Evaluating the FRACTAL Component Model in a Distributed Chat Application System

Devadatta Sadhu
Department of Computer Science
Colorado State University
Fort Collins, CO 80533
(970) 231-5997
sadhu@cs.colostate.edu

ABSTRACT

Adaptive software has evolved as a separate discipline, owing to the need for software systems and applications that adapt to changes in context they are deployed in. Component Based Software is often considered a suitable paradigm to engineer adaptive software. The FRACTAL Component Model provides an open set of control capabilities that may facilitate the incorporation of adaptive features into an application. This paper presents an evaluation of the FRACTAL Component Model by re-engineering an existing Chat Room Application. I port this application to JULIA, a Java implementation of FRACTAL, and then adaptive features are added to the application. In this paper, I evaluate the capabilities of this middleware in terms of how it supports dynamic incorporation of adaptive features in existing applications. For the adaptations it supports, our intention is to find out the ease it provides to add the new adaptations. I have used design metrics for the purpose of measuring the effectiveness of this Component Model.

Keywords

Component-based programming; Levels of Control; Controllers; JULIA; Maven; FRACTAL ADL

7. INTRODUCTION

An adaptive software system is a type of specialized software system that is created to respond to changes in the needs or desires of the user. The key element of adaptive system is flexibility. An adaptive application may be required to support dynamic re-configurations of some features of the application at runtime, fluctuations in the application contexts, and the applications internal state. The system may choose suitable variants of services to better adapt to such situations. These variants may provide better quality of service, offer new services which are not available in the previous context, or discard services that are no longer useful or cause hindrances.

Component-based software engineering (CBSE) is a branch of software engineering which emphasizes the separation of concerns in respect of the wide-ranging functionality available throughout a given software system. Component-based development is used to build and maintain software systems by using existing software components, such as a web service or a module that encapsulates a set of related functions [9].

Certain characteristics of components are

- Components are capable of performing a task in isolation; i.e. without being composed with other components.
- Components may be developed independently from each other.
- The purpose of composition is to enable cooperation between the constituent components.

Owing to the component characteristics, CBSE facilitates development of adaptive applications to a great extent. The components provide an ease in the addition of adaptive features in an existing application.

However, existing component-based frameworks provide limited support for extension and adaptation of dynamic features. A primary reason is the *lack of tailorability of Containers* (in case of Container based implementations of Component Models, such as EJB) [7]. There is no mechanism to configure an EJB container.

FRACTAL provides an alternative to Container-based Component Models. The FRACTAL Component Model consists of an open set of control capabilities which can be deployed as and when needed. The components in FRACTAL are configured and controlled through well-defined interfaces. Hence, FRACTAL is called an ***Open Component Model.***

In this paper, I evaluate the effectiveness of the FRACTAL Component Model in building an adaptive distributed software system. I build a Chat Application system in the Java platform, and then use the FRACTAL to introduce some adaptive features in the Chat Application. My objective is to evaluate how this middleware is conducive in supporting adaptive features in an existing application.

2. THE FRACTAL COMPONENT MODEL

The FRACTAL Component Model is intended to implement, deploy, and manage (i.e., monitor, control and dynamically configure) complex software systems [1].

FRACTAL is extensible. The control features of the FRACTAL components are not defined as a fixed set of specifications that all components must follow. The model allows for levels of control, ranging from no control (base components) to full-fledged introspection and dynamic re-configuration capabilities. This

provides the developer a prerogative in choosing the level of control as per the need of the application.

The main features of the model are [1]:

- Hierarchical components: Components that contain sub-components
- Shared Components: Sharing feature facilitates resource sharing while maintaining component encapsulation
- Introspection capabilities: Introspection to monitor and control the execution of a running system
- Dynamic re-configuration capabilities: Re-configuration to deploy and dynamically configure a running system

2.1. FRACTAL Features

2.1.1. FRACTAL Components

Figure 1 shows the architecture of the FRACTAL Component Model. A FRACTAL component is composed of a the content, which may recursively contain other sub-components. Each component is surrounded by a membrane that offers infrastructural services to the component. The membrane typically consists of several Controllers that allow the developer to tune the parameters which can alter the behavior of the component and its sub-components, as needed in the application. Controllers can have interceptors that can export the interfaces of a subcomponent its parent component. They can intercept the incoming and outgoing messages and can handle operation invocations through certain handlers, e.g. pre-handler and post-handler. A component can be primitive (a component that does not expose its content, but has at least one control interface) or, composite (a component that exposes its contents).

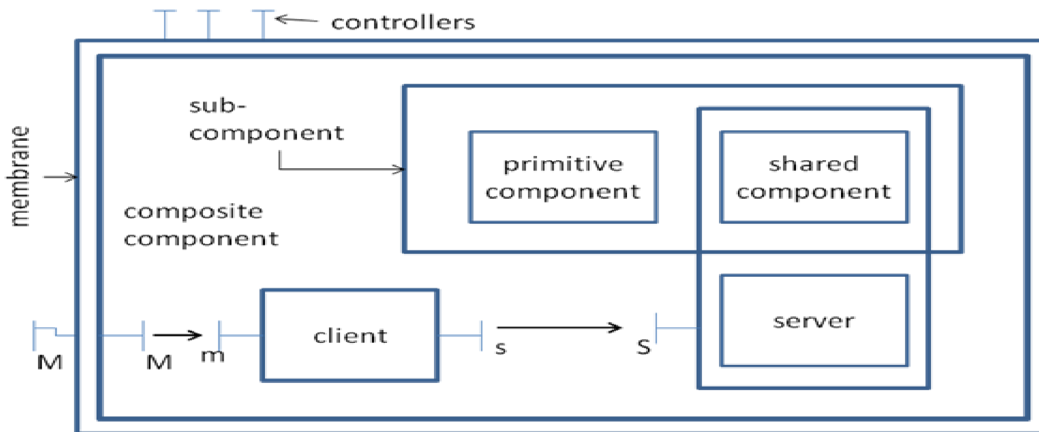

Figure 1: Architecture of the FRACTAL Component Model

2.1.2. Bindings

Communication between the FRACTAL components takes place only if their respective interfaces are bound. FRACTAL advocates two types of bindings: primitive bindings and composite bindings. A ***primitive*** binding is a binding between the client interface and the server interface, in the same address space, i.e., both the client and the server are two components within the same parent component. The messages emitted by the client interface should be received by the specified server interface. A ***composite*** binding is a binding between arbitrary numbers of component interfaces. These bindings are built out of a set of components bound by primitive bindings. This is conducive in building communication in a distributed configuration of FRACTAL components. The role of binding corresponds to the connector concept in other component models. It helps to mediate communication between the components [2].

2.1.3. Sharing

An ingenious feature of the FRACTAL component model is sharing of components. This feature turns out to be very helpful to achieve separation of 'aspects' in component-based applications. By this feature, the state of a component can be shared between two other components. [2].

2.2. FRACTAL Levels Of Control

The primary feature of the FRACTAL Component Model is that it does not enforce a fixed and pre-determined set of controls that need to be implemented in all the components. Instead, it allows different forms of membranes, each having individual set of controllers and interceptors. The implementation of the controlling features in FRACTAL can be classified into certain levels of controls.

At the lowest level, a FRACTAL Component does not provide any introspection or re-configuration functionalities. Such a component is called a *Base Component*, and is similar to Plain Old Java Objects (POJO).

However, even at the base level, the concrete implementation classes are separated from the interfaces. These base components are used by invoking operations in the component interfaces, which are bound by the Binders [3].

In the next level of control, FRACTAL Component Model supports Introspection functionality. The interfaces of a

component can be introspected in two ways: Component Introspection and Interface Introspection [1].

In the next higher level of control, the FRACTAL Component Model provides the features needed for dynamic re-configuration of application features. At this level, several types of controllers can be implemented to achieve the different levels of controls.

The Standard Controllers are:

- **Binding Control:** The BindingController interface is implemented to bind and unbind the client interfaces to other components through primitive bindings, as explained in Section 2.1.2
- **Super Control:** The SuperController can be used to get the sub-components of a Super Component.
- **Content Control:** The ContentController interface can be used to add and remove the sub-components in a component.
- **Attribute Control:** The AttributeController interface can be used to configure the properties of a component.
- **LifeCycle Control:** The LifeCycleController is implemented to support dynamic reconfiguration of the attributes of a component. This Controller provides minimal support to implement the above mentioned other controllers.

In the Final Level of Control, instantiation of new components can be handled. New components can be created by other component called ***Component Factories***. There are two kinds of Component Factories supported by the FRACTAL Model: the GenericFactory and the StandardFactory. Several kinds of components can be created by the GenericFactory Interface, while the StandardFactory Interface can create one particular type of components.

While building adaptive features in an application, the developer can use his own prerogative in choosing the optimal level of controls fitted for the application under consideration.

3. DISTRIBUTED CHAT APPLICATION SYSTEM

I developed a Distributed Chat Application System using Java RMI. The Chat Application consists of a centralized chat server and several chat clients. The application is structured in a two-tier hierarchy. The clients are grouped in different groups, such that the messages are sent through a level of hierarchy. Fault tolerance issues such as, Group Leader re-election when the existing Group Leader dies, load rebalancing of chat groups, etc are handled. I used this Chat Application System to build adaptive features on it, supported by the FRACTAL framework, JULIA. To run the application, I used Maven Application tool for FRACTAL.

3.1. Adaptive Features Implemented

I have extended the existing design by introducing the concept of a **Main Server**. This adds a **new level of hierarchy** to the chat communication system. The Main Server hosts a group of servers, which communicates with the Peer Leaders, which in turn, send message to the individual clients. If any one of the servers die, the client chat application does not crash; instead it binds with the Main Server, and communication continue unhindered. Also the load is now distributed to several servers; this, in turn, increases the efficiency of the application.

The Main Server can exist in two states --- Passive and Active:

Passive State of the Main Server: The Main Server monitors the state of the servers running; the servers communicate with the peer servers, while sending messages from one server group to another.

Active State of the Main Server: If a server dies, the clients connected to that server will be immediately bound to the Main Server. This will ensure uninterrupted communication between the clients. Thus Fault Tolerance can be handled smoothly.

The adaptive features implemented in this system are described below:

- **Dynamically Changing Client User Interface**: This is a cosmetic adaptation. If the client sends an encoded message requesting a change in its UI, the server will decode the message and change the client user interface as requested. This is a dynamic adaptation. The change in the user interface takes place as per the client request while chatting.

- **Addition of a New Level in Chat Group Hierarchy**: Maintaining a Main Server which will monitor the state of the Server as well as that of the Clients. The Main Server will be in the Passive State while monitoring the Server State.

- **Dynamic Change of the Main Server's State from Passive to Active:** If a Server dies, the Main Server will immediately change its state from Passive to Active. The clients connected to the crashed server will be bound to the Main Server. This will ensure unhampered communication between the clients, even when the Server dies.

- **Restarting Dead Servers/ Replacing Dead Servers**: If the central server crashes (accidentally or owing to some network problem), there will be a dynamic attempt to restart the server or to replace the dead server by starting a new one, so that the Chat Application runs smoothly.

4. DESIGN OF THE ADAPTIVE CHAT APPLICATION SYSTEM

The design of the Chat Application System with the proposed adaptive features is elaborated in this section

For the adaptations it supports, my intention will be to find out whether it provides the facility to add new adaptations by simply plugging them, rather than refactoring the existent code. I will also evaluate how well this platform helps to modularize the components of the MAPE-K model.

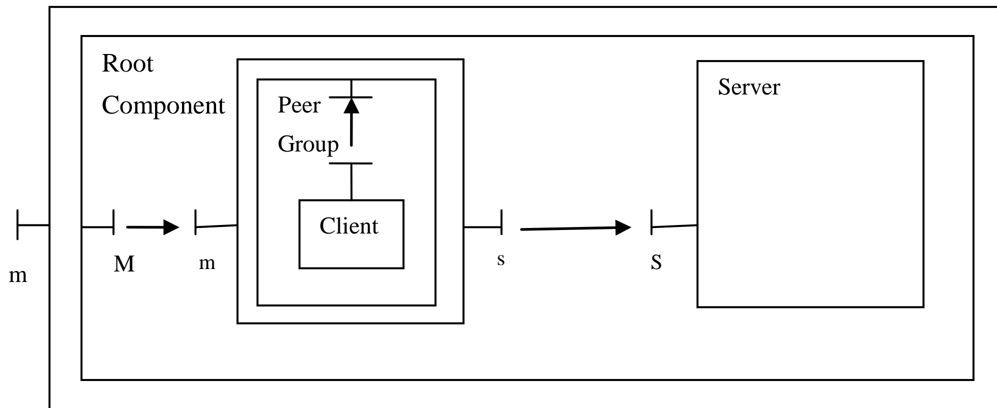

Figure 2: Component Model of the Chat Application

4.1. Component Diagram

Figure 2 exhibits the design adapted for this application.

A super-component, namely Root Component is created. The Server and the Peer Group are created as sub-components of the Super-component. The Clients are created as sub-components of the Peer Group component. The Root has two interfaces: an external interface, with which it can interact with other Root Components, and an internal interface, with which it can bind with its sub-components. The Server has an external interface, Service (S). The Peer Group has two types of interfaces: external interfaces, and internal interfaces to bind with its sub-components. The Peer Group uses one of its external interfaces(s) to bind with the Service interface (S) of the Server. It also binds through its other external interface (m) with the internal interface of the Root Component (M). Each Client sub-component has an external interface, with which it interacts with the internal interface of the Peer Group.

Since the Peer Group is bound with the Main interface of the Root Component, even if the Server dies and it gets unbound with the Service interface, message communication continues owing to its connection with the super-component.

4.2. Design For MAPE-K Abstraction

An abstraction of the MAPE-K model [6] has been intended to achieve through the design decisions. Implementation of the MAPE-K model makes a system component self-managing. The system component uses an automated method to collect the details it needs from the system analyzes those details to determine if something needs to be changed; creates a plan, or a sequence of actions, that specifies the necessary changes; and perform those actions. When these functions are automated, an intelligent control loop is formed that share the knowledge obtained through these actions. Figure 3 shows the architecture of the MAPE-K Model [6].

How the five components of the MAPE-K model, Monitor, Analyze, Plan, Execute and Knowledge are identified in this application are described in this section.

The Root Component can be used to **Monitor** the states of the Server.

If the server dies, the Root Component will **Analyze** the situation, **Plan** the next step and **Execute** accordingly (changing itself from Passive State to Active State).

The dynamic execution takes place based on the **Knowledge** defined in the policy.

The Server will analyze the messages sent by the client and plan the execution accordingly. If it is a simple text, Server will send it to other clients as message communication.

If the message is a special command (client requesting UI change), server will decode the special text and change the UI of the clients dynamically in the distributed system.

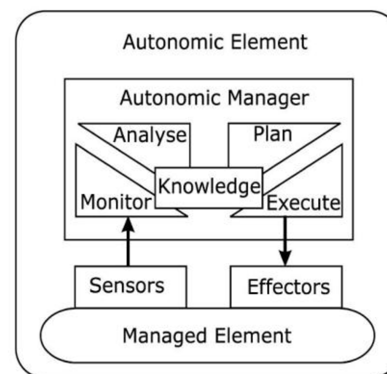

Figure3: MAPE-K Model

5. IMPLEMENTATION OF THE ADAPTIVE CHAT APPLICATION

I used the Java Version of the FRACTAL API. I port the Chat application in the JULIA framework, the Java implementation of the FRACTAL Membrane. I used the F4E (FRACTAL for Eclipse) plug-in of JULIA to work in the Eclipse workbench.

5.1. Implementation Details

FRACTAL has its own RMI Registry. I used the FRACTAL RMI Registry to build a distributed application. I created the following classes to implement this method:

ServerImpl : Concrete implementation of the Server Class

ClientImpl : Concrete implementation of the Client Class

Service: Server interface with which the Client binds

Main: Root interface with which the Client binds

ServerLauncher: Dynamic re-configuration policies of the Server are defined here

ClientLauncher: Dynamic re-configuration policies of the Client are defined here

In the ServerLauncher Class, I used the NamingService to call an instance of the FRACTAL registry. Next I bind the NamingService with the FRACTAL Bootstrap Component. I instantiated the following FRACTAL Components: **Root** and **Server**. In the ClientLauncher Class, I created a **Client** component. These components are created using the FRACTAL Generic Component Factory. I used the SuperController to create the Root Component. I added the Server and the Client as sub-components of the Root, using the ContentController [addFcSubComponent()].

The Root component has two interfaces: an external interface and an internal interface (Main Interface). The types of the interfaces are obtained from the FRACTAL Type Factory. The Server has an external interface: Service. I attached the Client component with two types of external interfaces: one to bind with the Main Interface of the Root component and another to bind with the Service Interface of the Server. I implemented the binding of the interfaces with the NamingService.

In the ServerImpl Class, I implemented the LifeCycleController. The ServerImpl extends the Service Interface. I applied the BindingController along with its relevant methods [listFc(), lookupFc(), bindFc()] in the ClientImpl. Each client creates a separate thread through Runnable interface. The client binds with the Service Interface of the Server through s interface, and with the Main interface of the Root through m interface.

In the ServerLauncher Class, I handled the adaptive feature of the Server state. If the Server dies, the unbindFc() method for the Client-Service interface is used. However, the client is still bound with the Main interace of the Root. This ensures continual communication between the clients, thereby dynamically handling a fault tolerance issue.

A UML diagram of the design of the adaptive application is shown in Figure 4.

A hashtable is used to store the colors defined as constants in the Java Color index. Now when a client sends a message, the message is decoded in the ClientLauncher at runtime. This ensures dynamic re-configuration.

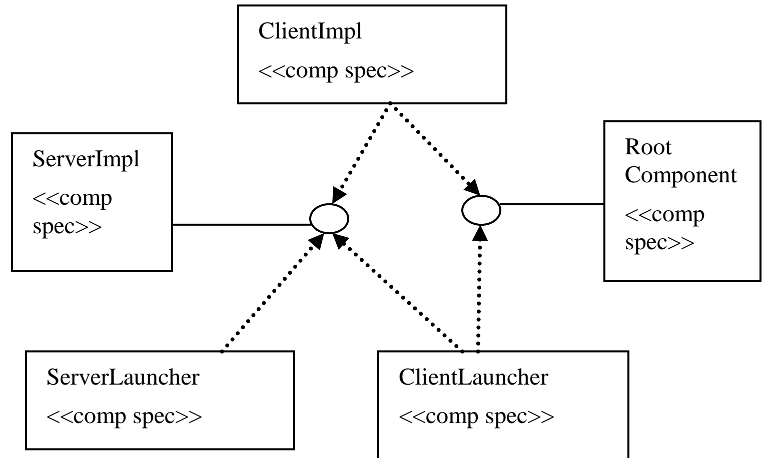

Figure 4: UML diagram showing Component Architecture

If the message stands out as a special text requesting a change in UI, the request is complied accordingly. The message should start with special characters. The color name stored in a substring of the message is then searched in the ColorMap Hashtable. Once the requested color index is found, the UI color is changed accordingly. Similarly, the FONT size and FONT color can be changed dynamically.

5.2. Execution With Maven

I have used Maven tool for FRACTAL to run this adaptive application. I have configured the xml file for Maven called pom.xml (Project Object Model) to specify the bindings. The host and the port can be set through proxies in the pom.xml. Arguments are passed in the xml file to dynamically create unique clients.

The application can be run by these steps:

1. Run the FRACTAL Registry
2. Run the FRACTAL Bootstrapping Component
3. Run the ChatServer
4. Run the ChatClients

The BootStrapping Component acts as a Monitor to introspect the state of the Servers.

6. Comparison Between FRACTAL Model And OO Model

I used the Software Metrics, Eclipse Metrics v1.3.6, to collect data in order to measure the characteristics of the software system

and evaluate how the FRACTAL Component Model has facilitated in improving a design originally developed in the Object Oriented Model platform. I used the Chidamber and Kemerer metrics [8] to measure the design quality achieved after using FRACTAL.The metrics data are collected for the existing Chat Application developed on the OO framework. After porting the application on the FRACTAL CBSE framework, the Metrics are generated again. The data obtained from the Metrics are presented below:

Table 1: Software Metrics Analysis

Metrics	OO Framework	CBSE Framework
Total Lines of Code	1790	2097
Afferent Coupling[1]	6	0
Efferent Coupling[2]	5	4
Lack of Cohesion of Methods[3]	0.977	0.823

It is observed that although the lines of code have increased (adaptive features are implemented), CBSE has improved the quality of the code to some extent. While coupling has decreased (note Afferent Coupling is 0), the cohesion of methods has increased.

7. CONCLUSIONS

From the design metrics, I find that using the FRACTAL middleware, the quality of the code has improved to a great extent. However, the results obtained here is just for single distributed adaptive system. Experiments should be conducted on other distributed adaptive systems as well.

While working with this middleware, I found that it truly provides the developer a choice of specifications from the levels of control to apply as per the requirements of the applications.

The separation of the interfaces and the concrete implementations enhances the adaptability property of a software system. This, in turn, allows dynamic introspection and runtime (and/or deployment time) configurations of the adaptive features.

The platform has its own RMI Registry. This makes it independent of other rmi services (say, Java RMI). The model has its own Generic Factories. The components can be instantiated from these factories, following the factory framework and standards. The Model has a BootStrapping Component, which facilitates the monitoring service within the Model.

FRACLET, an annotation based FRACTAL programming model, can be used for Service based applications.

In spite of all these capabilities listed above, there are limitations of this model.

It does not support Java RMI. I developed my initial application on the Java RMI platform. I had to do major refactoring of the existing code so as to get rid of the Java RMI part and then include FRACTAL RMI Registry.

There is too much of dependency on different execution tools. While FRACTAL ADL depends on ADL files for execution, the FRACTAL RMI & API depends on Maven for running the application.

8. ACKNOWLEDGEMENT

I performed this experimental evaluation in a term project in the Software Engineering course: CS518- Distributed Software Systems Development. I would like to thank my instructor, Dr. Sudipto Ghosh for his guidance.

9. REFERENCES

[1] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq,Vivien Quema,and Jean-Bernard Stefani., “The FRACTAL Component Model and its Support in Java”, SP&E, 36: 1257-1284, 2006.

[2] E.Bruneton et al., “The FRACTAL Component Model”, Object Web Consortium, v2.0-3, Feb, 2004

[3] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V.,Stefani, J. B., "An Open Component Model and Its Support in Java", Proc. of CBSE’04, Edinburgh, UK, May 2004

[4] Thierry Coupaye and Jean-Bernard Stefani, "FRACTAL Component-Based Software Engineering", ECOOP workshop, 2006

[5] The FRACTAL Projects, OW2 Consortium

[6] IBM. Autonomic Computing White Paper, “An Architectural Blueprint for Autonomic Computing”

[7] Pichler R, Ostermann K, Mezini M. “On aspectualizing component models”, Software: Practice and Experience 2003

[8] Chidamber, S.R., Kemerer, C.F., "A metrics suite for object oriented design," Software Engineering, IEEE Transactions on , vol.20, no.6, pp.476-493, Jun 1994

[9] Component-based software engineering, http://en.wikipedia.org/wiki/Component-based_software_engineering

[1] Afferent Coupling: High afferent coupling indicates an object has too much responsibility

[2] Efferent Coupling: High efferent coupling suggests the object isn't independent enough

[3] Lack of Cohesion of Methods: A measure for the number of not connected method pairs in a class representing independent parts having no cohesion