# Design Patterns and Algorithmic Skeletons: A Brief Concordance

Adriana E. Chis and Horacio González–Vélez

**Abstract** Having been designed as abstractions of common themes in object-oriented programming, patterns have been incorporated into parallel programming to allow an application programmer the freedom to generate parallel codes by parameterising a framework and adding the sequential parts. On the one hand, parallel programming patterns and their derived languages have maintained, arguably, the best adoption rate; however, they have become conglomerates of generic attributes for specific purposes, oriented towards code generation rather than the abstraction of structural attributes. On the other hand, algorithmic skeletons systematically abstract commonly-used structures of parallel computation, communication, and interaction. Although there are significant examples of relevant applications—mostly in academia—where they have been successfully deployed in an elegant manner, algorithmic skeletons have not been widely adopted as patterns have. However, the ICT industry expects graduates to be able to easily adapt to its best practices. Arguably, this entails the use of pattern-based programming, as it has been the case in sequential programming where the use of design patterns is widely considered the norm, as demonstrated by a myriad of citations to the seminal work of Gamma et al. [6] widely known as the Gang-of-Four. We contend that an algorithmic skeleton can be treated as a structural design pattern where the degree of parallelism and computational infrastructure are only defined at runtime. The purpose of this chapter is to systematically explain how design patterns can be efficiently mapped into algorithmic skeletons and consequently benefit application programmers. We illustrate our approach using a visitor design pattern and a task farm algorithmic skeleton.

A. E. Chis · H. González–Vélez
Cloud Competency Centre, National College of Ireland, Dublin 1, Ireland e-mail: `{adriana.chis,horacio}@ncirl.ie`

## 1 Introduction

Parallel programming aims to capitalise on concurrency, the execution of different sections of a given program at the same time, in order to improve the overall performance of the program, and, eventually, that of the whole system. Despite major breakthroughs, parallel programming is still a highly demanding activity widely acknowledged to be more difficult than its sequential counterpart, and one for which the use of efficient programming models and structures has long been sought. These programming models must necessarily be performance-oriented, and are expected to be defined in a scalable structured fashion to provide guidance on the execution of their jobs and assist in the deployment of heterogeneous resources and policies.

Furthermore, it is widely acknowledged that one of the major challenges of the multi/many-core era is the efficient support of parallel programming models that can predict and improve performance for diverse heterogenous architectures [9]. Furthermore, the "Berkeley View" work established the importance of not only producing realistic benchmarks for parallel programming models based on patterns of computation and communication, but also developing programming paradigms which efficiently deploy scalable task parallelism [2]. Such decoupling has allowed them to be efficiently deployed on different dedicated and non-dedicated architectures including symmetric multiprocessing, massively parallel processing, clusters, constellations, and clouds.

*Design patterns* have been conceived as abstractions of common themes in object-oriented programming [5, 6]. *Parallel patterns* aim to further expand this concept by decoupling the detail or implementation from the structure of a parallel program in order to transfer any performance improvements in the system infrastructure while preserving the final result.

*Algorithmic skeletons* abstract commonly-used patterns of parallel computation, communication and interaction [3]. Skeletons provide a clear and consistent behaviour across platforms, with the underlying structure depending on the particular implementation.

While diverse authors have established the importance of patterns and skeletons in parallel programming from a design point of view [8, 10], a significant programmer-oriented approach should arguably benefit applicative environments and development projects.

We propose an initial direct mapping of design patterns and skeletons in order to have "marry" the well-known, accepted design pattern approach with the programmer-oriented functional algorithmic skeleton paradigm.

This chapter is structured as follows. Firstly, section 2 provides a brief introduction to design patterns. Secondly, section 3 describes the algorithmic skeleton paradigm. Thirdly, section 4 describes our methodological mapping of a simple design pattern to an algorithmic skeleton. Finally, section 5 presents our conclusions.

## 2 Design Patterns

Computers have been traditionally programmed with a sequential frame of mind, but parallel solutions require a different way of approaching and dissecting a problem. They require a holistic analysis and understanding of the system architecture, the programming paradigm, and the problem constraints. Parallel computing requires calculations to be synchronised, staged, and/or communicated over a number of different phases. Message-passing, threads, load-balancing, and semaphores are matters restricted to the expert software developers and, arguably, lack some high-level design features required for large-scale software development endeavours.

Having defined a *pattern* as a core solution to a problem that recurrently occurs in a given context, Alexander et al. introduced a pattern language to describe tens of patterns applied in civil engineering [1]. Subsequently, design patterns have documented solutions to recurrent software design problems. Gamma et al. present 23 design patterns [6].

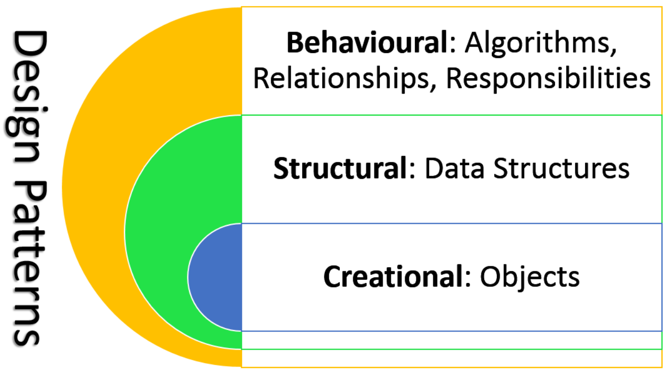
Design Patterns
Behavioural: Algorithms, Relationships, Responsibilities
Structural: Data Structures
Creational: Objects

**Fig. 1** Traditional Classification of Design Patterns: Creational, Structural and Behavioural.

The authors classify the design patterns based on their purpose into three main categories, namely *creational patterns*, *structural patterns* and *behavioural patterns* as illustrated in Figure 1.

Creational patterns Used to build objects such that they can be decoupled from the implementing system.
Structural patterns Used to form large data structures from many disparate objects.
Behavioural patterns Used to manage algorithms, relationships, and responsibilities between objects.

Furthermore, Gamma at el. provide another classification of the design patterns based on the patterns' scope in *object patterns* and *class patterns*. As the name suggests the former category of patterns specify relationships between objects, whereas the latter category of patterns encodes relationships between classes and subclasses.

A complete description of these design patterns can be found in the seminal book by Gamma et al. [6]. The authors document each pattern using a template. The core elements of describing a pattern are: the pattern name; the problem, which presents details about a problem in a given context; the solution in form of a generic design solution which incorporates the relationships and interactions between objects and classes; and the consequences of using a given pattern. A number of core design patterns are presented in Table 1.

**Table 1** Examples of Design Patterns (Creational, Structural, and Behavioural) [6]

| *Category* | *Pattern* |
|---|---|
| Creational | Abstract Factory, Builder, Factory Method, Prototype, Singleton |
| Structural | Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy |
| Behavioural | Chain of Responsibility, Command, Interpreter, Iterator, Mediator Memento, Observer, State, Strategy, Template Method, Visitor |

## 3 Algorithmic Skeletons

Cole pioneered the field with the definition of skeletons as "specialised higher-order functions from which one must be selected as the outermost purpose in the program", and the introduction of four initial skeletons: divide and conquer, iterative combination, cluster, and task queue [3]. His work described a software engineering approach to high-level parallel programming using a skeletal (virtual) machine rather than the deployment of a tool or language on a certain architecture.

In essence, algorithmic skeletons systematically abstract commonly-used structures of parallel computation, communication, and interaction. Skeletal parallel programs are typically expressed by interweaving parameterised skeletons using descending composition and control inheritance throughout the program structure, analogously to the way in which sequential structured programs are constructed [4]. This high-level parallel programming technique, known as *structured parallelism*, enables the composition of skeletons for the development of programs where the control is inherited through the structure, and the programmer adheres to top-down design and construction. Thus, it provides a clear and consistent behaviour across platforms, while their structure depends on the particular implementation.

Since skeletons enable programmers to code algorithms without specifying the machine-dependent computation and coordination primitives, they have been positioned as coordination enablers in parallel programs. A complete survey on algorithmic skeletons and their frameworks can be found in [7].

Despite its elegance and potential, it is important to state that structured parallelism still lacks the necessary critical mass to become a mainstream parallel programming technique. Its principal shortcomings are its application space, since it can only address well-defined algorithmic solutions, and the lack of a specification to define and exchange skeletons between different implementations.

Skeletons can be broadly categorised into three types based on their functionality as shown in table 2.

Data-parallel skeletons Work typically on bulk data structures. Their behaviour establishes functional correspondences between data, and their structure regulates resource layout at fine-grain parallelism, e.g. MPI collectives.

Task-parallel skeletons Operate on tasks. Their behaviour is determined by the interaction between tasks, and their coarse-grain structure establishes scheduling constraints among processes, e.g., task farm and pipeline.

Resolution skeletons Delineate an algorithmic method to undertake a given family of problems. Their behaviour reflects the nature of the solution to a family of problems, and their structure may encompass different computation, communication, and control primitives, e.g, the divide-and-conquer and dynamic programming skeletons.

From a coordination point of view, data-parallel skeletons are typically input/output intensive as they operate on memory, and even disk, stored data structures, while resolution are computational-intensive as they deploy complex algorithms with demanding computational requirements. Task parallel can be construed as task schedulers with intimate knowledge of the program structure.

**Table 2** A taxonomy for the algorithmic skeleton constructs based on their functionality

| *Skeleton* | *Scope* | *Main Coordination Characteristic* | *Examples* |
|---|---|---|---|
| Data-Parallel | Data structures | I/O intensive | map, reduce |
| Task-Parallel | Tasks | Scheduling | task farm, pipeline |
| Resolution | Family of problems | Computational-intensive | divide-and-conquer, branch-and-bound |

### 3.1 A Classification for Algorithmic Skeletons

This section elaborates on the functionality associated with the specific algorithmic skeletons listed in Table 2.

- Data-parallel (see Fig. 2)

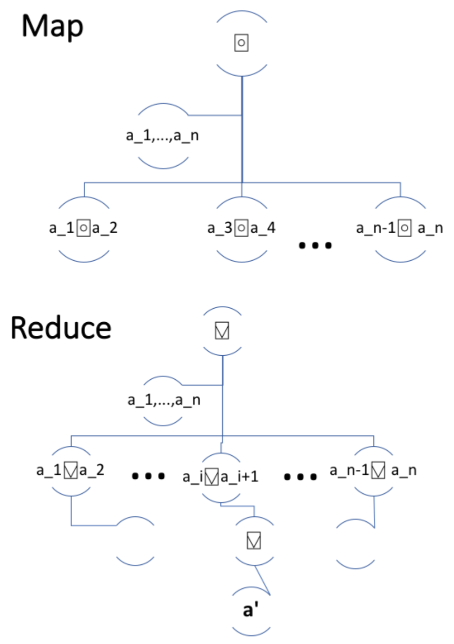

**Fig. 2** Two Data Parallel Skeletons: Map and Reduce.

  – *Map* specifies that a function or a sub-skeleton can be applied simultaneously to all the elements of a list to achieve parallelism. The data parallelism occurs because a single data element can be split into multiple data, then the sub-skeleton is executed on each data element, and finally the results are united again into a single result. The *map* skeleton can be conceived as single instruction, multiple data parallelism.
  – *Reduce*, also known as *scan*, is employed to compute prefix operations in a list by traversing the list from left to right and then applying a function to each pair of elements, typically summation. Note that as opposed to *map*, it maintains aggregated partial results.

- Task-parallel (see Fig. 3)

  – *Task Farm* or simply *farm* embeds the ability to schedule independent tasks in a divisible workload across multiple computing nodes.

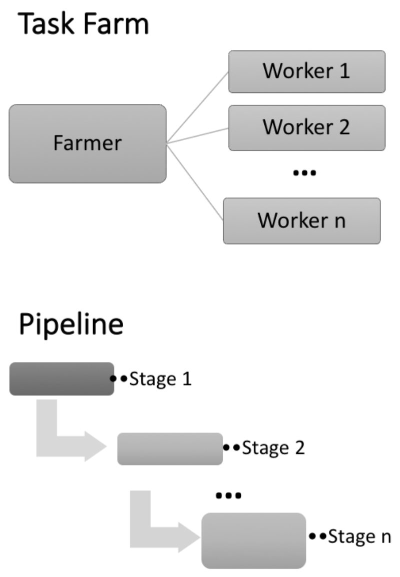

**Fig. 3** Two Task Paralel Skeletons: Task Farm and Pipeline.

  – *Pipe* enables staged computations, where parallelism can be achieved by computing different stages simultaneously on different inputs. The number of stages provided by *pipe* can be variable or fixed.

- Resolution (see Fig. 4)

  – *Divide & Conquer* (d&c) calls are recursively applied until a condition is met within a optimisation search space. Its semantics are as follows. When an input arrives, a condition component is invoked on the input. Depending on the result two things can happen. Either the parameter is passed on to the sub-skeleton, or the input is split with the split component into a list of data. Then, for each list element the same process is applied recursively. When no further recursions are performed, the results obtained at each level are merged. Eventually, the merged results yield to one result which corresponds to the final result of the d&c skeleton.

  – *Branch & Bound* (b&b) divides recursively the search space (branch) and then determines the elements in the resulting sub-spaces by mapping an objective function (bound). The merged results also produce one result which corresponds to the final result of the b&b skeleton.

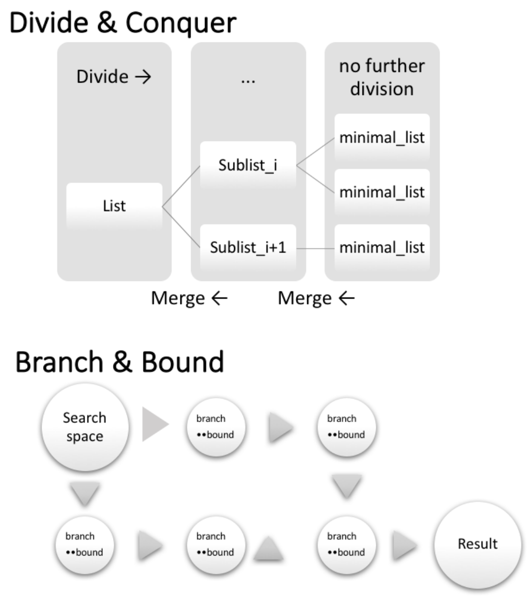

**Fig. 4** Two Resolution Skeletons: Divide & Conquer and Branch & Bound.

## 4 Mapping Patterns and Skeletons

In this section we show how the Visitor pattern, a behavioural design pattern, can be mapped to the Task Farm skeleton by documenting the latter pattern using the design pattern template and identifying the commonalities between the two.

Design patterns comprise intent, motivation, participants, and collaborations, and consequences. Given that structural design patterns have been conceived to create added functionality via object augmentation, they can be “made” parallel. That is to say, a standard compound structural pattern can have parallel characteristics which can be instantiated dynamically. On the other hand, algorithmic skeletons operate on the notion of changing underlying computational resources and therefore detach the structure from the behaviour of the program.

We contend that an algorithmic skeleton can be treated as a structural design pattern where the degree of parallelism and computational infrastructure are only defined at runtime. From this perspective, the programmer task is arguably simplified by completely detaching the structure and behaviour as originally intended, and additionally increasing its consistency and programmability through the design pattern characteristics.

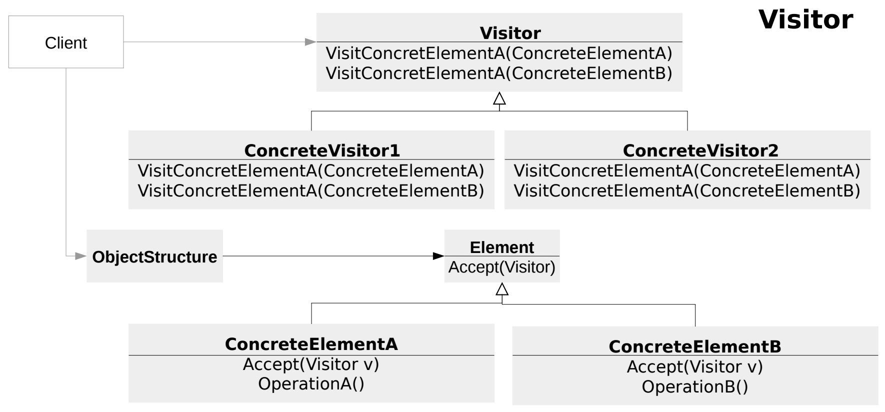
Visitor
Client
Visitor
VisitConcretElementA(ConcreteElementA)
VisitConcretElementA(ConcreteElementB)
ConcreteVisitor1
VisitConcretElementA(ConcreteElementA)
VisitConcretElementA(ConcreteElementB)
ConcreteVisitor2
VisitConcretElementA(ConcreteElementA)
VisitConcretElementA(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()

**Fig. 5** Visitor Pattern (source: [6])

## *Visitor Pattern – a Behavioural Pattern*

As introduced by Gamma et al. [6], the *Visitor* pattern separates structure from computation by enabling new operations on existing object structures without modifying the structures. Fig. 5 presents the generic solution of the Visitor pattern. The following description is adapted from [6]:

Intent Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Motivation/Description Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Applicability The visitor pattern is useful in the following scenarios:

- An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
- The classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If

the object structure classes change often, then it's probably better to define the operations in those classes.

Participants (consult the pattern structure shown in Fig. 5)

Visitor declares a visit method for each of the concrete elements that need to be traversed.
ConcreteVisitor implements each visit method declared in the Visitor. Usually, each ConcreteVisitor keeps track of a local state for the visited concrete element. The state is going to be updated while recursively traversing the structure.
Element declares an accept method which allows passing in a Visitor as a parameter
ConcreteElement declares an accept method which allows passing in a Visitor as a parameter
ObjectStructure offer a mechanism to allow a visitor to visit the elements

Collaborations

- A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.
- When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.

## *Task Farm: Compound Structural*

Let us present an example based on a task parallel algorithmic skeleton, the task farm, as introduced in Section 3. We shall therefore formalise its description by using the notation for design patterns.

Intent A Task Farm enables the creation of a variable number of independent tasks to be allocated to distinct computational "worker" nodes by a central scheduling node "farmer". Farms can be nested recursively to enable a worker to become a farmer of additional nodes.
Motivation Farms are especially useful to offload large numbers of independent tasks to several nodes. Typically there are many more tasks than nodes. As nodes can have different architectures (e.g. based on CPUs or GPUs) and, consequently, distinct computational characteristics, the farmer requires to allocate tasks using greedy or other heuristics scheduling mechanisms. Furthermore, computational resources may not necessarily be dedicated, can be geographically distributed, and have variable latencies, making the overall scheduling dynamic and complex.
Applicability Farms are particularly useful to offload embarrassingly-parallel computations where the ordering and finish times of independent tasks are not subject to hard constraints.

Participants

| | |
|---|---|
| Farmer | the process which divides and allocates tasks to workers. |
| Worker | the processes which receive tasks and compute results based on given function. |

Collaborations

- A client that uses the Task Farm skeleton must create a Farmer object to create an object structure for Workers. By traversing the object structure for each Worker element, a Farmer assigns tasks to each Worker.
- When an element (Worker) completes a task (or a series of them), it calls the Farmer operation that corresponds to its class.

We notice that if we perform a pairwise comparison that the visitor design pattern and the task farm skeleton can be similar in nature. For instance, the intent of both is to perform a series of tasks without altering the nature of the structure and both have an architecture-independent approach. While the Task Farm deals with processing heterogeneity by using scheduling mechanisms, the Visitor pattern makes no assumption on the nature of the underlying infrastructure.

## 5 Conclusions

This initial mapping approach of patterns to skeletons has shown that, in principle, parallel programming structures can be formally documented using a design pattern notation to strengthen its nature and, most certainly, its readability .

With respect to the analysis of the mapping problem, the findings of this work provide an initial idea to document large parallel programming endeavours. This tacitly reinforces the notion that although parallel programming is complex, well-documented parallel patterns can help to ease the burden.

From a performance standpoint, it is arguable that the overall performance of algorithmic skeletons can be assumed to be unaltered as the design pattern notation is mostly static. Furthermore, by assuming a skeleton is a pattern whose degree of parallelism is determined at run-time, there is an intrinsic reinforcement to the decoupling of computation from coordination.

However, it is a fact there are substantial avenues of research that need to be explored to fully formalise a design pattern approach to skeletons.

## *Acknowledgements*

The authors would like to acknowledge the contribution of the ICT COST Action IC1406 “High-Performance Modelling and Simulation for Big Data Applications (cHiPSet)” `http://chipset-cost.eu/`.

## References

1. Alexander, C., Ishikawa, S., Silverstein, M.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press, New York (1977)
2. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
3. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing. MIT Press/Pitman, London (1989)
4. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Computing **30**(3), 389–406 (2004)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Abstraction and reuse of object-oriented design. In: ECOOP'93, *Lecture Notes in Computer Science*, vol. 707, pp. 406–431. Springer, Kaiserslautern (1993)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman, Boston (1995)
7. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software: Practice and Experience **40**(12), 1135–1160 (2010). DOI 10.1002/spe.1026
8. Mattson, T.G., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Software Patterns Series. Addison-Wesley Professional, Boston (2004)
9. Mittal, S., Vetter, J.S.: A survey of cpu-gpu heterogeneous computing techniques. ACM Comput. Surv. **47**(4), 69:1–69:35 (2015)
10. Rabhi, F.A., Gorlatch, S. (eds.): Patterns and Skeletons for Parallel and Distributed Computing. Springer, London (2003)