Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Engineering 30 (2012) 1129 – 1137

Procedia Engineering

www.elsevier.com/locate/procedia

International Conference on Communication Technology and System Design 2011

A novel adaptive re-configuration compliance design pattern for autonomic computing systems

Vishnuvardhan Mannava[a], T. Ramesh[b] a*

[a]Department of Computer Science and Engineering, KL University, Vaddeswaram, 522502, A.P, India
[b]Department of Computer Science and Engineering, National Institute of Technology, Warangal, 506004, A.P, India

Abstract

The growing complexity in the systems due to growing size need a new mechanism to enable the system to self-manage, releasing administrators of low-level task management while delivering an optimized system. The Autonomic systems are: self-configuring, self-healing, self-optimizing, self-protecting. They sense the environment in which they are operating and automatically take action to change their own behaviour or the environment. There are no well established methodologies that a designer of an autonomic system can exploit to drive their work. So, the Current autonomic systems are ad hoc solutions in which each system is designed and implemented from scratch. This paper proposes adaptive reconfiguration compliance pattern for autonomic computing systems that can propose the reconfiguration rules and can learn new rules at runtime. The pattern is described using a java-like notation for the classes and interfaces. A simple UML class and Sequence diagrams are depicted.

© 2011 Published by Elsevier Ltd. Selection and/or peer-review under responsibility of ICCTSD 2011
Open access under CC BY-NC-ND license.

Keywords: Autonomic Computing; Design Patterns; Dynamic Adaptability and Reconfiguration.

1. Introduction

In order to get over the growing size and complexity of current systems, suggestion of mechanisms that are able to automatically adapt the systems to new scenarios is essential. Mechanisms are needed to make the systems self-managing to overcome rapidly growing complexity and to enable their further growth. So, the management of such huge size and complex systems is hard and expensive to be done by

* Vishnuvardhan Mannava. Tel.: +91- 08644-238262
E-mail address: vishnumannava@gmail.com.

using solely the human operators. The autonomic systems sense the environment in which they are operating and take action to change their own behavior or the environment with a minimum effort.

Design patterns can be classified using two orthogonal classification schemes. The first option is to classify the patterns according to their purpose: creational, structural, or behavioral. Creational patterns focus on object creation. Structural patterns focus on describing the composition of classes or objects. Behavioral patterns depict the method of interaction and distribute the responsibility of classes or objects. Thus far, we have only identified structural and behavioral adaptation design patterns.

The second option is to classify the patterns according to their adaptation functions: monitoring, decision-making, and reconfiguration [12]. Monitoring patterns focus on probing components and distributing the information across a network to interested clients. Decision-making patterns focus on identifying when a reconfiguration is needed and selecting a reconfiguration plan that will yield the desired behavior [1]. Reconfiguration patterns focus on safely adding, removing, or modifying components at run time to adapt a program. Thus far, we have identified and proposed several design patterns for each area.

Monitoring [2] and decision-making patterns [3][4] are what we consider to be adaptation enabling design patterns. These design patterns provide the necessary infrastructure to perform introspection and intercession. Although monitoring and decision-making design patterns do not reconfigure an application, without these, a developer would have to manually perform these tasks at run time. Therefore, reconfiguration patterns depend upon monitoring and decision-making design patterns.

For instance, a traffic shaper [5] can be considered. The packets come into the shaper, it first assigns them a priority (with respect to some traffic classification rules) and then it chooses how to manage them. Based on the predefined strategy it’s all ready available on the strategy class. Some packets can be put inside higher priority queues, others in lower priority queues and/or can be discarded by the system because are unsuitable for the traffic shape it has to provide. The pattern provided is a generic solution that can be easily adapted to specific situations.

The strategy of the system is predefined based on the input stream strategy plan will chosen by the TRIGGER REPOSITERY. Classifier of the system will classifies the input and output stream based on the strategy that are applied on input. And apply strategy based on the priority of the process.

This memory management system reconfiguration of modules depends on the memory usage of the modules. If a certain module has occupied more memory and the usage is null then it has to be killed and another module has to be invoked for which the user has requested. If certain module has occupied more memory and usage the module is more then it will create child module of previously executing module.

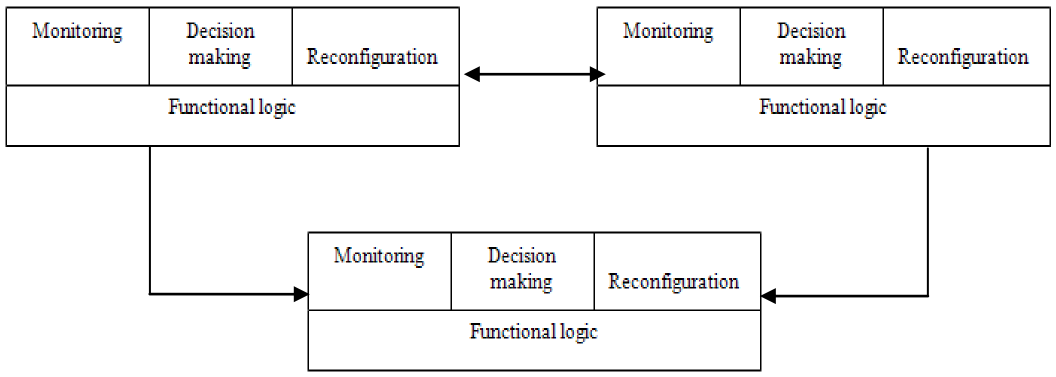

Figure 1: Autonomic Elements

2. Related work

In this section we present some works that deal with different aspects of autonomic systems and their design. Gomaa et al. proposed several patterns for dynamically reconfiguring specific types of software architectures at run time [1]. Sterritt and Bustard in their paper [6] discuss the type of system architecture needed to support such objectives. They propose a design template based on a simple characterization of autonomic systems. In [7], Gilbert exploits the analogy of autonomic human behavior with object behavior as an abstraction used to identify opportunities for concurrency. The paper provides a pattern that exploits such abstraction. In [8] is presented a technique to approximate the form of the data flow distribution. Their estimation can lead to a good data management, but their proposal has not autonomic features. Pendarakis et al. [9] propose a characterization of the traffic generated by distributed application. They present a system for autonomic management of network resources (such as local link bandwidth) to effect a desired balance between concurrently executing processes on a data flow processing node. Solomon et al. [10] outline a component-based architecture for autonomic computing and propose a set of seven components for building autonomic systems. While our approach focuses on the programming model of autonomic computing systems, we propose a general architecture for autonomic applications.

3.Adaptive Reconfiguration Compliance Desing Pattern

Figure 2 shows the Class Diagram for Adaptive Reconfiguration Compliance Design Pattern. Autonomic behavioural design pattern's aim is to provide a general repeatable solution easing the design of autonomic memory management systems. A Module memory management system is characterized by three components that can be represented in the following way:

- MEMORY USAGE DATA STREAM**:** a group of elements or set of elements there is no functional dependencies M (usually the memory usage of modules) $M_i \in M$.
- CLASSIFIER: classifies MEMORY USAGE DATA STREAM AND OUTPUTSSTREAM depending on the policy supplied by the administrator. Classifier function f applied to each element *of* MEMORY USAGE DATA STREAM $f(M)$; *i.e.* $f(M)>0;$
- OUTPUTSTREAM: a group of elements $f(o)$ such that each element is $m_{i'=}f(M)$ with $m_i \in M$ and $m_{i'=O}$

The CLASSIFIER receives the input from the MEMORY USAGE DATA STREAM and then classifies based on administrator policy send to OUTPUTSTREAM, in case the CLASSIFIER is a traffic shaper the policy is specified by the network administrator. Let's suppose that the CLASSIFIER has a priori knowledge of the nature of all possible elements it has to classify, and it is able to manage every possible distribution of the items coming from the input data flows without any performance loss. Nevertheless, in a more realistic scenario, the input items come from the input data flows with a non-predictable distribution. Hence the CLASSIFIER could behave in a strange or inefficient way.

Let's suppose a memory management system where the reconfiguration of modules depends on the memory usage of the modules. If a certain module has occupied more memory and the usage is null then it has to be killed and another module has to be invoked for which the user has requested OR if a module has occupied more space in the memory as well as rapid use of memory then the CLASSIFIER will create other instance of same module.

The data prioritize based on the size, the CLASSIFIER will emit long data flows as low priorities, resulting bad utilization of the network channel. In this case, it could be fruitful to replace the management strategy deriving from the administrator policy. If a designer has sufficient knowledge about the data flows of input element, it is sufficient to exploit the GoF strategy design pattern to give to the system the ability to replace its strategy dynamically. Unfortunately, it is impossible to have such a priori knowledge [11]. Hence, in order to accomplish the task to classify items in an efficient way, the CLASSIFIER needs to behave in an autonomic way.

This paper proposes a generic autonomic pattern for Autonomic Computing System designers to achieve automaticity with a minimal effort. We use strategy pattern for autonomic systems, Systems are designed to analyse data flows of memory usage data or stream and classify each data flow item depending on a specific management policy. We named as "memory usage of modules". We have a system which behaviour is driven by an external entity: the strategy. The strategy perfectly handle self configuration. This strategy uses three other entities: TRIGGER REPOSITORY, INFERENCE ENGINE and STRATEGY.

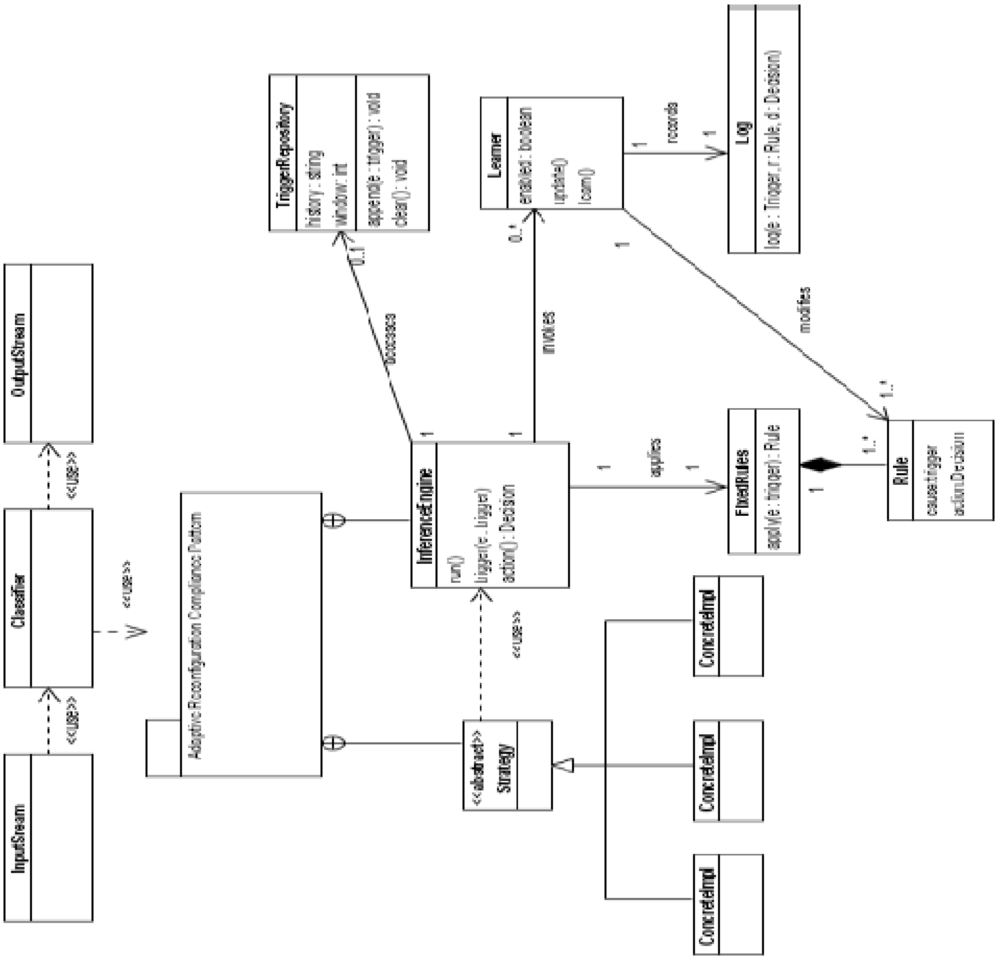

Figure 2: Adaptive Reconfiguration Compliance Design Pattern for Autonomic Computing Systems

The strategy of the system is predefined based on the input stream strategy plan will chosen by the TRIGGER REPOSITERY. Classifier of the system will classifies the input and output stream based on the strategy that are applied on input. And apply strategy based on the priority of the process. Strategy of system categories based input stream if the memory usage of the input stream is more then it will create another thread for the presently executing module or memory usage of the module is more and the utility of the system is less then it will kill the process. We extended the pattern template used by Gamma et al. [11] for describing design patterns with Behavioural and Constraints fields.

4. Participants

- TRIGGER REPOSITORY: It contains all previous Triggers. This trigger is used by the Learner for further reconfigurations plans [3].

- INFERENCE ENGINE: This class contains set of Rules that are applied to either a single Trigger or a history of Triggers and producing an action in the form of a Decision. It takes as input the current Strategy configuration and the TRIGGER REPOSITORY.

- STRATEGY: It consists of group of classes; based on the input module it will choose the Concrete Implementation.

- FIXED RULES: This class contains a collection of Rules that guide the Inference Engine in producing a Decision. The individual Rules stored within the Fixed Rules can be changed at run time[3].

- RULE: Represents a relationship between a Trigger and a Decision. A Rule evaluates to true if an incoming Trigger matches the Trigger contained in the Rule.

- LEANER: Applies on-line and statistical-based algorithms to infer new Rules in the system. This is an optional feature of the Case-based Reasoning design pattern.

- LOG: This class is responsible for recording which reconfiguration plans have been selected during execution. Each entry is of the form Trigger Rule Decision.

The CLASSIFIER forward all the module that are received from the MEMORY USAGE DATA STREAM to STARATEGY it will choose appropriate policy based on the input modules, before classifying the items STRATEGY access its own configuration by invoking the INFERENCE ENGINE. The INTERFACEENGINE read the past input/output from the TRIGGER REPOSITORY and it take decision based on the STRATEGY. After the reconfiguration step the INTERFACE ENGINE computes the output values accordingly to its new configuration and stores both the input and the computed output into TRIGGER REPOSITORY. Finally, the INTERFACE ENGINE sends the computed output back to the CLASSIFIER then retrieved to MEMORY USAGE DATA STREAM.

4.1 Collaborations

- Strategy and INTERFACE ENGINE interact to implement the chosen reconfiguration plan. INTERFACE ENGINE passes all data required by the reconfiguration plan to the strategy when the reconfiguration is called. Alternatively, the INTERFACE ENGINE pass itself as an argument to Strategy operations. That lets the strategy call back on the INTERFACE ENGINE as required.
- INTERFACE ENGINE forwards requests from its clients to its strategy. Clients usually create and pass a Concrete Strategy object to the context; there after clients interact with the context exclusively. There is often a family of Concrete Strategy classes for a client to choose from.

4.2Applicability

Use the memory management pattern when

- The strategy chooses the reconfiguration plan based on the memory usage of the modules.
- You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
- If the strategy used again and again it will store the results in the TRIGGER REPOSITERY, other module need the same strategy then it will take the strategy from the TRIGGER REPOSITERY.

- No need of calculating reconfiguration plans again and again if it is all ready there in the TRIGGER REPOSITERY then INTERFACE ENGINE invoke the plan and post the OUTPUTSTREAM values.

5. Interfaces definition for the pattern entities

The entities involved in the behavioural pattern presented can be represented by classes. In this section we used java syntax to report five out of seven different programming interfaces that the classes representing the entities must implement. The first three interfaces represent the methods of a much summarized view of the data flow-management-system: the MEMORY USAGE DATA STREAM, the OUTPUTSTREAM and the CLASSIFIER. The last four interfaces are presented to describe the methods of the entities to be provided to make the data flow management system autonomic: the Strategy, the INFERENCE ENGINE and the TRIGGER REPOSITORY.

The CLASSIFIER is a thread consisting in an infinite loop which performs three operations: a Read on the Input- Data flow, an elaborate (described below) on the Strategy and a Write on the OUTFLOW.

Classifier.java
```
public class Classifier extends Thread
{
        static void Run()
        {
        }
        Public static void main(String[] args)
        {
                Run();
        }
}
```
ConcreteStrategyAlpha.java
```
Public class ConcreteStrategyAlpha implements Strategy
{
        Public void Elaborate(String str) {}
}
Public class ConcreteStrategyInt implements Strategy
{
        publicvoid Elaborate(String str)
        {
        }
}
```

Inferenceengine.java
```
Public class Inferenceengine
{
        Public void trigger(Triggerrepository e)
        {
        }
        Public void Eval(String str)
        {
        }
}
```
Fixedrules.java
```
Public class Fixedrules
{
       public void apply(Inferenceengine e)
       {
        Rule r=new Rule(e);
       }
}
```

Strategy.java
```
public interface Strategy
{
    public void Elaborate(String str);
}
```

The Strategy class interface provides three methods. The first one (Elaborate) is invoked by the CLASSIFIER to activate the management activity of the strategy.

To demonstrate the efficiency of the pattern we took the profiling values using the Netbeans IDE and plotted a graph that shows the profiling statistics when the pattern is applied and when pattern is not applied. This is shown in figure 4.Here X-axis represents the runs and Y-axis represents the time intervals in milliseconds.Below simulation shows the graphs based on the performance of the system if the pattern is applied then the system performance is high as compared to the pattern is not applied.

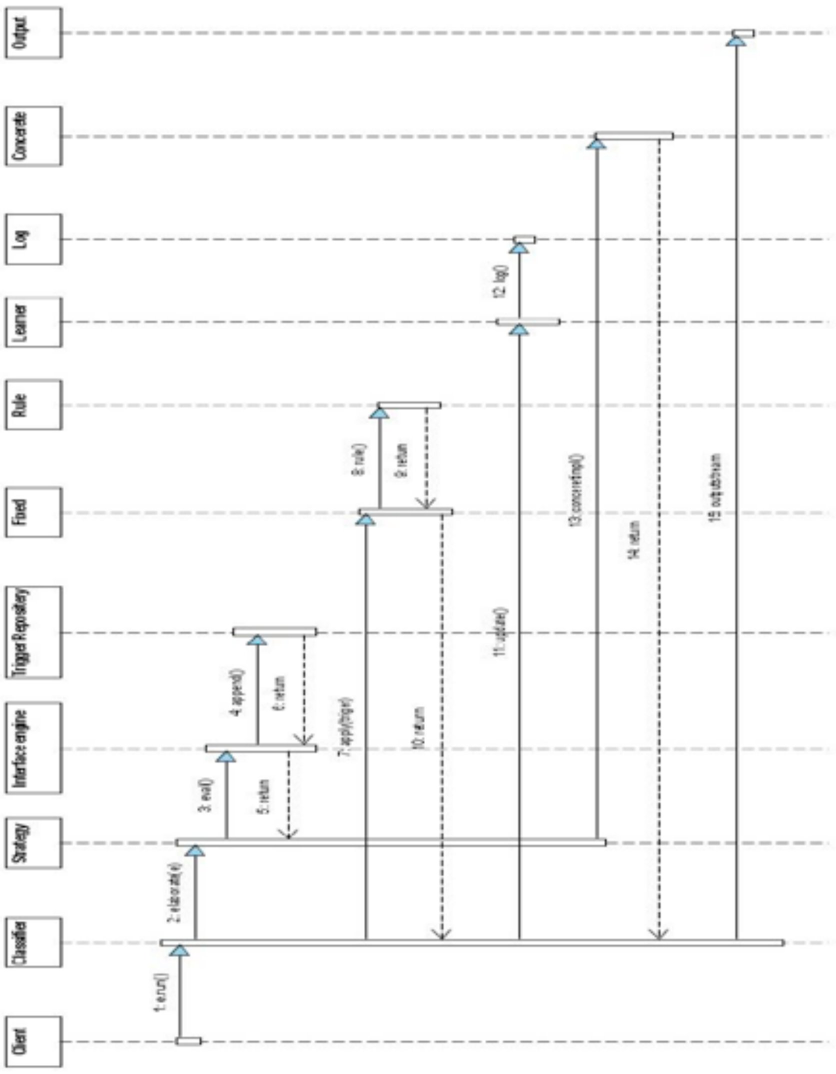

Figure 3: Sequence diagram for Adaptive Reconfiguration Compliance Design Pattern for Autonomic Computing Systems

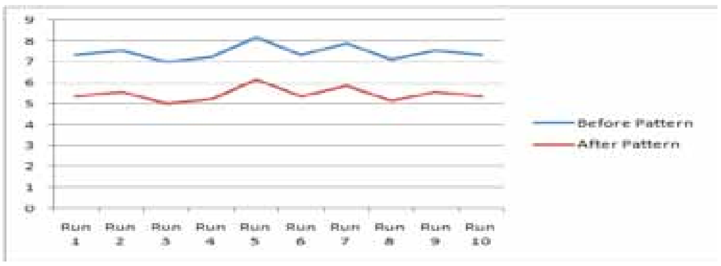

Figure 4: Profiling statistics before applying pattern and after applying pattern

6. Conclusion and Future work

In this paper, we presented a pattern easing the design and the implementation of runtime reconfiguration-systems. The pattern can be easily incorporated into the design and implementation process packaging it inside a component around which develops the autonomic computing systems and can learn new rules dynamically. An interesting direction for future research concerns Meta-programming and Aspect Oriented Programming: code transformation tools that, starting from a set of high level specification, are able to generate the code needed to provide autonomic behaviour to certain class of systems.

7. References:

[1] H. Gomaa and M. Hussein, "Software reconfiguration patterns for dynamic evolution of software architectures" In WICSA'04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture, page 79, Washington, DC, USA, 2004.

[2] Vishnuvardhan Mannava and T. Ramesh, "A Novel Adaptive Monitoring Compliance Design Pattern for Autonomic Computing Systems, Springer, Communications in Computer and Information Science, 1, Volume 190, Advances in Computing and Communications, Part 3, Pages 250-259, 2011

[3] Vishnuvardhan Mannava and T. Ramesh, "A Novel Event Based Autonomic Design Pattern for Management of Webservices", Springer, Communications in Computer and Information Science, 1, Volume 198, Advances in Computing and Information Technology, Part 1, Pages 142-151, 2011

[4] V. S. Prasad Vasireddy, Vishnuvardhan Mannava and T. Ramesh, "A Novel Autonomic Design Pattern for Invocation of Services", Springer, Communications in Computer and Information Science, 1, Volume 196, Advances in Network Security and Applications, Part 2, Pages 545-551, 2011

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. Ietf rfc 2475.
http://tools.ietf.org/html/rfc2475, December 1998.

[6] Roy Sterritt and Dave Bustard, "Towards an autonomic computing environment", In Proc. of the 14th International Workshop on Database and Expert Systems Applications, 2003.

[7] John W. Gilbert. Privatethread, "A software pattern for the implementation of autonomic object behavior" In Workshop on design patterns for concurrent, parallel, and distributed object-oriented systems (OOPSLA 95), 1995.

[8] Nittaya Kerdprasop and Kittisak Kerdprasop, "Density estimation technique for data stream classification", In DEXA '06: Proceedings of the 17th International Conference on Database and Expert Systems Applications, Washington, DC, USA, 2006. IEEE Computer Society.

[9] Dimitrios Pendarakis, Jeremy Silber, and Laura Wynter, "Autonomic management of stream processing applications via adaptive bandwidth control", In ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Bogdan Solomon, Dan Ionescu, Marin Litoiu, and Mircea Mihaescu, "Towards a real-time reference architecture for autonomic systems" In SEAMS '07: Proc. of the 2007.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Pp: 105-155, 1995.

[12] Andres J. Ramirez and Betty H.C. Cheng, "Design Patterns for Developing Dynamically Adaptive Systems", In the Proceedings of the 5th International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'10), Cape Town, South Africa.