SindhUniv. Res. Jour. (Sci. Ser.) Vol. 51 (03) 541-546 (2019)

http://doi.org/10.26692/sujo/2019.03.85

SINDH UNIVERSITY RESEARCH JOURNAL (SCIENCE SERIES)

Making the Software Architecture Explicit in Java Programs to Enable Dynamic Evolution

G. LAGHARI[++], SAADNIZAMANI[*], SEHRISHNIZAMANI[*], M.MEMON[+], A. H. ABRO[***], M. Y. KOONDHAR[***]

Institute of Mathematics and Computer Science, University of Sindh, Jamshoro, Pakistan

Received 09th November 2018 and Revised 14th June 2019

Abstract: Software architecture helps in developing and understanding software applications at high-level abstraction. Yet, programming languages like Java do not directly support those abstractions. In this paper, we provide the support for architectural abstractions in Java. The support is provided in a middleware that, besides application development and initialization at architectural level, also supports dynamic evolution in the running applications. We demonstrate the use and benefit of the approach with an example scenario.

Keywords: Software Architecture, Dynamic Evolution, Software Components.

1. INTRODUCTION

In software development, it is customary that developers describe the software architecture of the application diagrammatically comprising the boxes and lines. The boxes symbolize computational components while the lines correspond to the interconnection those between components (Garlan and Schmerl 2002). Clearly, the box-line model does not succinctly represent the true architecture of the application. The lines do not adequately specify the interaction type rather it is left to the developers to make their own interpretations. When the system is implemented the mapping between the implementation its initial architecture is lost. Consequently, to support the maintenance of such systems, developers rely on reverse engineering techniques to identifying the key classes in a software system to understand the system (Wang, *et. al.* 2017). Thus, researchers have emphasized the software architecture defined as the organization of computational components and their interconnection through connectors (Shaw and Garlan 1996, Garlan 2000). The components encapsulate functionality and the connectors mediate interactions between them (Taylor, *et al.* 2009). Software architecture, therefore, ensures that system satisfies the requirements including performance, reliability, or interoperability etc. (Garlan 2000). Yet, the programming languages like Java do not provide the mechanisms to directly write software components and connectors.

In this paper, we propose an architecture-centric approach to application development where the application development is specified as composition of components and connectors. As the application is specified as composition of components and connectors, our approach also supports architecture-centric dynamic evolution of applications where the components can be dynamically added, removed, or replaced. This dynamic evolution is supported by maintaining a runtime model of the application architecture.

Dynamic evolution is necessary in applications which are deployed and started but their shutdown or restarting them is undesirable. However, traditional programming lack the support for this. Hence, there exist research effort to support dynamic evolution (Gomaa and Hussein 2004, Oreizy, *et. al.* 1998).

The contributions of this paper include:

- An architecture-centric approach to develop applications in java and enable their dynamic evolution.
- The development of architecture-centric middleware that supports initialization of applications specified at software architectural level and dynamic modifiability of those applications via architectural actions.
- A small set of configuration commands and the component model as well as their implementation.

In the rest of the paper, Section 2 explains the architecture-centric approach. Section 3 provides particular implementation details about architecture-centric middleware, configuration commands, and the component model. Section 4 highlights some related work. While Section 5 finally concludes the paper.

[++]Corresponding Author: gulsher.laghari@usindh.edu.pk

[*]Department of Information Technology, Sindh University Campus Mirpurkhas, Pakistan.

[**]Sindh University Laar Campus @ Badin, Pakistan.

[***]Information Technology Centre, QUEST, Nawabshah, Pakistan.

Note: Part of this paper comes from M.Phil. thesis titled "Policy-based Context-aware Architectural Adaptation in Pervasive Computing" by the first author.

2. ARCHITECTURE-CENTRIC APPROACH

The application development in our approach is based on software architecture—composition of components and connectors.

First, we provide overview of the key concepts. The core architectural element component is the compact executable element with provided and required services. The provided services are the tasks performed by the component while the required services are the tasks provided by other components. Both provided and required services are specified at ports. On the other hand, connectors are architectural elements that correspond to the lines in traditional box-line architecture and mediate the communication between the provided service of one component and the required service of the other component (Abdelkrim, *et. al.* 2009).

Architecture-centric middleware

This middleware is at the core of our approach. It has two essential responsibilities: initialize the application from the initial description of the software architecture of the application and dynamically modify the application.

Application is initially specified as composition of components and connectors (software architecture) using configuration commands (Section 3.2).When this specification is input to the middleware, it loads the necessary components and connectors effectively initializing the application. Additionally, the middleware also builds and maintains a runtime model of the architecture as seen in its initial description. This model is causally connected to the executing units, any change in the model is immediately reflected in the running application.

As the dynamic changes in application essentially change the architecture of the application, thus the model reflects the current architecture. Some examples of the dynamic changes include to add totally new component as a new feature, remove any unwanted components, or replace old component with new component that might implement new strategy or improved algorithm different from the previous implementation.

Those dynamic changes are applied by the middleware by modifying the runtime model via architectural actions such as add, remove, or replace a component.

Notable features of the architecture-centric middleware owing to use of the software architecture centric approach include: providing loose coupling between entities and operating at a high-level of abstraction. The limitations include: a component is able to provide a single required service to another component and a limited set of configuration commands. The middleware might also not work well with large scalable systems.

3. IMPLEMENTATION

In this section, we provide the implementation details of the prototype implementation of architecture-centric middleware, the configuration language, and the component model.

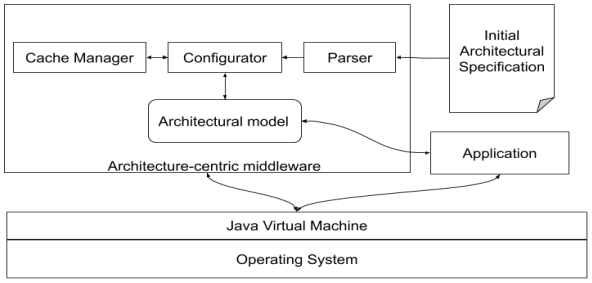

Fig. 1 The architecture-centric middleware.

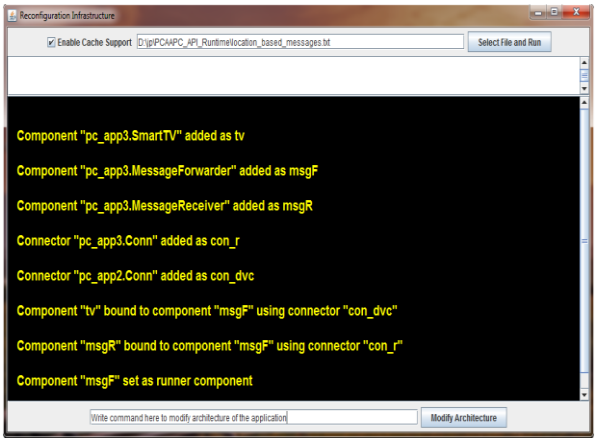

Fig. 2 The graphical user interface.

3.1. Architecture-centric middleware

This architecture-centric middleware is the core that provides the application initialization and dynamic application evolution at architectural level. The middleware is implemented in Java and runs in the java virtual machine (JVM). The overall architecture of the middleware and how the application is executed in it is depicted in **Fig.1**. It also provides a graphical user interface for the user to initialize and modify the application and comprises several sub-components described below.

User Interface

The graphical user interface is organized into four vertically stacked sections (**Fig. 2**). In the top section at right side the button labeled as “Select File and Run” is used to input the file describing the initial software architecture of the application. Once the file is selected,

the application is initialized from this initial description, the complete path of the selected file is displayed in the textbox. The check box with label "Enable Cache Support" if checked before the architecture is loaded, enables the cache support for performance enhancement.

The text pane with white background in following section prints the text from standard output. While the one with black background shows the actions that the middleware performs.

To modify the running application dynamically, the modification command can be typed in the text box in the last section. Then the button click will trigger the reconfiguration actions to modify the architectural model in memory and subsequently modify the application.

Parser

The parser first checks the syntax of configuration commands (Section 3.2) and organizes them as a list to be used by Configurator. When the file describing the initial architecture is selected, the Parser reads the contents and performs line-by-line syntax checks. In case of any syntax errors, the Parser aborts the further process. Then, the Parser prepares the list of commands and passes it to the Configurator.

Similarly, the commands issued at runtime also go through the Parser.

Configurator

The Configurator serves for the two primary roles. First, it is responsible to load the components, initialize them, and finally bind them together to initialize the application. Importantly, it builds the model of the software architecture as a first-class citizen in the memory, which is causally connected to the executing components.

The other key role is to dynamically modify the application. This dynamic modification may be to load new components in memory and replace them with the existing components. These modification actions are applied to the model in the memory and thereby to the running application.

This model of the software architecture of the application, initialized from the initial software architecture description, always represents current architecture of the application. It has references to executing units. Dynamic evolution of the application is carried out by change in the model. Any change in the model is immediately reified in running application also.

Cache Manager

As new components are added in the architectural model in the memory and removed from it, the references to those components are also removed from the model and the components are also garbage collected from the memory. Thus, when the very same component needs to be added into the application again, it is reloaded fresh in memory. Consequently, these component-reloads incur cost in terms of time.

These costs can be avoided by enabling cache support. Then, the Cache Manager maintains references to the components removed from the model. Though they are removed from the model, yet they reside in memory. If in the future they are needed, their reference is obtained from Cache Manager and they are reused in the application without requiring reload.

3.2. Configuration Commands

The mechanism for dynamic evolution of the application in architecture-centric middleware is based on software architecture where the application initial application composition as well as dynamic composition of components is carried out via software architecture. Initially, the composition (specification) of the components is expressed via the configuration commands that come as part of the middleware. The initial specification specifies the components and the connectors used as well as their interconnection. At runtime when the application needs to evolve, the software architecture of the application represented in the model is recomposed to reflect the new behavior.

The configuration commands are highly declarative in nature. The provided list of those commands is small yet sufficient to compose and recompose the application.

The commands and their description follows.

- *Add command.* This command is used to add the components as well as connectors in the application architecture. This command can be used both to specify the composition of the initial architecture and to modify the application architecture at application runtime. The syntax of the command is as follows.

add component *className* as *identifier*

Where the `add`, `component`, and `as` are keywords. The *class Name*needs to be the fully qualified name of the Java class that implements the component. The *identifier* is the reference to a component in model.

Likewise, the syntax to add a connector is as follows.

add connector *className* as *identifier*

Where the `add`, `connector`, and `as` are keywords. The *`class Name`*needs to be the fully qualified name of the Java class that implements the connector. The `identifier` is the reference to a connector in model.

- *Bind command.* This command specifies the interconnections of components through connectors. It binds together two components with a connector. Where one component provides the service and another component requires that service. The connector is the intermediary for the communication between provider and receiver components. The command syntax is as follows.

bind *identifier1* at *port1* to *identifier2* at *port2* using *identifier3*

Where the `bind`, `at`, and `using` are keywords. The `identifier1` is the reference to the component providing the required service at `port1`.The `identifier2` is the reference to the component requiring, at `port2`, the provided service. The `identifier3` is the reference to the connector to be used as intermediary for communication between provider component and the component requiring the service.

- *Replace command.* This is the command primarily providing the dynamic evolution of the application architecture. This essentially replaces the old components with new components, thus effectively changing the behavior of running application. The command syntax is as follows.

replace component *identifier1* with *identifier2*

Where the `replace`, `component`, and `with` are keywords. The `identifier1` is the reference to a component to be replaced and`identifier2`is the reference to the new component.

Start command. This command is intended to execute the application. Application entry point is the component that implements I Runner. The command syntax is as follows.

start *identifier*

Where the `start` is keyword and `identifier` is refers to the startup component.

Component Model

As the applications are composition of software components and connectors, they need to be modeled.

Both the component and connector are first class entities. The components provide the core services. Each component implements a particular task. Some components can provide services while others can require them. On the other hand, a connector facilitates the interaction between components. A component is supposed to export its service through provided interface and use external services at the required interface. Currently, the component is able to provide a single service, yet it can require many services as per need.

A connector has two interfaces, the provider component is plugged in at one interface while the other component requiring the service is plugged in at another interface. The connector facilitates the communication through method invocations via connector. The middleware accompanies a Java API with basic interfaces and classes to implement both components and connectors. The details of these are provided in next section.

Component Model API

The API is collection of some basic interfaces and classes which are written in Java.

- *Component interface and Class.* The interface to implement in a component class is `I Component`. It has signatures for following methods.

- `voidinitialize().`

This method is to be used for component initialization such as connecting to the database server etc. are accomplished.

- `Output do Required(Stringport, Input in).`

By implementing this method, a component can provide service to other components. Where `port` is connection point and `in` is an object of type `Input`. `Input` is used to pass the data to the method. Similarly, the return type `Output` models data to be returned back by the method. Both `Input` and `Output` are interfaces.

- `void do Provided(String port).`

Similar to previous method, in this method a component can use services of another component implemented in `do Required`.

- `void set Connector(I Connector connector, String port).`

This method is used to bind the component and connector.

- `Hashtable<String, IConnector>getConnectors().`

This method can be used to get a list of all theconnectors where the component is plugged in.

Moreover, the API, also comes with a default implementation of `I Component` as a `Component` class.

- *Connector Interface and Class.* The interface to implement the connector is `I Connector`. It has signatures for following methods.

- `void setProvided(IComponent provided).`

Using this method, the provider component is plugged in to the connector.

- `setRequireded(IComponent required).`

Likewise, the component requiring the service is plugged in to the connector using this method.

- `I Component get Required().`

The reference to the component requiring the service can be acquired via this method.

- `I Component get Provided().`

Likewise, the reference to the provider component can be acquired via this method.

- `Output do Required(String port, Input in).`

This method delegates the calls from one component to `do Required(String port, Input in)` in another component.

- `void do Provided(String port).`

Similarly, this method delegates the calls from one component to `do Provided(String port)` in another component.

Moreover, the API, also comes with a default implementation of `I Connector` as a `Connector` class. This default implementation only delegates massage passes between components, though any protocol for communication can be enforced.

```
1. add component my.components.SmartTV as tv
2. add component my.components.Forwarder as forwarder
3. add component my.components.Receiver as receiver
4. add connector middleware.api.connectors.Connector as receiverConnector
5. add connector middleware.api.connectors.Connector as deviceConnector
6. bind tv at devicePort to forwardPort at devicePort using deviceConnector
7. bind receiver at receiverPort to devicePort at receiverPort using receiver Connector
8. start forwarder
```

Fig. 2 Initial architecture of example application.

Input and Output Interfaces. The interfaces provide the pertinent tags to the objects that need to be passed to or returned from `doRequired(String port, Input in)`.

- *IRunner Interface*. This interface extends the `Runnable` interface from Java API. This is desirable for the component that needs to get control after the application initialization.

Example Application

Here we describe an example scenario and discuss how it can be implemented in architecture-centric approach and how it can be evolved.

Consider an application of message delivery based on the location where user's messages from remote service are presented on device preferably selected by the user. The user might select the mobile device for message or select TV.

In this application, there are three software components. `Message Receiver` component to receive the messages from remote service has only one provided interface where it provides the messages received from remote service. `Message Forwarder` component retrieves messages from `Message Receiver` component to send them to the selected device—`SmartTV` or `SmartPhone`. This component has two required interfaces, one interface to retrieve the messages from provided interface of `Message Receiver` and another interface to send retrieved messages to provided interface of the selected device. Lastly, third component represents the device. **(Fig. 3)** shows the architectural diagram of the application. Initially the application architecture can be described as shown in **(Fig.4).**

Fig. 3 Diagram of message delivery application.

When the application is initialized at executing, the messages are displayed on smart TV. If they need to be displayed on smart phone service, the component for smart phone can be added dynamically to replace the smart TV component. Following two command are needed to do it. This dynamic change is shown diagrammatically in **Fig. 5.** `add component my. components. Smart Phone as phone replace component tv with phone`

This dynamic change does not require to halt the system and restart it again, rather it is applied while the application is running.

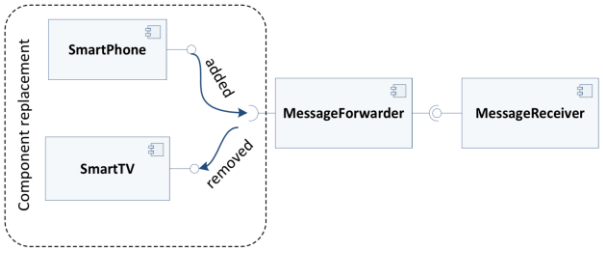

Fig. 5 Dynamic change in message delivery application.

4. RELATED WORK

This section provides some related work on architecture-based development.

The C2-style (Taylor, *et al.* 1995) approach to runtime software evolution defines the system as configuration of software components and connectors (Oreizy, *et. al.*1998). The event-based and layered style exploits connectors to mediate communication between components. The runtime modification in applications is

done with architectural changes that include add, remove, and replace components to reconfigure the application.

Building upon this approach, also outline a software architectural framework for software self-adaptation (Oreizy, *et al.* 1999). Adaptation management monitors the running application, its operating environment, and plans corresponding changes to be applied.

ArchJava, an extension to Java, provides Java like language constructs to support application development at architectural level with seamless mapping between software architecture of the application and its implementation (Aldrich, *et. al.* 2002). Software architecture is specified in the form of composite components comprising subcomponents connected with one another. Components are connected together with connect construct effectively binding required and provided methods. Hence, connectors are not first-class entities. As ArchJava provides a compiler to translate the architectural definition into implementation, thus it does not support dynamic evolution of the application, also it does not have even mechanisms to remove components rather relies on garbage collector.

SOFA 2.0 component system build on SOFA and provides component model with a specific goal to support dynamic reconfiguration meaning dynamic modification of application architecture (Bures, *et. al.* 2006). SOFA 2.0 provides language independent abstraction yet generates implementation code in Java. The application executes in distributed environment comprising of deployment docks, which are containers containing JVM and SOFA 2.0 runtime. Dynamic reconfiguration is mainly the dynamic update of a component in terms of component replacement. SOFA 2.0, however, favors to reflect dynamic reconfiguration at the design time.

5. CONCLUSION

Software architecture provides the basis to develop and understand software applications. Conventionally, software architecture has been described diagrammatically as box-lines model. This box-line schematic does not provide true mapping of the architecture and implementation.

To overcome this, the research on software architecture emerged which views the software architecture as configuration or composition of components at high-level of abstraction. However, these notions are not directly supported in programming languages.

In this paper, we have provided an architecture-centric approach to application development in Java. Owing to the use of high-level architectural abstractions, this also supports dynamic evolution of applications while they are executing. We achieved this with the help of an architecture-centric middleware, a component model, and small set of configuration commands. We also demonstrated the use of this approach by building and modifying an example application at architectural level. Initial experience indicates that this is a viable approach at small scale projects.

REFERENCES:

Abdelkrim, A., and M. Oussalah. (2009) First-Class Connectors to Support Systematic Construction of Hierarchical Software Architecture. The Journal of Object Technology, Chair of Software Engineering, 8 (7), 107-130.

Aldrich, J. and D. Notkin, (2002). ArchJava: connecting software architecture to implementation. In Proceedings of the 24th International Conference on Software Engineering. ICSE, 187-197. IEEE.

Bures, T., P. Hnetynka, and F. Plasil, (2006). Sofa 2.0: Balancing advanced features in a hierarchical component model. In Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06) 40-48. IEEE.

Garlan, D. (2000). Software architecture: a roadmap. Proceedings of the Conference on The Future of Software Engineering. Limerick, Ireland, ACM: 91-101.

Garlan, D. and B. Schmerl (2002). Model-based adaptation for self-healing systems. Proceedings of the first workshop on Self-healing systems. Charleston, South Carolina, ACM: 27-32.

Oreizy, P., M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, D. S. Rosenblum and A. L. Wolf (1999). "An Architecture-Based Approach to Self-Adaptive Software." IEEE Intelligent Systems 14(3): 54-62.

Shaw, M. and D. Garlan (1996). Software architecture: perspectives on an emerging discipline, Prentice Hall Englewood Cliffs.

Taylor, R. N., N. Medvidovic and E. M. Dashofy (2009). Software Architecture: Foundations, Theory, and Practice, Wiley Publishing.

Taylor, R. N., N. Medvidovic, K. M. Anderson, J. E. James Whitehead and J. E. Robbins (1995). A component- and message-based architectural style for GUI software. Proceedings of the 17th international conference on Software Engineering. Seattle, Washington, United States, ACM: 295-304.

Wang, J., Y. Yang, and W. Su, (2017). Identifying key classes of object-oriented software based on software complex network. *2nd International Conference on System Reliability and Safety (ICSRS)* 444-449. IEEE.