Preface

Christopher Alexander was the first to introduce patterns as a form of describing accumulated experiences in the field of architecture. He defines a *pattern* as a construct made of three parts: a *context*, a set of *forces* and a *solution* (Alexander, Ishikawa, & Silverstein, 1977). The *context* reflects the conditions under which the pattern holds (Alexander, 1979). The *forces* occur repeatedly in the context and represent the *problem(s)* faced (Alexander, 1979). The *solution* is a configuration that allows the forces to resolve themselves (i.e., balances the forces) (Alexander, 1979). Alexander's patterns comprise commonly encountered problems and their appropriate solutions for the making of successful towns and buildings in a western environment. Alexander called a set of correlated patterns a *pattern language*, because patterns form a vocabulary of concepts used in communications that take place between experts and novices.

Ward Cunningham and Kent Beck (1987) were inspired by Alexander's work and decided to adapt it to software development. Their first five patterns dealt with the design of user interfaces. This marked the birth of patterns in the software field. Nowadays, software patterns are so popular that they are being applied in virtually every aspect of computing. Moreover, the concept of patterns is being adapted to many other fields, such as management, education, and so forth.

There are many kinds of software patterns based on different categorizing criteria. If the focus is on the software design phase, patterns are classified according to their abstraction level into architectural patterns, Design patterns, and idioms. This book mainly focuses on Design patterns. As such, in this preface, the terms Design patterns and patterns can be used interchangeably.

A pattern can be defined as a description of a proven (successful or efficient) *solution* to a recurring *problem* within a *context*. The above definition keeps the essence of Alexander's original definition by mentioning the three pillars of a pattern (context, problem, and solution). Reusing patterns usually yields better quality software within a reduced time frame. As such, they are considered artifacts of software reusability.

Patterns are published mostly within collections or catalogs. However, the most influential publication of Design patterns is the catalog by the "*Gang of Four*" (GoF) (Gamma, Helm, Johnson, & Vlissides, 1995), which listed 23 patterns classified under three categories: creational, structural, and behavioral. All patterns recorded in the GoF catalog were described in great detail and share an identical format of presentation.

Patterns are used as a way of improving software design productivity and quality for the following reasons:

- Patterns capture previous design experiences, and make it available to other designers. Therefore, designers do not need to discover solutions for every problem from scratch.
- Patterns form a more flexible foundation for reuse, as they can be reused in many ways.
- Patterns serve as a communication medium among software designers.
- Patterns can be considered microarchitectures, from which bigger software architectures can be built.

Well-established engineering disciplines have handbooks that describe successful solutions to known problems. Though as a discipline, software engineering is a long way from that goal, patterns have been useful for software engineers to reuse successful solutions.

Currently most patterns are described using a combination of textual descriptions, object-oriented (OO) graphical notations such as unified modeling language (UML) (Rumbaugh, Jacobson, & Booch, 1998), and sample code fragments. The intention is to make them easy to read and use and to build a pattern vocabulary. However, informal descriptions give rise to ambiguity, and limit tool support and correct usage. Tool support can play a great role in automated pattern mining, detection of pattern variants, and code generation from pattern specification.

Hence, there is a need for a formal means of accurately describing patterns in order to achieve the following goals:

- Better understand patterns and their composition. This will help know when and how to use them properly in order to take full advantage of their inherent benefits.
- Resolve the following issues regarding relationships between patterns such as duplication, refinement, and disjunction. Resolving the above-mentioned questions will ease the process of pattern repository management.
- Allow the development of tool support in activities related to patterns.

Many formal approaches for pattern specification have been emerging as a means to cope with the inherent shortcomings of informal descriptions. Despite being based on different mathematical formalisms, they share the same goal, which is accurately describing patterns in order to allow rigorous reasoning about them, their instances, their relationships and their composition and facilitate tool support for their usage. It is important to note that formal approaches to Design pattern specifications are not intended to replace existing informal approaches, but to complement them.

Currently, there is no single avenue for authors actively involved in the field of formal specification of Design patterns to publish their work. There has been neither a dedicated conference nor a special journal issue that covers precisely this field. Since this book contains chapters describing different Design pattern formalization techniques, it will contribute to the state-of-the-art in the field and will be a one-stop for academicians, research scholars, students, and practitioners to learn about the details of each of the techniques.

The book is organized into XVI chapters. A brief description of each of the chapters follows. These were mainly taken from the abstracts of the chapters.

Chapter I describes Balanced pattern specification language (BPSL), a language intended to accurately describe patterns in order to allow rigorous reasoning about them. BPSL incorporates the formal specification of both structural and behavioral aspects of patterns. The structural aspect formalization is based on first-order logic (FOL), while the behavioral aspect formalization is based on temporal logic of actions (TLA). Moreover, BPSL can formalize pattern composition and instances of patterns (possible implementations of a given pattern).

Chapter II describes the Design pattern modeling language (DPML), a notation supporting the specification of Design pattern solutions and their instantiation into UML design models. DPML uses a simple set of visual abstractions and readily lends itself to tool support. DPML Design pattern solution specifications are used to construct visual, formal specifications of Design patterns. DPML instantiation diagrams are used to link a Design pattern solution specification to instances of a UML model, indicating the roles played by different UML elements in the generic Design pattern solution. A prototype tool is described, together with an evaluation of the language and tool.

Chapter III shows how formal specifications of GoF patterns, based on the rigorous approach to industrial software engineering (RAISE) language, have been helpful to develop tool support. Thus, the object-oriented design process is extended by the inclusion of pattern-based modeling and verification steps. The latter involving checking design correctness and appropriate pattern application through the use of a supporting tool, called DePMoVe (design and pattern modeling and verification).

Chapter IV describes an abstraction mechanism for collective behavior in reactive distributed systems. The mechanism allows the expression of recurring patterns of object interactions in a parametric form, and to formally verify temporal safety properties induced by applications of the patterns. The authors present the abstraction mechanism and compare it to Design patterns, an established software engineering concept. While there are some obvious similarities, because the common theme is abstraction of object interactions, there are important differences as well. Authors discuss how the emphasis on full formality affects what can be expressed and achieved in terms of patterns of object interactions. The approach is illustrated with the Observer and Memento patterns.

In **Chapter V**, authors have investigated several approaches to the formal specification of Design patterns. In particular, they have separated the structural and behavioral aspects of Design patterns and proposed specification methods based on first-order logic, temporal logic, temporal logic of action, process calculus, and Prolog. They also explore verification techniques based on theorem proving. The main objective of this chapter is to describe their investigations on formal specification techniques for Design patterns, and then demonstrate using these specifications as the methods of reasoning about Design pattern properties when they are used in software systems.

Chapter VI presents the Spine language as a way of representing Design patterns in a suitable manner for performing verification of a pattern's implementation in a particular source language. Spine is used by a proof engine called Hedgehog, which is used to verify whether a pattern is correctly implemented.

Chapter VII presents a viewpoint based on intent-oriented design (IOD) that yields simple formalisms and a conceptual basis for tools supporting design and implementation from an intent-oriented perspective. The system for pattern query and recognition (SPQR) is an automated framework for analysis of software systems in the small or the large, and detection of instances of known programming concepts in a flexible, yet formal, manner. These concepts, when combined in well-defined ways to form abstractions, as found in the Design patterns literature, lead to the automated detection of Design patterns directly from source code and other design artifacts. The chapter describes the three major portions of SPQR briefly, and uses it to facilitate a discussion of the underlying formalizations of Design patterns with a concrete example, from source code to completed results.

Chapter VIII describes techniques for the verification of refactorings or transformations which introduce Design patterns. The techniques use a semantics of object-oriented systems defined by the object calculus and the pattern transformations are proved to be refinements using this semantics.

Chapter IX describes a UML-based pattern specification language called role-based metamodeling language (RBML), which defines the solution domain of Design patterns in terms of roles at the metamodel level. The chapter discusses benefits of the RBML and presents notation for capturing various perspectives of pattern properties. The Observer, Interpreter, and Iterator patterns are used to describe RBML. Tool support for the RBML and the future trends in pattern specification are also discussed.

In **Chapter X**, the formal specification of a Design pattern is given as a class operator that transforms a design given as a set of classes into a new design that takes into account the description and properties of the Design pattern. The operator is specified in the Slam-Sl specification language, in terms of pre- and postconditions. Precondition collects properties required to apply the pattern and post-condition relates input classes and result classes encompassing most of the intent and consequences sections of the pattern.

Chapter XI describes a formal, logic-based language for representing pattern structure and an extension that can also represent other aspects of patterns, such as intent, applicability, and collaboration. This mathematical basis serves to eliminate ambiguities. The chapter explains the concepts underlying the languages and shows their utility by representing two classical patterns, some concurrent patterns and various aspects of a few other patterns.

Chapter XII introduces an approach to define Design patterns using Semantic Web technologies. For this purpose, a vocabulary based on the Web ontology language OWL is developed. Design patterns can be defined as RDF documents instantiating this vocabulary, and can be published as resources on standard Web servers. This facilitates the use of patterns as knowledge artefacts shared by the software engineering community. The instantiation of patterns in programs is discussed, and the design of a tool is presented that can x-ray programs for pattern instances based on their formal definitions.

Chapter XIII presents a novel approach allowing the precise specification of patterns as well as retaining the patterns' inherent flexibility. The chapter also discusses tools that can assist practitioners in determining whether the patterns used in designing their systems have been implemented correctly. Such tools are important also during system maintenance and evolution to ensure that the design integrity of a system is not compromised. The authors also show how their approach lends itself to the construction of such tools.

Chapter XIV introduces the user requirements notation (URN), and demonstrates how it can be used to formalize patterns in a way that enables rigorous trade-off analysis while maintaining the genericity of the solution description. URN combines a graphical goal language, which can be used to capture forces and reason about trade-offs, and a graphical scenario language, which can be used to describe behavioral solutions in an abstract manner. Although each language can be used in isolation in pattern descriptions (and have been in the literature), the focus of this chapter is on their combined use. It includes examples of formalizing Design patterns with URN together with a process for trade-off analysis.

Chapter XV describes an extended compiler that formalizes patterns, called pattern enforcing compiler (PEC). Developers use standard Java syntax to mark their classes as implementations of particular Design patterns. The compiler is then able to use reflection to check whether the classes do in fact adhere to the constraints of the patterns. The checking possible with our compiler starts with the obvious static adherence to constraints, such as method presence, visibility, and naming. However, PEC supports dynamic testing to check the runtime behavior of classes and code generation to assist in the implementation of complex patterns. The chapter gives examples of using the patterns supplied with PEC, and also examples of how to write your own patterns and have PEC enforce these.

Chapter XVI presents Class-Z, a formal language for modelling OO Design patterns. The chapter demonstrates the language's unique efficacy in producing precise, concise, scalable, generic, and appropriately abstract specifications modelling the GoF Design patterns. Mathematical logic is used as a main frame of reference: the language is defined as a subset of first-order predicate calculus and implementations (programs) are modelled as finite structures in model theory.

References

Alexander, C. (1979). *The timeless way of building*. Oxford University Press.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A pattern language: Towns, buildings, construction*. Oxford University Press.

Beck, K., & Cunningham, W. (1987). Using pattern languages for object-oriented programs (Tech. Rep. No. CR-87-43). Tektronix Inc, Computer Research Laboratory.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented systems*. Addison-Wesley Professional.

Rumbaugh, J., Jacobson, I., & Booch, G. (1998). *The unified modeling language reference manual*. Addison-Wesley Professional.