Pattern-based Software Design and Adaptation

Hassan Gomaa
Department of Computer Science
George Mason University
Fairfax, Virginia 22030, USA
hgomaa@gmu.edu

***Abstract* - This paper describes how software architectural design patterns can be used to build software systems and how software adaptation patterns can be used to dynamically adapt them at run-time. The software architectural design patterns consist of architectural structure patterns and architectural communication patterns. This paper also describes the concept of software adaptation patterns and how they can be used in software system adaptation**.

Keywords - Software design; Unified Modeling Language (UML); software architectural design patterns; software adaptation.

I. INTRODUCTION

Software design methods have advanced considerably over the past three decades from structured methods for centralized systems to advanced design methods for distributed applications and product lines. Recent developments are the emergence of component-based design, software product line design, service-oriented architectures, and applying software architectural and design patterns in the design process.

This paper describes how software architectural patterns can be used to help build and evolve software applications. The approach involves integrating software architectural patterns into a model-driven software development process. The patterns consist of architectural structure patterns and architectural communication patterns. This paper then describes how software architectural patterns can be used for software adaptation.

The paper is organized as follows. Section II describes software architectural design patterns. Section III describes how these software architectural design patterns are applied and integrated to create and evolve software architectures for software applications. Section IV describes the role of software adaptation in software development. Section V describes the concept of software adaptation patterns while Section VI describes several software adaptation patterns that have been developed. Section VII describes the process of run-time adaptive change management for software adaptation patterns. Section VIII provides concluding remarks.

II. ARCHITECTURAL DESIGN PATTERNS

Software architectural patterns [2, 10] provide the skeleton or template for the overall software architecture or high-level design of an application. These include widely used architectures [1] such as client/server and layered architectures. Design patterns [3] address smaller reusable designs than architectural patterns, such as the structure of subsystems within a system. The description is in terms of communicating objects and classes customized to solve a general design problem in a particular context.

Basing a candidate software architecture on one or more software architectural patterns helps in designing the original architecture as well as evolving the architecture. This is because the adaptation and evolutionary properties of architectural patterns can also be studied and this assists with an architecture-centric evolution approach [7].

There are two main categories of software architectural patterns [10]. Architectural structure patterns address the static structure of the software architecture. Architectural communication patterns address the message communication among distributed components of the software architecture.

Most software systems can be based on well understood software architectures. For example, the client/server software architecture is prevalent in many software applications. There is the basic client/service pattern, with one service and many clients. An example of this pattern is given in Figure 1 in which an ATM Client communicates synchronously with a Banking Service by sending a message and waiting for a response. However, there are also many variations on this theme, such as multiple client / multiple service architectures and brokered client/service architectures. Furthermore, with a client/service pattern, evolution can be introduced by replacing a service with a newer version or adding new services, which are discovered and invoked by clients. New clients can be added that discover services provided by one or more servers.

 ISBN: 978-1-61208-158-8

Many real-time systems [4] provide overall control of the environment through centralized control, decentralized control, or hierarchical control. Each of these control approaches can be modeled using a software architectural pattern. In a centralized control pattern, there is one control component, which executes a state machine. It receives sensor input from input components and controls the external environment via output components. In a centralized control pattern, evolution takes the form of adding or modifying input and/or output components that interact with the control component, which executes a state machine.

In a distributed control pattern, control is distributed among several control components, each of which controls local input and output components (Figure 2). The control components communicate with each other to inform each other of new events, as shown in Figure 2.

Another architectural pattern that is worth considering because of its desirable properties is the layered architecture. A layered architectural pattern allows for ease of extension and contraction [13] because components can be added to or removed from higher layers, which use the services provided by components at lower layers of the architecture.

In addition to the above architectural structure patterns, certain architectural communication patterns [10] also encourage adaptation and evolution. In software architectures, it is often desirable to decouple components. The Broker, Discovery, and Subscription/Notification patterns encourage such decoupling. With the broker patterns, services register with brokers, and clients can then discover new services. Thus a software system can evolve with the addition of new clients and services. A new version of a service can replace an older version and register itself with the broker. Clients communicating via the broker would automatically be connected to the new version of the service.

The Subscription/Notification pattern also decouples the original sender of the message from the recipients of the message, as shown in Figure 3. A client (Operator Interaction) subscribes to receive event notifications from a service (Alarm Handling Service). The Alarm Handling Service will then multicast event notifications, whenever they are received from the Event Monitor, to all subscribing Operator Interaction clients.

III. APPLYING SOFTWARE ARCHITECTURAL PATTERNS

A very important decision is to determine which architectural patterns—in particular, which architectural structure and communication patterns—can be applied in the design of a given software application. It is necessary to decide first which architectural structure patterns can be used for the application and then which architectural communication patterns.

In many applications, architectural patterns, including client/service and control patterns, can be integrated with the layered pattern. Integrating the client/service pattern with the layered pattern involves placing clients at higher layers than services, since clients depend on services. With a centralized control pattern, the control component is placed at a higher layer than the components it controls. With the distributed control pattern, the control components are all at the same level since communication between them is peer-to-peer. With a hierarchical control pattern, since the high level controller sends overall control commands to the lower level control components, it is placed at a higher layer in the hierarchy. Typically, communication patterns are used to facilitate the integration of the architectural structure patterns.

Consider an example of applying and integrating software architectural patterns. A distributed emergency control and monitoring system is to be designed. The layered pattern is applied to facilitate the pattern integration. This system has client/service characteristics in that user interaction clients can request emergency monitoring status and alarm status from emergency monitoring services. This system also has distributed control characteristics, since there are control components at different locations that receive local sensor data, such as fire or smoke sensors, and can output commands to local actuators, e.g., to switch on sirens or sprinkler systems. The system could be constructed by integrating the client/service pattern (see Figure 1) and the distributed control pattern (see Figure 2). Integrating these architectural structure patterns could be achieved by using architectural communication patterns such as a Broker pattern or a Subscription/Notification pattern (see Figure 3).

For this application, the Distributed Controller, Event Monitor, and Operator Interaction components are all clients of the Event Handling Service (see Figure 1) and are therefore placed at higher layers in a layered architecture. The Distributed Controller (of which there are multiple interconnected instances as shown in Figure 2) and Event Monitor components send asynchronous event messages to the Event Handling Service. Operator Interaction uses the subscription/notification pattern (Figure 3) to subscribe to the Event Handling Service, which sends multicast notifications to all subscribing clients for each new event it receives. The integration of these software architectural patterns is depicted in Figure 4.

 ISBN: 978-1-61208-158-8

IV. SOFTWARE ADAPTION

Software adaptation addresses software systems that need to change their behavior during execution. In self-managed and self-healing systems, systems need to monitor the environment and adapt their behavior in response to changes in the environment [12].

Software adaptation can take many forms. It is possible to have a self-managed system which adapts the algorithm it executes based on changes it detects in the external environment. If these algorithms are pre-defined, then the system is adaptive but the software structure and architecture is fixed. The situation is more complex if the adaptation necessitates changes to a software component or more widely to the architecture. In order to differentiate between these different types of adaptation, adaptations can be classified as follows within the context of distributed component-based software architectures:

a) Behavioral adaptation. The system dynamically changes its behavior within its existing structure. There is no change to the system structure or architecture.

b) Component adaptation. Dynamic adaptation involves changing one component with another that has the same interface. The old component has to be dynamically replaced by a new component while the system is executing.

c) Architectural adaptation. The software architecture, consisting of multiple components, has to be modified as a result of the dynamic adaptation. Old components must be dynamically replaced by new components while the system is executing.

Model based adaptation can be used in each of the above forms of dynamic adaptation, although the adaptation challenge is likely to grow progressively from behavioral adaptation through architectural adaptation.

V. CONCEPTS OF SOFTWARE ADAPTATION PATTERNS

The software architecture is composed of distributed software architectural patterns, such as client/server, master/slave, and distributed control patterns (Section III), which describe the software components that constitute the pattern and their interconnections. For each of these architectural patterns, there is a corresponding software adaptation pattern [9], which models how the software components and interconnections can be changed under predefined circumstances, such as replacing one client with another in a client/server pattern, inserting a control component between two other control components in a distributed control pattern, etc.

A software adaptation pattern defines how a set of components that make up an architecture or design pattern dynamically cooperate to change the software configuration to a new configuration given a set of reconfiguration commands. A software adaptation pattern requires state- and scenario-based reconfiguration behavior models to provide for a systematic design approach. The adaptation patterns are described in UML with adaptation integration models (using communication or sequence diagrams) and adaptation state machine models [8, 9].

An adaptation state machine defines the sequence of states a component goes through during dynamic adaptation from a normal operational state to a Passive state (in which it does not initiate any new transactions but completes existing transactions), to a quiescent (idle) state, as shown in Figure 5. Once quiescent, the component is idle and can be removed from the configuration, so that it can be replaced with a different version of the component.

VI. SOFTWARE ADAPTATION PATTERNS

Several adaptation patterns have been developed and are described below:

a) The Master-Slave Adaptation Pattern is based on the Master-Slave pattern [2]. A Master component, which sends commands to slaves and then combines responses, can be removed or replaced from the configuration after the responses from all slave components have been received. Slave components can be removed or replaced after the Master is quiescent.

b) The Centralized Control Adaptation Pattern is based on the Centralized Control pattern, and can be used in real-time control applications [10]. The removal or replacement of any component in the configuration requires the Central Controller to be quiescent.

c) The Client / Service Adaptation Pattern is based on the Client / Service pattern [10]. A client can be added to or removed from the configuration after completing the service request it initiated. A Service can be removed or replaced after completing the current service request.

d) The Decentralized Control Adaptation Pattern is based on the Decentralized Control pattern and can be used in distributed control applications [10]. A control component in this Adaptation Pattern notifies its neighboring control components if it plans to become quiescent. The neighboring components cease to communicate with this component but can continue with other processing.

VII. ADAPTIVE CHANGE MANAGEMENT

Adaptive change management is provided by a change management model [9, 11], which defines the precise steps involved in dynamic reconfiguration to transition from the current software run-time configuration to the new run-time configuration. For each software adaptation

 ISBN: 978-1-61208-158-8

pattern, the change management model describes a process for controlling and sequencing the steps in which the configuration of components in the pattern is changed from the old configuration to the new configuration [5].

A component that needs to be replaced has to stop being active and become quiescent, the components that it communicates with need to stop communicating with it; the component then needs to be unlinked, removed and replaced by the new component, after which the configuration needs to be re-linked and restarted. A dynamic software reconfiguration framework is designed and implemented to initiate and control the steps of the change management model for automatic reconfiguration of the architecture from one run-time configuration to another [5, 8].

For example, if it is necessary to replace one of the control components in Figure 2, the control component would need to transition to a quiescent state in which it has completed its current operation and has notified its neighboring components that it is no longer communicating with them. It can then be removed from the configuration and be replaced with a new component (e.g., with enhanced functionality), which is then activated and resumes execution and interaction with its neighbors.

VIII. CONCLUSIONS

This paper has described how software architectural patterns can be applied and integrated in the design of software applications. The paper has also described how the application can be dynamically adapted at run-time to replace one component with another. It is also possible to create and integrate executable design patterns [14]. Current research is investigating software design and adaptation patterns for software product lines [6], as well as how these patterns can be used to assist with software evolution.

REFERENCES

[1] L. Bass, P. Clements, R. Kazman, “Software Architecture in Practice”, Addison Wesley, Reading MA, Second edition, 2003.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, “Pattern Oriented Software Architecture: A System of Patterns”, John Wiley & Sons, 1996.

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison Wesley, 1995.

[4] H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison-Wesley Object Technology Series, 2000.

[5] H. Gomaa and M. Hussein, “Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures”, Proc. Fourth Working IEEE Conf. on Software Architecture, Oslo, Norway, June, 2004.

[6] H. Gomaa, “Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures”, Addison-Wesley, 2005.

[7] H. Gomaa, “A Software Modeling Odyssey: Designing Evolutionary Architecture-centric Real-Time Systems and Product Lines”, Proc. ACM/IEEE 9th Intl. Conf. on Model-Driven Eng., Lang. and Systems, Springer Verlag LNCS 4199, Pages 1-15, Genova, Italy, Oct. 2006.

[8] H. Gomaa and M. Hussein, Model-Based Software Design and Adaptation, Proc. ACM/IEEE ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, Minneapolis, MN, May 2007.

[9] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, and D. Menascé, Software Adaptation Patterns for Service-Oriented Architectures, Proc ACM Software Applications Conf. (SAC), Sierre, Switzerland, March 2010.

[10] H. Gomaa, “Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures”, Cambridge University Press, February 2011.

[11] J. Kramer and J. Magee, The Evolving Philosophers Problem: Dynamic Change Management, IEEE Trans. on Software Eng., Vol. 16, No. 11, Nov. 1990.

[12] J. Kramer and J. Magee, “Self-Managed Systems: an Architectural Challenge”, Proc Intl. Conf. on Software Engineering, Minneapolis, MN, May 2007.

[13] D. Parnas, "Designing Software for Ease of Extension and Contraction", in Software Fundamentals, edited by D. Hoffman & D. Weiss, Addison Wesley, 2001.

[14] R. Pettit and H. Gomaa, “Modeling Behavioral Design Patterns of Concurrent Objects”, Proc. Intl. Conf. on Software Eng., Shanghai, China, May 2006.

 ISBN: 978-1-61208-158-8

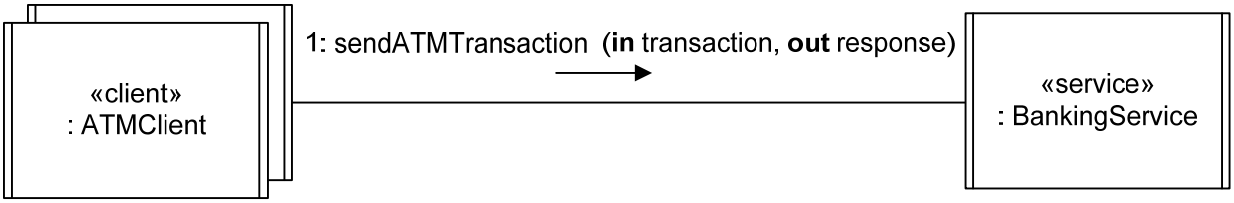

Figure 1: Client/Service pattern

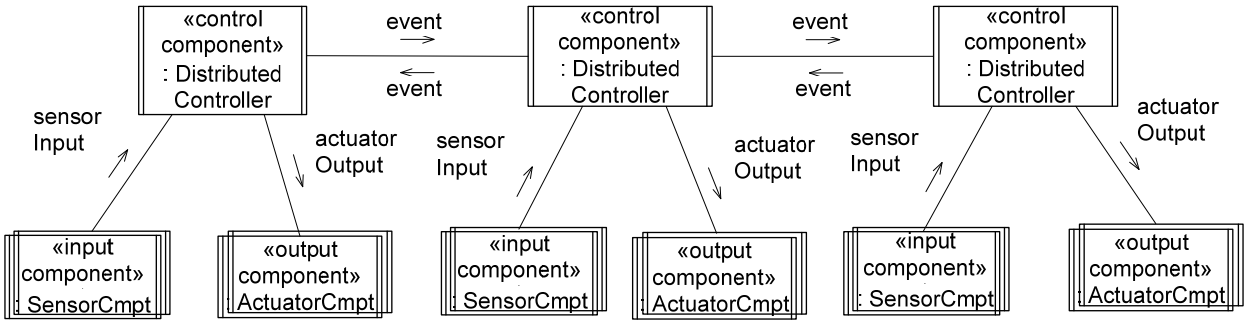

Figure 2: Distributed Control pattern

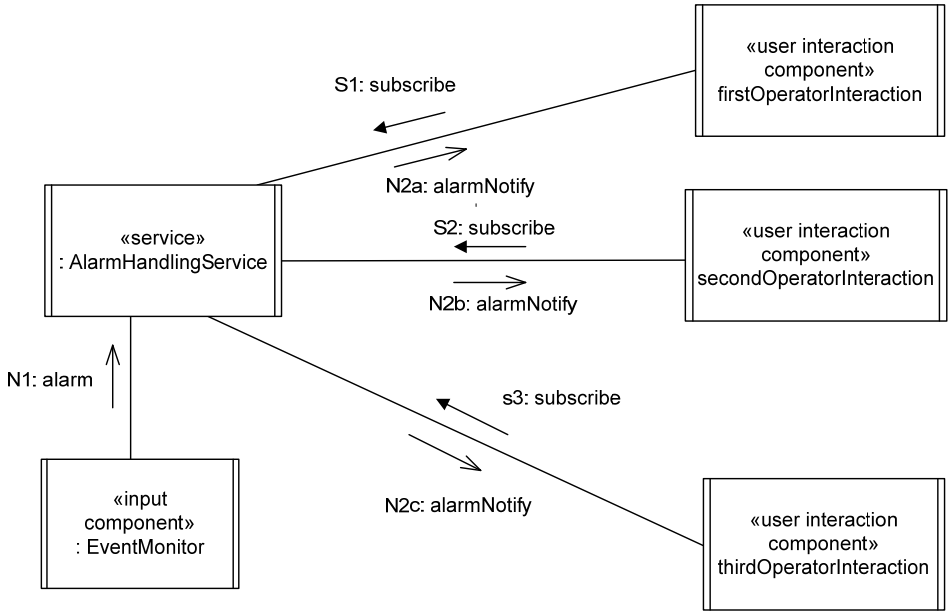

Figure 3: Subscription/Notification pattern

 ISBN: 978-1-61208-158-8

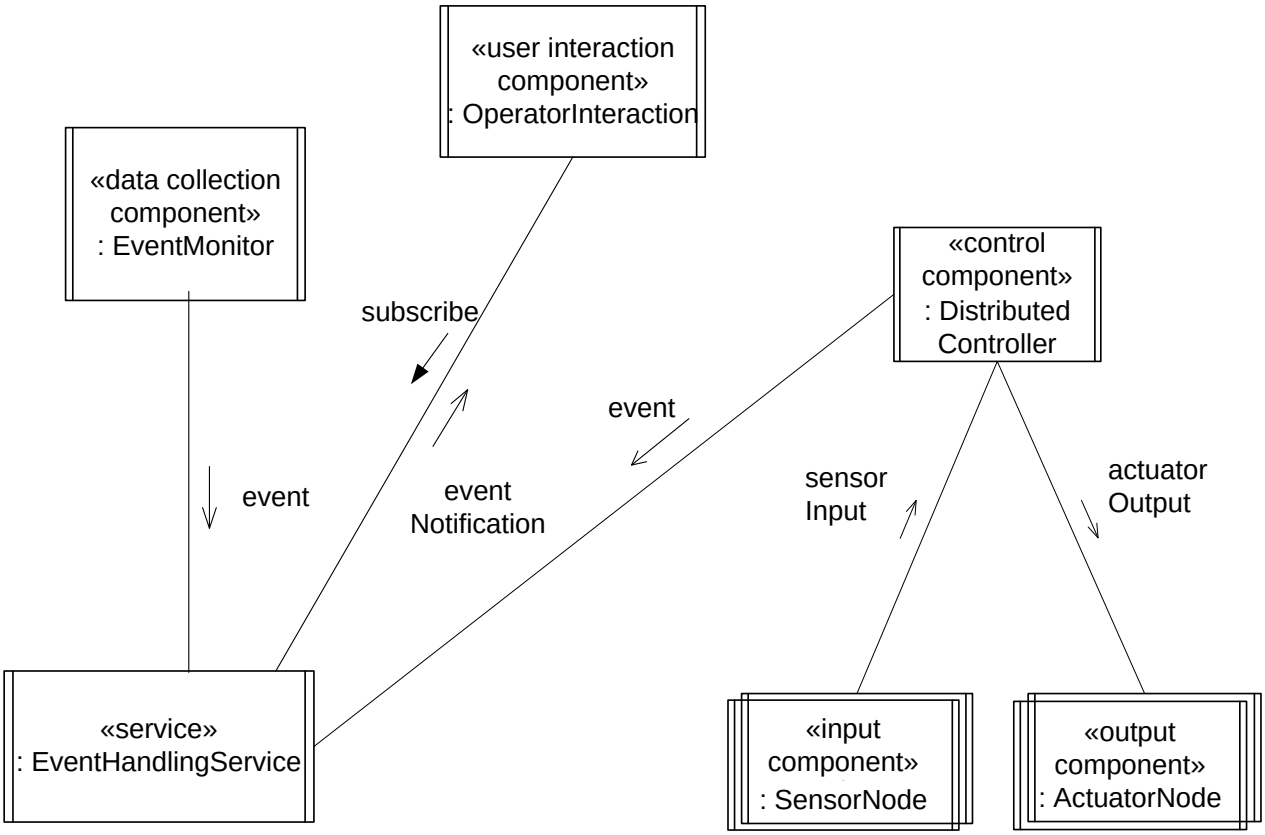

Figure 4: Emergency Monitoring and Control System

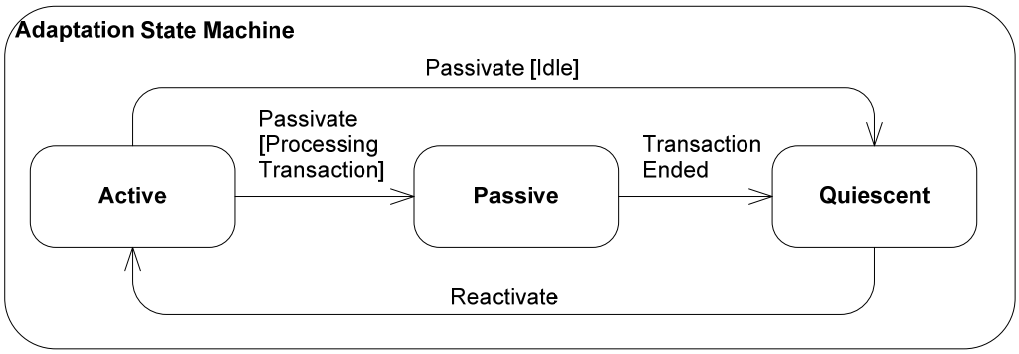

Figure 5: Software Adaptation State Machine