Abstract: The invention concerns a technology for reusing a software component, and presents a method of eliminating dependency of service components on a business entity of a specific application so that binary components independently developed can be reused without modifying source codes. Unlike the conventional methods of developing software by defining a business entity first and then developing service components according to the specification thereof, with the invention, a component developer defines any data type to be used in implementing business logic without consideration for a specific business entity and then develops a service component on the basis of the definition, and an application developer then develops software which coordinates and interworks with the data type defined by the component developer and the attribute type of the business entity actually to be used in an application by means of a data type converter when assembling the components developed as such.
Description
A METHOD FOR DECOUPLING SERVICE COMPONENTS FROM APPLICATION-SPECIFIC BUSINESS ENTITIES

Technical Field
The invention relates to a software development technology based on object-oriented components, and more particularly to a technology that binary components independently developed can be reused in order to develop new software via assembling without modifying source codes. Even more particularly, the invention relates to a technology for eliminating close dependency of service components on business entities which hinders software development via assembling after first developing the components.

Background Art
Since Douglas McIlroy disclosed a method of producing software via mass production and assembling of components in the software engineering conference of NATO in 1968, there has been a need of improving productivity by reusing software modules in the industry for a long time. Such an interest in reusable software modules led to appearance of component models, e.g., CORBA/CCM of OMG and COM/COM+ of Microsoft after 1990 through object-oriented programming in late 1980s. Currently, it is a mainstream in the IT industry to produce software based on a component framework, e.g, EJB and .NET in object-oriented language such as Java and C#.

However, the aforementioned state-of-the-art object-oriented component technologies have not been used for developing software by reusing components which was the motive of devising the technologies in spite of the advancement since then. Currently available technologies have advanced while developing all of components required in an application in a direction to aim simple module interaction and to enhance reusable efficiency in codes, rather than aiming reusability of binary components. As a result, a method of developing software has been used which requires strict type safety in order to detect errors early in the compiling step. In this type-based programming, since compiling can be carried out, provided that a specific type for interaction (class, interface, user defined data type, etc.) has been obtained in developing components to use the identification information for codes, it is accordingly inevitable that the components are closely coupled to a specific type, and further to another specific component. This is a method opposite to loose coupling which is emphasized by those who assert reusability. The components developed in this method cannot be reused in a different environment without modifying source
codes (as seen in the case of active binding technology, it is not necessarily in-
compatible to observe type safety and reuse binary components).

[4] In order to reuse service components which implemented business logic in various
applications, like hardware elements, without modifying source codes, they should not
be dependent on a specific application environment. In order for any component not to
be dependent on a specific application environment, it should be loosely coupled to
another specific component and also loosely coupled to a business entity responsible
for data transfer between components which is a main concept consisting of a specific
application. This accordingly means that each service component must not comprise a
code which uses a specific type of instance defined externally other than the type
provided as a class library in a platform or framework.

[5] However, since service components use the types defined externally, e.g., the
interface of other service components, business entities of an application, etc., for
codes as described above, they are inevitably dependent on a specific application. In
order to use them in a different application, it is required to modify the source code. In
order to modify a source code, the source code should be known. However, it is highly
unlikely to be able to get a source code of a component developed by a third party.
Although the source code is known, it is not easier to analyze the source code prepared
by a third party and then to modify it without errors than to develop a new required
component. Therefore, on the basis of the conventionally available component tech-
nologies, it is fundamentally impossible to reuse service components independently
developed by a third party in terms of binary codes, and it is not economically feasible
in terms of source codes.

[6] As described above, in order to reuse service components already developed when
developing a new application, it is essential that each component should not be
dependent on a specific type defined by a third party. One feasible method to comply
with the constraint on the basis of the current object-oriented component technologies
which constrain type safety is that each component defines an interface type or user
defined data type in a relevant component itself to use it in preparing source codes,
wherein the component should be dependent on the type in order to perform its
function. When reusing the service component developed in this manner subsequently
for developing an application, coupling to external types actually dependent on is then
coordinated out of the component with type information revealed with metadata. For
this coordination, it is required to settle different interface between service components
assembled in a relationship of client-server, and to settle type nonconformity between
the data type used by the service component instead of a business entity and an actual
business entity of an application.

[7] The technology which enables software to be developed in a manner of assembling
and reusing components after developing them first is not supported by conventionally available component technologies, e.g., Java, Dotnet, etc. The active binding technology not commercially available yet (disclosed in Korea Patent Registration No. 10-0697246 and entitled Method of and system for developing software with metadata extended in component based environment ) and its upgraded version (disclosed in Korea Patent Application No. 10-2007-0113441, and entitled Method of developing and executing software by reusing components and injecting dependency ) propose a method of enabling self-sufficing coding by defining an interface required for requesting a service in each component thereby to eliminate dependency between service components and to settle interface nonconformity by means of a glue component in assembling the components.

The active binding technology provides a method of assembling components by loosening coupling between service components and by means of interface coordination in order to provide an essential basis required for reusing the service components independently developed without modifying source codes. However, this technology does not address an issue that service components are dependent on a specific application business entity. A business entity which is a main concept consisting of an application and which transfers data in a form of parameters and resulting values between service components in a presentation layer, a business logic layer and a data processing layer reflects requirements of users from application to application thus to be uniquely defined. In a conventionally available development environment for strict type safety, since compiling and operation is enabled only when the business entity type specific to an application conforms to the data type handled by each method of the service components, the business entity is first defined on the basis of which service components are developed. Therefore, in the current development environment, if a method of coordinating the data type of service components and the business entity type of an application is not further used although different interfaces between service components are coordinated by means of active binding, there is even less possibility of assembling service components developed independently by a third party without modifying source codes in order to develop a new application.

Considering the current technologies related to reusing service components as described above, there is a need of new technology for loosening coupling of service components to a business entity in addition to the active binding technology which settled the issue of loosening coupling between service components in order to enhance software productivity by developing and reusing components not dependent on a specific application environment.

Disclosure of Invention
Technical Problem

The invention was devised to solve the aforementioned problems in reusing service components independently developed without modifying source codes. It is an object of the invention to provide a method for which a service component developer defines a component data type optionally which is not dependent on a business entity type of a specific application, and implements business logic with the type information in order to eliminate dependency on the business entity type of the service components.

It is another object of the invention to provide a method of coordinating the data type of a component itself defined optionally by a component developer with the business entity type actually used in an application thereby to implement a data type converter component capable of doing interaction, when assembling service components developed as described in the aforementioned manner to develop the application.

It is still another object of the invention to provide a technology of enabling the service components developed by a third party to handle business entities of a new application without errors by a framework creating objects with reference to dependency injection information files and then injecting a data type converter object in the data type of the service component itself, when running the application developed in the manner described above.

Advantageous Effects

In the invention as described above, the inventor proposes a new object-oriented component structure comprising a self-defined data type in order to avoid a service component to be dependent on a specific application environment. According to the structure, there is presented a method of the data type of a service component with the data type of a business entity actually used in an application by means of a data type converter when reusing the service component already developed by a third party to develop the application. In the invention, the inventor also proposes a method of a service component handling a business entity actually used in an application by allowing a framework to create objects and inject dependency between objects when running the application, the service component having been developed independently of the business entity of a specific application.

The invention overcomes the current technical constraint that a service component is inevitably dependent on the business entity of a specific application while utilizing the advantage of conventional object-oriented programming, e.g., intuitive coding, type safety and efficient error detection by means of aforementioned methods, the service component processing business logic. The invention enables service components to be reused when developing a new application without modifying source codes in combination with the active binding technology which loosens coupling between service
components, the service components having been independently developed by a third party.

Therefore, the invention significantly improves reusability of service components, achieves application development cost to be reduced and a development period to be shortened via reusable components and also implements easy software production and maintenance. Such technical outcomes of the invention provides a practical basis required for realizing the idea of CBD for improving software productivity in a method of developing and accumulating software components like hardware parts, purchasing and assembling the required components in the market to develop an application. Furthermore, the invention will be able to contribute to enhancing a software ecosystem by creating and furthering the market for components in order to endow more business opportunities to small-scale software providers.

**Brief Description of Drawings**

The features and advantages of the present invention will become apparent from the following detailed description of preferred embodiments thereof illustrated with reference to the accompanying drawings, in which:

Fig. 1 shows a conceptual structure of interworking between service components developed according to the invention with a business entity of a specific application;

Fig. 2 shows a flow chart illustrating a process of developing a service component according to the invention;

Fig. 3 shows a flow chart for the step of defining the data type of a service component itself and a data type converter interface type in the process of developing the service component according to the invention, in more detail;

Fig. 4 shows a flow chart illustrating the process of developing a data type converter component according to the invention;

Fig. 5 shows a process illustrating the sequence of processing service calls of a client while each component interacts in running software developed according to the invention;

Fig. 6 shows a common object code in which the root data type and custom attributes for annotation have been defined in a preferred embodiment of the invention;

Fig. 7 shows a business entity object code which an application actually uses in another preferred embodiment of the invention;

Fig. 8 shows a service object code of a service component to be reused in another preferred embodiment of the invention;

Fig. 9 shows a data type code of a service component itself to be reused in another preferred embodiment of the invention;

Fig. 10 shows a data type converter interface code of a service component to be
reused in another preferred embodiment of the invention;

Fig. 11 shows a data type converter object code implemented in assembling an application in another preferred embodiment of the invention;

Fig. 12 shows a dependency injection information file between the data type converter object and the data type of a service component itself, the data type converter object being created in assembling an application, in another preferred embodiment of the invention.

Best Mode for Carrying out the Invention

In order to achieve the aforementioned object, the invention presents a method of implementing a service component embedded with a data type of its own instead of a business entity so that it may not be dependent on the business entity of a specific application in an environment for object-oriented component based software development; a method of implementing a data type converter component for converting the data type of a service component to the data type of a business entity actually used in an application when reusing service components developed in the manners as described above for developing an application; and a method in which service components to be reused can handle business entities newly defined without errors by a framework creating objects with reference to dependency injection information files and injecting the data type converter object in the data type of the service component itself, when running the application developed in the aforementioned manner.

Hereinafter, the method of developing and running software according to the invention will be described in detail with reference to the accompanying drawings.

Fig. 1 shows a conceptual structure in which a service component developed according to the invention interworks with the business entity of a specific application. The service component 13 according to the invention has a structure comprising, in addition to a service object 131, a data type 132 of the component itself defined to be used in coding instead of the business entity of an application for interworking, and a data type converter interface from which a data type converter object to be implemented is originated for converting a business entity type to the data type of the component itself.

In the environment for developing a service component while there is no business entity of an application actually to interwork according to the invention, a root data type 111 is essential which is commonly used by a component developer and an application developer as a data type that all of business entity types should inherit for developing service components. A root data type defined in a format only with a name is provided to all developers who have employed the development manner according to the invention as a type of common type package 11.
When developing a new application while reusing service components developed according to the invention, an application developer defines a business entity object 121 which inherits the root data type 111. When the business entity is provided, he implements a data type converter object 141 for converting the business entity type to the data type 132 of a service component itself to be reused. This converter object is completed by implementing the methods of the data type converter interface 133 defined in the service component.

A file 16 is then created which shows inter-dependency with respect to the service component 13, the business entity 12, and the data type converter component 14 developed and assembled as described above, then arranged in a framework 15 and executed. The framework then creates objects of the components according to the dependency information file and injects the reference of the data type converter object in the data type of the service component itself. By means of this measure, the attribute of the business entity handled by the service method of the service component object is converted to the corresponding attribute of the data type of the service component. In this state, the service method performs necessary tasks.

Fig.2 shows a flow chart illustrating the process of developing a service component according to the invention. The sequence of developing a service component is described below.

When it is required to use a business entity defined as a user defined data type for a parameter or return value in the step S12 while developing a service method of a service component object in the step S11, there is defined a method signature in the step S13, with a root type which all of business entity types should inherit instead of using the business entity type of a specific application. Subsequently, in the step S14, there are defined the data type of the service component itself to be used instead of the business entity of the specific application in a service method code and a converter interface type to be implemented by a data type converter to be made in developing an application later.

In the step S15, there is prepared a code for converting the variable of the root type received as a parameter of the service method to the data type of the service component itself by means of a conversion operator, the code being a first statement of the service method. In the step S16, the service method of the service component being developed is completed, using the variable of the data type of the component itself, instead of using the business entity object received as a parameter.

Fig.3 is a flow chart illustrating, in more detail, the step S14 of defining the data type of a component itself and a data type converter interface type in the process of developing the service component according to the invention. The sequence of defining the data type of a service component itself and the converter interface type will be
described herein below. In the step S141, there is prepared a data type class of the
component itself in the component being developed and a field variable of the root
type is declared for accommodating business entity objects. In the step S142, there is
defined a creator with a parameter of the root type in the data type class of the
component itself. In the step S143, a conversion operator is implemented in the class
which converts, in the class, the business entity type to the data type of the component
itself by means of the creator already defined, the business entity type having inherited
the root type. In the step S144, the type and name of the attributes to be used in the
service method of the component object being developed are specified in the class, and
there are declared a get accessor and a set accessor for each attribute. In the step S145,
a converter interface type is defined which consists of methods to be used for
converting each of the attributes of the actual business entities for interworking in the
get accessor and the set accessor declared in the step S144 in the component being
developed to the attribute of the data type of the component itself. In the step S146,
there are declared the field and attribute of the converter interface type as a variable for
accommodating the reference of the converter object to be implemented later in the
data type class of the component itself in order to enable the function of dependency
injection of the framework in run time. Lastly in the step S147, the get accessor and the
set accessor for each attribute of the data type of the component itself are completed,
using the variable of the converter object reference filed declared in the step 146, to
finish the process of defining the data type of the component itself.

[39]  Fig.4 is a flow chart illustrating the process of developing a data type converter
component for converting the attribute of the data type of a service component itself to
the corresponding attribute of a business entity when running an application according
to the invention. The sequence for implementing a data type converter component will
be described hereinbelow. In the step S21, there is created a data type converter
component development project. In the step S22, after identifying all of the converter
interface types defined in the service component, one interface is selected which was
not implemented with a converter class among them. In the step S23, a data type is
identified which corresponds to the converter interface selected in the step S22 among
the defined data types of the component itself. In the step S24, a class is created which
implements the selected converter interface prior to the data type converter component
development project. In the step S25, a business entity is identified which corresponds
to the identified data type of the component among business entities of a specific app-
lication. In the step S26, one method is selected which was not implemented among
the methods of the selected converter interface. In the step S27, a pair of attributes are
identified which the selected method convert and interwork with, between the attribute
of the data type of the component itself and the attribute of the business entity. In the
step S28, codes are prepared for converting the values of identified attributes in the converter class to finally complete implementation of the selected method. If there remains any method not implemented among the methods of the selected converter interface, the process from the step of selecting methods S26 not having been implemented to the step of preparing attribute conversion codes S28 is repeated until all methods are implemented, in order to complete the converter class for implementing the selected converter interface, in the step S29. If there is any interface not implemented with the converter class among the identified converter interface types, the process from the step of selecting interfaces S22 not having been implemented to the step of implementing interface methods S29 is repeated until all converter interfaces are implemented, in order to complete the process of implementing the data type converter class, in the step S30. In the step S31, there is created a file with the dependency injection information between the converter object implemented through the aforementioned process and the corresponding data type of the component itself, in order to compile a project and then create a data type converter component in the step S32.

Fig.5 shows a process illustrating the sequence that each component processes service requests of a client in running software developed according to the invention. The process will be described in the following for creating an object, injecting inter-object dependency and processing service requests of a client in a dependency injection framework. When an external client, e.g., a user interface, etc., requests an object of a service component arranged in a framework in the step S41, the framework creates an object of the relevant component and an object of a data type converter class(es) associated therewith with reference to the dependency injection information file in the steps S42 and S43. The framework then injects the reference of the created data type converter object(s) to the attribute for injecting dependency of the data type of the component itself corresponding to each converter object in the step S44. The framework then returns a relevant component object to the client which requested the service in the step S45. When the client sends the reference of a business entity as a parameter while invoking a service method of the component object received in the step S46, the invoked method of the service component object converts the business entity object received from the client as a parameter to the data type object of the component itself in the step S47. When the invoked service method access a specific attribute of the data type object of the component itself for carrying out its service in the step S48, the accessor of the accessed attribute invokes the type conversion method related to the relevant attribute of the data type converter object and simultaneously sends the reference of the business entity object in the step S49. The invoked type conversion method converts the attribute of the business entity object to the data type
attribute of the component itself corresponding thereto. The service method invoked by
the client repeats from the step of accessing a specific attribute of the data type of the
component itself S48 to the step of invoking the type conversion method related to the
relevant attribute S49 as many times as required to complete its service provision and
returning the result of service performance to the client in the step S51.

Hereinafter, in order to more specifically disclose the method of developing and
running software according to the invention, one preferred embodiment of the
invention will be described in detail in which the method is applied to the Dotnet
framework which is one of current representative development platforms and executed.
Those skilled in the art will readily appreciate by means of the result of the em-
bodyment that the method of developing and executing software according to the
invention is immediately applicable to other object-oriented component technology
platforms having a similar structure, e.g., Java, as well as the Dotnet framework.

The embodiment to be described hereinbelow has a simplified configuration of a
hotel customer information management system developed in C# programming
language in the Dotnet framework according to the invention. It is an example of
preparing a service component for carrying out customer information management, a
data type converter component for coordinating a business entity which is actually
used in an application with the data type of the component itself, a business entity, a
common object in which the root type, etc., which all business entities should inherit is
defined and a dependency information file for directing a framework to create objects
and inject dependency. In this example, a minimum amount of codes are shown which
are related directly to understanding the concept of the invention.

Fig.6 shows the code of common objects CommonObjects in which the root data
type and the custom attribute for annotation are defined, wherein the business entity of
an application developed according to the invention should commonly inherit the root
data type and the custom attribute. Such a common object will be provided as a basic
class library of a development tool to be implemented for practicing the invention. The
common object code of the embodiment comprises a root data type 111 only with a
name of EntityRoot, a custom attribute 112 showing that a specific interface is the data
type converter interface, a custom attribute 113 showing that a specific class is of the
data type of the component itself, and a custom attribute 114 showing that a specific
attribute is a dependency injection path injected with the reference of an object on
which the type comprising the attribute is dependent.

Fig.7 shows the code of the business entity object HotelCustomer 121 actually used
in the application of the embodiment. In this code, it is shown that the business entity
object HotelCustomer inherits the root data type EntityRoot 111. In this business
entity, 6 attributes of CustId, etc., are declared.
Fig. 8 shows the service component object CustomerMgt 131 in the code of the service component ServiceComponent obtained from a third party in order to reuse it in the application of the embodiment. This service component object comprises a method NotifyCustomer 131-1 which receives and sends an actual business entity object of the application and an e-mail address as a parameter, and a method UpdateEmailAddress 131-2 which receives the business entity object and an e-mail address as a parameter to update the customer's e-mail address. The parameter type of both of the methods is declared as a root data type EntityRoot defined in the common object in order to receive the reference of the actual business entity object as a parameter, and, for the parameter name, customer optionally specified is used while the name of the business entity object for actual interworking (HotelCustomer in this embodiment) is not known. Both of the methods comprise, as a first statement, a code for converting the business entity object received as the parameter customer to the data type MyCustomer of the component itself. Before preparing this statement, it is required first to define the data type of the component itself and the data type converter interface.

Fig. 9 shows the data type of the service component itself MyCustomer 132 in the code of the service component ServiceComponent obtained from a third party in order to reuse it in the application of the embodiment. On the upper part of the data type, the custom attribute 132-1 is added which shows it is the data type of the component itself defined in the common object. In this data type of the component itself, there are declared a field variable bizEntity 132-2 for receiving and holding the reference of the actual business entity object for interworking with the type and a field variable typeConverter 132-3 for being injected with and holding the reference of the data type converter object from the framework in run time. There is defined an attribute TypeConverter 132-5 as a path for receiving the reference of the data type converter object from the framework in run time. On the upper part of the attribute, a custom attribute 132-4 is added as annotation showing that this attribute is a dependency injection position. There are then defined a creator 132-6 for specifying that the data type object of the component itself should have the root data type defined in the common object as a parameter when the data type was created, and a conversion operator 132-7 for converting the business entity object for later interworking to the data type object of the component itself. There are defined attributes 132-8 of the data type of the service component itself used in the methods of the service component object as well, which shows that, the attributes, the number, name, type and the like for interworking in this embodiment are different from the attributes of the actual business entity.

The data type code of the service component itself described above is seen complicated at a glance but most parts thereof are prepared according to a given rule. Therefore, it is automatically prepared on the basis of a minimum amount of in-
formation newly entered if a tool implementing such a rule is used.

For this embodiment, if a service component developer only specifies and enters the name of data type of the component itself MyCustomer and the name and type int CustomerNumber, etc., of the data type attributes of the component itself to be used in the service methods in the tool, there are automatically created other codes using the entered information, the field variable name which the tool can optionally specify, the root data type and the custom attribute annotation already specified.

Fig. 10 shows a data type converter interface MyCustomerConverter 133 in the code of the service component ServiceComponent obtained from a third party to be reused in an application of the embodiment. On the upper part of the interface, there is added a custom attribute 133-1 as annotation showing that this is a data type converter interface. This converter interface consists of Get/Set methods for converting each attribute of the data type of the service component itself used in the service method of the service component object to corresponding business entity attributes. This code can be automatically created on the basis of the type and name of the data type attributes of the component itself already given in the process of defining the data type of the service component itself in Fig.9 as well.

Fig. 11 shows the code of data type converter component object MyCustomerConverter 141 for converting the data type MyCustomer of the service component itself and the attributes of the business entity HotelCustomer actually used in the application in the embodiment.

The data type converter object implements the methods included in the data type converter interface MyCustomerConverter 133 defined in the service component Service Component. When implementing the methods, an application developer identifies the attribute of the actual business entity corresponding to the data type attribute of the component itself then handled by the corresponding methods, to prepare a code for converting the type. In this embodiment, the attribute CustomerNumber of the data type MyCustomer of the component itself is int type but the attribute CustID of the business entity HotelCustomer corresponding thereto is declared as a string type. Therefore, in order to link these two attributes, the CustID value is converted to the int type with Convert.ToInt32() in the method GetCustomerNumber, and the value new Value is converted to the string type with the ToStringO method in the method SetCustomerNumber.

Since the code for implementing the data type converter method as described above is also prepared on a rule basis, it is possible to automatically create most parts of the code using an automation tool which implements the relevant rule. When the tool selects an interface which a developer wants to implement in the data type converter interface list which the tool presents by reading the metadata of the service component,
the tool reads out the data type of the component itself corresponding to the selected interface, and the metadata of the business entity corresponding to the data type then to present each attribute list. When the developer selects and links attributes in the list on both sides for interworking, the code of the data type converter object is automatically created. If there is a semantic difference between attributes corresponding to each other (for example, meters are used on one side but yards are used on the other side), the developer may inject appropriate conversion codes to coordinate semantic inconsistency.

Fig. 12 shows a dependency injection information file which directs a framework to create a service component object and a data type converter object, and then to inject the reference of the data type converter object in the attribute for injecting dependency of the data type of the service component itself, when arranging and executing components implemented according to the embodiment of the invention in the framework. This file can be automatically created on the basis of the information already given in the process of implementing the data type converter object.

Hereinafter, with reference to Figs.5 to 12, a preferred embodiment of the invention will be described in more detail, in which the components interwork when running an application developed according to the invention.

When an application user prepares contents of an e-mail to be sent to a specific hotel customer and commands transmission, the user interface module requests to the framework, as shown in Fig.5 (S41), an instance of the service object CustomerMgt (Fig.8, 131) of the service component ServiceComponent for executing the corresponding function. The framework creates an instance of CustomerMgt and a data type converter object associated therewith MyCustomerConverter (Fig.11, 141) with reference to the dependency injection information file (Fig.12). After injecting the instance reference of MyCustomerConverter in the attribute TypeConverter (Fig.9, 132-5) for injecting dependency of the data type MyCustomer (Fig.9, 132) of the component itself, the framework then returns the instance of CustomerMgt to the user interface.

The user interface module sends, as a parameter, the interface reference of customer's business entity HotelCustomer (Fig.7, 121) who will receive the e-mail and the contents of the e-mail while invoking the service method NotifyCustomer (Fig.8, 131-1) for sending the e-mail in the service object CustomerMgt. The invoked NotifyCustomer then converts the HotelCustomer received as a parameter to the object MyCustomer to save it in the variable cust, using the conversion operator 'MyCustomer' (Fig.9, 132-7) defined in the data type MyCustomer of the component itself.

When the NotifyCustomer accesses the attribute EmailAddress(Fig. 9, 132-8) of the data type MyCustomer of the component itself, the Get accessor of EmailAddress
sends the instance reference of the business entity object HotelCustomer while invoking the attribute conversion method GetEmailAddress of the data type conversion object MyCustomerConverter. The method GetEmailAddress of invoked MyCustomerConverter returns the e-mail address of the customer stored in the attribute Email of HotelCustomer to the Get accessor of the attribute EmailAddress of MyCustomer. In this process, the e-mail attribute value of the business entity HotelCustomer type is converted to an attribute value EmailAddress of the data type MyCustomer of the component itself. As a result, the e-mail address of the customer is stored in the local variable mailAddr of the service method NotifyCustomer, the e-mail address having been entered in the attribute Email of HotelCustomer instance received as a parameter. NotifyCustomer through such a data type conversion and data transfer process then transmits the contents of the e-mail received as a parameter message to the e-mail address of the customer stored in mailAddr.

The following process described as an example of another task is the same as that of sending an e-mail: assuming that an application user enters a new e-mail address and commands change in order to change the e-mail address of customers of a specific hotel, the user interface module requests, to the framework, an instance of the service object CustomerMgt of the service component ServiceComponent for executing a corresponding function. The framework then creates instances of CustomerMgt and the data type converter object MyCustomerConverter associated therewith, with reference to the dependency injection information file, then injects the instance reference of MyCustomerConverter in the attribute TypeConverter for injecting dependency of the data type MyCustomer of the component itself and then returns the instance of CustomerMgt to the user interface module.

When the user interface module then sends the instance reference of the business entity HotelCustomer and a new e-mail address as a parameter while invoking the service method UpdateEmailAddress (Fig. 8, 131-2) for changing the e-mail address in the service object CustomerMgt, the invoked UpdateEmailAddress converts HotelCustomer received as a parameter to the object MyCustomer to store it in the variable cust, using the conversion operator MyCustomer defined in the data type MyCustomer of the component itself.

When UpdateEmailAddress then arranges the new e-mail address value newAddress received as a parameter in custEmailAddress, the Set accessor of the attribute EmailAddress of MyCustomer sends the instance reference of the business entity object HotelCustomer and the new e-mail address as a parameter while invoking the type conversion method SetEmailAddress of the data type converter object MyCustomerConverter. The method SetEmailAddress of the invoked MyCustomerConverter arranges the new e-mail address received as a parameter in the attribute Email of
HotelCustomer. Through this process, the service method UpdateEmailAdresse which uses the data type of the component itself can change the attribute value of the business entity to the contents received from an application user. The method UpdateEmailAddress sends the reference of the business entity object with the changed e-mail address to the component on the data service layer as a parameter to commit the database to change the hotel customer information.

As seen from the exemplary embodiments, the invention enables a service component previously developed independently of a business entity type to handle values of the business entity type actually used in an application by means of a data type converter component implemented in developing the application.

It should be noted that the aforementioned embodiments illustrate rather than limit the invention, and that those skilled in the art will be capable of designing many alternative embodiments without departing from the scope of the invention as defined by the appended claims. In the claims, any reference signs placed in parentheses shall not be construed as limiting the claims. Use of the word "comprising", and its conjugates, does not exclude the presence of elements or steps other than those listed in any claims or the specification as a whole. The singular reference of an element does not exclude the plural reference of such elements and vice-versa. The mere fact that certain measures are recited in mutually different dependent claims does not indicate that a combination of these measures cannot be used to advantage.
Claims

A method of developing a component, not dependent on a business entity of a specific application in an object-oriented software development environment with a means for defining a root data type which all business entities should commonly inherit and a data type of the component itself, in order to enable components independently developed by a third party to be reused in a binary format without modifying source codes, characterized by comprising the steps of:

a. using the root data type instead of a business entity type if required for a developer to use a specific business entity as a data type of a parameter or return value while defining the service method of a component object;
b. the developer invoking a means for defining the data type of the component itself in the software development environment and then defining the data type of component itself and a data type converter interface according to the guidance of the means (this step comprises the following substeps (b1 to b7):

b1. when the developer enters any data type name in the means of defining the data type of the component itself, the means creating a data type class of the component itself in the same name and declaring a field variable in a root data type for accommodating the business entity object therein;
b2. the means for defining the data type of the component itself defining a creator method with a parameter in a root data type in the data type class of the component itself on the basis of the data type name received in the step (b1);

D
b3. the means for defining the data type of the component itself implementing a conversion operator method for converting the business entity type inherited with the root data type to the data type of the component itself, using the creator defined in the step (b2);
b4. the means declaring a Get accessor and a Set accessor of each attribute in the data type class of the component itself when the developer enters the type and any name of an attribute(s) to be included in the data type of the component itself in the means for defining the data type of the component itself;
b5. the means for defining the data type of the component itself defining a data type converter interface consisting of methods to be used when the Get accessor and the Set accessor declared in the step (b4) convert the attribute of the actual business entity to the data type attribute of the component itself;
Db6. the means for defining the data type of the component itself declaring a dependency injection field and an attribute of the interface type defined in the step
(b5) as a variable for accommodating the reference of the data type converter object to be implemented; and
b7. the means for defining the data type of the component itself completing the Get accessor and the Set accessor of each attribute declared in the step (b4), using the variable of the field declared in the step (b6), to finish the process of defining the data type of the component itself;
c. the developer preparing a code for converting the variable in a root data type used as a parameter of the service method in the step (a) to the data type of the component itself defined in the step (b), as a first statement of the service method; and
d. the developer using the variable converted to the data type of the component itself in the step (c) whenever he has to use the business entity object received as a parameter while preparing the rest part of the code of the service method.

In a method of assembling service components developed according to the method as claimed in claim 1 without modifying source codes in an object-oriented software development environment with a means for implementing a data type converter to develop a new application, a method of developing software, characterized in that a method of implementing a data type converter to be used in coordinating nonconformity between the data type of the component itself and a business entity type actually to be used in a specific application comprises the steps of:
a. a developer selecting a service component for which implementing a data type converter is required in a software development environment and invoking a means for implementing the data type converter;
b. the means for implementing a data type converter identifying and showing all of data type converter interfaces defined in the service component selected in the step (a) and the developer selecting one interface among them which has not been implemented with a data type converter class;
c. the means for implementing a data type converter identifying the data type of the component itself corresponding to the data type converter interface selected in the step (b);
d. the means for implementing a data type converter creating a class for implementing the data type converter interface selected in step (b);
e. the means for implementing a data type converter identifying and showing all of the business entities of the application and the developer selecting a business entity among them corresponding to the data type of the component itself selected in the step (c);
f. the means for implementing a data type converter identifying and showing all
of the methods included in the data type converter interface selected in the step
(b) and the developer then selecting one method among them to be implemented;
g. the means for implementing a data type converter identifying and showing all
of the attributes of the data type of the component itself identified in the step (c)
and the attribute of the business entity identified in the step (e) and the developer
then selecting a pair(s) of attributes which the method selected in the step (f)
handles among them;
h. the means for implementing a data type converter creating a data type
converter object code in the class created in the step (d) for interworking with the
attributes selected in the step (g), and the developer adding a type conversion
statement to complete the process of implementing the method selected in the
step (f) as far as the type of the attributes does not conform to each other;
i. the means for implementing a data type converter showing the state of im-
plementing the method of the data type converter interface selected in the step
(b) and the developer repeating the steps from (f) to (h) to complete the class of
implementing the data type converter interface selected if there is any method to
be implemented;
j. the means for implementing a data type converter showing the state of im-
plementing the class for the data type converter interface identified in the step (b)
and the developer then repeating the steps from (b) to (i) until all the interfaces
are implemented if there is any interface to be implemented;
k. when the developer selects a statement for creating a dependency injection in-
formation file in the means for implementing a data type converter, the means
creating a file of the dependency injection information between the data type
converter object and the data type of the component corresponding thereto as a
file, the data type converter object having been implemented through the steps
(b) to (j); and
l. the developer issuing a compiling command for the class prepared through the
steps (a) to (k) then to create a data type converter component.

A method of running object-oriented component based software on a computer
equipped with an environment for running the object-oriented software, char-
acterized by a process of running an application on a framework provided with a
function of injecting dependency, using a service component developed
according to the method as claimed in claim 1 and a data type converter
component developed according to the method as claimed in claim 2, comprising
the steps of:
a. a framework creating an object of a specific service component and an object
of a data type converter class(es) associated therewith with reference to a de-
pendency injection information file when a client process external to the framework requests the object of the corresponding specific service component arranged in the framework;

b. the framework injecting the reference of the data type converter object(s) created in the step (a) in the data type of the service component itself corresponding to each converter object with reference to the dependency injection information file;

c. the framework returning the service component object which was created in the step (a) and underwent the dependency injection process in the step (b) to the client process which requested the service component object;

d. the client process sending the reference of a business entity as a parameter while invoking a service method of the service component object received in the step (c);

e. the method of the service component object invoked in the step (d) converting the business entity object received as a parameter from the client process to the data type object of the component itself;

f. the service method invoked in the step (d) accessing a specific attribute of the data type object of the component itself;

g. the accessor of the attribute accessed in the step (f) sending the reference of the business entity object while invoking a type conversion method associated with the attribute of the data type converter object;

h. the type conversion method invoked in the step (g) converting the attribute of the business entity object to the data type attribute of the component itself corresponding thereto; and

i. the service method invoked in the step (d) repeating the steps (f) to (h) as many times as required to complete performance of its own service.
[Fig. 1]

Start to implement a component service method

Required to use user defined data type as a parameter or return value?

Use the root type instead of the business entity type to define a method signature

Define the data type of the component itself and the converter interface type to be used instead of the business entity

Convert the root type variable received as a parameter to the data type of the component itself

Use the data type variable of the component itself to complete implementation of the component service method
1. Declare the field variable of the root type
2. Define a creator with a parameter of the root type
3. Use the creator to implement a conversion operator for converting the root type to the data type of the component itself
4. Declare the attribute of the data type of the component itself (type and name)
5. Define the data type converter interface
6. Declare the field and attribute of the converter interface type for accommodating the reference of the converter object
7. Use the converter reference field variable to complete the Get/Set accessor of each attribute
Create a data type component development project.

Identify all of converter interfaces of the service component and then select one from those not implemented with the converter.

Identify those corresponding to the selected converter interface among the data types of the service component itself.

Create a converter class implementing the converter interface selected in the converter component development project.

Identify those corresponding to the data type of the component itself identified among the business entity objects.

Select one not implemented among the methods of the selected converter interface.

Identify those that the selected method should convert among the attributes of the business entity and the data type of the component.

Prepare codes for converting the values of the identified attributes thus to complete implementation of the identified method.

Create dependency injection information file between the converter object and the data type of the component itself.

Compile the data type converter component development project.
namespace CommonObjects
{
    public class EntityRoot
    {
    }

    [AttributeUsage(AttributeTargets.Interface)]
    public class TypeConverterInterfaceAttribute : System.Attribute
    {
    }

    [AttributeUsage(AttributeTargets.Class)]
    public class SelfDefinedTypeAttribute : System.Attribute
    {
    }

    [AttributeUsage(AttributeTargets.Property)]
    public class DependenciesInjectionAttribute : System.Attribute
    {
    }
}
namespace ApplicationEntities
{
    using CommonObjects;

    public class HotelCustomer : EntityRoot
    {
        public string CustId;
        public string Name;
        public string PostCode;
        public string Address;
        public string Email;
        public DateTime LastVisit;
    }
}

namespace ServiceComponent
{
    using CommonObjects;

    public class CustomerRst
    {
        public void NotifyCustomer(EntityRoot customer, string message)
        {
            MyCustomer cust = (MyCustomer)customer;
            string mailAddr = cust.EmailAddress;

            // Send 'message' to 'mailAddr'
            
        }

        public void UpdateEmailAddress(EntityRoot customer, string newAddress)
        {
            MyCustomer cust = (MyCustomer)customer;
            cust.EmailAddress = newAddress;

            // Update customer information in datastore with 'cust'
            
        }
    }
}
namespace ServiceComponent
{
    [SelfDefinedType]
    public class MyCustomer
    {
        [DependencyInjection]
        public static IMyCustomerConverter typeConverter;

        private EntityRoot bizEntity;

        public MyCustomer(EntityRoot entity)
        {
            this.bizEntity = entity;
        }

        public static explicit operator MyCustomer(EntityRoot entity)
        {
            MyCustomer myCustomer = new MyCustomer(entity);
            return myCustomer;
        }

        public int CustomerNumber
        {
            get { return typeConverter.SetCustomerNumber(bizEntity); }
            set { typeConverter.SetCustomerNumber(bizEntity, value); }
        }

        public string CustomerName
        {
            get { return typeConverter.SetCustomerName(bizEntity); }
            set { typeConverter.SetCustomerName(bizEntity, value); }
        }

        public string EmailAddress
        {
            get { return typeConverter.SetEmailAddress(bizEntity); }
            set { typeConverter.SetEmailAddress(bizEntity, value); }
        }
    }
}

namespace ServiceComponent
{
    [TypeConverterInterface]
    public interface IMyCustomerConverter
    {
        int GetCustomerNumber(EntityRoot entity);
        void SetCustomerNumber(EntityRoot entity, int newValue);

        string GetCustomerName(EntityRoot entity);
        void SetCustomerName(EntityRoot entity, string newValue);

        string GetEmailAddress(EntityRoot entity);
        void SetEmailAddress(EntityRoot entity, string newValue);
    }
}
using CommonObjects;
using ApplicationEntities;
using ServiceComponent;

class MyCustomerConverter {
    public int GetCustomerNumber(EntityRoot entity)
    {
        HotelCustomer hotelCustomer = (HotelCustomer)entity;
        return System.Convert.ToInt32(hotelCustomer.CustId);
    }

    public void SetCustomerNumber(EntityRoot entity, int newValue)
    {
        HotelCustomer hotelCustomer = (HotelCustomer)entity;
        hotelCustomer.CustId = newValue.ToString();
    }

    public string GetCustomerName(EntityRoot entity)
    {
        HotelCustomer hotelCustomer = (HotelCustomer)entity;
        return hotelCustomer.Name;
    }

    public void SetCustomerName(EntityRoot entity, string newName)
    {
        HotelCustomer hotelCustomer = (HotelCustomer)entity;
        hotelCustomer.Name = newName;
    }

    public string GetEmailAddress(EntityRoot entity)
    {
        HotelCustomer hotelCustomer = (HotelCustomer)entity;
        return hotelCustomer.Email;
    }

    public void SetEmailAddress(EntityRoot entity, string newEmail)
    {
        HotelCustomer hotelCustomer = (HotelCustomer)entity;
        hotelCustomer.Email = newEmail;
    }
}

<?xml version="1.0" encoding="utf-8" ?>
<components>
    <component id="customerMgt"
        class="ServiceComponent.CustomerMgt">
        <selfDefinedDataTypes>
            <selfDefinedDataType class="ServiceComponent.MyCustomer">
                <property name="TypeConverter">
                    <ref local="myCustomerConverter" />
                </property>
            </selfDefinedDataType>
        </selfDefinedDataTypes>
    </component>
    <component id="myCustomerConverter"
        class="ServiceComponent.Converter.MyCustomerConverter"/>
</components>