“Designing a Three-Tier Architecture Pattern Language”

Thomas Neumann
e-mail: mail@thomasneumann.org

Introduction

The experiences described in this proposal are the extract out of two projects. The first project integrated three different commercial of the shelf application to a new system and made them accessible via a web browser. The second project was the redesign of the Internet site of a bank. Kiosk terminals that enable the customer of the bank to browse through information of the different bank products also access this system.

Common to both systems is that they use a layered architecture. Roughly spoken this architecture consists of two separate layers. The topmost layer is dedicated for the user interface. The lower layer is responsible for the application logic itself. The user interface layer is structured according to the Model-View-Controller pattern. The application logic uses the Business Component System pattern.

Overview

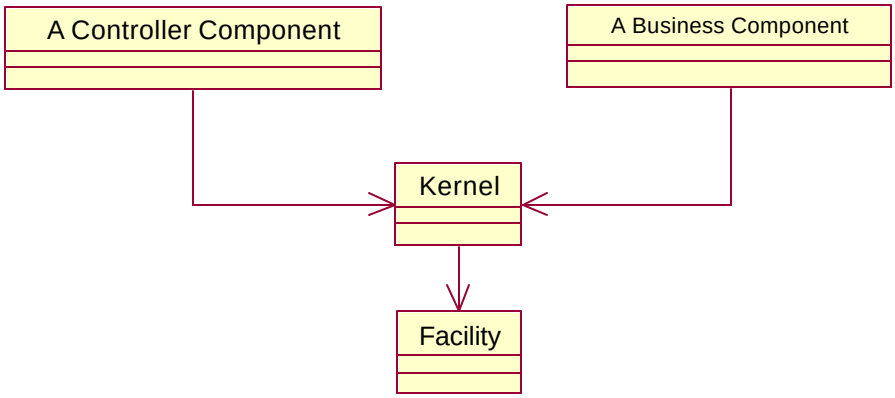
The Model-View-Controller pattern is a way of breaking an application, or even just a piece of an application's interface, into three parts: the model, the view, and the controller. The user input, the modeling of the external world, and the visual feedback to the user are separated and handled by Model, View and Controller objects. The Controller interprets input events from the user and maps these user actions into commands that are sent to the model and/or view to effect the appropriate change. The Model manages one or more data elements, responds to queries about its state, and responds to instructions to change state. The View manages a rectangular area of the display and is responsible for presenting data to the user through a combination of graphics and text.

The Business Component System pattern divides the system into coarse-grained components. These components are coupled as loosely as possible by using Request objects to interact with each other. Components are categorized into Controller and Business components. It has also an additional specialized component that provides infrastructure services.

The Model of the Model-View-Controller pattern communicates with the Controller Component of the Business Component System.

The rest of this position paper focuses on the Business Component System pattern, since there is already a lot of information on the Model-View-Controller pattern around.

Business Component System Pattern



Architectural overview of the Business Component System

Business Component – A Business Component is a subsystem that covers an autonomous business concept within the business component system – e.g. customer, account and provides services related to this domain segment. Services are method calls that could be used context free like create a new customer. These services are invoked by "sending" a Request Object to the Business Component – *COMMAND* Pattern. A Request Object identifies the service to be invoked and its input parameters. A Business Component may be distributed across multiple hosts and is uniquely identified by it's own logical name. This name is used to address Request objects.

Class *Business Component*	***Collaborators*** *Kernel*
Responsibility • Provide services for an autonomous business concept	

Controller Component – A Controller Component enables clients to access services provided by Business Components. They control the interaction on the user interface and invoke the related services on Business Components.

Class *Controller Component*	***Collaborators*** *Kernel*
Responsibility • Provide applications access to the Business Components	

Kernel - The Kernel Component represents the Broker in the overall system and hosts the infrastructure services.

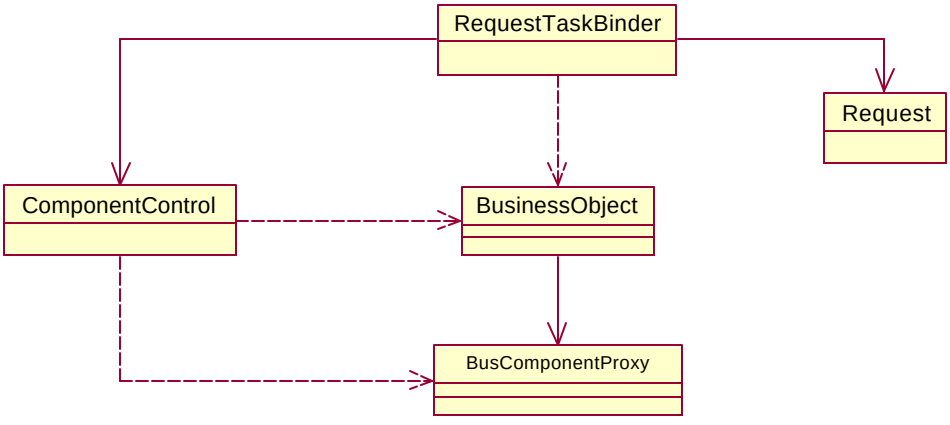
Class *Kernelt*	***Collaborators*** *Facilities*
Responsibility • Declares an interface for the needed Prototype operations.	

Facilities – Facilities are infrastructure services that are needed by several Business Components and do not represent an autonomous business concept – e.g. generation of unique identifiers, audit trail, …

Class *Facilities*	***Collaborators***
Responsibility • Provide infrastructure services to the rest of the Business Component system	

Business Component Overview

A Business Component covers an autonomous business concept and all related services. This business concept is modeled through a group of classes that work closely together also called business objects. E.g. the account business component hosts the account class itself and all derived classes, like debitor account, creditor account … . Besides this first level classes it contains also the related second level classes – or dependent objects – like account position, account owner. The services provided by the Business Component are no context dependent and thus combine various interactions on the object model. E.g. one service might be 'create new credit offer'. It includes checking the existence of the customer, creation of a new credit offer object and storing the credit offer in a database.



First class objects of a Business Component

Request – Specifies the service to be invoked, input and output parameter and context information for service invocation.

RequestTaskBinder – takes over the incoming request and activates the service specified. There

might be a dedicated object for each service, or a single object implement multiple services, or the request may be forwarded to another business component….

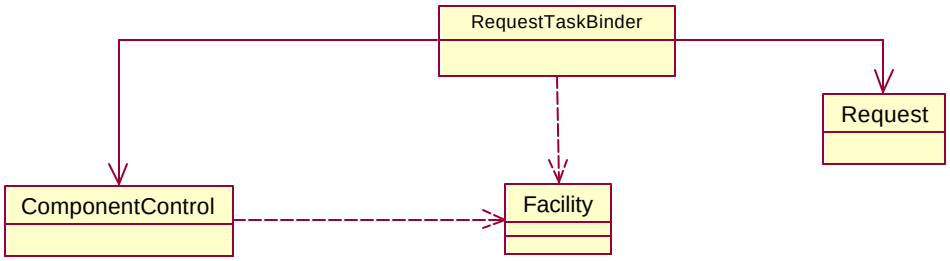
ComponentControl –is responsible for the life cycle services within a Business Component.

BusComponentProxy – links one Business Component to another one. As such it is designed as a smart *PROXY* which means it is more than just a plain image of the linked Business Component. More it represents the view of a Business Component on the linked Business Component. E.g. the credit owner in the Credit Component is the proxy of the Customer Business Component. The BusComponentProxy encapsulates the communication with other Business Components via Requests. Results obtained from Requests are transformed into an object representing the view on the linked Business Component.

Business Object – model the autonomous business concept of the Business Component e.g. credit, customer ... They are encapsulated by the Business Component and can only be accessed by Requests that are sent to the RequestTaskBinder.

Kernel a special Business Component

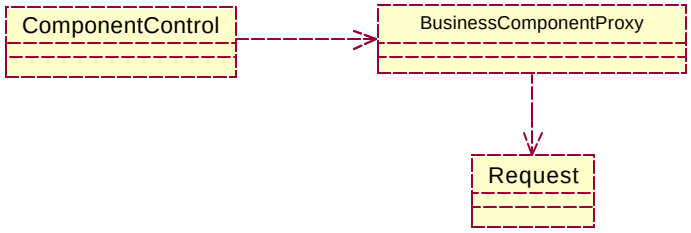
The Kernel plays the role of the Broker. All Business Components communicate via Requests with the Kernel. It hosts Facilities. Facilities provide infrastructure services for the Business Component System e.g. generation of unique identifiers, auditing … Business Components invoke infrastructure services via Requests that are sent to the Kernel. In turn the Kernel invokes the corresponding Facilities.



First class objects of the Kernel Component

Controller Component

A Controller Component is the entry point for applications into the Business Component System e.g. an application for creating credit offers in the front office. Such an application might offer a sophisticated user interface or is a background job. It uses a Component Control to access the BusinessComponentProxy. The responsibility of the BusinessComponentProxy is to control the communication between the application and the Business Components in the system.



First class objects of a Controller Component

Different categories of Business Components

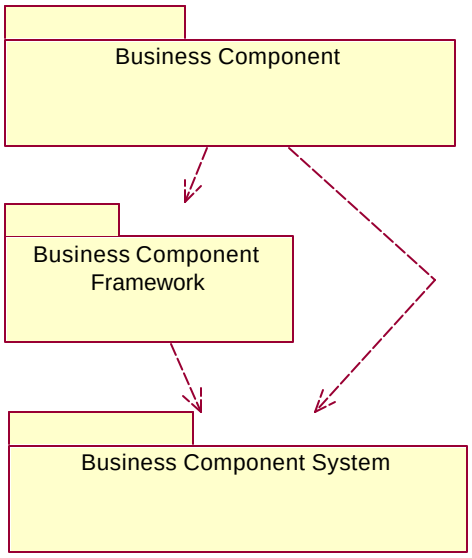
One requirement for this architecture is that it should be used for a broad range of systems in the commercial domain. This covers for instance

- small systems where only one application is used to access different Business Components
- systems that implement business processes spanning multiple organizational units
- larger systems that integrate newly written Business Components with business logic implemented in already existing systems.

This brings up the need to have an architecture that is flexible enough so that it could be easily adapted to cover such a range of different systems. One approach for doing this is to categorize the Business Components within the system.

- **Entity Business Components** contains Business Objects that model a business entity. E.g. credit, customer, account, …
- **Business Process Components** contains Business Objects that model a business process. A business process mediates multiple business entities. E.g. identify customer before creating a credit offer, …
- **Adapter Components** represents a bridge to other systems like legacy systems or systems implemented using another component technologie.

Each category of Business Component is structured according to the general pattern described above and might have its own framework. This results in a system that is built from multiple frameworks. Each framework helps to solve a small problem within the Business Component System. The different frameworks and Business Components are connected through Requests. This results in a system with the following layering.



Layering of the Business Component System

Business Component Layer – this layer hosts the implementation of an autonomous business concept e.g. customer, account, deletion of a customer, …

Business Component Framework Layer – gives help in building Business Components by providing services that are common to this category of Business Components. There might be several frameworks hosted in this layer according to the categories of business components found in the Business Component System.

Business Component System Layer – represents the transport layer of the Business Component System. It knows all the Business Components currently available in the system and routes the Request Objects accordingly.

This layering results in a system that is not build from one large monolithic framework with all its drawbacks in complexity and dependency. Instead, as described in the *LAYERED COMPONENT FRAMEWORK*[1] pattern, multiple frameworks are used. Each framework targets only a certain aspect in the system domain. This results also in a system of very loosely coupled components, since all components depend only on the Business Component Framework they are made of and on the Business Component System Layer. There is no direct dependency between two components. For instance an Account component does not directly use any interfaces of a Customer Component to implement the 'account owner' object. The Account Component just specifies the services it needs – e.g. 'check Customer' – to its environment. The environment of the Account Component consists of the Business Component Framework and the Business Component System Layer. It's the responsibility of this environment to route a service request to the Business Component that is able to satisfy this service. This means that the Account Component itself does not know anything about other Business Components in the system. It only knows the services it needs and tells it environment to satisfy these services. A similar layering could also be found in the chapter "Component Architecture" in Szyperski's book[2].

Communication between different Business Components.

In most cases Business Components communicate with each other by specifying a Request object and transmitting it to the target component. The originating Business Component is blocked until the requested service is executed. But there are cases where the originating Business Component is not interested in the result of the service. E.g. writing an audit trail of system activities. The originating Business Component just specifies the audit information and wants to be sure that it is written in any case into the audit trail. This communication model is different from the one described at the beginning.

Since the communication between Business Components is located in the Business Component System layer there is a dedicated place in the architecture that is responsible to implement these different communication models. Two frequently used communication models are:

- **Request Reply –** The flow of control in the originating Business Component is interrupted until the calculation of the result has been returned.
- **Guaranteed Delivery** – The flow of control in the Business Component is not interrupted until the result of the request is calculated. The Request is only transmitted to the receiver and the result is calculated sometimes in the future without interaction with the requestor. Thus the requestor receives never a result. This communication model is useful to send audit items to the auditing facility.

---

[1] see [Eskelin]

[2] [Szyperski] "Component architecture", p. 273

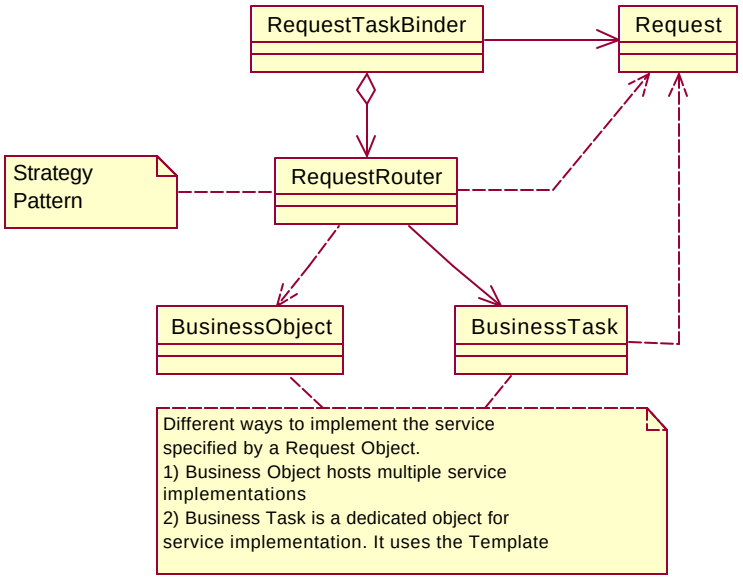
Levels of Component Granularity

The term component is used on several abstraction levels within the whole system e.g. architecture, design, ... Dependent on the abstraction level granularity of component changes from coarse to fine -grained. A critical task in component-based architecture is to map these different levels of component granularity to the component technology used[3].

- A **Business Component** represents an autonomous business concept thus it is a reusable piece of business logic e.g. management of all accounts. It is comparable to a subsystem and offers service calls. One service call is a context free method invocation.
- A **Distributed Component** encapsulates a small number of closely related Business Objects e.g. an account. One Distributed Component could be implemented using standard component technology, e.g. EJBs or DCOM. Each Business Component is made of several Distributed Components.
- A **Language Class** is a class in an object-oriented programming language. It is used by a developer to build Distributed Components.

Mapping of Business Components on Distributed Components

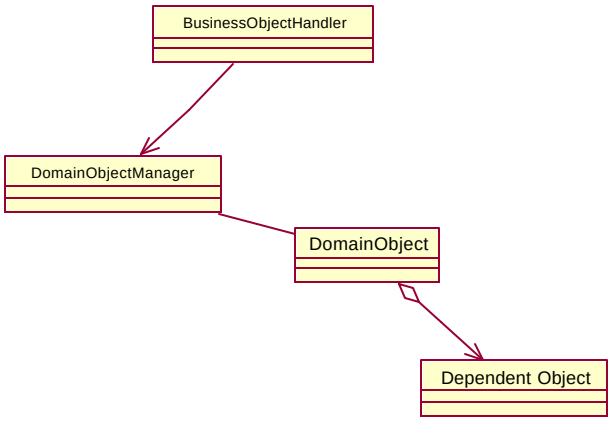
Each Business Component contains a Component Adapter – which is a Distributed Component itself. It corresponds to the RequestTaskBinder from the picture "First class objects of a Business Component". The Component Adapter receives a Request Object and performs the requested service. There are different ways how a service could be implemented. One way is a dedicated object for each service. Another way is that one object hosts multiple services. As a third way the Request Object is forwarded to another Business Component. These different behavior are encapsulated in different language classes using the *STRATEGY* Pattern.



Design of the Component Adapter

[3] See also [Herzum]

Another Distributed Component captures all the logic related to one Business Object as described in the chapter "First class objects of a Business Component". This Distributed Component hosts the Business Object itself and all of its associated dependent objects.



Design of a Business Object

Related Patterns

- ***MESSAGE PASSING BROKER***[4] - In many aspects this pattern is similar to the *MESSAGE PASSING BROKER* Pattern. But the Business Component System Pattern extends the *MESSAGE PASSING BROKER* Pattern by adding infrastructure services as extensions to the Broker itself.

- ***LAYERED COMPONENT FRAMEWORK*** – It is important in framework development to have a modular design instead of one monolithic piece of software that prevents developers from adapting this piece of software to unforeseen requirements. Because of this we have introduced a layering that divides the system into three layers. This gives the Business Component System the ability to be adapted to new requirements by implementing dedicated Business Component Frameworks or Facilities.

- ***COMMAND*** - Motivation behind the *COMMAND* pattern is that you want to ask services from an object without knowing anything about the object that provides these services. This capability of the *COMMAND* pattern is used in the Business Component System pattern to decouple components from each other. The usage of the *COMMAND* pattern leads to the point where a requesting component doesn't need to know anything about the structure of the object model in the component that provides the requested service. This leads to the advantage that the object model could be changed without any impacts on the requesting component.

- ***STRATEGY*** – The *STRATEGY* pattern is used to encapsulate the algorithm of handling an incoming Request object in the RequestTaskBinder. This enables a Business Component either to activate a Task Object or to forward the Request object to another Business Component.

- ***FAÇADE*** – The *FAÇADE* Pattern provides an abstract interface to a group of objects. In the Business Component System the *FAÇADE* pattern is used to provide an interface on the Business Objects. These Business Objects model the autonomous business concept of a

---

[4] [Buschmann] Broker Pattern, p ???

Business Component. The *FAÇADE* interface simplifies the implementation of a service in the RequestTaskBinder by combining multiple interactions with the Business Objects into one method call.

- ***CONWAYS LAW*** – This organizational pattern puts an emphasis on the fact that the organizational structure in project should be complementary to the structure of the architecture. Building the Business Component System out of loosely coupled Business Components makes it possible to have teams assigned to each Business Component. This results in having the team border aligned with the Business Component border which improves parallel development of Business Components.

- ***PROXY –*** A *PROXY* is a placeholder for another object. In the Business Component System the *PROXY* is used to link two Business Components together.

Bibliography

[Kruchten] Kruchten, 4+1 View, Rational, http://www.rational.com/products/whitepapers/350.jsp

[Gamma] Gamma, E. e.a., Design Patterns, Elements of Reusable Object-Oriented Software-Design, Addison-Wesley, Bonn 1995

[Buschmann] Buschmann, e.a., Pattern-Oriented Software Architecture, a System of Patterns, John Wiley & Sons Ltd, 1996

[Szyperski] Szyperski, Component Software, Beyond Object-Oriented Programming, Addison-Wesley, 1998

[Eskelin] Eskelin, Layering Frameworks in Component Based Development, http://www.eskelin.com/patterns/PLoP99/LayeredComponentFramework.pdf

[Herzum] Herzum, P. e.a., Business Component Factory: A Comprehensive Overview of Component Based Development for the Enterprise, John Wiley & Sons, Inc.