New Perspectives in Autonomic Design Patterns for Stream-Classification-Systems*

Patrizio Dazzi
HPC Lab – ISTI/CNR
IMT – Lucca
Italy
patrizio.dazzi@isti.cnr.it

Francesco Nidito
Computer Science Dept.
University of Pisa
Italy
francesco.nidito@di.unipi.it

Marco Pasquali
HPC Lab – ISTI/CNR
IMT – Lucca
Italy
marco.pasquali@isti.cnr.it

ABSTRACT

Nowadays, systems are growing in size and are becoming more and more complex. Such a complexity suggests a new need for mechanisms that enable the system to self-manage, freeing administrators of low-level task management whilst delivering an optimized system. Autonomic systems sense their operating environment and automatically take action to change the environment or their own behavior. They are able to achieve it with a minimum of human effort. This is because they are: self-configuring, self-healing, self-optimizing and self-protecting. Current autonomic systems are ad hoc solutions: each system is designed and implemented from scratch i.e., there are not standard (or well-established) methodologies that autonomic system designers and/or programmers can exploit to drive their work. In this paper, we propose a design pattern that can be easily exploited by the stream-classification-systems designer to achieve autonomicity with a minimal effort. The pattern is described using a java-like notation for the classes and interfaces. A simple UML class diagram is depicted.

Categories and Subject Descriptors

F.1.1 [**Computation by abstract devices**]: Models of Computation—*Self-modifying machines, Unbounded-action devices*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms

Autonomic Computing, Adaptive Strategies

Keywords

autonomicity, behavioral design pattern

*This research is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265), the FP6 GridCOMP project partially founded by the European Commission (Contract FP6-034442)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASE Workshop on Automating Service Quality, November 2007, Atlanta, Georgia, USA
©2007 ACM ISBN: 978-1-59593-878-7 /07/11...$5.00

1. INTRODUCTION

The growing size and complexity of current systems suggests a new need for mechanisms able to automatically adapt the systems to new scenarios. A need for mechanisms making the systems self-managing in order to overcome their rapidly growing complexity and to enable their further growth. Indeed, the management of such systems, characterized by a huge size and complexity, is too difficult and expensive to be done by using human operators only. On the other hand, the autonomic systems sense their operating environment and take action to change the environment or their own behavior with a minimum effort. This is because they are able to adapt themselves to new, somehow not previously taken into account, situations. Autonomic Computing is an initiative started by IBM in 2001, with the presentation of its manifesto [4]. Its ultimate aim is to create self-managing computer systems to overcome their rapidly growing complexity and to enable their further growth. The manifesto states that a system, to be autonomic, must have the following properties (that are simply listed because their analysis is beyond the scope of this paper): self-configuring, self-healing, self-optimizing and self-protecting. The IBM researchers outlined in the autonomic manifesto and in the "*Vision*" paper [5], the main aspects characterizing the autonomic computing:

- the features that an autonomic system should have;
- a possible evolution path for the autonomic computing (five evolution steps: base, managed, predictive, adaptive, autonomic);
- a highly abstract structure of an autonomic element (Figure 1).

Nevertheless, they did not provide any programming model or (behavioral) design pattern to ease the work of autonomic application (or system) designers. The lack of design and implementation methodologies for autonomic computing brings about a very expensive design and implementation of autonomic system. Current autonomic systems can be considered ad hoc solutions: each system is designed and implemented from scratch. In this paper, we propose a generic autonomic pattern for stream-classification-systems (SCS) that can be easily exploited by SCS designers to achieve autonomicity with a minimal effort. The stream-classification-systems are designed to analyze streams of data and to classify each stream item depending on a specific classification policy. A traffic shaper [1] is a good example of

these kind of systems. The packets come into the shaper, it first assigns them a priority (w.r.t. some traffic classification rules) and then it chooses how to manage them. 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. This paper is organized as follows: in Section 2 we present the proposed pattern describing its fundamental entities. In Section 3 we shortly introduce related work and in Section 4 we draw our conclusion sketching the path for possible future work.

2. AUTONOMIC DESIGN PATTERN FOR STREAM-CLASSIFICATION-SYSTEMS

In this section, we present our autonomic behavioral design pattern. Its aim is to provide a general repeatable solution easing the design of autonomic stream-classification-systems. A stream-classification-system is characterized by three components: an InputStream, a Classifier and an OutputStream. They can be represented in the following way:

- InputStream: a stream (or set) of independent elements I among which there are not functional dependencies.
- Classifier: a classification function (f) applied to each element of I.
- OutputStream: a stream (or set) of elements O such that each element is $e'_i = f(e_i)$ with $e_i \in I$ and $e'_i \in O$.

The Classifier retrieves each input element from the InputStream, then it classifies the element eventually sent to the OutputStream. Typically, the classification is driven by a policy specified by the classifier administrator e.g., 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 streams without any performance loss. In this case the classifier performs all classifications correctly, simply exploiting the information specified by the administrator policy. Nevertheless, in a more realistic scenario, the input items come from the input streams with a non-predictable distribution. Hence the classifier could behave in a strange or inefficient way. Let's

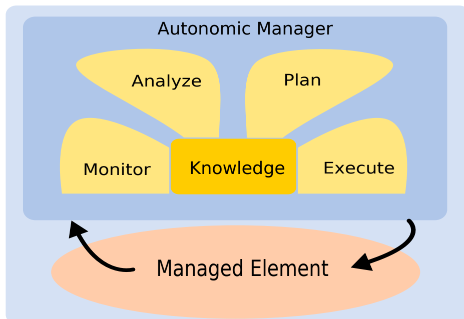

Figure 1: Structure of an autonomic element

suppose a shaping policy of a network traffic shaper that assigns a high-priority to IP packets whose size is less than 2 Kb, and it assigns a low-priority to the other packets. Moreover, suppose that the 80% of the network bandwidth is reserved for high-priority packets. In presence of long and very different streams, each one made of items characterized by very similar size, the classifier will emit long streams of high- or low-priorities, resulting in a bad utilization of the network channel. In this case, it could be fruitful to replace the classification strategy deriving from the administrator policy with a more suitable one. If a designer has sufficient knowledge about the streams of input element, it is sufficient to exploit the GoF strategy design pattern [2] to give to the system the ability to replace its strategy dynamically. Unfortunately, it is impossible to have such a priori knowledge. Hence, in order to accomplish the task to classify items in an efficient way, the Classifier needs to behave in an autonomic way. Conceiving our autonomic pattern for stream-classification-

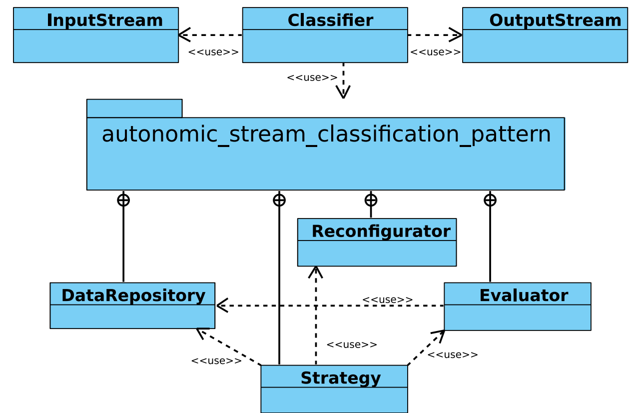

Figure 2: UML schema of the proposed autonomic strategy pattern.

systems we have been inspired by the GoF strategy pattern. As the strategy pattern, we have a system which behavior is driven by an external entity: the strategy. Moreover, the main entity of our pattern is called Strategy. On the contrary w.r.t the original strategy pattern, it is able to classify the packets, to evaluate itself and to change its behavior accordingly to some rules. To perform these tasks Strategy uses three other entities: DataRepository, Evaluator and Reconfigurator. Their behavior can be described as follows:

- DataRepository: it is an entity that holds up to a certain (finite) number of past input elements coupled with the respective computed outputs.
- Evaluator: it is an entity able to suggest a Strategy reconfiguration. It takes as input the current Strategy configuration and the DataRepository, than it suggests a change in the Strategy behavior.
- Reconfigurator: it takes as input the Strategy and the output of Evaluator, it is able to reconfigure the Strategy, in order to optimize it.

The Classifier forwards the items retrieved from the InputStream directly to the Strategy. Before classifying

the items, The STRATEGY evaluates its own configuration by invoking the EVALUATOR. The EVALUATOR reads the past input/output from the DATAREPOSITORY and then it evaluates the adherence of the classifier behavior w.r.t. the given classification policy. If the behavior is different from the expected one, the EVALUATOR suggests a change in the STRATEGY configuration. If the system needs to be reconfigured, STRATEGY invokes the RECONFIGURATOR passing to it the suggestions proposed by the EVALUATOR. The RECONFIGURATOR retrieves the tuning parameters of the STRATEGY through which it changes the configuration and, in consequence, the behavior of STRATEGY. After the reconfiguration step the STRATEGY computes the output values accordingly to its new configuration and stores both the input and the computed output into DATAREPOSITORY. Finally, the STRATEGY sends the computed output back to the CLASSIFIER that in turn send it to the OUTPUTSTREAM. A UML schema of the packages and classes that implements our autonomic strategy pattern is depicted in Figure 2. The higher part of the figure represents the classification system, made up of the CLASSIFIER entity, the INPUTSTREAM entity and the OUTPUTSTREAM entity. In the lower part of the figure, it is represented our autonomic pattern belonging to a package. The pattern package is made up of four entities: the EVALUATOR, the STRATEGY, the DATAREPOSITORY and the RECONFIGURATOR.

2.1 Interfaces definition for the pattern entities

The entities involved in the behavioral pattern presented can be represented by classes. In this section we use a Java-like syntax to report seven different programming interfaces that the classes representing the entities must implement. The first three interfaces represent the methods of a very summarized view of the stream-classification-system: the INPUTSTREAM, the OUTPUTSTREAM and the CLASSIFIER. The last four interfaces are presented to describe the methods of the entities to be provided to make the stream-classification-system autonomic: the STRATEGY, the EVALUATOR, the RECONFIGURATOR and the DATAREPOSITORY. Figure 3 presents the INPUTSTREAM class interface. This

```
class InputStream {
   Element Read( void );
}
```

Figure 3: The InputStream class interface.

interface must provide a public method, called READ, that permits the CLASSIFIER to retrieve data elements from the input stream. The type of the elements belonging to the input stream is ELEMENT. ELEMENT must present a suitable interface for its analysis and classification. Figure 4

```
class OutputStream {
   void Write( ClassifiedElement e );
}
```

Figure 4: The OutputStream class interface.

presents the OUTPUTSTREAM class interface. This interface must provide a public method, called WRITE, that enables the CLASSIFIER to write the classified elements into the output stream. The elements of the output stream have type CLASSIFIEDELEMENT, that represents an element after the classification process. A possible CLASSIFIEDELEMENT class can contain an instance of the ELEMENT class that originated it and a QoS field, set to the appropriate value for the subsequent usage by the application reading such output values. A convenient way to represent the class of CLASSIFIER

```
class Classifier extends Thread {
   void Run();
}
```

Figure 5: The Classifier class interface.

(presented in Figure 5) is to use a subclass of the Thread class. The CLASSIFIER is a thread consisting in an infinite loop which performs three operations: a *Read* on the INPUTSTREAM, an *elaborate*(described below) on the STRATEGY and a *Write* on the OUTPUTSTREAM. Figure 6 presents the

```
class Strategy {
   void Elaborate( Element );
   List GetTunableParameters();
   void SetTunableParameters( List );
}
```

Figure 6: The Strategy class interface.

STRATEGY class interface. It provides three methods. The first one (*Elaborate*) is invoked by the CLASSIFIER to activate the classification activity of the strategy. The second (*GetTunableParameters*) and third (*SetTunableParameters*) methods are invoked by the RECONFIGURATOR to get and to set the STRATEGY tuning parameter in order to optimize the classification performances. The EVALUATOR class interface

```
class Evaluator {
   Evaluation Evaluate( DataRepository );
}
```

Figure 7: The Evaluator class interface.

is presented in Figure 7. It defines only one public method: *Evaluate*. This method takes as input a DATAREPOSITORY object storing a finite set of CLASSIFIEDELEMENT. The EVALUATOR analyzes such data in order to evaluate the behavior of the current strategy, comparing it to the one expected by the administrator. If the behavior is not compliant to the one specified by the administrator's policy, the *Evaluate* method returns an evaluation object describing the *distance* between the current behavior and the desired one. The instance is defined as the difference between the optimal behavior and the actual behavior. Its nature strictly depends on the application: it can be a single numeric value or a complex tuple. Figure 8 presents the RECONFIGURATOR class interface. It defines the *Reconfigure* method that takes as input: the evaluation returned by the EVALUATOR and a reference to the STRATEGY object. If the returned *distance*, computed by the EVALUATOR, is different from zero, the RECONFIGURATOR retrieves the strategy tuning parameters in

```
class Reconfigurator {
   void Reconfigure( Evaluation, Strategy );
}
```

Figure 8: The Reconfigurator class interface.

order to change the behavior of the STRATEGY object. The

```
class DataRepository {
   void Store( Element, ClassifiedElement);
   List GetData();
}
```

Figure 9: The DataRepository class interface.

DATAREPOSITORY class interface is presented in Figure 9. It defines two methods: *Store* and *GetData*. The former is used by the STRATEGY class to store each input item and the classification it received. The latter is used by the EVALUATOR to retrieve a *List* of the data stored.

2.2 Polytope

In the previously described system, each input may trigger the reconfiguration activity. This operation can be expensive and sometimes the optimization gain can be less than the reconfiguration overhead. To address this problem we introduce the concept of *polytope*. A polytope consists in a particular subset of the possible values returned by an EVALUATOR invocation. When an EVALUATOR invocation returns a value belonging to the *polytope* the reconfiguration step is not performed i.e., the polytope is a geometrical area in which the values returned by the EVALUATOR are free to move without triggering a reconfiguration. A geometrical interpretation of the polytope is the following: each value v returned by the EVALUATOR can be seen as a coordinate vector which points are real numbers ($v \in \mathbb{R}^n$), and the polytope as a subspace in $\mathbb{R}^n$ centered in the value representing the optimal stream-classification. A strategy reconfiguration is required if and only if the value returned by the EVALUATOR relies outside the polytope meaning that the classification behavior is quite bad. As a consequence, the RECONFIGURATOR plays a slightly different role. The RECONFIGURATOR reconfigures the STRATEGY to move the possible next status inside the polytope perimeter (accordingly to the past history only).

3. RELATED WORK

In this section we present some works that deal with different aspects of autonomic systems and their design. Sterritt and Bustard in their paper [9] 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 [3], 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 exploit such abstraction. In [6] is presented a technique to approximate the form of the data stream distribution. Their estimation can lead to a good data classification, but their proposal has not autonomic features. Pendarakis et al. [7] propose a characterization of the traffic generated by distributed applications. 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 stream processing node. Solomon et al. [8] outline a component-based architecture for autonomic computing and propose a set of seven components for building autonomic systems. While our approach focus on the programming model of autonomic SCSs, they propose a general architecture for autonomic applications.

4. CONCLUSION AND FUTURE WORK

In this paper, we presented a pattern easing the design and the implementation of autonomic stream-classification-systems. The pattern can be easily incorporated into the design and implementation process packaging it inside a component (or a library) around which develop the autonomic SCS. 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 behavior to certain class of systems.

5. REFERENCES

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

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995.

[3] 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.

[4] IBM. "autonomic computing: Ibm perspective on the state of information technology". http://www.research.ibm.com/autonomic/manifesto, 2001.

[5] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1), January 2003.

[6] 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.

[7] 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.

[8] 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 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Washington, DC, USA, 2007. IEEE Computer Society.

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