Design Patterns in Enterprise Application Integration for e-Learning Arena

Sidhant Rajam, Ruth Cortez, Alexander Vazhenin, Subhash Bhalla
Graduate Department of Information Technologies and Project Management
University of Aizu, Fukushima, Japan
{m5132202, m5132203, vazhenin, bhalla} @u-aizu.ac.jp

ABSTRACT
Pattern based design is an effective way to avoid an expensive process of reinventing, rediscovering and revalidating agnostic software artifacts. The Enterprise Application Integration (EAI) leverages the reusability factor of an application by applying decoupling and location transparency in the communication of the disparate applications and services. Design patterns are reusable solutions to solve recurring issues pertaining to the Functional, Non-Functional and Implementation tasks. The e-Learning is an ever growing and expanding arena. It has huge number of disparate applications and services that can be exposed over a ubiquitous media, such as the Internet, to the various kinds of end users. Therefore, the EAI is an important aspect in the e-Learning Arena in order to increase the high reusability and application decoupling factors. In this paper, we are imitating the Model-View-Controller (MVC) design patterns in order to explore the other composite patterns for an efficient integration of the applications and services. The demarcation of a Functional (View) and an Implementation (Model) task can be achieved deliberately by inducing an Integrator (Controller). The Controller can be further enriched to encapsulate certain Non-Functional activities such as security, reliability, scalability, and routing of request. This enables the separation of an Integration Logic from that of a Functional Logic (Client Application) and an Implementation Logic (Service). The Controller can be viewed by using the compound design pattern of the Enterprise Service Bus (ESB). This paper discusses how the Dependency Injection pattern is used in the ESB pattern for the integration of the e-Learning applications.

Categories and Subject Descriptors
D.2.12 [Software]: Interoperability – Distributed objects, Data mapping.

General Terms
Design, Algorithm, and Human Factors.

Keywords
Design Patterns, e-Learning, Enterprise Application Integration, Web services, Service Oriented Architecture, Enterprise Service Bus, and Dependency Injection.

1. INTRODUCTION
The e-Learning has seen the galloping advances over the past few years because of the consistently improving technology. The key drivers of such whopping advancements are: (a) high esteemed cloud computing technology (b) increasing penetration of high-speed broadband, (c) smart network-enabled devices, (d) the expansion of digital screen resolution and surface area, (e) powerful and high speed computing devices [3]. These technological advances are introducing new types of social interactions. The Web 2.0 technologies promote collaboration among learning communities, through tools such as Wikis and Social Networking, which have already found its roots among all types of generations [17]. This altogether paves a way for a paradigm shift from the traditional class room oriented learning methodologies to the next generation Learning Management Systems (LMS).

The LMS facilitates exchange of tools, functionalities, semantics and learning workflow. The end user based customization of an e-Learning platform is possible through integrating a wide range of underlying e-Learning services [5]. The inception of an e-Learning capacity for education is no longer restricted by physical, time and location constraints. Internet based e-Learning arena faces the challenge of managing the learning objects in an open and scalable architecture [14]. Another challenge is creating scalable technologies which support an arbitrary number of users while offering them with a personalized and customizable learning environment [14]. Therefore, it is obvious that disparate e-Learning applications and services are being developed over heterogeneous technologies and platforms. Thereby, reusability and interoperability are the important aspects for developing and exposing e-Learning services.

An interface based wrapper over the LMS services can abstract the low level service logic and forms a standardized contract in order to compose or assemble the learning services that solve complex problems. If integration logic is detached from the interfaced service component, the service component becomes more autonomous and reusable [7,8]. In order to leverage these interoperability and reusability virtues at further heights, the non-functional logic such as Quality of Service (QoS), security, reliability, scalability etc. can be taken out from the interfaced...
service component [17,18]. The service component can concentrate only on the business logic and becomes interoperable with other architectures and frameworks. The interface defines the metadata and the overall behavior of the service [8]. Reusability of an existing solution is the key characteristic behind the Enterprise Application Integration (EAI). Hence, it is very important to demarcate the integration logic and the non-functional logic from the e-Learning applications and services in order to achieve high reusability and interoperability [10,12]. The demarcated logic can be visualized in the form of a middleware solution that sits between a service invoker and a service provider. This middleware solution can also be enhanced for certain complex operations such as data and protocol transformation, request message routing and service composition or workflow. Such decoupled applications and services require a flexible integrator as a controller.

The design patterns are important means to capture reusable design knowledge that provides a proven solution to the recurring problems which might arise in a particular context or domain [7,10,12,21]. The design patterns based approach is helpful to avoid expensive cycle of revalidating, reinventing and rediscovering common software solutions [21]. The Model-View-Controller (MVC) is also one of the compound design pattern solutions for web based application development [11,16]. The MVC is an ideal and effective pattern to contemplate the e-Learning arena. The view layer holds functional requirements of an application while the model holds the business logic of a service. The controller acts as a middleware to control the information exchange between the view and the model. In our current research, we are enhancing the controller to form an Enterprise Service Bus (ESB) that holds the service integration logic and the non-functional logic [2,6]. We are proposing a design pattern based approach for the seamless integration among disparate e-Learning applications and services [7]. The Dependency Injection (DI) pattern meets most of the requirements. In this paper, we are exploring the DI design pattern to find a solution for integrating and decoupling the e-Learning services and applications [9].

The remainder of this paper is organized as follows. In Section 2, some approaches are reviewed related to the e-Learning systems design strategies, implementation of MVC design patterns and the EAI. Section 3 introduces and describes an overview of the DI pattern. Section 4 describes our earlier research work related to the development of an e-Learning model and the ESB. Section 5 explains our approach about how to use the DI pattern in the ESB. The conclusion and future work are discussed in Section 6.

2. RELATED WORK
The first generation of e-Learning platforms includes applications that were mainly associated with the delivery and interoperability of the presentation and content design within a particular platform. On top of the content sharing, the second generation also focuses on sharing learning objects, workflow, semantic exchange, and service oriented approach [5,17,18]. The contemporary e-Learning platforms introduce a new idea by which users can dynamically compose the required services from a vast rage of the e-Learning services [5]. Considering these requirements, an e-Learning platform based on an open architecture, high reusability, interoperability, flexibility and compatibility with external systems is required. In our research work related to the e-Learning arena, we have proposed the e-Learning Computational Cloud (eLC^2) based on MVC design patterns [4,17]. The eLC^2 is a software development platform for deploying the e-Learning services.

As shown in the Fig. 1(A), in the original MVC design pattern [16], the view updates itself from the model, via the Observer pattern. Hence, the original MVC design pattern works like a closed loop wherein the view talks to the controller, which connects to the model, and this in turn talks to the view [11,16]. On the contrary, the eLC^2 collaborate the reusable services in the model that is wrapped inside the reconfigurable controller which acts as an interface integrator and transforms the request/response parameters of the end user view to the model. Therefore, the eLC^2 has redefined the original MVC design patterns by removing the direct link between the model and the view and by reorganizing the controller in the middle [17,18]. The redefined version of the MVC design pattern is shown in Fig. 1(B), in which the controller acts as a single point of contact for the view layer and the model layer.

The controller is further enriched to form the ESB [18]. The model is normalized into the business logic layer and the database layer, as shown in Fig. 1(B). The Data Access Object (DAO) pattern is used to transfer the state of the data from database layer to the business access layer. This design is mainly motivated from the Apache's Struts and SpringSource's Spring frameworks [1,19]. Both Struts and Spring frameworks are developed in Java 2 Platform Enterprise Edition (J2EE) and imitates MVC design patterns. These frameworks use Extensible Markup Language (XML) configuration files which considers XML elements as descriptors in order to deploy objects [1,19]. Struts follows only MVC design pattern based web application development. On the other hand, Spring is a combination of multiple design patterns including MVC, Aspect Oriented Programming (AOP), Java Database Connectivity (JDBC) Integration Framework and EJB Integration Framework. Simultaneously, it has a wide range of the view components including JavaServer Pages (JSP), Extensible Stylesheet Language Transformation (XSLT), Java Server Faces (JSF), Velocity, Freemarker etc. for content generation. Spring framework supports DI pattern with Inversion of Control (IoC) to manage the JavaBean objects using runtime callbacks. Hence, Spring framework enables the developers to mix and match multiple frameworks such as Apache Struts and libraries while developing and deploying the applications [19]. The service object that is managed by the Spring IoC container is called as a bean. A bean is an object which is instantiated, assembled and managed by a Spring IoC container. The Spring container stores XML based configuration metadata that reflects the bean and its dependencies.
In our current research, we have introduced the DI pattern in the controller that is enhanced to form the ESB [18]. The controller is acting as a centralized container that mediates a request received from the view in order to find and invoke the appropriate service which is deployed in the model. The ESB is built on Service Oriented Architecture (SOA) methodologies. The ESB is provisioned for the service virtualization, which is a service plumbing that focuses on building new functionalities and services without bothering about how the services will be exposed, consumed and maintained [18]. The service virtualization has abstracted peer services for transparent service lookup.

3. OVERVIEW OF DEPENDENCY INJECTION PATTERNS

Dependency Injection design pattern was initially proposed by Martin Fowler in 1996 and later on enhanced it in 2004 [9]. DI pattern is influenced by Johnson and Foote's proposal on IoC [20]. It enables the client application to consume a service without knowing in advance the specifics or origins of the consumed service. Therefore, the end user application is not required to hold the knowledge about the interface, behavior and implementation of a service. The DI pattern hides the service logic implementation that is not apparent from the interface [15,20]. The client object neither looks up for its dependencies, nor knows the location or class of the dependencies. The client in the DI pattern has a passive role that requires an external mechanism to pass the dependent objects. This mechanism is called as Injection. The main benefit of the DI pattern is that it removes the dependency from concrete implementation of the business logic [9].

Mathematical problems often require processing various standard formulae. In Fig. 2, it is shown an example of a Formula engine that sets and processes any mathematical formula. The client class is the FormulaEngine which processes a given formula. If the required formula is hard-coded into the FormulaEngine, the FormulaEngine remains stuck to a specific formula and loose the reusability and extendibility factors. As the main function of the FormulaEngine is a formula, this formula becomes its dependency. This dependency is declared as an interface that describes the behavior of a formula. An interface focuses on the requirements of the client and describes what functionality a client will receive from an object, avoiding tight coupling the client to the object's implementation [11]. The FormulaEngine holds a 'use' type of association with the dependency interface of a Formula. The Formula interface can be implemented in many forms for different kinds of formulae such as simultaneous equation, Taylor series, mathematical derivatives and integration. The FormulaImplementer class implements the Formula interface for one of such formulae. All the functions that are declared in the Formula interface are implemented in the FormulaImplementer class. When the FormulaImplementer class is instantiated, it becomes a dependency object of the data type formula for the FormulaEngine. This dependency object is required by the client class to complete its operation.

If the client finds and instantiates the dependency object, it becomes tightly coupled with the dependency implementation class. The client is not responsible to receive the dependent object [9]. An external mechanism known as the Assembler is used to inject the dependent object into the client object. The class Assembler can be further extended to form a container to control all the activities of the client class.

The important property of the DI pattern is that the client does not have any dependencies onto the Assembler for locating the dependent object [9]. Therefore, the Assembler is responsible to (1) instantiate the client object (FormulaEngine), (2) understand the dependency data of the client object (Formula interface), (3) locate and instantiate the dependent object (FormulaImplementer), and (4) inject the dependent object into the client object (setFormula(p_formula:Formula)). The injection of the dependent object can be done in various ways such as Interface Injection, Setter Injection as shown in the Fig. 2, and Constructor Injection. The setter type of DI is better than that of constructor or interface because it allows changing the object over the lifetime of the client object. Also by using setter injection method, the client object can be instantiated without even knowing about the dependency object.

The DI is a very useful alternative to the Service Locator Pattern where every client class has a dependency on the registry class of Service Locator [9]. In the Service Locator pattern, the
client class has to ask for the dependent object explicitly by issuing a message to the Service Locator class. On the other hand, in the DI pattern there is no explicit request message call, but the dependent object is injected in the client class from the external mechanism of the Assembler class. Hence, DI pattern is a IoC where the control of the client class is maintained by a container [20]. By introducing DI with IoC, the managed object is decoupled from any hard-coded dependencies on its collaborators including expensive initialization logic, and thereby the object can be tested in isolation. As a result of this, the managed object is decoupled from an overhead of the collaborator's lookup action and the object does not have to bother about the reference of the collaborator. We are following a Java implementation for the wiring of the dependency externalization.

4. ENTERPRISE SERVICE BUS

As discussed in Section 2, from our previous research, the redefined MVC design pattern is used to decouple the view layer and the model layer by introducing the controller layer in the middle [16]. The controller is further enhanced to perform non-functional logic and integration logic. Therefore, the main role of the controller is to integrate a service interface and ensure that certain QoS properties related to the integration such as security, performance, and availability work by negotiating the Service Level Agreements (SLA) with the e-Learning service consumer and the service provider. The controller is viewed by using the compound design pattern of the ESB [18]. In the ESB approach, integration broker capabilities are distributed across a loosely coupled integration network. The integration components are independently deployed and managed [2]. The core design pattern in the ESB is the DI pattern that manages the integration components.

Fig. 3 describes how the e-Learning applications and services are collaborated through the ESB by using the MVC design pattern. The view layer consists of an e-Learning Application Portal and an External System such as MOODLE which is a Course and Learning Management System. The model layer consists of e-Learning services (business logic) and data. The controller layer is the ESB for the purpose of interface integration between disparate and heterogeneous e-Learning applications (i.e. view layer) and services (i.e. model layer). The controller layer can communicate in two modes with the ESB, via Business-To-Client (B2C) and/or Business-To-Business (B2B). The integration task mainly involves Data Transformation, Protocol Transformation, Message Routing (Mediation), and Service Composition [18].

Figure 2. Dependency Injection Pattern.

Figure 3. Enterprise Service Bus services collaboration in MVC design pattern.
The DI pattern is implemented in a Service Orchestration Engine. The view layer communicates to the ESB through web services interface with XML Simple Object Access Protocol (SOAP) message over Hypertext Transfer Protocol (HTTP) or HTTPS protocols. The request from the view layer follows an itinerary based messaging. The itinerary message represents a set of discrete message routing operations. The ESB separates the service definition from the mechanism for locating and invoking the services. Depending on the request parameters given in the message itinerary, the ESB invokes the appropriate service components including Protocol and Message (Data) Transformation, Ad-hoc Service Stack [18].

The Ad-hoc Service Stack is a pool of auxiliary services. These auxiliary services are responsible for Security, Monitoring, State Maintenance (or Session Control), Auditing and Logging. However, the auxiliary services are non mandatory. Its usage is decided as per the SLA document which is mutually agreed by both the service consumer (view layer) and the service provider (model layer). Every application and its corresponding request are mapped for certain auxiliary services and it is documented in the SLA. The Service Virtualization mediates a request from the view layer in order to invoke and execute the desired e-Learning service which is implemented in the model layer. All the service interfaces are registered in the Service Repository and Registry. The Event and Transaction Controller manages the web services transactions across the distributed systems including Local Database, and Message Queues, for two-phase atomic commit and concurrency control. The Data Model for Service Configuration is used for the service related data processing such as service definition, transactional data etc. The ESB acts as a container for instantiating the dependent and the client objects as well as keeping the references of those objects in the service pool and then ESB injects the dependent objects into the client objects. The view layer knows only the ESB and it needs to fill up an itinerary request form which is defined by the ESB. This form contains all the information about the required functionalities by the view layer.

The view layer issues a high level request such as "Solve Taylor Series Equation" and provides the input data payload. The ESB contains the service mapping information for the "Solve Taylor Series Equation" request in the Service Registry and Repository. The services are deployed and maintained in the model layer. By using this service mapping, the ESB composes the endpoint services that are required to execute the "Solve Taylor Series Equation" request. The Service Virtualization mediates the respective services by using Service Mediation Pattern. This approach does not need the Universal Description Discovery and Integration (UDDI) registries and thereby, the view layer does not need to know the endpoint services. Therefore, it increases the service interoperability and reusability.

5. DEPENDENCY INJECTION FOR ENTERPRISE SERVICE BUS

As discussed in Section 4, the DI is implemented in the service orchestration engine of the ESB. The integration components such as Protocol Transformer, Data or Message Transformer, and Service Virtualizer, are registered in the local data model or database of the service orchestration engine. The integration components are considered as client classes in the ESB. Every integration component declares its dependencies in the form of an interface which is implemented by some implementer component. This implementer component is also registered in the orchestration engine. Therefore, the integration components are managed by the orchestration engine which acts as a container for the ESB.

The local database schema for storing the mappings of the integration components is shown in Table 1. One component can have many interfaces or dependencies. Hence, the columns Component ID and Dependency Injection Interface form a composite key for this mapping to uniquely identify particular component. As discussed in Section 3, there are three types of dependency injections such as setter, constructor and interface based. These injection types are denoted by the column Dependency Injection Type. The column Injection Function Name denotes which function will be injected for the dependent object. The Main Function Name denotes the bootstrap function of the class mentioned in the column Component Class. Table 1 shows the tuples of various integration components such as Data Transformer (DATA_TNSFR), Protocol Transformer (PROTOCOL_TNSFR) and Service Virtualizer (SERVICE_VRTL).

<table>
<thead>
<tr>
<th>Component ID</th>
<th>Component Class</th>
<th>Dependency Injection Interface</th>
<th>Dependency Injection Type</th>
<th>Injection Function Name</th>
<th>Main Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_TNSFR</td>
<td>DataTransformer</td>
<td>DataInterface</td>
<td>Setter</td>
<td>setDataTnsfr</td>
<td>beginDataTnsfr</td>
</tr>
<tr>
<td>SERVICE_VRTL</td>
<td>ServiceVirtualizer</td>
<td>ServiceVrtlInterface</td>
<td>Setter</td>
<td>setServiceVrtl</td>
<td>beginServiceVrtl</td>
</tr>
</tbody>
</table>

The local database schema for storing the mappings of the integration components is shown in Table 1. One component can have many interfaces or dependencies. Hence, the columns Component ID and Dependency Injection Interface form a composite key for this mapping to uniquely identify particular component. As discussed in Section 3, there are three types of dependency injections such as setter, constructor and interface based. These injection types are denoted by the column Dependency Injection Type. The column Injection Function Name denotes which function will be injected for the dependent object. The Main Function Name denotes the bootstrap function of the class mentioned in the column Component Class. Table 1 shows the tuples of various integration components such as Data Transformer (DATA_TNSFR), Protocol Transformer (PROTOCOL_TNSFR) and Service Virtualizer (SERVICE_VRTL).

Table 2. Dependency Implementer Mapping Table.

<table>
<thead>
<tr>
<th>Component ID</th>
<th>Dependency Interface</th>
<th>DI Implementer ID</th>
<th>DI Implementer</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_TNSFR</td>
<td>DataInterface</td>
<td>XML_DATA_TNSFR</td>
<td>XMLDataTnsfr</td>
</tr>
<tr>
<td>DATA_TNSFR</td>
<td>DataInterface</td>
<td>CSV_DATA_TNSFR</td>
<td>CSVDatatnsfr</td>
</tr>
<tr>
<td>DATA_TNSFR</td>
<td>DataInterface</td>
<td>JSON_DATA_TNSFR</td>
<td>JSONDataTnsfr</td>
</tr>
<tr>
<td>PROTOCOL_TNSF</td>
<td>ProtocolTnsfInterface</td>
<td>HTTP_PROTOCOL_TNSF</td>
<td>HTTPProtocolTnsf</td>
</tr>
<tr>
<td>PROTOCOL_TNSF</td>
<td>ProtocolTnsfInterface</td>
<td>FTP_PROTOCOL_TNSF</td>
<td>FTPProtocolTnsf</td>
</tr>
</tbody>
</table>
The schema for storing the mappings of the dependency implementer components or dependency classes is described in Table 2. This table refers to the Table 1 of Integration Components by the foreign keys of Component ID and Dependency Interface. The column DI Implementer ID is mapped to the foreign keys. The column DI Implementer Class denotes the name of the dependency implementation class. Table 2 shows the tuples of various kinds of DI implementers, for example, in case of Data Transformation there are XML Data Transformer (XML_DATA_TNSFR), Comma-Separated Values or CSV Transformer (CSV_DATA_TNSFR) and JavaScript Object Notation or JSON Transformer (JSON_DATA_TNSFR).

An algorithm for implementing the DI pattern in the service orchestration engine is explained in Fig. 4. As discussed in the previous Section, the orchestration engine receives an itinerary based request message from the view layer. This message is processed by the orchestration engine in order to analyze what are the various integration components, and the respective dependency implementer classes required in order to execute the request. The description about this processing is out of the scope of this paper. Therefore, the algorithm assumes that the request message in req_msg is analyzed. The details about integration components in comp_tobe_executed and DI interfaces in DI_interfaces_required are known.

The orchestration engine loads the components mapping in inte_comp_map_schema, as given in Table 1, into the memory at the runtime. The orchestration engine also loads the DI implementation mapping in dep_imple_map_schema, as given in Table 2. Then, the orchestration engine traverses the list of all the integration components in comp_tobe_executed within a loop and instantiates a component in an every iteration to form a client object. During the same iteration, it instantiates the corresponding dependency interface implementer class and injects that dependency object into the client object. At the end of the every iteration, the orchestration engine invokes the main function of the integration component in order to execute the required functionality.

As discussed in Section 3, we are using Java J2EE platform to implement the DI in the ESB. To explain one of the integration components of data transformer, a Java code snippet is shown in Fig. 5. It reflects the DI pattern discussed in Section 3. The class Controller acts as an Assembler, which is a service orchestration engine in the case of the proposed ESB. The class DataTransformer is registered as an integration component in the Controller. There are various types of data formats as shown in Table 2, such as XML, CSV, and JSON. The class DataTransformer converts or transforms one type of data format into another. Therefore, the class DataTransformer depends on the function of data transformation. Thereby, this function can be considered as a dependency for DataTransformer. The dependency function is declared in the DataTnsfrInterface by two methods. Firstly read the input data format by readInputData() and secondly convert the input data format to the output data format by convertToOutputDataFormat(). The output data format is same as that of model layer. Every type of data transformation can be realized by a separate dependency implementer class that implements DataTnsfrInterface dependency interface.

The Fig. 5 uses XML data transformation and the class XMLDataTnsfr implements the DataTnsfrInterface. The interface implementer class XMLDataTnsfr is registered in the Controller. The Controller receives a request message (String reqMessage) from the view layer, and the request message is processed by a function orchestrationEngine. At the beginning, the function orchestrationEngine instantiates the DataTransformer integration component in order to form a client object, then it instantiates XMLDataTnsfr in order to form a dependency object. The dependency object is injected into the client object by using setter type of injection. After this, the client object is ready to process the data transformation task. The orchestrationEngine invokes the main function of the DataTransformer to begin the XML data transformation. The DataTransformer does not need to perform expensive initialization logic and therefore, the dependent objects can be tested separately in isolation. As a result of this autonomy, there is a potential scope for a modular programming paradigm such as AOP.

Figure 4. Algorithm for DI implementation in Controller’s Service Orchestration Engine.
The crosscutting concerns often refer to non functional logic such as security, monitoring, auditing and logging, synchronization, and QoS. The AOP is used to capture and encapsulate the crosscutting concerns within an independent modular unit and describes how the concerns code should be integrated into the client code [13]. The AOP complements Object Oriented Programming (OOP) by providing program components and aspect languages with different abstraction and composition mechanism. The fundamental unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect.

6. CONCLUSIONS

In this paper, we have seen that contemporary e-Learning applications and services are rather diverse and disparate with respect to the various development platforms and architectures. The communication gap and integrity are improving gradually. In order to achieve the high level of application and service reusability, it is more effective to integrate the applications and services rather than rebuilding because redevelopment is a costly affair. The design patterns based approach enables us to reuse the proven and consolidated design knowledge, supporting the development of high-quality software systems.

We have imitated the MVC design pattern and the controller is extended to form the ESB. The ESB performs high level functionalities so that the integration logic becomes independent from both the e-Learning applications as well as services. The ESB bolstered with DI pattern has reinforced the EAI by facilitating service loose coupling, global service contract, service implementation autonomy, service reusability, service modularity, statelessness and dynamic discoverability of the services. The DI pattern benefits the software development by bringing forth an abstraction layer between high level modules and low level modules. As a result of this abstraction layer, the integration components can be developed, deployed, tested and replaced independently. Therefore, the low level integration tasks can be seamlessly orchestrated. The DI pattern enables the ESB to grow or shrink as required by the network and workload being supported. The integration components become more extensible because it can be used with different kinds of implementations of the dependent code without undergoing any modifications in it. Due to such loose-coupling attribute, the DI can be used as "Plug-Points" of an application framework.

The ESB pattern ensures the robust communication between e-Learning applications and services, an itinerary based intelligent routing, sophisticated transaction and transformation of service data. The integration capabilities of the ESB are developed and implemented as separate services, which are injected as dependent objects into the service orchestration engine. The ESB abstracts the low level details of the Message Oriented Middleware (MOM) by removing the dependencies on message routines. Data exchange between web services happens over the open standards.

Figure 5. Example of Dependency Injection in Java for Data Transformation.
and protocols such as HTTP or HTTPS. Therefore, the ESB can be used for an effective EAI in e-learning, e-Business, e-Governance arenas. In a nutshell, the agility that is ensured by the SOA methodology can be fulfilled by using the ESB. Hence, the ESB can be contemplated as a standardized way to integrate heterogeneous e-Learning applications.

In our future research, on top of DI pattern, we are planning to introduce AOP methodologies in the ESB implementation. The AOP enhances the development ability to express the separation of crosscutting concerns or modular codes that are necessary for a well-designed, maintainable software system [13]. At the time of the actual development, the crosscutting concerns result in tangled and scattered code. The AOP separates tasks which should not be entangled with the crosscutting concerns, in order to provide better procedures for the interoperability and encapsulation.

7. ACKNOWLEDGMENT
This work is supported by the Japanese Society for the Promotion of Science, Grant-in-Aid for Scientific Research: 21500948, 2009-2011.

8. REFERENCES