Are you interested in new directions of software development? Would you like to have a detailed understanding of domain-specific languages? If you answered yes to either of these questions, then you should explore this book, which contains original academic work about current research in different areas of domain-specific languages. The book’s mission is to give a comprehensive overview of the research in the field of domain-specific languages. Domain-specific languages represent an emerging technology, but their application is currently unduly limited by a lack of reliable knowledge available to potential domain-specific language developers. To remedy this situation, this book provides comprehensive material for anyone who would like to introduce domain-specific languages into their software engineering process. This book’s ambition is to provide new results and answers to several open problems in domain-specific language research. The book is therefore indispensable for researchers and practitioners in the field of software engineering, as well as for educators who would like to introduce domain-specific languages into their curriculum.

There exist a plethora of computer languages. Some of them, such as Java and C#, are general-purpose programming languages that have many common features that can be used to write programs for a wide range of applications. These languages are the primary tools of the programmer. However, not all general-purpose computer languages are programming languages. For example, UML is a general-purpose modeling language for the abstract specification and documentation of many different kinds of software, and Z is a general-purpose formal specification language. Generality is a mixed blessing. Broad applicability often results in suboptimal expressiveness in any particular application domain, hence the motivation for domain-specific languages, which sacrifice generality in exchange for enhanced expressiveness in a particular domain. By providing notations and constructs tailored toward a particular application domain, domain-specific languages offer substantial gains in expressiveness and ease of use compared with general-purpose languages for the domain in question, with corresponding gains in productivity and reduced maintenance costs. By reducing the amount of domain and software development expertise needed, domain-specific languages expand their application to a larger group of software developers compared to general-purpose languages. These benefits have often been observed in practice and are supported by quantitative studies, although perhaps not as many as one would expect. The advantages of specialization are equally valid for programming, modeling, and specification languages.

Domain-specific languages have been around for a long time. BNF, for instance, dates back to the late 1950s, and represents an example of a domain-specific language that can be used to describe the syntax of a language. Although there are examples of many domain-specific languages from times past, it has only been recently that workshops and conferences emerged that have a focus on domain-specific languages. The development of domain-specific language is itself a significant software engineering
task, requiring a considerable investment of time and resources. Without appropriate methodology and tools, the investment for domain-specific language support can exceed the savings that might be obtained from their usage. Hence, domain-specific language development should be based on strong software engineering principles with clearly identified phases (e.g., decision, analysis, design, implementation, deployment, maintenance) and artifacts. Despite the recent attention that domain-specific languages have attracted from researchers, many challenges still exist that need to be solved before domain-specific languages will become fully adopted by domain experts, end-users, and professional programmers. For example, general-purpose languages are often designed with great care and comply to language design principles. Unfortunately, this is not always the case for the design and development of domain-specific languages. Design and implementation of general-purpose languages may last several months, while development of domain-specific languages should be more cost effective. However, there is a serious threat that such cost-effectiveness will negatively impact language design and/or implementation. In this respect, some open problems are:

- How should results from domain analysis drive the language design process?
- Should a domain-specific language be designed by a domain expert, general-purpose language designer or software language engineer?
- How much domain analysis and language design is actually needed?
- Can we skip some initial domain-specific languages development phases? What are the consequences?
- Can we build tools that support earlier phases of domain-specific language development?

There are no straightforward answers to these questions, and actual decisions depend on many factors, such as domain-specific language end-users, project budget, time to market, and domain-specific language life span. After deciding to invest into domain-specific language development, a domain analysis should be performed with the aim to build the domain model, which is an explicit representation of the common and the variable properties of the system in a domain. The domain model should also describe the semantics of domain concepts and the dependencies between these concepts. Some typical domain analysis activities are analysis of similarity, analysis of variations, and analysis of combinations. Despite the existence of many domain analysis methods (e.g., FODA and FAST), they are rarely used in domain-specific language development. As a result, domain analysis is usually done informally and in an incomplete manner. There is an urgent need in domain-specific language research to identify the reasons for such informal development and possible solutions for improvement. An initial observation is that information gathered during domain analysis cannot be automatically used in the language design process. Another reason might be that complete domain analysis is often complex and may be outside of software engineers’ capabilities.

Designing a language involves defining the constructs in the language and giving the semantics to the language, whether formal or informal. The semantics of the language describe the meaning of each construct in the language, but also some fixed behavior that it is not specified by the program. Designing a language is highly creative, but not an easy task. As stated before, information gathered in the domain analysis should be used along with additional constraints (e.g., capability to perform various analysis, readability). Of particular importance for domain-specific languages is the level of abstraction that can be represented in the language and the degree to which it may be analyzed. From previous experiences with programming language design and from criteria for programming language evaluation, researchers...
developed several criteria for good language design, such as readability, writeability, reliability, and cost. However, the criteria for language design are often contradicting (e.g., reliability vs. cost of execution, readability vs. writeability). Hence, language designers should find good trade-offs among these factors. To help language designers in this process, several rules of thumb have been proposed, such as:

- Don’t include untried ideas - consolidation, not innovation.
- Simplicity is really the key - avoid complexity. Too many solutions make the language hard to understand.
- Avoid requiring something to be stated more than once.
- Automate mechanical, tedious, or error-prone activities by providing higher level features.
- Regular rules, without exceptions, are easier to learn, use, describe, and implement.

With respect to the language design phase, another important research question is whether domain-specific language design is radically different from general-purpose language design? The domain-specific language designer has to keep in mind the special characteristics of domain-specific languages, as well as the fact that users of the language may not be programmers. General-purpose languages are designed traditionally by computer scientists. The general-purpose language design process was driven by their personal aesthetics and theoretical judgments resulting in a computer language they would like to use. A domain-specific language for end-users should not be designed by such intuition because general-purpose language designers are not necessarily the targeted end-users. Therefore, domain-specific language design should be informed by empirical studies, involvement of end-users, or by psychology of programming research.

After a domain-specific language is designed, it should be implemented by building a domain-specific translator (e.g., compiler or interpreter). A vast array of implementation choices exists: interpreter/compiler, preprocessing, embedding, extensible compiler/interpreter, commercial of-the-shelf (COTS) components, or some hybrid variation. In the compiler/interpreter approach, standard compiler/interpreter techniques are used to implement a domain-specific language, such that a complete static semantic analysis is done on the program written in the domain-specific language. Using the preprocessing approach, the domain-specific language constructs are simply translated into constructs in a base language (e.g., Java or C++). In the preprocessing approach, static semantic analysis of a domain-specific program is limited to the capabilities available in the base language. There are important sub-patterns of the preprocessing approach (e.g., macro processing, source-to-source transformation, pipeline, and lexical processing). In the embedding approach, existing mechanisms in the host language (e.g., operator overloading, using functions as operators and vice versa) are used to build a library of domain-specific operations. The existing compiler/interpreter for the host language is completely reused, which means that a domain-specific program is also syntactically and semantically valid as a general-purpose program. In the extensible compiler/interpreter approach, the general-purpose compiler/interpreter is extended to be able to cover domain-specific constructs. In the COTS approach, existing tools and/or notations are applied to a specific domain (e.g., XML-based domain-specific languages). Blending particular implementation approaches is also possible and popular. Although various implementation approaches are beneficial, they bring another open research issue: which implementation approach is the most suitable from the point of implementers and from the point of language users? Finally, the differences and advantages/disadvantages in the implementation from the grammarware and modelware point of view have not yet been investigated deeply in the domain-specific language research community. Grammar-based domain-
Specific languages are those languages where structure (syntax) is defined by grammars, while structure of metamodel-based domain-specific languages is defined by metamodels. There is still lively debate among domain-specific language researchers about which syntax specification is the most appropriate.

A solution to the aforementioned problems would enable new horizons in domain-specific language and end-user development that would introduce a new paradigm shift in software development. The domain-specific language field is growing quickly, and it will take some time for stabilization of these issues. This book, comprised of 20 chapters, presents several recent developments in various aspects of domain-specific language development. The book is divided into the following cohesive sections, which can be read independently:

**Section 1**: Internal domain-specific languages (Chapters 1 – 7),
**Section 2**: Domain-specific language semantics (Chapters 8 – 10),
**Section 3**: Domain-specific language tools and processes (Chapters 11 – 15),
**Section 4**: Domain-specific language examples (Chapters 16 – 20).

The term “internal domain-specific language” is specifically related to the design and implementation phases of domain-specific language development. Approaches to domain-specific language design can be further characterized along two orthogonal (independent) dimensions:

- The formal nature of the design description
- The relationship between the domain-specific languages and existing languages

The first dimension is about informal versus formal language design. In an informal design, the specification is usually in some form of natural or computer language, which may include a set of illustrative domain-specific language programs. A formal design consists of a specification written using one of the available formal definition methods (e.g., regular expressions and grammars for syntax specifications, and attribute grammars, denotational semantics, operational semantics, or abstract state machines for semantic specification). The second dimension is about language exploitation versus language invention. In the former case, and often the easiest way, the design of a domain-specific language is based on an existing language, where the existing language is:

- **Partially Used**: Piggyback pattern (e.g., only Java statements are used)
- **Restricted**: Language specialization pattern
- **Extended**: Language extension pattern

One possible benefit of the language exploitation design pattern is familiarity for programmers. The implementation of the domain-specific language is also often easier with this pattern. If there is no relationship between the domain-specific language under development and an existing computer language, then a new language has to be invented. This requires the specification of the new language’s syntax and semantics. Note that language exploitation vs. language invention is completely a design decision, and virtually any implementation pattern can be chosen after that to implement such a domain-specific language. For example, a new compiler or preprocessor can be implemented even when extending an existing language. Of course, such an extension would be in many cases easier to implement by embedding.
An internal domain-specific language is an extension of an existing computer language that is implemented by embedding. Hence, there is no need to write a compiler/interpreter for such a newly developed domain-specific language, because a host language compiler/interpreter is completely reused without any changes. An “external domain-specific language” refers to those domain-specific languages that require the construction of a new domain-specific language translator (usually requiring the implementation of a syntax and/or semantic analyzer). Note that if an existing computer language is only partially used or specialized, the existing host compiler/interpreter cannot be used without changes; such a domain-specific language is not internal. Hence, an internal domain-specific language is just a special case of the language exploitation design pattern (only for language extension pattern) with the requirement that such a domain-specific language can be implemented by embedding. An internal domain-specific language determines both the design and the implementation phase of a particular domain-specific language. On the other hand, an internal domain-specific language cannot be called an embedded domain-specific language because the term “embedding” has been mistakenly used in the domain-specific language community for two purposes and is a source of much confusion.

Firstly, domain-specific language embedding has been used as a synonym for a domain-specific language design phase where the domain-specific language is implemented by reusing existing language constructs by extension, specialization, or by a piggyback approach. Secondly, domain-specific language embedding was used originally as a synonym for a particular domain-specific language implementation approach where the domain-specific language is implemented by using the host language feature of function composition. This embedding is of the purest sense. It is clear that the term “embedding” is overloaded and should not cover both phases of design and implementation. To distinguish these two cases, two patterns emerged for the domain-specific language design phase: language exploitation versus language invention, while embedding in a pure sense is only one out of many different implementation approaches (compiler/interpreter, preprocessing, embedding, extensible compiler/interpreter, COTS, and hybrid). Hence, the terms “heterogeneous embedding” and “homogeneous embedding” have been also used to distinguish these two cases. The heterogeneous embedding indicates the case where a domain-specific language is designed by language exploitation (extension, specialization, piggyback) and not implemented by an embedding approach. As readers may notice, the terminology might be quite confusing - heterogeneous embedding uses the term “embedding” for the design phase and not for the implementation phase, where such a type of language is not implemented by embedding. On the other hand, ‘homogeneously embedded domain-specific language’ is a domain-specific language designed by language exploitation using only an extension sub-pattern and implemented by embedding (using macros, function composition, and libraries that provide domain-specific constructs) without changes in host compiler/interpreter. Hence, homogeneously embedded domain-specific languages is a synonym for internal domain-specific languages, while heterogeneous embedding is only one particular approach that can be used for external domain-specific languages, which covers also those domain-specific languages designed by the invention design pattern (Figure 1).

There are several benefits of internal domain-specific languages. Because the syntax and semantics are borrowed from a host language, there is less need to define such an internal domain-specific language formally. Hence, much less effort is needed to develop an internal domain-specific language. Many language features can be implemented with relative ease due to the reuse from a host language. Inherently, designing such an internal domain-specific language is less error prone. In general, it can be said that internal domain specific languages are well designed by default. Internal domain-specific languages may also reuse other language-based tools (e.g., editor, debugger) and/or whole integrated de-
velopment environments of a host language. However, internal domain-specific languages may suffer from suitable domain notation and an inability to perform domain-specific analysis, verification, optimization, parallelization, and transformations (AVOPT). Appropriate or established domain-specific notations are usually beyond the limited user-definable notation offered by general-purpose languages. The importance of suitable notations should not be underestimated as they are directly related to the productivity improvement associated with the use of domain-specific languages. The degree to which internal domain-specific language can establish appropriate domain-specific notation depends on the flexibility of a host language. In this respect, general-purpose languages offer different possibilities.

The first four chapters of this book investigate appropriateness of different general-purpose programming languages (Lisp in Chapter 1, C++ in Chapter 2, Haskell in Chapter 3, and Scala in Chapter 4) as host languages for domain-specific notations. An additional goal of Chapter 3 is to show semantic-driven design where a language developer first concentrates on semantic domains and their operations, while an appropriate domain notation comes afterward. Appropriate or established domain-specific notation for anything more than a trivial domain-specific language is usually hard to achieve using the internal approach. One of the objectives of Chapter 4 is to experiment with a non-trivial domain-specific language to achieve an established notation using an internal domain-specific language. An appropriate or established domain-specific notation is much easier, and can often be achieved using an external domain-specific language because different syntax analyzers can be applied (e.g., generalized LR parser, backtracking parsers). The decision on whether to choose an internal or external domain-specific language has profound consequences on later phases. The final decision depends on many factors, and some are discussed in Chapter 5. The discussion on internal domain-specific languages concludes in Chapters 6 and 7, which discuss iterative development of internal domain-specific languages, and design principles/patterns for internal domain-specific languages. Despite their limitations, internal domain-specific languages are formidable competitors to external domain-specific languages. Even with improved domain-specific toolkits, it will remain the most cost-effective solution in many cases. Section 1 (Chapters 1 -7) of this book is unique in covering the details of internal domain-specific languages. Hopefully, the chapters in this book will become a standard source of knowledge for developing internal domain-specific languages.

The semantics of domain-specific languages are discussed in Section 2 (Chapters 8 – 10). Although formalizing domain-specific language semantics is often not needed for internal domain-specific languages, it is crucial for external domain-specific languages. With a formalization of a language, the meaning of domain-specific language constructs can be defined unambiguously. Also, other language-based tools can
be constructed automatically from formal specifications (as with general-purpose languages). Although these benefits occur for grammar-based domain-specific languages where different semantic formalisms have been extensively used (e.g., attribute grammars, denotational semantics, operational semantics), the opposite is true for metamodel-based domain-specific languages, where syntax is formally specified, not by grammars, but metamodels. Chapters 8 and 9 discuss different aspects of formally defining the semantics of metamodel-based domain-specific languages. In Chapter 10, the formal semantics of Kermeta, a popular framework for metamodel-based domain-specific languages, is given. By providing such formal semantics, any domain-specific language developed with Kermeta is formally specified.

Domain-specific language tools and processes are discussed in Section 3 (Chapters 11 – 15). While using a dedicated domain-specific language, the tools for building domain-specific language translators are indispensable and expedite the domain-specific language development. However, there are very few such tools. To be productive, programmers need a domain-specific language translator and also other development tools (e.g., debugger, simulator, analyzer). Hence, Chapter 11 discusses domain-specific language development with the XMF notation in the XModeler tool and compares the tool with some other such available tools. The topic of constructing and using a particular domain-specific language debugger is the focus of Chapter 12. The development of a domain-specific language can be expedited by inferring the domain-specific notation, as presented in Chapter 13. Finally, after development, a newly developed domain-specific language should go through the process of verification and validation. Two key questions are related to this activity: Did we develop a domain-specific language that can solve the problems efficiently for which it was designed? Is the newly developed domain-specific language using a natural notation that is easy to use by end-users? Such questions should be answered during an evaluation of domain-specific languages. This topic is discussed in Chapter 14. Building a domain-specific language is generally not the only goal. A domain-specific language often must be integrated into an existing software engineering process through customization. This introduces several challenges, which are discussed in Chapter 15.

Finally, in Section 4 (Chapters 16 – 20) different examples of domain-specific languages are presented. In Chapter 16, domain-specific languages for spatial computing are discussed and systematically classified, and in Chapter 17 domain-specific languages for information system development are investigated. In Chapters 18 and 19, the domain-specific languages for high-level parallelization and for reconfigurable conveyor systems are presented, respectively. In Chapter 20, a domain-specific language for model-driven software evolution concludes the spectrum of different domain-specific languages.

There were 38 chapter proposals that were received in response to the call for chapters (CFC) issued in May 2011. Out of 38 two-page proposals, 30 were selected, and authors were invited to submit full chapters by October 2011. After the reviewing process, there were 20 chapters selected for inclusion into this book (thus, an overall 53% acceptance rate).

Marjan Mernik
University of Maribor, Slovenia & University of Alabama at Birmingham, USA
April 11, 2012