1.1 The Concept of a Software Pattern

Current interest in software patterns was originally inspired by the work of the British architect Christopher Alexander and his colleagues [AIS+77] [Ale79]. Alexander was the first to describe what he called a *pattern language*, which mapped various problems in building architecture to proven solutions. In Alexander's terms, a pattern is *'a three-part rule, which expresses a relation between a certain context, a problem, and a solution'* [Ale79].

Since the mid-1990s, pattern-based design has been adapted for use by the software development community. The resulting software patterns are literary forms that describe recurring designs used in software development. They have been used extensively in the development of object-oriented systems, and have been highly effective in capturing, transferring and applying design knowledge at different levels of software design [Ram98]. In general, patterns exist in any context in which there are design decisions to be made.

Software patterns focus on capturing and systematizing successful experience and techniques used in previous software development. They describe successful solutions to common software problems with the intention of creating handbooks of good design and programming practices for software development. Their long term goal is to gather design experience and techniques for software development. Even though much work remains before that goal is reached, two decades of applying pattern-oriented software architectures and techniques have shown that software patterns help developers reuse successful software practices [POSA1] [POSA2] [POSA4] [POSA5]. Moreover, they help developers to communicate their experience better, reason about what they do and why they do it.

Software patterns are found at every level of software development: from the programming language level (the 'language idioms') to entire software systems, known as 'architectural patterns'. They are also commonly used to describe software processes. Moreover, classic algorithms and data types can be considered as programming-language level pattern-like entities. In particular, software patterns are viewed as well-documented design descriptions for software design.

What is a Pattern?

Defining a software pattern is not easy. Inside the pattern community it is generally accepted that a pattern is *'a recurring solution to a standard problem'* [Cop94] [Gab96]. In a wider sense, a pattern is *'a way to capture and systemize proven practice in any discipline'* [AIS+77] [Ale79].

For our purposes we consider a software pattern as *a function-form relation that occurs in a context, where the function is described in problem domain terms as a group of unresolved trade-offs or forces, and the form is a structure described in solution domain terms that achieves a good and acceptable equilibrium among those forces*. This defini-

tion of a software pattern is consistent with the previous definitions and relates software patterns with software design.

In general, the concept of software patterns is not confined to a particular software domain. As software patterns express recurring designs, they can be used to document design decisions at any level in any software domain. This generality is particularly important for parallel software design: software patterns are useful in documenting the design decisions in any aspects of a complete parallel system: for example, to document hardware systems or subsystems, communication and synchronization mechanisms, partitioning and mapping policies and so on.

An Example: The Manager–Workers Pattern

To show how software patterns are applied to parallel programming, a well-known example is presented in this section: the Manager–Workers pattern. This is a simple and classical example, presented in many parallel programming books and publications [Hoa78] [And91] [FP92] [Fos94] [KSS96] [Har98] [And00] [Ort04].

The Manager–Workers organization is one of the simplest patterns for parallel programs. It is often used to solve problems in which a single algorithm is applied independently to many different pieces of data. A manager (usually associated with the main process of the parallel program) partitions work (commonly the pieces of data to process) among a set of workers. These are launched together and executed simultaneously, assigning each one a separate portion of work. The manager waits for all workers to complete their work, then continues. A diagram showing the structure of the Manager–Workers pattern is shown in Figure 1.1.

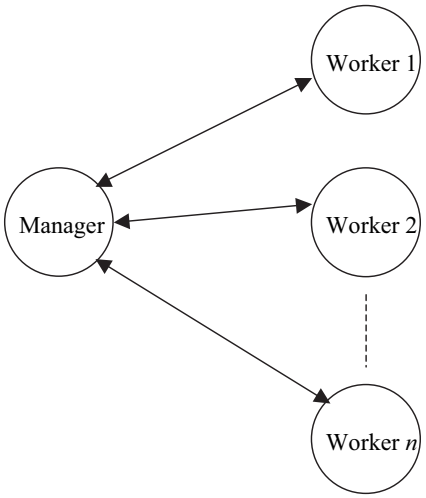


Figure 1.1: A Manager–Workers organization block diagram

The Manager–Workers pattern describes a simple kind of parallel execution, used when the amount of data on which to operate is known in advance and where it is easy to partition such data into roughly equal parts whose operation does not depend on each other. The absence of data dependencies is a key requirement that ensures no synchronization is required among the workers. A summary of the Manager–Workers pattern [Ort04] is shown in Figure 1.2.

Name:
Manager-Workers pattern.

Context:
Start the design of a parallel program, using a particular programming language for certain parallel hardware. Consider the following context constraints:

1. The problem involves tasks of a scale that would be unrealistic or not cost-effective for other systems to handle, and lends itself to be solved using parallelism.
2. The parallel hardware platform or machine to be used is given.
3. The main objective is to execute the tasks in the most time-efficient way.

Problem:
The same operation is required to be repeatedly performed on all the elements of some ordered data. Data can be operated without a specific order. However, an important feature is to preserve the order of data. If the operation is carried out serially, it should be executed as a sequence of serial jobs, applying the operation to each datum one after another. Generally, performance as execution time is the feature of interest, so the goal is to take advantage of the potential simultaneous execution in order to carry out the whole computation as efficiently as possible.

Forces:
The following forces should be considered:

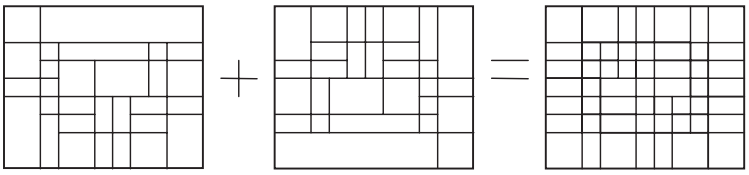
1. Preserve the order of data. However, the specific order of operation on each piece of data is not fixed.
2. The operation can be performed independently on different pieces of data.
3. Data pieces may exhibit different sizes.
4. The coordination of the independent computations has to take up a limited amount of time in order not to impede performance of other processing elements.
5. Mapping the processing elements to processors has to take into account the interconnection among the processors of the hardware platform.

Solution:
Introduce activity parallelism by having multiple data sets processed at the same time. The most flexible representation is the Manager-Workers approach. This structure is composed of a manager component and a group of identical worker components. The manager is responsible of preserving the order of data. On the other hand, each worker is capable of performing the same independent computation on different pieces of data. It repeatedly seeks a task to perform, performs it, and repeats; when no tasks remain, the program is finished.

Figure 1.2: A summary of the Manager–Workers pattern

To illustrate an application of the Manager–Workers pattern, we present a case study based on the Polygon Overlay problem [Ort04] [WL96]. The objective of this case study is to obtain the overlay of two rectangular maps, *A* and *B*, each covering the same area, which is decomposed into a set of non-overlapping rectangular polygons. This type of problem is common in geographical information systems, in which one map represents, for example, soil type, and another, vegetation. Their conjunction is thus an overlay that represents how combinations of soil type and vegetation are distributed. Overlaying both maps therefore creates a new map consisting of the non-empty polygons in the geometric intersection of *A* and *B*.

To simplify this problem for practical purposes, all polygons are considered as non-empty rectangles, with vertices on a rectangular integer grid $[0\ldots N]\square[0\ldots M]$ (Figure 1.3). Both input maps have identical extents, each completely covered by its rectangular decomposition.



A *B* *A* + *B*

Figure 1.3: The polygon overlay problem for two maps *A* and *B*

A sequential solution to this problem iterates through each polygon belonging to *A* and finds all intersections with any polygon in *B*. Although this is an effective solution, it can run slowly, depending on the number of polygons into which each map is divided. It is possible to obtain intersections in parallel, however, since the overlaying operation of two polygons can be performed potentially independently of the overlay of any other two polygons.

For experienced parallel programmers, developing a parallel solution for the Polygon Overlay problem is straightforward: simply link the concrete requirements of functionality of the problem with a concrete solution based on a parallel technology. Moreover, since experienced programmers understand typical structures of parallel programs, they would immediately recognize a solution to the problem based on the Manager–Workers pattern, as well as its partitioning policies, its communication and synchronization mechanisms, its mapping strategies and so on.

Nevertheless, consider novice parallel programmers, who might learn about the Manager–Workers pattern and parallel systems by reading the literature [And91] [Fos94] [Har98] [And00], but cannot adequately and efficiently exploit such knowledge to solve the Polygon Overlay problem. The main problem faced by novice parallel programmers is their lack of design experience, which could prevent them from linking the functional-

ity of the problem with a parallel programming solution. The typical effects of this lack of experience are design problems that might be detected late in subsequent development, for example in the form of poor performance or deadlocks during execution.

The main objective of this book is to show how a solid understanding of groups of software patterns for parallel programming during the design process can enable novice programmers to leverage the knowledge of experienced parallel programmers. Such novices must find pattern(s) that describe (or nearly describe) their problem, understand whether the forces match the constraints of such a problem, grasp the solution description(s) and map them to a design. During this process parallel programmers can start to formulate their own body of experience. Although this process may sound simple, we will show how it works for the Polygon Overlay problem using the Manager–Workers pattern as a design guide.

As described in the *Context* section of the Manager–Workers pattern (Figure 1.2), we are just about to start the design of a parallel program. In parallel programming, the programming language and the parallel hardware are typically given resources. Nevertheless, let us assume that the Polygon Overlay problem involves *tasks of a scale that would be unrealistic or not cost-effective for a sequential system to handle* (Figure 1.2). The solution to the Polygon Overlay problem thus lends itself to using parallelism, as explained later when describing the parallel solution.

Note also that the Polygon Overlay problem matches the Problem description provided by the pattern (Figure 1.2), since it involves only a single overlaying operation that is performed repeatedly on all the rectangles, which are ordered inside each map. The rectangles can be overlaid without a specific order. It is important to preserve the order of rectangles in the final result, however, so we need to keep track of which rectangle in *A* is overlaid with which rectangle in *B*. As mentioned earlier, if the overlaying is performed serially, it would be executed as a sequence of serial jobs, applying the same operation to each rectangle iteratively, which takes a long time to run. Nevertheless, we can take advantage of the independence between overlaying different sections of both maps, and hence perform the whole overlaying process as efficiently as possible.

Notice that most of the forces, as described in the Manager–Workers pattern (Figure 1.2) are present in the Polygon Overlay problem:

- The Polygon Overlay problem requires that its solution preserves the order of rectangles from maps *A* and *B*. Nevertheless, notice that all pairs of rectangles, one from *A* and one from *B*, can be overlaid without a specific order among them.
- The overlaying can be performed independently between any rectangle from *A* and any rectangle from *B*.
- Although rectangles have different sizes, the overlaying operation requires a representation of the rectangles (normally, their coordinates within the map).
- The Manager–Workers organization ensures that adding a new worker does not affect the rest of the workers, but it can influence the total execution time of the parallel program.

Considering the previous analysis of the context, problem, and forces for the Polygon Overlay problem, our conclusion is to use the Manager–Workers pattern to create a parallel solution. Such a parallel solution can be described as follows (Figure 1.2): using the Manager–Workers pattern, a set of workers do the actual polygon overlaying by simultaneously finding intersections for each sub-map in *A* with each sub-map in *B*. For the two input maps, the manager divides all the polygons belonging to *A* into sub-maps, and for each of them the workers find all the intersections with a sub-map of *B* (Figure 1.4). The key for the parallel solution is to limit the part of both maps, *A* and *B*, that workers must examine to find the overlaps. The manager is responsible for tracking which sub-map is sent to which worker so that each overlaying is performed in the right order. At the end of the whole process, each worker returns its result map to the manager, which assembles them into a complete result map.

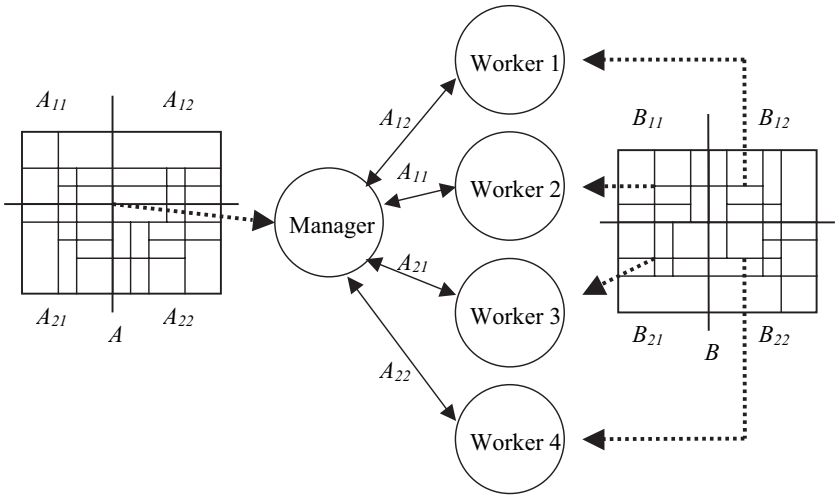


Figure 1.4: A Manager–Workers block diagram for solving the Polygon Overlay problem

The solution to the Polygon Overlay problem using the Manager–Workers pattern can be developed further to obtain a complete parallel program. Nevertheless, our objective with this case study is simply to show how a software pattern can be used to design a solution from a problem description, so we stop here. Several questions, however, arise from this example, such as 'Why use the Manager–Workers pattern to solve the Polygon Overlay problem?', 'Why not use another pattern?', 'What are the characteristics and features of this problem that lead us to select Manager–Workers pattern as a description of the coordination of its solution?'. The rest of this book attempts to provide answers to questions like these; first, however, the following sections address other issues about software patterns.

1.2 Pattern Description, Organization and Categorization

Describing Patterns: The POSA Form

Software patterns are usually documented in several forms. These forms are known as *pattern schemata*, *pattern forms* or *pattern templates*. Numerous examples of these templates can be found in the literature [GHJV95] [POSA1] [PLoP1]. The typical form is a collection of sections that characterize different aspects of a software pattern. The collection of sections varies from author to author and from domain to domain.

In parallel programming, as in other software domains, the most common forms are the 'Gang of Four' (GoF) form [GHJV95] and the 'Pattern-Oriented Software Architecture' (POSA) form [POSA1]. Both forms use diagrams based on Unified Modeling Language (UML) and plain text. This book uses the POSA form to describe software patterns. This form uses the following sections [POSA1]:

- *Name*. A word or phrase that essentially describes the pattern.
- *Brief*. A description of the pattern stating what it does.
- *Example*. A real-world example that shows the existence of a problem and the need for the pattern.
- *Context*. The situation or circumstance in which the pattern is applied.
- *Problem*. A description of the conflict the pattern solves, including a discussion about the forces.
- *Solution*. A description of the fundamental principle of the solution which serves as base for the pattern.
- *Structure*. A detailed specification (usually based on UML diagrams) describing structural aspects of the pattern.
- *Dynamics*. Typical scenarios describing the behavior through time of the participants within the pattern. Normally UML sequence diagrams are used.
- *Implementation*. Guidelines for implementing the pattern.
- *Example resolved*. Restating the Example section, this section presents a discussion about any important aspects of solving the problem proposed as the example.
- *Known uses*. Example uses of the pattern (at least three) taken from existing systems.
- *Consequences*. Benefits and liabilities that occur when applying the pattern.
- *See also*. References to other patterns that solve similar problems, or to patterns that help to refine the pattern being defined.

Pattern Languages and Systems: Organizing Patterns

In general – and independently of the domain – patterns are distilled from successful designs, which means that the main source of patterns is the analysis of existing successful solutions, identifying their recurring forms and designs. This discovery and documentation of patterns produces a large number of them: every day someone somewhere discovers a pattern and works on documenting it. Nevertheless, patterns are only useful if they can be organized in a way that makes them easy to select and use. Normal practice is to gather related patterns into structured pattern collections [POSA5].

When the pattern organization process advances, it often yields a network of relations between patterns known as a *pattern language* or *pattern system*. These networks are collections of interrelated patterns that can be used to describe or design a concrete system in a domain [PLoP1]. The term 'pattern language' was originally suggested by Alexander et al. [AIS+77]: the term 'pattern system' was proposed later by Buschmann et al. [POSA1].

A pattern language or system is a set of patterns complete enough for design within a domain. It is a method for composing patterns to synthesize solutions to diverse objectives [POSA1]. Hence software patterns become the building blocks for design, or suggest important elements that should be presented in the software system. Each software pattern suggests instructions for solution structure or contains a solution fragment. The fragments and instructions are merged to yield a system design.

Software Pattern Categories

Software patterns cover various levels of scale and abstraction. They range from those that help in structuring a software system into subsystems, through those that support the refinement of subsystems and components, to those that are used to implementing particular design aspects in a specific programming language. Based on a description such as this, software patterns are commonly grouped into three categories, each one consisting of patterns having a similar level of scale or abstraction [POSA1]:

- Architectural patterns. *'An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationship between them'.*
- Design patterns. *'A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context'.*
- Idioms. *'An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language'.*

In this book we are concerned about architectural patterns as high-level software patterns used for specifying the coordination of parallel software systems, about design patterns as refinement schemes for inter-component communication, and about idioms as low-level patterns used for describing synchronization mechanisms in different languages.

1.3 Summary

This chapter has briefly introduced the reader to the field of software patterns. Addressing issues such as the concept of a software pattern, its description, organization and categorization, this chapter has provided a simple introduction intended to help clarify the remaining patterns for parallel software design presented in this book.