Architectural and Design Patterns for Developing Autonomic Computing Systems

Vishnuvardhan Mannava\textsuperscript{1,2}, Dr. T. Ramesh\textsuperscript{2}

\textsuperscript{1}Department of Computer Science and Engineering, KL University, Vijayawada
\textsuperscript{2}Department of Computer Science and Engineering, National Institute of Technology, Warangal

Abstract: As computational complexity of systems continues to increase, the amount of maintenance required to keep them operational will also increase. Autonomic systems have the aim of reducing the amount of maintenance required by performing certain levels of maintenance themselves. This paper describes a preliminary investigation into identifying design patterns associated with the self-monitoring aspects of autonomic computing systems. We studied many autonomic-related research and project implementations available from the literature and open sources to find autonomic-oriented design patterns that support the development of autonomic systems.

1. Introduction:

Given the complexity of the present day enterprise IT infrastructure, its exploitation, operation, and maintenance require considerable efforts and funds. The idea of building intelligent management software for the IT infrastructure automation was for a long time an aim of many academic and industry research groups. A higher automation step in the supervision and control of all IT infrastructure elements is represented by the autonomic computing. To address the growing complexity due to the advances in computing and communication technologies for distributes software autonomic systems are generally difficult to specify, design, verify, and validate [1]. In addition, the current lack of reusable design expertise that can be leveraged from one adaptive system to another further exacerbates the problem. To address this problem, we have identified design patterns for adaptive and autonomic systems.

Most adaptive systems, including autonomic systems, comprise three key elements: monitoring, decision-making, and reconfiguration. Monitoring enables an
application to be aware of its environment and detect conditions warranting reconfiguration; decision-making determines what set of monitored conditions should trigger a specific reconfiguration response; and reconfiguration enables an application to change itself in order to fulfill its requirements. Not only must developers design and implement each of these elements correctly, they must also carefully determine their interactions. For instance, if the monitoring process fails to report a significant environmental change, then the decision-making process may incorrectly determine whether a reconfiguration is warranted or not. Unfortunately, until recently, most approaches have addressed adaptation in ad hoc manners [2]. To address these concerns, researchers provided adaptation-enabling frameworks [3, 4, 5], middleware [6, 7], and language-based support [8, 9]. These approaches, however, tend to be tightly coupled with specific domains or technologies, thus limiting their fitness with respect to the problem being addressed. Design patterns, on the other hand, work at the modeling level of abstraction, thereby possibly increasing the amount of design reuse when compared to other approaches.

This research is aimed at providing a strong software engineering foundation for autonomic based system development. The Structure of the paper is of as follows Section: 1 Address Autonomic System Characteristics. Section: 2 describe the Architectural Patterns for autonomic Systems. Section: 3 describe the Design Patterns for autonomic Systems

2. Autonomic Systems

Autonomic systems incorporate self- properties such as Self-configuration, self-healing, self-optimization, and self-protection. Self-configuration refers to the ability to reconfigure components and their interactions. Self-healing refers to the ability of automatically discovering and correcting faults. Self-optimization refers to the ability to optimize behavior based on requirements and constraints. Self-protection refers to ability to detect and fend-off attacks.

Design patterns can also be used to aid in the design and construction of autonomic computing systems comprising some number of autonomous elements. Each autonomous element is instrumented with monitoring, decision-making, and reconfiguration processes.
This paper describes the autonomic design patterns and how they can be used to construct adaptive and autonomic systems. These autonomic oriented patterns facilitate the separate development of the functional and adaptive logic.

3. Architectural Patterns for Autonomic Systems

Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them. The following are the Architectural patterns identified from literature and open sources based on Self-configuration, Self-healing, Self-optimization and Self-protection characteristics of the Autonomous systems

3.1 Microkernel pattern

Since the microkernel approach in operating system has been proved to be effective, some researchers try to abstract this approach to a kind of architecture pattern to guide the design of adaptive systems [10, 11]. The pattern structure is illustrated in Fig.1.

The microkernel serves as a socket for plugging in new features and has the responsibility of coordinating service collaboration. Internal server extends the functionality of microkernel, which makes microkernel small. External server is a layer of abstraction built over atomic services and provides programming interfaces for user applications. User application calls adaptor for service. The adaptor invokes methods of external servers on behalf of clients. The introducing of adaptor avoids tight coupling between user applications and underlying servers.
Fig. 1. Microkernel pattern Structure

The microkernel architectural pattern is expected to be applied to software systems that are under the pressure of constant evolution so as to adapt to changing requirements. However, distributed computing systems are complex in nature. There is no hard and fast rule that can be stated about what constitutes a specific microkernel and what is the corresponding structure.

**Known Uses:** KAON Semantic Middleware and Application Server [12], PKUAS Application Server [13] [37]

### 3.2 Reflection Pattern

The Reflection architectural pattern [23] provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms. In this pattern, an application is split into two parts. A meta level provides information about selected system properties and makes the software self-aware. A base level includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behavior. The pattern structure is illustrated in Fig. 2.

Fig. 2 Reflection Pattern Structure
Known Uses
CodA [14], OLE 2.0 [15], OpenCorba: a Reflective Open Broker[16][24]

3.3 Adaptability Aspects Architectural pattern

This architectural pattern is intended to show how to use aspects [17] in order to better structure adaptive applications, which are able to change their behavior in response to context changes [18], such as the device new localization or its resources state.

The Adaptability Aspects architectural pattern presents five elements or modules:

**Base Application:** The core application functionalities, such as business and GUI code, and possibly persistence and distribution code, but no adaptability code.

**Adaptability Aspects:** The aspects implementing the adaptability concern. They specify how the behavior of the base application functionalities should be changed to adapt to contextual changes. This element delegates several tasks to the auxiliary classes.

**Auxiliary Classes:** Classes used by the aspects to provide the adaptive behavior. Its isolation from the aspect is intended to improve reuse. They communicate with the adaptation data provider module in order to obtain dynamic data for the adaptation. Besides that, for developing the auxiliary classes, the developers of this module do not necessarily need to know an AOP language. The Adaptability Aspects developer or the system architect may simply specify the interfaces of these classes and what they should do. Then, from these specifications, these classes can be built and have their methods invoked by the aspects.

**Context Manager:** Module responsible for analyzing context changes and triggering adaptive actions implemented by the aspects. It can also be called by the aspects to obtain information about the context. Its implementation can be based on a variation of the Observer pattern [19], or on its implementation with aspects [20]. In this way, new mechanisms for accessing the context can be easily supported without significant impact on the application.

**Adaptation Data Provider:** Classes responsible for providing data for dynamic adaptations according to context changes. This means that the same context change can lead to different behaviors in different moments according to the data provided by this module. These classes can be organized as an Adaptive Object-Model (AOM) [21].

These elements and their inter-relation are shown in Fig 3. They are represented there using the UML package notation. Each package represents a logical part of
the code, but each of these parts can be implemented using several programming language-specific packages. The arrows represent the dependency between the packages. It is important to notice that the relation between Context Manager and Adaptability Aspects is double-way but this does not imply in internal packages double-way relations. Sometimes a Context Manager package uses an Adaptability Aspects package to notify about context changes or an Adaptability Aspects package may use a Context Manager to request some information about the context in a certain execution point. Another important observation is that we have a different kind of dependency between Base Application and Adaptability Aspects because the latter is capable of crosscutting the former.

![Fig 3 Adaptability Aspects Architectural Pattern Structure](image)

### 3.4 Metamorphic Architecture Pattern

The multi-agent system should meet the system’s reconfiguration and adaption demands. But, in large, distributed systems, it is hard to design and manage the system development without architecture. And it becomes very important to reuse the architectural and component features at the different levels of realization. An architecture pattern could also give guidelines for system development related to the specific domain.

It has been found that there are six basic agent design patterns involved in the development of metamorphic adaptive multi-agent systems. These agent design patterns can be catalogued as two types. One is the architectural pattern: the Metamorphic Architecture Pattern. The other includes these component or method patterns: Mediator Pattern, Task Decomposition Pattern, Virtual Clustering Pattern, Partial Cloning Pattern, and Prototyping Pattern. The pattern structure is illustrated
Architectural and Design Patterns

in Fig. 4

At the architectural level, the metamorphic pattern can be composed from the low-level component or method patterns. Through interaction of these low-level agent patterns, the adaptive behaviour of the system architecture can be achieved. Following the rules inherited from the agent patterns, the development of multi-agent system can meet the desired goal of reconfigurability.

Known Uses:
Metamorph: an Adaptive multi-agent Architecture for Advanced Manufacturing Systems [22]

![Fig. 4 Structure in the Metamorphic Architectural Pattern](image)

4. Design Patterns for Autonomic Systems

Gomaa [3] introduced a set of design patterns for dynamically reconfiguring specific types of software architectures. These design patterns leverage the concept of dynamic change management by specifying the behavior required to dynamically reconfigure master/slave, server/client, centralized, and decentralized architectures. Most importantly, Gomaa’s reconfiguration patterns identify when it is safe to perform a reconfiguration based on the application’s architecture. To this end, they used hierarchical UML state diagram templates to depict, at a high level of abstraction, the behavior required to reconfigure these system architectures. While these reconfiguration design patterns provide a valuable reference for developers implementing dynamically adaptive systems, their contents are not organized in a template format, such as Gamma’s design patterns [25]. Moreover, the set of reconfiguration design patterns are neither presented, nor integrated, within the context of an adaptive system comprising monitoring and decision-making processes. Table 1 shows the Design patterns identified from literature and open sources based on Self-configuration, Self-healing, Self-optimization and Self-protection characteristics of the Autonomous systems
<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Description</th>
<th>Selected Sources and Known Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sensor Factory</strong></td>
<td>Deploy sensors across a distributed infrastructure and probe components</td>
<td>Resource Monitoring for network-aware applications [26], Rainbow Adaptation Framework [5, 27],</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A Distributed Monitoring Service Architecture (MonALISA) - via SNMP [28], SNMP3-Agent [29].</td>
</tr>
<tr>
<td><strong>Reactive Monitoring</strong></td>
<td>Perform introspection on a component and dynamically alter a sensor’s behavior</td>
<td>Reflection Design Pattern [27], InSCT (A generic instrumentation framework for collecting dynamic information) [31], Adaptive Exception Monitor [32], Java Reflection In Action [33].</td>
</tr>
<tr>
<td><strong>Content-based Routing</strong></td>
<td>Route monitoring information based on the content of the message</td>
<td>Rainbow Adaptation Framework [27], Senia Routing [34], JAMM Monitoring System [35], Rebeca [36].</td>
</tr>
<tr>
<td><strong>Case-based Reasoning</strong></td>
<td>Rule-based approach to selecting a reconfiguration plan</td>
<td>Decentralized self-adaptive component-based system [38], Rainbow Adaptation Framework [27], Architectural Approach to Autonomic Computing [39], Earth Management Application (Ontology-based mobile agents) [40], Kinetics eXtreme (KX) Framework [41].</td>
</tr>
<tr>
<td><strong>Divide &amp; Conquer</strong></td>
<td>Systematically decompose a complex reconfiguration plan into simpler reconfiguration plans</td>
<td>Task Decomposition [42], Rainbow Adaptation Framework [27, 24], Care-O-Bot II (uses metric-FF) [43, 44], An Architectural Approach to Autonomic Computing [39], Simple Hierarchical Ordered Planner (SHOP2) [45], Proactive Control, Monitoring and Maintenance (PCMM) Modules for Autonomous Systems [46], Task Control Architecture (TAC) for Mobile Robots [47].</td>
</tr>
<tr>
<td><strong>Adaptation Detector</strong></td>
<td>Interpret monitoring data and determine when an adaptation is required</td>
<td>SmartEvents (part of XUES) [48], PBX - Design Patterns for Software Health Monitoring [49], Java Agents for Monitoring and Management (JAMM) - event gateway [55], Kinetics eXtreme (KX) [41].</td>
</tr>
<tr>
<td><strong>Architectural-based</strong></td>
<td>Provide an architecture-based approach for selecting reconfiguration plans</td>
<td>Rainbow Adaptation Framework [27, 24], MADAM [50], Distributed Configuration Routing (DCR) [41], C2 [52], Kinetics eXtreme (KX) [41].</td>
</tr>
<tr>
<td><strong>Tradeoff-based</strong></td>
<td>Systematically select a reconfiguration plan that best balances multiple objectives</td>
<td>Rainbow Adaptation Framework [27], Unity (Autonomic Prototype by IBM) [54], MADAM [50], Utility Based Allocation [56].</td>
</tr>
<tr>
<td><strong>Component Insertion</strong></td>
<td>Safely insert and initialize a component at run time</td>
<td>Zneus.com - Rainbow framework [16], Software Reconfiguration Patterns [2], Monitor - Dynamic Reconfiguration in Distributed Systems [57], Evolving Philosophers - Dynamic Change Management [58].</td>
</tr>
<tr>
<td><strong>Component Removal</strong></td>
<td>Safely remove a component at run time</td>
<td>Zneus.com - Rainbow framework [53], Software Reconfiguration Patterns [2], Evolving Philosophers - Dynamic Change Management [58].</td>
</tr>
<tr>
<td><strong>Server Reconfiguration</strong></td>
<td>Safely reconfigure a server - client component architecture at run time</td>
<td>Zneus.com - Rainbow framework [53], Server/Client Reconfiguration Pattern [2].</td>
</tr>
<tr>
<td><strong>Decentralized Reconfiguration</strong></td>
<td>Safely insert and remove components from decentralized component architecture at run time.</td>
<td>Unity - IBM Autonomic System [54], Software Reconfiguration Patterns [2], Evolving Philosophers - Dynamic Change Management [58].</td>
</tr>
</tbody>
</table>
5. Conclusion and Future Scope:

This research is aimed at providing a strong software engineering foundation for autonomic based system development. This paper describes the autonomic design patterns; however, it couldn't address how they can be used to construct adaptive and autonomic systems is the immediate Future Scope. Several directions for future work are possible. First, additional design patterns can be identified from Agent Orient Deign and Aspect Oriented Design. Second, additional design patterns for adaptation and Autonomic computing could be identified and integrated with the set of design patterns listed in this paper. Third, we could examine how these design patterns can be inserted into a non-adaptive application through the use of aspect-oriented techniques

References:


[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.


GUIDELINES FOR CONTRIBUTORS

➢ Manuscripts may be submitted in triplicate in A4 size Hardcopy along with a CD containing softcopy. Softcopy may be in MS-word (.doc) format single column in Times New Roman 12 point for running matter, not exceeding seven pages. Details of Title, authors, addresses, e-mail address, etc., may be mentioned clearly. For details on Research paper format, one may refer to klce.ac.in/ijst.htm.

➢ A lot of expenditure is involved in processing the paper. To meet the part of expenditure, each Research paper may be sent along with processing fee of Rs.1000/- in the form of a DD in favor of “INTERNATIONAL JOURNAL OF SYSTEMS & TECHNOLOGIES (IJST), payable at Vijayawada.

➢ Research paper along with processing fee DD for Rs 1000/- is to be sent to the Chief Editor, IJST.

SUBSCRIPTION DETAILS OF THE JOURNAL

➢ The Annual Subscription of International Journal of Systems & Technologies is Rs.1, 250/- ($60).

➢ 20% discount is admissible to Educational Institutions.

➢ Payments in respect of Subscription must be sent by Demand Draft, in favor of “INTERNATIONAL JOURNAL OF SYSTEMS & TECHNOLOGIES, payable at Vijayawada, Andhra Pradesh, India.

Address for Communication:

The Chief Editor, IJST,
K L University,
Green Fields, Vaddeswaram,
GUNTUR – 522 502.