardo de Almeida Falbo. A software measurement task ontology. In *SAC*, pages 311–318. ACM, 2013.

[BDJ+03] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First experiments with the atl model transformation language: Transforming xslt into xquery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, volume 37, 2003.

[Ber12] Olivier Berger. Linked data descriptions of debian source packages using adms.sw. In *Proc. of Semantic Web Enabled Software Engineering*, 2012.

[Béz04] J. Bézivin. In Search of a Basic Principle for Model Driven Engineering. *Novatica Journal, Special Issue*, 5(2):21–24, 2004.

[BGGN08] Andrea Brühlmann, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. Enriching Reverse Engineering with Annotations. In *Proceedings of MoDELS 2008 (Model Driven Engineering Languages and Systems)*, volume 5301 of *LNCS*, pages 660–674. Springer, 2008.

[BHBL09] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[BJ06] Jean Bézivin and Frédéric Jouault. Using ATL for Checking Models. *ENTCS*, 152:69–81, 2006.

[BJB08] M. Barbero, F. Jouault, and J. Bézivin. Model Driven Management of Complex Systems: Implementing the Macroscope's Vision. In *Proceedings of ECBS '08*, pages 277–286. IEEE, 2008.

[BJRV05] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick 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.

[BJV04a] J. Bézivin, F. Jouault, and P. Valduriez. On the need for Megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop*, 2004.

[BJV04b] Jean Bezivin, Frederic Jouault, and Patrick Valduriez. On the Need for Megamodels. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development at the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[BLK09] Steven Bird, Edward Loper, and Ewan Klein. *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

[BMB11] Vanessa Araujo Borges, José Carlos Maldonado, and Ellen Francine Barbosa. Towards the establishment of supporting mechanisms for modeling and generating educational content. In *SAC*, pages 1202–1207, 2011.

[BNRM08] Ellen Francine Barbosa, Elisa Yumi Nakagawa, Ana C. Riekstin, and José Carlos Maldonado. Ontology-based development of testing related tools. In *SEKE*, pages 697–702, 2008.

[Boh13] Shannon Bohle. What is e-science and how should it be managed? *Nature. com, Spektrum der Wissenschaft (Scientific American), http://www. scilogs. com/scientific_and_medicallib raries/what-is-e-science-and-how-should-it-be-managed*, 2013.

[Bos04] Jan Bosch. Software architecture: The next step. In *Software Architecture*, volume 3047 of *Lecture Notes in Computer Science*, pages 194–199. Springer Berlin Heidelberg, 2004.

[Bou09] Maged N. Kamel Boulos. Semantic Wikis: A Comprehensible Introduction with Examples from the Health Sciences. *Journal of Emerging Technologies in Web Intelligence*, 1(1):94–96, 2009.

[CCA+13] Valerio Cosentino, Jordi Cabot, Patrick Albert, Philippe Bauquel, and Jacques Perronnet. Extracting business rules from cobol: A model-based framework. In *Proc. WCRE 2013*, pages 409–416. IEEE, 2013.

[CJB99] B. Chandrasekaran, John R. Josephson, and V. Richard Benjamins. What Are Ontologies, and Why Do We Need Them? *IEEE Intelligent Systems*, 14:20–26, 1999.

[Cro07] Nigel Cross. From a design science to a design discipline: Understanding designerly ways of knowing and thinking. In Ralf Michel, editor, *Design Research Now*, Board of International Research in Design. BirkhÃďuser Basel, 2007.

[CS13] D. Correa and A. Sureka. Integrating issue tracking systems with community-based question and answering websites. In *Software Engineering Conference (ASWEC), 2013 22nd Australian*, pages 88–96, June 2013.

[CSH07] Thomas M Connolly, Mark Stansfield, and Thomas Hainey. An application of games-based learning within software engineering. *British Journal of Educational Technology*, 38(3):416–428, 2007.

[dCAG14] Victorio A de Carvalho, João Paulo A Almeida, and Giancarlo Guizzardi. Using reference domain ontologies to define the real-world semantics of domain-specific languages. In *Advanced Information Systems Engineering*, pages 488–502, 2014.

[DDT99] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why FAMIX and not UML. In *Proceedings of UMLâĂŹ99*, volume 1723, 1999.

[DEH+05] Ernst-Erich Doberkat, Gregor Engels, Jan Hendrik Hausmann, Marc Lohmann, Jörg Pleumann, and Jens Schröder. Software engineering and elearning: The musoft project. *eleed Journal*, 2005.

[Dek13] Makx Dekkers. Asset description metadata schema (adms). `https://dvcs.w3.org/hg/gld/raw-file/default/adms/index.html`, jun 2013. [Online; accessed 08.07.2013].

[dG11] Mathieu d'Aquin and Aldo Gangemi. Is there beauty in ontologies? *Applied Ontology*, 6(3):165–175, 2011.

[DGD06] Dragan Djuric, Dragan Gasevic, and Vladan Devedzic. The Tao of Modeling Spaces. *Journal of Object Technology*, 5(8):125–147, 2006.

[DJ03] Melis Dagpinar and Jens H. Jahnke. Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison. In *Proc. of WCRE 2003*, pages 155–164. IEEE, 2003.

[DKM13] Zinovy Diskin, Sahar Kokaly, and Tom Maibaum. Mapping-aware megamodeling: Design patterns and laws. In *Proc. of SLE 2013*, volume 8225 of *LNCS*, pages 322–343. Springer, 2013.

[DMC12] Zinovy Diskin, Tom Maibaum, and Krzysztof Czarnecki. Intermodeling, Queries, and Kleisli Categories. In *Fundamental Approaches to Software Engineering* -

15th International Conference, FASE 2012. Proceedings, volume 7212 of *LNCS*, pages 163–177. Springer, 2012.

[DP09] Stéphane Ducasse and Damien Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Trans. Software Eng.*, 35(4):573–591, 2009.

[DQPvS03] Remco M. Dijkman, Dick A. C. Quartel, Luís Ferreira Pires, and Marten van Sinderen. An Approach to Relate Viewpoints and Modeling Languages. In *7th Int'l Enterprise Distributed Object Computing Conference (EDOC 2003), Proceedings*, pages 14–27. IEEE, 2003.

[DRGP13] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[DRLP13] Coen De Roover, Ralf Lämmel, and Ekaterina Pek. Multi-dimensional exploration of API usage. In *Proc. of ICPC 2013*, pages 152–161. IEEE, 2013.

[DRR+05] Björn Decker, Eric Ras, Jörg Rech, Bertin Klein, and Christian Hoecht. Self-organized reuse of software engineering knowledge supported by semantic wikis. In *Proceedings of the Workshop on Semantic Web Enabled Software Engineering (SWESE)*, page 76, 2005.

[DXC11] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In *Models in Software Engineering - Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers*, volume 6627 of *LNCS*. Springer, 2011.

[DYG+12] Stefan Dietze, Hong Qing Yu, Daniela Giordano, Eleni Kaldoudi, Nikolas Dovrolis, and Davide Taibi. Linked education: interlinking educational resources and the web of data. In *SAC*, 2012.

[EBGR01] Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Software Eng.*, 27(7):630–650, 2001.

[Egy03] Alexander Egyed. A scenario-driven approach to trace dependency analysis. *Software Engineering, IEEE Transactions on*, 29(2):116–132, 2003.

[EKKM08] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and Continuous Checking of Structural Program Dependencies. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 391–400. ACM, 2008.

[EKS03] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *Software Engineering, IEEE Transactions on*, 29(3):210–224, 2003.

[ES07] J. Euzenat and P. Shvaiko. *Ontology matching*. Springer-Verlag, 2007.

[ES13] Jérôme Euzenat and Pavel Shvaiko. Classifications of ontology matching techniques. In *Ontology matching*, pages 73–84. Springer, 2013.

[FA98] Roberto Fiutem and Giuliano Antoniol. Identifying design-code inconsistencies in object-oriented software: a case study. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 94–102. IEEE, 1998.

[FACF14] Oscar Franco-Bedoya, David Ameller, Dolors Costal, and Xavier Franch. Queso - A quality model for open source software ecosystems. In *ICSOFT-EA 2014 - Proceedings of the 9th International Conference on Software Engineering and Applications, Vienna, Austria, 29-31 August, 2014*, pages 209–221, 2014.

[Fav04a] Jean-Marie Favre. CacOphoNy: Metamodel-Driven Architecture Recovery. In *11th Working Conference on Reverse Engineering (WCRE 2004), Proceedings*, pages 204–213. IEEE, 2004.

[Fav04b] Jean-Marie Favre. CacOphoNy: Metamodel-Driven Architecture Recovery. In *Proceedings of WCRE 2004 (Working Conference on Reverse Engineering)*, pages 204–213. IEEE, 2004.

[Fav05a] Jean-Marie Favre. Foundations of meta-pyramids: Languages vs. metamodels – Episode II: Story of thotus the baboon. In *Language Engineering for Model-*

Driven Software Development, number 04101 in Dagstuhl Seminar Proceedings, 2005.

[Fav05b] Jean-Marie Favre. Foundations of Model (Driven) (Reverse) Engineering: Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, 2005.

[FB07] Andrea Forte and Amy Bruckman. Constructing text: : Wiki as a toolkit for (collaborative?) learning. In *Int. Sym. Wikis*, pages 31–42, 2007.

[FFK+92] A. Finkelsetin, A. Finkelstein, J. Kramer, J. Kramer, B. Nuseibeh, B. Nuseibeh, L. Finkelstein, L. Finkelstein, M. Goedicke, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. 1992.

[FGLP11] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. Empirical Language Analysis in Software Linguistics. In *Software Language Engineering - Third International Conference, SLE 2010, Revised Selected Papers*, volume 6563 of *LNCS*, pages 316–326. Springer, 2011.

[FH09] Peter Fox and James A Hendler. Semantic escience: encoding meaning in next-generation digitally enhanced science. *The Fourth Paradigm*, 2, 2009.

[FLL+12a] J-M Favre, Ralf Lammel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. Linking documentation and source code in a software chrestomathy. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 335–344. IEEE, 2012.

[FLL+12b] Jean-Marie Favre, Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. Linking documentation and source code in a software chrestomathy. In *WCRE*, pages 335–344, 2012.

[FLSV12] Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz, and Andrei Varanovich. 101companies: A Community Project on Software Technologies and Software

Languages. In *Proceedings of TOOLS 2012 (Int'l Conference on Objects, Models, Components, Patterns)*, volume 7304 of *LNCS*, pages 58–74. Springer, 2012.

[FLV12] Jean-Marie Favre, Ralf Lämmel, and Andrei Varanovich. Modeling the linguistic architecture of software products. In *MoDELS*, pages 151–167, 2012.

[FM06] Liliana Favre and Liliana Martinez. Formalizing mda components. In *Proc. of ICSR 2006*, pages 326–339, 2006.

[FN04a] J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. *ENTCS*, 127(3), 2004. Proceedings of the SETra Workshop.

[FN04b] J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. *ENTCS*, 127(3), 2004. Proc. of the SETra Workshop.

[FRJ09] Martin Feilkas, Daniel Ratiu, and E Jurgens. The loss of architectural knowledge during system evolution: An industrial case study. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 188–197. IEEE, 2009.

[Gas08] Holger Gast. Patterns and traceability in teaching software architecture. In *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, PPPJ 2008*, volume 347 of *ACM International Conference Proceeding Series*, pages 23–31. ACM, 2008.

[GCS13] Carlos Gómez, Brendan Cleary, and Leif Singer. A study of innovation diffusion through link sharing on stack overflow. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 81–84, 2013.

[GD11] Matan Gavish and David Donoho. A universal identifier for computational results. *Procedia Computer Science*, 4:637–647, 2011.

[GFK+04] Aldo Gangemi, Frehiwot Fisseha, Johannes Keizer, Jos Lehmann, Anita Liang, Ian Pettman, Margerita Sini, and Marc Taconet. A core ontology of fishery and its use in the fishery ontology service project. 2004.

[GG07] Ismênia Galvão and Arda Goknil. Survey of Traceability Approaches in Model-Driven Engineering. In *EDOC 2007*, pages 313–326. IEEE Computer Society, 2007.

[GGM+02] Aldo Gangemi, Nicola Guarino, Claudio Masolo, Alessandro Oltramari, and Luc Schneider. Sweetening ontologies with dolce. In *Knowledge engineering and knowledge management: Ontologies and the semantic Web*, pages 166–181. Springer, 2002.

[GLM+08] Ioannis Giannoukos, Ioanna Lykourentzou, Giorgos Mpardis, Vassilis Nikolopoulos, Vassilis Loumos, and Eleftherios Kayafas. Collaborative e-learning environments enhanced by wiki technologies. In *PETRA*, page 59, 2008.

[GM11] Pieter Van Gorp and Steffen Mazanek. Share: a web portal for creating and sharing executable research papers. In *ICCS*, pages 589–597, 2011.

[Goe11] Stijn Goedertier. Asset description metadata schema for software. `https://joinup.ec.europa.eu/asset/adms_foss/description`, dec 2011. [Online; accessed 25.04.2015].

[Gou13] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR'13, pages 233–236, 2013.

[GPFI+] AsunciÃşn GÃşmez-PÃ‘rez, Mariano FernÃąndez, Facultad De Informtica, E. U. Politcnica, and Campus Universitario Ctra. Towards a method to conceptualize domain ontologies.

[Gra96] Robert M. Grant. Prospering in dynamically-competitive environments: Organizational capability as knowledge integration. *Organization Science*, 7(4):375–387, 1996.

[Gra07] Bas Graaf. *Model-Driven Evolution of Software Architectures*. Dissertation, Delft University of Technology, 2007.

[Gro06] Object Management Group. *Object Constraint Language Object Constraint Language, OMG Available Specification, Version 2.0*, 2006.

[Gru95] Thomas R Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, 43(5):907–928, 1995.

[GS09] Peter Godfrey-Smith. *Theory and reality: An introduction to the philosophy of science*. University of Chicago Press, 2009.

[GTS10] Lars Grammel, Christoph Treude, and Margaret-Anne Storey. Mashup environments in software engineering. In *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, Web2SE '10, pages 24–25. ACM, 2010.

[Gui05] Giancarlo Guizzardi. *Ontological foundations for structural conceptual models*. CTIT, Centre for Telematics and Information Technology, 2005.

[Had03] Said Hadjerrouit. Toward a constructivist approach to e-learning in software engineering. In *E-Learn: World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education*, pages 507–514. Association for the Advancement of Computing in Education (AACE), 2003.

[Ham08] Mohamed Hamada. An integrated virtual environment for active and collaborative e-learning in theory of computation. *IEEE Transactions on Learning Technologies*, 1(2):117–130, 2008.

[HAMM10] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *Proceedings of WCRE 2010 (Working Conference on Reverse Engineering)*, pages 35–44. IEEE, 2010.

[HB11a] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011.

[HB11b] Tom Heath and Christian 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.

[HHJW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type Classes in Haskell. *TOPLAS*, 18(2):109–138, 1996.

[HHT11] Harry Halpin, Patrick J. Hayes, and Henry S. Thompson. When owl:sameAs isn't the Same Redux: A preliminary theory of identity and inference on the Semantic Web. In *Proc. of LHD 2011)*, pages 25–30, 2011.

[HMMP10] Rich Hilliard, Ivano Malavolta, Henry Muccini, and Patrizio Pelliccione. Realizing Architecture Frameworks Through Megamodelling Techniques. In *Proc. of ASE'10*, pages 305–308. ACM, 2010.

[HN03] Jimmy C Huang and Sue Newell. Knowledge integration processes and dynamics within the context of cross-functional projects. *International Journal of Project Management*, 21(3):167 – 176, 2003.

[HNS99] C. Hofmeister, R.L. Nord, and D. Soni. Describing software architecture with uml. In *Software Architecture*, volume 12 of *IFIP âĂŤ The International Federation for Information Processing*, pages 145–159. Springer US, 1999.

[HOJT05] William H. Harrison, Harold Ossher, Stanley M. Sutton Jr., and Peri L. Tarr. Supporting aspect-oriented software development with the Concern Manipulation Environment. *IBM Systems Journal*, 44(2):309–318, 2005.

[How08] James Howison. Cross-repository data linking with RDF and OWL: Towards common ontologies for representing FLOSS data. In *WoPDaSD (Workshop on Public Data at International Conference on Open Source Software)*, 2008.

[HSG10] Regina Hebig, Andreas Seibel, and Holger Giese. On the unification of megamodels. In *Proceedings of the 4th Int'l Workshop on Multi Paradigm Modeling (MPM'10) at the 13th IEEE/ACM Int'l Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo, Norway*, 10 2010.

[HSGPML14] Brian Henderson-Sellers, Cesar Gonzalez-Perez, Tom McBride, and Graham Low. An ontology for {ISO} software engineering standards: 1) creating the infrastructure. *Computer Standards & Interfaces*, 36(3):563 – 576, 2014.

[HTT] Tony Hey, Stewart Tansley, and Kristin Tolle. Jim gray on escience: A transformed scientific method. *Fourth Paradigm*.

[Hue97] Gérard Huet. The Zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. http://www.cs.nott.ac.uk/ gmh/book.html.

[HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, 2011.

[HZMK08] Matthias Hauswirth, Dmitrijs Zaparanuks, Amirhossein Malekpour, and Mostafa Keikha. The JavaFest: a collaborative learning technique for Java programming courses. In *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, PPPJ 2008*, volume 347 of *ACM International Conference Proceeding Series*, pages 3–12. ACM, 2008.

[ICH12] Aftab Iqbal, Richard Cyganiak, and Michael Hausenblas1. Integrating floss repositories on the web. 2012.

[IH12] Aftab Iqbal and Michael Hausenblas. Integrating developer-related information across open source repositories. In *IRI*, pages 69–76, 2012.

[IUHT09] Aftab Iqbal, Oana Ureche, Michael Hausenblas, and Giovanni Tummarello. Ld2sd: Linked data driven software development. In *SEKE*, pages 240–245, 2009.

[Jac04] J.-P. Jacquot. A full Java post-graduate curriculum. In *Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java, PPPJ 2004*, volume 91 of *ACM International Conference Proceeding Series*, pages 46–51. ACM, 2004.

[Jan14] Slinger Jansen. Measuring the health of open source software ecosystems: Beyond the scope of project health. *Information & Software Technology*, 56(11):1508–1519, 2014.

[JK06] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proc. of MODELS 2005, Satellite Events*, volume 3844 of *LNCS*, pages 128–138. Springer, 2006.

[Jon72] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11âĂŞ21, 1972.

[JSB12] Ethan K. Jackson, Wolfram Schulte, and Nikolaj Bjørner. Detecting Specification Errors in Declarative Languages with Constraints. In *Proc. of MODELS 2012*, pages 399–414, 2012.

[JVB+10] Frédéric Jouault, Bert Vanhooff, Hugo Bruneliere, Guillaume Doux, Yolande Berbers, and Jean Bézivin. Inter-dsl coordination support by combining megamodeling and model weaving. In *SAC*, pages 2011–2018, 2010.

[KBA02a] I. Kurtev, J. Bézivin, and M. Aksit. Technological Spaces: an Initial Appraisal. In *CoopIS, DOA 2002 Federated Conferences, Industrial track*, 2002.

[KBA02b] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, 2002.

[KDSG14] Eirini Kalliamvakou, Daniela Damian, Leif Singer, and Daniel M German. The code-centric collaboration perspective: Evidence from github. Technical report, Technical Report DCS-352-IR, University of Victoria, 2014.

[KFH+12a] Iman Keivanloo, Christopher Forbes, Aseel Hmood, Mostafa Erfani, Christopher Neal, George Peristerakis, and Juergen Rilling. A Linked Data platform for mining software repositories. In *Proc. of MSR 2012*, pages 32–35. IEEE, 2012.

[KFH+12b] Iman Keivanloo, Christopher Forbes, Aseel Hmood, Mostafa Erfani, Christopher Neal, George Peristerakis, and Juergen Rilling. A Linked Data platform for mining software repositories. In *Proc. of MSR 2012*, pages 32–35. IEEE, 2012.

[KFH+12c] Iman Keivanloo, Christopher Forbes, Aseel Hmood, Mostafa Erfani, Christopher Neal, George Peristerakis, and Juergen Rilling. A linked data platform for mining software repositories. In *MSR*, pages 32–35, 2012.

[KFRC11] Iman Keivanloo, Christopher Forbes, Juergen Rilling, and Philippe Charland. Towards sharing source code facts using linked data. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, SUITE '11, pages 25–28. ACM, 2011.

[KG11] Siim Karus and Harald Gall. A study of language usage evolution in open source software. In *Proceedings of MSR 2011 (Mining Software Repositories)*, pages 13–22. IEEE, 2011.

[KGB+14] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101. ACM, 2014.

[KK12] Bernhard Katzmarski and Rainer Koschke. Program complexity metrics and programmer opinions. In *Proc. of ICPC 2012*, pages 17–26. IEEE, 2012.

[KKS+11] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni. Design Defects Detection and Correction by Example. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 81–90, June 2011.

[KLL09] Sungwon Kang, Seonah Lee, and Danhyung Lee. A framework for tool-based software architecture reconstruction. *Int'l Journal of Software Engineering and Knowledge Engineering*, 19(2):283–30, 2009.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proc. of ECOOP 1997*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[KLRB11] Peter Kraker, Derick Leony, Wolfgang Reinhardt, and Günter Beham. The case for an open science in technology enhanced learning. *International Journal of Technology Enhanced Learning*, 3(6):643–654, 2011.

[KLV05] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, 2005.

[KLW06] Kostas Kontogiannis, Panos Linos, and Kenny Wong. Comprehension and Maintenance of Large-Scale Multi-Language Software Applications. In *Proceedings of ICSM 2006 (Int'l Conference on Software Maintenance)*, pages 497–500, 2006.

[KMA04] Andrew Jensen Ko, Brad A Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 199–206. IEEE, 2004.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27:97–111, 1984.

[KS96] Gerald Kotonya and Ian Sommerville. Requirements engineering with viewpoints. 1996.

[Küh06a] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.

[Küh06b] Thomas Kühne. Matters of (Meta-) Modeling. *Software and Systems Modeling*, 5:369–385, 2006.

[KUM11] Shingo Kawamura, Minoru Uehara, and Hideki Mori. A Method for Project Member Role Assignment in Open Source Software Development Using Self-Organizing Maps, 2011.

[KWDE98] Bernt Kullbach, Andreas Winter, Peter Dahm, and Jürgen Ebert. Program Comprehension in Multi-Language Systems. In *Proceedings of WCRE 1998 (Working Conference on Reverse Engineering)*, pages 135–143, 1998.

[KYNM06] Toshinobu Kasai, Haruhisa Yamaguchi, Kazuo Nagano, and Riichiro Mizoguchi. Building an ontology of IT education goals. *International Journal of Continuing Engineering Education and Life Long Learning*, 16:1–17, 2006.

[Lae13] Software chrestomathies. *Science of Computer Programming*, 2013.

[Läm13] Ralf Lämmel. Software chrestomathies. Liber amicorum Paul Klint. Available online `http://softlang.uni-koblenz.de/chrestomathy/`, 2013.

[Lay04] B. Layton. *Coptic Gnostic Chrestomathy: A Selection of Coptic Texts with Grammatical Analysis and Glossary*. Peeters Pub, 2004.

[Lin00] Marcia C Linn. Designing the knowledge integration environment. *International Journal of Science Education*, 22(8):781–796, 2000.

[Lip11] Miran Lipovaca. *Learn You a Haskell for Great Good!* no starch press, 2011. http://learnyouahaskell.com/.

[LJ03a] Ralf Lämmel and Simon L. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proc. of TLDI'03*, pages 26–37. ACM, 2003.

[LJ03b] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proc. of TLDI 2003*, pages 26–37. ACM, 2003.

[LK94] Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[LLSV14a] R. Lammel, M. Leinberger, T. Schmorleiz, and A. Varanovich. Comparison of feature implementations across languages, technologies, and styles. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 333–337, Feb 2014.

[LLSV14b] Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. Comparison of Feature Implementations across Languages, Technologies, and Styles. In *Proc. of IEEE CSMR-WCRE 2014*. IEEE, 2014.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical report, University of Utrecht, 2001. Technical Report no. UU-CS-2001-27.

[LM06a] Ralf Lämmel and Erik Meijer. Mappings Make Data Processing Go 'Round. In *Generative and Transformational Techniques in Software Engineering, Int'l Summer School, GTTSE 2005. Revised Papers*, volume 4143 of *LNCS*, pages 169–218. Springer, 2006.

[LM06b] Ralf Lämmel and Erik Meijer. Revealing the x/o impedance mismatch - (changing lead into gold). In *SSDGP*, pages 285–367, 2006.

[LM07] R. Lämmel and E. Meijer. Revealing the X/O impedance mismatch (Changing lead into gold). In *Spring School on Datatype-Generic Programming, Lecture Notes*, volume 4719 of *LNCS*, pages 285–367. Springer, 2007.

[LMV13] Ralf Lämmel, Dominik Mosen, and Andrei Varanovich. Method and tool support for classifying software languages with wikipedia. In *SLE*, pages 249–259, 2013.

[LOV02] Christoph Stoermer Liam OâĂŹBrien and Chris Verhoef. Software architecture reconstruction: Practice needs and current approaches. Technical report, Carnegie Mellon, 2002.

[LP07] Patrick Connor Linskey and Marc Prud'hommeaux. An In-depth Look at the Architecture of an Object/Relational Mapper. In *Proceedings of SIGMOD 2007*, pages 889–894. ACM, 2007.

[LPT06] Josh Lerner, Parag Pathak, and Jean Tirole. The Dynamics of Open Source Contributors. In *Allied Social Science Associations 2006 Annual Meeting. AEA Conference Papers*, page 14 pages, 2006.

[LR11] Kevin Lano and Shekoufeh Kolahdouz Rahimi. Model-driven development of model transformations. In *Theory and Practice of Model Transformations, Fourth Int'l Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, LNCS, pages 47–61. Springer, 2011.

[LSV14] Ralf Lämmel, Thomas Schmorleiz, and Andrei Varanovich. The 101haskell Chrestomathy—A Whole Bunch of Learnable Lambdas. In *Postproceedings of IFL 2013*, 2014. 12 pages. To appear in ACM DL.

[Lun08] Mircea Lungu. Towards reverse engineering software ecosystems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 428–431. IEEE, 2008.

[LV14] Ralf Lämmel and Andrei Varanovich. Interpretation of Linguistic Architecture. 16 pages. Accepted at ECMFA 2014, 2014.

[LVL+14] Ralf Lämmel, Andrei Varanovich, Martin Leinberger, Thomas Schmorleiz, and Jean-Marie Favre. Declarative Software Development (Distilled Tutorial). In *Proc. of PPDP 2014*. ACM, 2014. 6 pages.

[MAB07] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. In *SIGMOD '07: Proceedings of the 2007*

ACM SIGMOD international conference on Management of data, pages 461–472. ACM, 2007.

[Man13] Many contributors. Rosetta Code, 2013. Wiki: `http://rosettacode.org` — Accessed on 20 Mar 2014.

[May04] Richard E Mayer. Should there be a three-strikes rule against pure discovery learning? *American psychologist*, 59(1):14, 2004.

[MBM13] Diego Mendez, Benoit Baudry, and Martin Monperrus. Empirical evidence of large-scale diversity in api usage of object-oriented software. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 43–52. IEEE, 2013.

[MCG14] Tom Mens, Maëlick Claes, and Philippe Grosjean. ECOS: ecological studies of open source software ecosystems. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, pages 403–406, 2014.

[Mei06] Erik Meijer. There is no impedance mismatch: (language integrated query in visual basic 9). In *OOPSLA Companion*, pages 710–711, 2006.

[Mes10] Jill P Mesirov. Computer science. accessible reproducible research. *Science (New York, NY)*, 327(5964), 2010.

[MFB09] Pierre-Alain Muller, Frédéric Fondement, and Benoit Baudry. Modeling Modeling. In *Model Driven Engineering Languages and Systems, 12th Int'l Conference, MODELS 2009, Proceedings*, volume 5795 of *LNCS*, pages 2–16. Springer, 2009.

[MFBC11] Pierre-Alain Muller, Frédéric Fondement, Benoit Baudry, and Benoît Combemale. Modeling modeling modeling. *Software and Systems Modeling*, pages 1–13, 2011.

[MG08] Shah J. Miah and John Gammack. A mashup architecture for web end-user application designs. In *Second IEEE International Conference on Digital Ecosystems and Technologies*, 2008.

[MM03] Andrian Marcus and Jonathan I. Maletic. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In *Proc. of ICSE 2003*, pages 125–137. ACM, 2003.

[MMM+11] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design lessons from the fastest q&a site in the west. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 2857–2866. ACM, 2011.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of SIGSOFT FSE 1995 (ACM SIGSOFT Symposium on Foundations of Software Engineering)*, pages 18–28. ACM, 1995.

[MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Trans. Software Eng.*, 27:364–380, 2001.

[MS05] David G Messerschmitt and Clemens Szyperski. Software ecosystem: understanding an indispensable technology and industry. *MIT Press Books*, 1, 2005.

[MS12] Philip Mayer and Andreas Schroeder. Cross-language code analysis and refactoring. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, pages 94–103. IEEE, 2012.

[MSC+01] Spiros Mancoridis, Timothy S. Souder, Yih-Farn Chen, Emden R. Gansner, and Jeffrey L. Korn. REportal: A Web-Based Portal Site for Reverse Engineering. In *Proceedings of WCRE 2001 (Working Conference on Reverse Engineering)*, pages 221–230. IEEE, 2001.

[MSRM04a] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223, Nov 2004.

[MSRM04b] Andrian Marcus, Andrey Sergeyev, Václav Rajlich, and Jonathan I. Maletic. An Information Retrieval Approach to Concept Location in Source Code. In *Proc. of WCRE 2004*, pages 214–223. IEEE, 2004.

[MT07] Shailey Minocha and Peter G Thomas. Collaborative learning in a wiki environment: Experiences from a software engineering course. *New Review of Hypermedia and Multimedia*, 13(2):187–209, 2007.

[MV11] Bart Meyers and Hans Vangheluwe. A framework for evolution of modelling languages. *Sci. Comput. Program.*, 76(12):1223–1246, 2011.

[MVDBK14a] Kim Mens, MGJ Van Den Brand, and Holger M Kienle. Guest editors' introduction to the 4th issue of experimental software and toolkits (est-4). *Science of Computer Programming*, 79:1–5, 2014.

[MvdBK14b] Kim Mens, M.G.J. van den Brand, and Holger M. Kienle. Guest editorsâĂŹ introduction to the 4th issue of experimental software and toolkits (est-4). *Science of Computer Programming*, 79(0):1 – 5, 2014.

[MVG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, March 2006.

[MWHH06] Daniel L. Moise, Kenny Wong, H. James Hoover, and Daqing Hou. Reverse Engineering Scripting Language Extensions. In *14th Int'l Conference on Program Comprehension (ICPC 2006). Proceedings*, pages 295–306. IEEE, 2006.

[Nas05] Naslavsky, Leila and Alspaugh, Thomas A and Richardson, Debra J and Ziv, Hadar. Using scenarios to support traceability. In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, pages 25–30. ACM, 2005.

[NCH+11] Piotr Nowakowski, Eryk Ciepiela, Daniel HarÄŹÅijlak, Joanna Kocot, Marek Kasztelnik, Tomasz BartyÅĎski, Jan Meizner, Grzegorz Dyk, and Maciej Malawski. The collage authoring environment. *Procedia Computer Science*, 4(0):608 – 617, 2011. Proceedings of the International Conference on Computational Science, {ICCS} 2011.

[NdAFA13] Julio Cesar Nardi, Ricardo de Almeida Falbo, and João Paulo A Almeida. Foundational ontologies for semantic integration in eai: a systematic literature review. In *Collaborative, Trusted and Privacy-Aware e/m-Services*, pages 238–249. Springer, 2013.

[Nie11] Michael Nielsen. *Reinventing Discovery: The New Era of Networked Science*. Princeton University Press, 2011.

[NL12] Oscar Nierstrasz and Mircea Lungu. Agile software assessment (Invited paper). In *Proceedings of ICPC 2012 (Int'l Conference on Program Comprehension)*, pages 3–10. IEEE, 2012.

[NS00] Lee Naish and Leon Sterling. Stepwise enhancement and higher-order programming in prolog. *Journal of Functional and Logic Programming*, 2000(4), 2000.

[Obe06] Daniel Oberle. *Semantic management of middleware*, volume 1. Springer Science & Business Media, 2006.

[OLG+06] Daniel Oberle, Steffen Lamparter, Stephan Grimm, Denny Vrandecic, Steffen Staab, and Aldo Gangemi. Towards ontologies for formalizing modularization and communication in large software systems. *Applied Ontology*, 1(2):163–202, 2006.

[Ome02] Borys Omelayenko. Integrating Vocabularies: Discovering and Representing Vocabulary Maps. In *Proc. of ISWC 2002*, volume 2342 of *LNCS*, pages 206–220. Springer, 2002.

[ON08] Shaul Oreg and Oded Nov. Exploring motivations for contributing to open source initiatives: The roles of contribution context and personal values. *Computers in Human Behavior*, 24(5):2055–2073, 2008.

[OSG08] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. O'Reilly Media, 2008. `http://book.realworldhaskell.org/`.

[PDMG14] Francis Palma, Johann Dubois, Naouel Moha, and Yann-Gaël Guéhéneuc. Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach. In *Proceedings of ICSOC 2014*, volume 8831 of *LNCS*, pages 230–244. Springer, 2014.

[Per11] Vassilios Peristeras. Open government metadata, September 2011.

[PFGJ02] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri. Revealer: A Lexical Pattern Matcher for Architecture Recovery. In *Proceedings of WCRE 2002 (Working Conference on Reverse Engineering)*, pages 170–. IEEE, 2002.

[PGM+07] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *Software Engineering, IEEE Transactions on*, 33(6):420–432, 2007.

[PGS02] Domenico M. Pisanelli, Aldo Gangemi, and Geri Steve. G.: Ontologies and information systems: the marriage of the century. In *In Proceedings of LYEE Workshop*, 2002.

[PMTG14] Francis Palma, Naouel Moha, Guy Tremblay, and Yann-Gaël Guéhéneuc. Specification and Detection of SOA Antipatterns in Web Services. In *Proceedings of ECSA 2014*, volume 8627 of *LNCS*, pages 58–73. Springer, 2014.

[PTRC07] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[PXT+11] Xin Peng, Zhenchang Xing, Xi Tan, Yijun Yu, and Wenyun Zhao. Iterative context-aware feature location (NIER track). In *Proc. of ICSE 2011*, pages 900–903. ACM, 2011.

[RBB97] Martin Rajman, Romaric BESANÃĞON, and R. Besancon. Text Mining: Natural Language techniques and Text Mining applications. In *Proc. of DS-7*, pages 7–10. Hall, 1997.

[RDL08] Anna Riccioni, Enrico Denti, and Roberto Laschi. An experimental environment for teaching Java security. In *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, PPPJ 2008*, volume 347 of *ACM International Conference Proceeding Series*, pages 13–22. ACM, 2008.

[RFD+08] Daniel Ratiu, Martin Feilkas, Florian Deissenboeck, Jan Jürjens, and Radu Marinescu. Towards a Repository of Common Programming Technologies Knowledge. In *Proc. of the Int. Workshop on Semantic Technologies in System Maintenance (STSM)*, 2008.

[RFJ08] Daniel Ratiu, Martin Feilkas, and Jan Jurjens. Extracting domain ontologies from domain specific APIs. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 203–212. IEEE, 2008.

[RH06] Francisco Ruiz and José R Hilera. Using ontologies in software engineering and technology. In *Ontologies for software engineering and software technology*, pages 49–102. Springer, 2006.

[RJ01] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *Software Engineering, IEEE Transactions on*, 27(1):58–93, 2001.

[RJJ+08] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008*, pages 111–122. ACM, 2008.

[Ron12] Ronald Bourret. Xml data binding resources, 2001–2012.

[Rue01] Fritz Ruehr. The Evolution of a Haskell Programmer, 2001. Website: `http://www.willamette.edu/~fruehr/haskell/evolution.html` — Accessed on 20 Mar 2014.

[RW02] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *Proc. of IWPC 2002*, pages 271–280. IEEE, 2002.

[RYR13] Mohammad Masudur Rahman, Shamima Yeasmin, and Chanchal K. Roy. An ide-based context-aware meta search engine. In *WCRE*, pages 467–471, 2013.

[SAtDL04] Maarten W. A. Steen, David H. Akehurst, Hugo W. L. ter Doest, and Marc M. Lankhorst. Supporting Viewpoint-Oriented Enterprise Architecture. In *8th Int'l Enterprise Distributed Object Computing Conference (EDOC 2004), Proceedings*, pages 201–211. IEEE, 2004.

[SCBR06] Margaret-Anne D. Storey, Li-Te Cheng, R. Ian Bull, and Peter C. Rigby. Shared waypoints and social tagging to support collaboration in software development. In *Proceedings of CSCW 2006 (Computer Supported Cooperative Work)*, pages 195–198. ACM, 2006.

[SCFC09] 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.

[SdAFV13] Erica F. Souza, Ricardo de Almeida Falbo, and N. L. Vijaykumar. Ontologies in software testing: A systematic literature review. In *ONTOBRAS*, volume 1041 of *CEUR Workshop Proceedings*, pages 71–82. CEUR-WS.org, 2013.

[SDdOV08] Leonardo Silva, Samuel Domingues, and Marco Tulio de Oliveira Valente. Non-invasive and non-scattered annotations for more robust pointcuts. In *Proceedings of ICSM 2008 (Int'l Conference on Software Maintenance)*, pages 67–76. IEEE, 2008.

[SE93] Steve Easterbrook School and Steve Easterbrook. Domain modelling with hierarchies of alternative viewpoints. 1993.

[SFFC+13] Leif Singer, Fernando Figueira Filho, Brendan Cleary, Christoph Treude, Margaret-Anne Storey, and Kurt Schneider. Mutual assessment in the social programmer ecosystem: An empirical investigation of developer profile aggregators. In *Proceedings of the 2013 conference on Computer supported cooperative work*, pages 103–116. ACM, 2013.

[SFS14] Leif Singer, Fernando Marques Figueira Filho, and Margaret-Anne D. Storey. Software engineering at the speed of light: how developers stay current using twitter. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 211–221, 2014.

[SG96] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.

[SG05] Sergey Sosnovsky and Tatiana Gavrilova. Development of Educational Ontology for C-Programming. In *Proceedings of the XI-th International Conference Knowledge-Dialogue-Solution*, volume 1, pages 127–132. FOI ITHEA, 2005.

[SKG+13] Zéphyrin Soh, Foutse Khomh, Y-G Guéhéneuc, Giuliano Antoniol, and Bram Adams. On the effect of program exploration on maintenance tasks. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 391–400. IEEE, 2013.

[SL16] Thomas Schmorleiz and Ralf Lämmel. Similarity management of 'cloned and owned' variants. In *Proc. of SAC 2016*. ACM, 2016. 6 pages.

[Smi01] Fois introduction: Ontology—towards a new synthesis. In *Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001*, FOIS '01, pages .3–.9, 2001.

[Smi11] Eefke Smit. Abelard and héloise: Why data and publications belong together. *D-lib magazine*, 17(1):7, 2011.

[SMSB11] Dimitrios L Settas, Georgios Meditskos, Ioannis G Stamelos, and Nick Bassiliades. SPARSE: A symptom-based antipattern retrieval knowledge-based system using Semantic Web technologies. *Expert Systems with Applications*, 38(6):7633–7646, 2011.

[SNG10] Andreas Seibel, Stefan Neumann, and Holger Giese. Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. *Software & Systems Modeling*, 9(4):493–528, 2010.

[SSFS11] Ansgar Scherp, Carsten Saathoff, Thomas Franz, and Steffen Staab. Designing core ontologies. *Applied Ontology*, 6(3):177–221, 2011.

[SSS06] Martin Sulzmann, Tom Schrijvers, and Peter J. Stuckey. Principal Type Inference for GHC-Style Multi-parameter Type Classes. In *Proc. of APLAS 2006*, volume 4279 of *LNCS*, pages 26–43. Springer, 2006.

[SSSS01] Steffen Staab, Rudi Studer, Hans-Peter Schnurr, and York Sure. Knowledge processes and ontologies. *IEEE Intelligent Systems*, 16(1):26–34, 2001.

[SZ92] John F. Sowa and John A. Zachman. Extending and formalizing the framework for information systems architecture. *IBM systems journal*, 31(3):590–616, 1992.

[Tho03] Dave Thomas. The Impedance Imperative: Tuples + Objects + Infosets = Too Much Stuff! *Journal of Object Technology*, 2(5):7–12, September–October 2003.

[Tho11] Simon Thompson. *Haskell: The Craft of Functional Programming (3rd edition)*. Addison-Wesley, 2011. `http://www.haskellcraft.com/craft3e/Home.html`.

[Til08] Stefan Tilkov. Rest anti-patterns @ONLINE, jul 2008.

[TKB10] Jonas Tappolet, Christoph Kiefer, and Abraham Bernstein. Semantic web enabled software analysis. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(2):225–240, 2010.

[TMWW93] Scott R. Tilley, Hausi A. Müller, Michael J. Whitney, and Kenny Wong. Domain-Retargetable Reverse Engineering. In *Proceedings of ICSM 1993 (Int'l Conference on Software Maintenance)*, pages 142–151. IEEE, 1993.

[TRT14] Federico Tomassetti, Giuseppe Rizzo, and Marco Torchiano. Spotting automatically cross-language relations. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 338–342. IEEE, 2014.

[TS10] Eli Tilevich and Myoungkyu Song. Reusable enterprise metadata with pattern-based structural expressions. In *Proceedings of AOSD 2010 (Aspect-Oriented Software Development)*, pages 25–36. ACM, 2010.

[TT14] Federico Tomassetti and Marco Torchiano. An empirical assessment of polyglotism in github. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 17. ACM, 2014.

[TTV13] Federico Tomassetti, Marco Torchiano, and Antonio Vetro. Classification of language interactions. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 287–290. IEEE, 2013.

[TVT+13] Federico Tomassetti, Antonio Vetró, Marco Torchiano, Markus Voelter, and Bernd Kolb. A model-based approach to language integration. In *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, pages 76–81. IEEE Press, 2013.

[TWSM94] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Müller. Programmable reverse engineering. *Int'l Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[vDHK+04] Arie van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva. Symphony: View-Driven Software Architecture Reconstruction. In *Proceedings of WICSA 2004 (Working IEEE / IFIP Conference on Software Architecture)*, pages 122–134. IEEE, 2004.

[VFS13] B. Vasilescu, V. Filkov, and A. Serebrenik. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *Social Computing (SocialCom), 2013 International Conference on*, pages 188–195, Sept 2013.

[VJBB11] Andrés Vignaga, Frédéric Jouault, Marya Bastarrica, and Hugo Brunelière. Typing Artifacts in Megamodeling. *Software and Systems Modeling*, pages 1–15, 2011.

[VKP02] Antje Von Knethen and Barbara Paech. A survey on tracing approaches in practice and research. *Frauenhofer Institut Experimentelles Software Engineering, IESE-Report No*, 95, 2002.

[VKV+06] Max Völkel, Markus Krötzsch, Denny Vrandecic, Heiko Haller, and Rudi Studer. Semantic Wikipedia. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 585–594. ACM, 2006.

[VSG11] Thomas Vogel, Andreas Seibel, and Holger Giese. The Role of Models and Megamodels at Runtime. In *Models in Software Engineering - Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers*, volume 6627 of *LNCS*, pages 224–238. Springer, 2011.

[Wad92] Philip Wadler. The Essence of Functional Programming. In *Conference Record of POPL 1992*, pages 1–14. ACM, 1992.

[Wei13] Jim Weirich. OO example code, 2013. Website: `http://onestepback.org/articles/poly` — Accessed on 20 Mar 2014.

[WGGM14] Xiaowei Wang, Nicola Guarino, Giancarlo Guizzardi, and John Mylopoulos. Towards an Ontology of Software: a Requirements Engineering Perspective. In *Formal Ontology in Information Systems - Proceedings of the Eighth International Conference, FOIS 2014, September, 22-25, 2014, Rio de Janeiro, Brazil*, pages 317–329, 2014.

[WGGS92] Norman Wilde, Juan A Gomez, Thomas Gust, and Douglas Strasburg. Locating user functionality in old code. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 200–205. IEEE, 1992.

[WLJ13] Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study on developer interactions in stackoverflow. In *SAC*, pages 1019–1024, 2013.

[WPXZ13] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. Improving feature location practice with multi-faceted interactive exploration. In *Proc. of ICSE 2013*, pages 762–771. ACM, 2013.

[WvP10] Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and System Modeling*, 9(4):529–565, 2010.

[YYN+07] Yunwen Ye, Yasuhiro Yamamoto, Kumiyo Nakakoji, Yoshiyuki Nishinaka, and Mitsuhiro Asada. Searching the library and asking the peers: learning to use Java APIs on demand. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, PPPJ 2007*, volume 272 of *ACM International Conference Proceeding Series*, pages 41–50. ACM, 2007.

[Zac87] John A Zachman. A framework for information systems architecture. *IBM systems journal*, 26(3):276–292, 1987.

[Zac02] John Zachman. The zachman framework for enterprise architecture. *Zachman International*, 79, 2002.

[ZN04] Jianna J. Zhang and Huy Nguyen. An example oriented on-line java tutorial for university students. In John Waldron, editor, *Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java, PPPJ 2004, Las Vegas, Nevada, USA, June 16-18, 2004*, volume 91 of *ACM International Conference Proceeding Series*, pages 52–60. ACM, 2004.