Software Design Pattern Using : Algorithmic Skeleton Approach

S. Sarika

Lecturer,Department of Computer Sciene and Engineering
Sathyabama University, Chennai-119.
sarish_sar1@yahoo.co.in

***Abstract* - In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Many patterns imply object-orientation or more generally mutable state, and so may not be as applicable in functional programming languages, in which data is immutable or treated as such..Not all software patterns are design patterns. For instance, algorithms solve computational problems rather than software design problems.In this paper we try to focous the algorithimic pattern for good software design.**

***Key words* - Software Design, Design Patterns, Software esuability,Alogrithmic pattern.**

1. INTRODUCTION

Many studies in the literature (including some by these authors) have for premise that design patterns improve the quality of object-oriented software systems, because design patterns are supposed to improve the quality of systems[1], for example, a tangled implementation of patterns impacts negatively quality Also, patterns generally ease future enhancement at the expense of simplicity. There is little empirical evidence to support the claims of improved reusability, expandability and understandability as put forward in when applying design patterns. We present detailed results for three design pat-terns: Abstract Factory, Composite, Flyweight and three quality attributes: reusability, understandability, and expandability.

The following set of quality attributes, based on their relevance to design patterns[2].

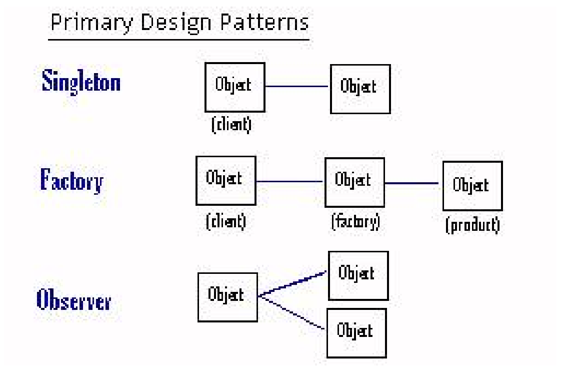

Fig 1. Primary design pattern

1.1Attributes related to design :

Expandability : The degree to which the design of a system can be extended.

Simplicity : The degree to which the design of a system can be understood easily.

Reusability : The degree to which a piece of design can be reused in another design.

1.2 Attributes related to implementation:

Learn ability : The degree to which the code source of a system is easy to learn.

Understandability : The degree to which the code source can be understood easily.

Modularity : The degree to which the implementation of the functions of a system are independent from one another.

1.3 Attributes related to runtime :

Generality: The degree to which a system provides a wide range of functions at runtime.

Modularity at runtime: The degree to which the functions of a system are independent from one another at runtime.

2. ALGORITHMIC SKELETON

In computing, **algorithmic skeletons** (a.k.a. Parallelism Patterns) are a high-level parallel programming model for parallel [3] [4] and distributed computing. Algorithmic skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by combining the basic ones.This section describes some well known Algorithmic Skeleton patterns. Additionally, the patterns signature in the Skandium library is provided for clarity.

FARM is also known as **master-slave**. Farm represents task replication where the execution of different tasks by the same farm are replicated and executed in parallel.

```
Farm(Skeleton<P,R> keleton){...}
```

PIPE represents staged computation. Different tasks can be computed simultaneously on different pipe stages. A pipe can have a variable number of stages, each stage of a pipe may be a nested skeleton pattern. Note that an n-stage pipe can be composed by nesting n-1 2-stage pipes.

```
<X> Pipe(Skeleton<P,X> stage1, Skeleton<P,X> stage2){...}
```

FOR represents fixed iteration, where a task is executed a fixed number of times. In some implementations the executions may take place in parallel.[5]

For(Skeleton<P,X> skeleton, int times){...}

WHILE represents conditional iteration, where a given skeleton is executed until a condition is met.

```
public While(Skeleton<P,P> skeleton, Condition<P> condition){...}
```

IF represents conditional branching, where the execution choice between two skeleton patterns is decided by a condition.

```
If(Condition<P> condition, Skeleton<P,R> trueCase, Skeleton<P,R> falseCase){...}
```

MAP represents *split*, *execute*, *merge* computation. A task is divided into sub-tasks, sub-tasks are executed in parallel according to a given skeleton, and finally sub-task's results are merged to produce the original task's result[6].

```
<X,Y> Map(Split<P,X> split, Skeleton<X,Y> skeleton, Merge<Y,R> merge){...}
```

D&C represents divide and conquer parallelism. A task is recursively sub-divided until a condition is met, then the sub-task is executed and results are merged while the recursion is unwound.

```
DaC(Condition<P> condition, Split<P,P> split, Skeleton<P,R> skeleton, Merge<R,R> merge){...}
```

SEQ does not represent parallelism, but it is often used a convenient tool to wrap code as the leafs of the skeleton nesting.

```
public Seq(Execute <P,R> execute){...}
```

3. EXAMPLE PROGRAM

The following example is based on the Java Skandium library for parallel programming.The objective is to implement an Algorithmic Skeleton based parallel version of the **QuickSort** algorithm using the Divide and Conquer pattern. Notice that the high-level approach hides Thread management from the programmer.

```
// 1. Define the skeleton program
Skeleton<Range, Range> sort = new DaC<Range, Range>(
new ShouldSplit(threshold, maxTimes),
    new SplitList(),
    new Sort(),
    new MergeList());
// 2. Input parameters
Future<Range> future = sort.input(new Range(generate(...)));
// 3. Do something else here.
// ...
// 4. Block for the results
Range result = future.get();
```

The first thing is to define a new instance of the skeleton with the functional code that fills the pattern (ShouldSplit, SplitList, Sort, MergeList). The functional code is written by the programmer without parallelism concerns.

The second step is the input of data which triggers the computation. In this case Range is a class holding an array and two indexes which allow the representation of a subarray. For every data entered into the framework a new Future object is created. More than one Future can be entered into a skeleton simultaneously.

The Future allows for asynchronous computation, as other tasks can be performed while the results are computed. The functional codes in this example correspond to four types Condition, Split, Execute, and Merge.

```
public class ShouldSplit implements Condition<Range>{
   int threshold, maxTimes, times;
   public ShouldSplit(int threshold, int maxTimes){
    this.threshold = threshold;
    this.maxTimes  = maxTimes;
    this.times     = 0;
  }
   @Override
  public     synchronized     boolean
condition(Range r){
    return   r.right   -   r.left   >
threshold &&
           times++ < this.maxTimes;
  }
}
```

4. ALGORITHMIC EFFICIENCY

In computer science, **efficiency** is used to describe properties of an algorithm relating to how much of various types of resources it consumes[7][8]. Algorithmic efficiency can be thought of as analogous to engineering productivity for a repeating or continuous process.

4.1 SOFTWARE METRICS

The two most frequently encountered and measurable metrics of an algorithm are:-speed or running time - the time it takes for an algorithm to complete, and 'space' - the memory or 'non-volatile storage' used by the algorithm **during its operation**.

Transmission size - such as required bandwidth during normal operation or size of Eternal memory-such as temporary disk space used to accomplish its task.

4.2 SPEED

The **absolute** speed of an algorithm for a given input can simply be measured as the duration of execution (or clock time) and the results can be averaged over several executions to eliminate possible random effects[9]. Most modern processors operate in a multi-processing & multi-programming environment so consideration must be made for parallel processes occurring on the same physical machine, eliminating these as far as possible.

4.3 MEMORY

It is possible to make an algorithm faster at the expense of memory. This might be the case whenever the result of an 'expensive' calculation is cached rather than recalculating it afresh each time[10]. The additional memory requirement would, in this case, be considered additional overhead although, in many situations, the stored result occupies very little extra space and can often be held in pre-compiled static storage, reducing not just processing time but also allocation & deallocation of working memory.

4.4.PRECOMPUTATION

Precomputing a complete range of results prior to compiling, or at the beginning of an algorithm's execution[11], can often increase algorithmic efficiency substantially. This becomes advantageous when one or more inputs is constrained to a small enough range that the results can be stored in a reasonably sized block of memory.

5. SOFTWARE VALIDATION VERSUS HARDWARE VALIDATION

An optimization technique that was frequently taken advantage of on legacy platforms was that of allowing the hardware (or microcode) to perform validation on numeric data fields[12]. The choice was to either spend processing time checking each field for a valid numeric content in the particular internal representation chosen or simply assume the data was correct and let the hardware detect the error upon execution. The choice was highly significant because to check for validity on multiple fields (for sometimes many millions of input records), it could occupy valuable computer resources. Since input data fields were in any case frequently built from the output of earlier computers[13]

6. CONCULSION

Different software development models will focus the desing effort at different points in the development process. Newer development models, such as algorithamic skeleton often employ desing case development and place an increased portion of the desing in the hands of the developer, before it reaches a formal team of designers. In this paper we foucs that the Algorithmic skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications.

7. REFERENCES

[1] Beck, Kent; Ward Cunningham (September 2009). "Using Pattern Languages for Object-Oriented Program". OOPSLA '87 workshop on Specification and Design for Object-Oriented Programming. OOPSLA '09.

[2] Baroni, Aline Lúcia; Yann-Gaël Guéhéneuc and Hervé Albin-Amiot (June 2003). "Design Patterns Formalization" (PDF). Nantes: École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes.]

[3] Erl, Thomas (2009). *SOA Design Patterns*. New York: Prentice Hall/PearsonPTR. pp. 864. ISBN 0-13-613516-1.

[4] http://soa.sys-con.com/node/809800

[5] Meyer, Bertrand; Karine Arnout (July 2006). "Componentization: The Visitor Example". *IEEE Computer* (IEEE) **39**

[6] Laakso, Sari A. (2003-09-16). "Collection of User Interface Design Patterns". University of Helsinki, Dept. of Computer Science. http://www.cs.helsinki.fi/u/salaakso/patterns/index.html. Retrieved 2008-01-31.

[7] Heer, J.; M. Agrawala (2006). "Software Design Patterns for Information Visualization". *IEEE Transactions on Visualization and Computer Graphics* **12** (5): 853. doi:10.1109/TVCG.2006.178..

[8] Chad Dougherty et al (2009). *Secure Design Patterns*. http://www.cert.org/archive/pdf/09tr010.pdf.

[9] Simson L. Garfinkel (2005). *Design Principles and Patterns for Computer Systems That Are Simultaneously Secure andUsable*..

[10] "Yahoo! Design Pattern Library". http://developer.yahoo.com/ypatterns/. Retrieved 2008-01-31.

[11] "How to design your Business Model as a Lean Startup?". http://torgronsund.wordpress.com/2010/01/06/lean-startup-business-model-pattern/. Retrieved 2010-01-06.

[12] Pattern Languages of Programming, Conference proceedings (annual, 2009) [1]

[13] McConnell, Steve (June 2004). "Design in Construction". *Code Complete* (2nd ed.). Microsoft Press. pp. 104. ISBN 978-0735619678. "Table 5.1 Popular Design Patterns"