ChefMaster: A Glue Framework for Dynamic Customization of Interacting Components

D Janaki Ram and Chitra Babu*

Distributed Object Systems Lab, Dept. of Computer Science & Engg.,
Indian Institute of Technology Madras,
Chennai - 600 036, India.
{djram@lotus.iitm.ernet.in,chitra@cs.iitm.ernet.in}
http://lotus.iitm.ac.in

Abstract. *Component composition is becoming increasingly popular during the past decade due to its claims of increased productivity and reduction in software development cost. Viewed from a different perspective, composition of components could make the application flexible by providing customized responses to different client requests. Towards this objective, this paper proposes a glue framework, ChefMaster that addresses this issue by coordinating the customization of a set of interacting components and composing them dynamically. In this framework, the behavioural extensions and the composition styles are separated from the component model, and are captured in a Connector Module(CM). CM is a set of scripts which are written using a specific composition language. Each script specifies the way in which the involved components should be customized and composed together. These scripts dictate the correlation among the customization of participating components. Since the scripts are interpreted, ChefMaster provides a customized response, catering individually to each client request by plugging the appropriate behavioural extensions to the group of participating components and composing them during run-time.*

Keywords: Grey box component, Behavioural extension, Composition, Script.

1 Introduction

In recent times, the issue of component composition is gaining more importance, because of the enhanced flexibility and increased productivity associated with it in developing software applications. Components are built in different flavors. Black-box components provide an abstraction, where none of the component internals are visible and only the pre- and post-conditions are specified. This has the advantage of allowing binary composition of components[1]. Further, different programming languages can be used for realizing the component implementations. Unfortunately, black-box specifications are insufficient for extending the behavior of the

* The work of this author was supported in part by a fellowship from Infosys Technologies Ltd., Bangalore.

components. On the other hand, the problem with the white-box components is that they expose too much detail to the deployers and hence create unwanted dependencies. Further, this methodology does not help the component developers in protecting their implementation know-how. Hence, there must be a middle ground between these two extreme ways of building components. A grey-box component is one, which reveals selectively some of its internal workings. This facilitates extension of component behavior[2, 3].
In the context of server side component architectures such as Enterprise Java Beans, it will be desirable, if each individual client request could get a customized response. The integration of the glue framework proposed in [4] with EJB, enabled the clients to customize the server side components. However, the customization was restricted to individual components. Using this framework, it is not possible to coordinate the behavioural extensions of a group of interacting components in a consistent manner.
The major contribution of this paper is the proposal of an augmented glue framework, *ChefMaster*, which uses a set of scripts

- to coordinate the various behavioral extensions for the interacting components to be composed.
- to vary the styles in which the components should be composed.

The rest of the paper is organized as follows. Section 2 presents the Glue object model briefly and also explains how it was used to achieve client-driven customization in the context of EJB architecture. The proposed framework *ChefMaster* is explained in detail in section 3. Section 4 discusses related work and Section 5 concludes the paper and outlines future research directions.

2 Glue Customization Framework

2.1 Glue Model

The aim of the Glue object model[5] is to achieve dynamic object adapation and customization, while composing the objects. The underlying philosophy of Glue object model is to relax the tight coupling between abstraction and encapsulation, in a systematic fashion, to achieve object-level behavioral customization. The rationale behind this philosophy is the observation that whenever an object needs to be used in contexts other than the ones for which it was originally designed, breaking the encapsulation becomes mandatory. It focuses on the need to separate Behavioral eXtension(BeX) and Behavioral Specialization(BeS) aspects in modeling. Glue model is based on the notion that an object exhibits immutable and mutable behavior. A Type-hole which is an interface consisting of a set of method declarations, whose definitions are deferred, captures this mutable behavior. A base class contains the fixed behavior and the Type-hole. A glue class is one which defines the methods declared in a Type-hole. There can be several glue classes that provide varying definitions for the same Type-hole. Dictated by the context, various mutable behaviors can be composed with the single immutable behavior.

These mutable behaviors can be dynamically plugged and unplugged, thus enabling the behavioral extension of a single object during its life-cycle.

The model defines four Type-hole relationships, namely, *in*, *out*, *part-of* and *using*. Each relationship abstracts a specific kind of compositional behavior. An *in* Type-hole enables messages to an object to be intercepted and manipulated at the receiving object's side. An *out* Type-hole enables message interception and manipulation at the sending object's side. They provide incremental definition of a method. A *part-of* Type-hole is used to model objects that are part of another object. The glue object which is in *part-of* relationship, has access to the private data of the base object. A *using* Type-hole relationship lets the base object use the glue object without exposing any of its private data.

2.2 Client Customization in EJB

An EJB container is an environment in which Enterprise JavaBeans execute. Its primary role is to serve as a mediator between an EJB and clients, thus providing all the background services. The client never communicates directly with the bean that encapsulates the business logic. Instead, it indirectly invokes the bean methods through its home interface and its remote interface, whose implementations are automatically provided by the container.

In EJB, the rules governing life-cycle, transactions, security and persistence of the enterprise bean are defined in an associated eXtensible Markup Language(XML) Deployment Descriptor(DD) file. These rules are defined declaratively at deployment time rather than programmatically at development time, and they tell the EJB container how to manage and control the bean. The container provider is responsible for enforcing at run-time, the security policies and transactional attributes defined during deployment, by providing the necessary tools. EJB provides only this level of customization. With this, the client cannot change any server functionality dynamically during run-time. However, allowing client side of the application to customize the server side components will be useful, when the the responsibilities cannot be clearly divided between the server and client. Janaki Ram et.al.,[4] discussed how this additional level of customization can be achieved by integrating glue framework with EJB.

Different roles played by the same bean object and the security policies associated with each bean object are abstracted into Type-holes within the bean class. Variety of role and security policy implementations are abstracted as glue classes, on the client side of the application. Figure 1 depicts this.The *plug* and *unplug* methods included in the augmented remote interface facilitate plugging of appropriate glue objects with the basic bean object, thus enabling the customization of the bean object at run-time. Janaki Ram et.al.,[4] discuss a detailed case study and its implementation details.

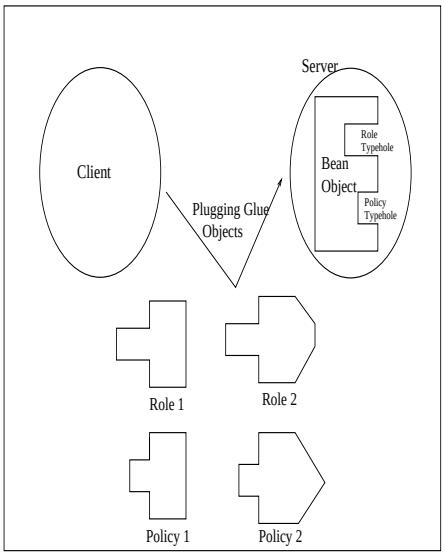

Fig. 1. Glue Customization

3 Dynamic Customization of Interacting Components

Although, the glue framework presented in the previous section facilitates dynamic client-driven customization in the context of EJB architecture, it has the following disadvantage:

- The behavioral extensions for the different bean are independent and disconnected. It cannot capture the dependencies among the behavioral extensions of a set of interacting bean components.

The following section describes the *ChefMaster* framework, whose objective is to make the server customize each client request individually, by coordinating the customization of a group of interacting components.

3.1 ChefMaster Framework

This section presents the *ChefMaster* framework, where the behavioural extensions for the set of interacting components are specified in an external entity called as Connector Module(CM). CM consists of a set of methods written in a specific composition language, where new keywords for plugging the behavioural extensions and operators for composition styles are introduced. A preprocessor converts the composition language code into regular scripting language code. Scripting languages will be more suitable than conventional programming languages for implementing these methods in CM, because of the flexibility offered by them. Such flexibility in composition is possible due to the dynamic typing approach

adopted by the scripting languages. Further, since scripting languages are interpreted, they also reduce the application turnaround time [6]. Each script uses specific operators to indicate the style in which the participating components should be composed.

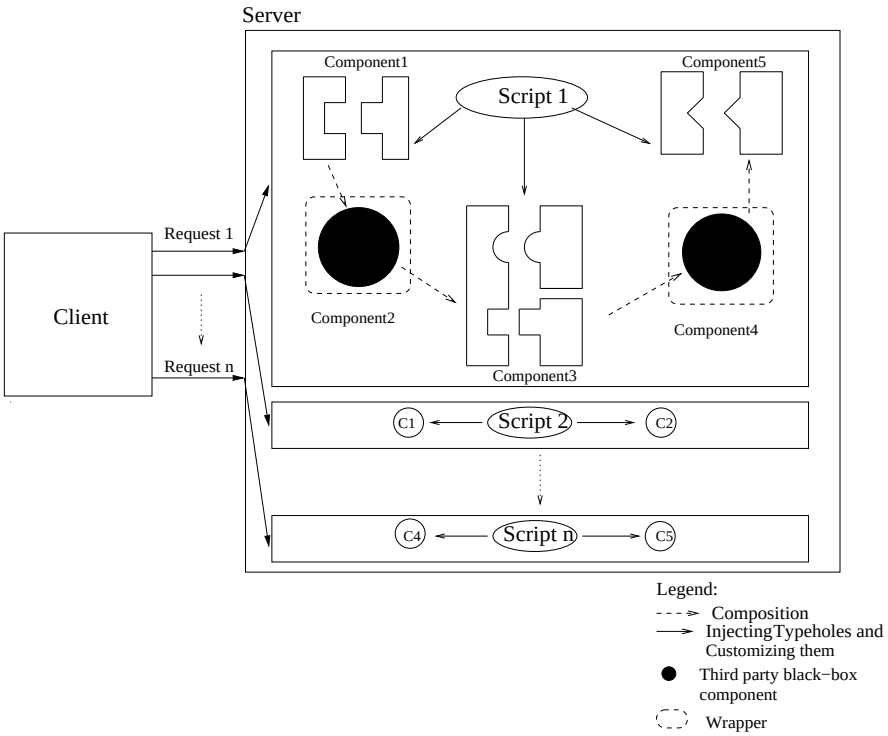

Fig. 2. ChefMaster Framework

Figure 2 pictorially represents the *ChefMaster* framework. The Connector Module essentially consists of a set of methods, where each method is an independent script. The composition style for the components to be composed, are specified in these scripts. Further, these scripts will decide how much "encapsulation breaking" has to be done on the components that it is intending to compose. Accordingly, the behavioural extensions for the set of interacting components will be injected by the scripts. However, this systematic break in encapsulation is possible only with grey-box components. For third party black-box components which are built outside of this framework, only the composition styles can be varied.

Once the client issues a request, the server will dispatch the appropriate script. The scripts will dynamically generate the glue objects that should be plugged to the various participating components,and then compose

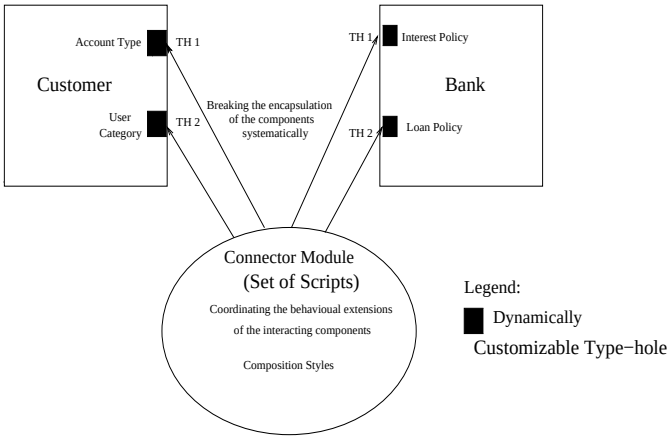

Fig. 3. Piercing Effect of Connector Module

them in the specified style. Since the scripts are interpreted, the glue objects that should customize the specified set of components can be varied in a consistent coordinated way so that each client request is individually provided with a response catered to its need. The script will decide the actual sequence of methods to be dispatched during run-time. *ChefMaster* framework provides more flexibility in generating customized response to client requests compared to the framework explained in the earlier section, because it can coordinate the plugging of glue objects across a set of interacting components. However, there is a penalty on the performance, because of the overhead associated with the dynamic method dispatch during run-time. One of the ways of improving the performance without giving up the flexibility offered by the framework is to componentize the script and make it inter-operate with the Java Virtual Machine(JVM). We feel that jython[7, 8] would be suitable from this perspective, because of its benefits of interoperability with java. Since *ChefMaster* treats the script itself as a coordination component, altering this component enables change in the coordination among the behavioural extension of the participating components.

3.2 Example

This section illustrates the workings of *ChefMaster* framework by taking a prototypical bank application.
Figure 4 shows the Graphical User Interface(GUI) screens for the sample bank application. The bank administrator might want to calculate the interest that should be paid on the various accounts maintained by different clients. In the provided GUI, the administrator will click on the "Calculate Interest" button. This will provide the next level form, where the name of the customer and the selection among the available account types are provided. This button click gets mapped to Script 1 by the

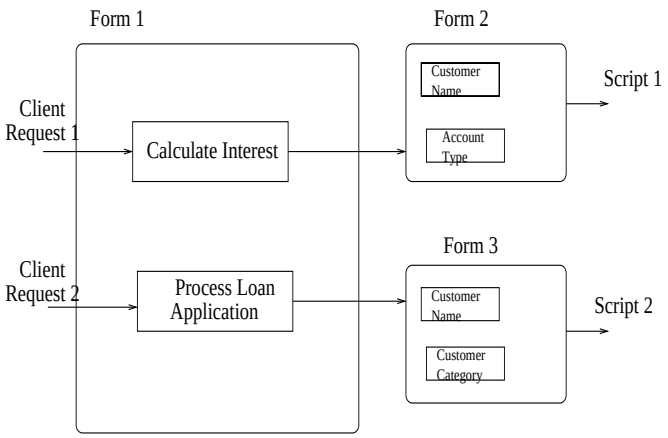

Fig. 4. Mapping Client Requests to Scripts

```
Script 1:

       [Plug(Customer.TH1, AccountTypeGlue)]. AccountInfo |

         [Plug(Bank.TH1, InterestPolicyGlue)]. CalculateInterest

Script 2:

       [ Plug(Customer.TH2, CustomerCategoryGlue)]. CategoryInfo |

        [ Plug(Bank.TH2, LoanPolicyGlue)]. ProcessLoan
```

Fig. 5. Sample Script

server code. Figure 5 provides the script code. The selection of the account type will generate the appropriate glue components to be plugged with "Customer" and "Bank" components. If the administrator chooses "Checking Account", the *CheckingAccountGlue* and *CheckingAccountPolicyGlue* components will be plugged to the "Customer" and "Bank" Components respectively. Further, the operator " |" makes the checking account details of the customer to be passed to the Bank component, where the plugged policy glue calculates the interest on that account and returns the result back to the administrator. If the administrator chooses "SavingsAccount" in a different request, *SavingsAccountGlue* and *SavingsAccountPolicy* components are plugged to "Customer" and "Bank" components. This will return to the administrator, the calculated interest for the amount in the savings account of the chosen customer.
If the initial request of the administrator is "Process Loan Application", the server will redirect the request to Script 2. Based on the chosen customer category which can be student, farmer, employee etc., the appropriate CustomerCategoryGlue and LoanPolicyGlue components will be plugged to the "Customer" and "Bank" components. This illustrates how the behavioural extensions for the various participating components are plugged in a consistent and coordinated fashion, for each client request.

3.3 Enhanced Client Customization in EJB

At present, in EJB architecture, the container abstraction is static. The static container restricts the way in which a set of interacting components should be composed. For a given client request, the method sequence that needs to be executed is decided at compile-time itself.
By integrating *ChefMaster* framework, the container itself can become dynamic. The infrastructure services provided by the present container will be the static non-varying part. The dynamic parts of the container are composition styles, and behavioural extensions to components. These will be managed by the set of scripts described in the previous subsection. The scripts will dynamically decide the behavioural extensions that should be plugged to the various participating components in the interaction and compose them at run-time. Thus, the script alters the method dispatch sequence for each client request and provides a customized response. The integration of *ChefMaster* with EJB is rather in its preliminary stage.

4 Related Work

Truyen et.al.,[9] has discussed a dynamic customization model called, *Lasagne*. In this model, distributed applications are developed as a combination of some core functionality and an unbounded set of possible extensions. This model is based on the concept of wrappers. The wrapper chain is dynamically constructed based on the composition policy specified. Though, this model has attempted to overcome some deficiencies of wrappers discussed in [10], by introducing a notion of component identity,

it still has to deal with the complexity of wrappers underneath. Further, it considers the extensions as self-contained. It cannot deal with grey-box components. However, *ChefMaster* can systematically inject behavioural extensions to grey-box components in a coordinated and consistent way.
Lumpe et.al.,[11,16] have designed a prototype composition language known as *Piccola*. This work assumes that all the components are pure black box entities. The composition language solely addresses the issue of establishing connection between the required and provided services of components. Even though it focuses on specifying connectors as first class entities, it does not deal with dynamically dictating behavioral extensions to the components to be composed. Our work considers all the third party components as black box and components developed in the native framework as grey-box. Hence, in the case of natively developed components, it is possible to dynamically extend their behavior externally through the script. Thus, *ChefMaster* can coordinate the customization of all participating components and compose them.
Hadas model[12] proposed by Ben-Shaul et al., provides a development environment, in which components can be built and adapted during their run-time. The application adaptability is the responsibility of a component called *ambassador*. While these ambassadors are similar to CORBA stubs in providing data marshaling and remote reference, they can be deployed dynamically. Even though the ultimate aim of Hadas matches the objective of our framework, Hadas uses mutable reflective component model, which uses the built-in meta-methods of the component to change its behavior. In contrast to this, in our framework, this functionality is externalized as connector module. Another shortcoming of Hadas is the loss of programming transparency because it significantly deviates from the conventional programming models. On the other hand, *ChefMaster* does not intend to replace CORBA, .NET or EJB, but provides a way of composing components built in these different frameworks.
Context relations[13] proposed by Linda Seiter et al., uses a context object in dynamically modifying the base object or set of base objects involved in a given collaboration. However, the same context has to travel along the entire collaboration. Whereas in *ChefMaster* framework, the number of Type-holes customized for the set of interacting components can be controlled and altered from the script, for each independent client request.
Contracts[14] explicitly capture the behavioral composition of a set of communicating objects. A contract defines preconditions required on participating objects to establish the contract, and the invariant to be maintained by these participants. Even though *ChefMaster* also addresses the behavioural composition of interacting components, it has the additional capability of dealing with grey-box components by systematically breaking their encapsulation. Further, contracts do not deal with customizing client requests, which is the major focus of *ChefMaster*.

5 Conclusions and Future Work

This paper proposed an augmented glue framework, *ChefMaster*, which provides individually catered response to different client requests, through

a set of scripts. These scripts dynamically coordinate the behavioral extensions for a set of components involved in that interaction and compose them at run-time. Since the script is interpreted, it offers increased flexibility in gluing the components in different ways, thus enabling customized response to client requests.
Currently, the *ChefMaster* framework has been implemented for a prototypical application. We intend to illustrate the power of the framework with a more complex example, which will clearly justify the need for dynamic coordination among interacting components. Our future plan also focuses on providing a formal description of the connector semantics. π-calculus[15] provides channel, an identity based on the unique sender and receiver. The possibility of giving first class status to methods based on the identities of sender and receiver is presently under scrutiny.

References

1. R. Wuyts and S. Ducasse. Composition languages for black-box components. In *Proceedings of the First OOPSLA workshop on Language Mechanism for Programming Components*, 2001.
2. M. Buchi and W. Weck. A plea for grey-box components. Technical Report 122, Turku Centre for Computer Science, Turku, Finland, 1997.
3. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.
4. D. Janaki Ram and Chitra Babu. A framework for dynamic client-driven customization. In *Proceedings of International Conference on Object-oriented Information Systems*, pages 245–258, University of Calgary, Canada, 2001.
5. D. Janaki Ram and O. Ramakrishna. The glue model for reuse by customization in object-oriented systems. Technical Report IITM-CSE-DOS-98-02, Indian Institute of Technology, Madras, India, 1998.
6. J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.
7. G. V. Rossum. Python. In S. Zamir, editor, *Handbook of object technology*. The CRC Press, 1999.
8. Jim Hugunin. Python and java: The best of both worlds. In *Proceedings of the 6th International Python Conference*, 1997.
9. E. Truyen and et.al.,. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of the 23rd International Conference on Software Engineering*, 2001.
10. U. Holzle. Integrating independently developed components in object-oriented languages. In *Proceedings of the ECOOP 93*, pages 36–56, 1993.
11. M. Lumpe and J. G. Schneider and O. Nierstrasz and F. Achermann. Towards a formal composition language. In *Proceedings of the ESEC 97 workshop on foundations of component-based systems*, pages 178–187, September 1997.

12. I. Ben-Shaul, O. Holder, and B. Lavva. Dynamic adaptation and deployment of distributed components in hadas. *IEEE Transactions on Software Engineering*, 27(9):769–787, September 2001.
13. L.M. Seiter and K.J. Lieberherr. Evolution of object behavior using context relations. *IEEE Transactions on Software Engineering*, 24(1):79–92, January 1998.
14. R. Helm, I.M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral composition in object-oriented systems. In *Proceedings of the Fifth ACM Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA)*, pages 169–180, October 1990.
15. Davide Sangiorgi and David walker. *The π-lalculus: A Theory of Mobile Processes.* Cambridge University Press, 2001.
16. M. Lumpe. *A π-calculus based approach for software composition.* Ph.D thesis. University of Berne, Switzerland, 1999.