

Towards a Re-engineering Method for Web Services Architectures

Claus Pahl and Ronan Barrett

Dublin City University, School of Computing
Dublin 9, Ireland
[cpahl|rbarrett]@computing.dcu.ie

Abstract. Recent developments in Web technologies – in particular through the Web services framework – have greatly enhanced the flexible and interoperable implementation of service-oriented software architectures. Many older Web-based and other distributed software systems will be re-engineered to a Web services-oriented platform. Using an advanced e-learning system as our case study, we investigate central aspects of a re-engineering approach for the Web services platform. Since our aim is to provide components of the legacy system also as services in the new platform, re-engineering to suit the new development paradigm is as important as re-engineering to suit the new architectural requirements.

Keywords: Software Re-engineering, Web Services Framework, Services-oriented Architectures, Architectural Transformation.

1 Introduction

Activities aiming to develop the Web from a document- to a services-centric environment are bundled in the *Web Services Framework* WSF [1]. The WSF philosophy is to open the Web for application-to-application use. It provides a framework based on description languages to describe services, a publication and discovery facility acting as a marketplace for providers and users of services, and protocols allowing services to be invoked in a distributed environment.

With the Web services technology becoming mature, it will impact existing Web-based and other distributed systems. Numerous software systems will be re-engineered for the Web services platform in the future. Re-engineering these systems as *service-oriented architectures* SOA [2] will become possible. The advantage of service-oriented architectures for Web-based environments in particular is a standardised interaction architecture [3, 4], allowing the flexible integration or exchange of components. This in turn will allow a different style of system design and implementation – more open and more based on sharing and reusing resources. Improved maintainability is another advantage.

Our aim is to investigate *central aspects of a re-engineering method* for the Web services platform. The Web services framework deployed as a service-oriented architecture [2] poses new architectural constraints on these systems.

This is an expected aspect, but we found that the new development style associated with the Web services framework also influences the re-engineering approach. Preparing an existing software component as a service not only for later deployment in a service-oriented architecture, but also as a software entity that can be made available through the WSF description and discovery mechanisms, is a central requirement that a re-engineering method needs to embrace.

The literature on *re-engineering* is rich; see e.g. [5, 6]. However, the focus is often on program design and implementation. Our contribution is a collection of re-engineering techniques specific to the WSF. In addition to program design questions such as modularisation, which has to be considered here in the context of architectures, the development aspects has turned out to be central. Re-engineering existing software entities as services and providing them to other users enables reuse and composition. We have focussed on architectural transformation to enable composition. Reference architectures and architectural patterns play a central role in coarse- and fine-granular architectural transformation.

We will use an *e-learning system* as our *case study* here to illustrate our findings [7]. This system is a Web-based, distributed system that is characterised by complex interaction processes. The existence of a reference architectures and standards for this area – such as the Learning Technology Standard Architecture LTSA [8] – will guide the re-engineering process.

Section 2 addresses the wider re-engineering context. In Section 3, we introduce our case study. We address the description and discovery of services in service-oriented architectures in Section 4. In Section 5, we focus on the architectural transformation and the assembly of services. We end with some conclusions.

2 Software Re-engineering

Software re-engineering is a central method to deal with *software evolution* [5, 6]. The requirements concerning a software system or the environment in which the system is running will inevitably change. The ability to adapt software to these changing needs (e.g. through re-engineering) is paramount.

Software re-engineering is concerned with the reimplementing of legacy software in order to improve its *maintainability* [6]. In our case, we are also interested in enabling *reuse* and *composition*, allowing components of the legacy system to be provided (and discovered) as services. This obviously widens the classical re-engineering focus. Re-engineering usually involves techniques such as source code translation, program structure improvement, modularisation, and data re-engineering [6]. *Modularisation* becomes more important, leading us to focus on *architectures* and *architectural transformation* [9]. This needs to be embedded into a concern for *development process aspects* for the target platform.

Modularisation is the process of (re)organising a software system such that related parts form coherent components. A service-oriented architecture is the target of the architectural transformation [9, 10]:

- *reference architectures* determine the architecture on a coarse-grained level,
- *architectural patterns* determine the architecture on a fine-grained level.

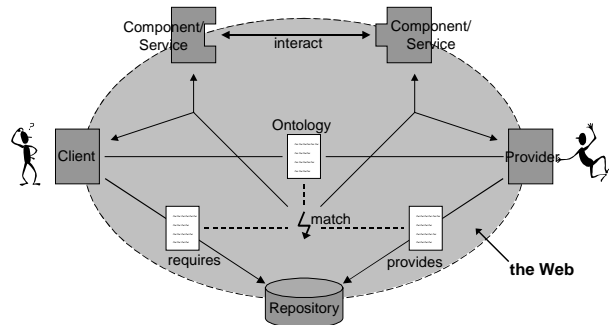


Fig. 1. A Web-based Development Process for Service-oriented Architectures.

Both introduce *standardisation* into the architectural transformation process.

We have already emphasised the importance of *description and discovery* for the targeted development process. Again, standards determine this context:

- *domain-specific notations* address the application domain,
- *software/service-specific notations* address software-related aspects.

An *ontology* [11, 12] – we have used the term to refer to the combined knowledge about domain-specific aspects (metadata) and software service aspects (interface specification) – forms the central description approach.

Developing software systems based on shared repositories of reusable components or services requires a different development style compared to the classical 'from-scratch' approach [13]. The Web as an open and distributed environment opens a new dimension to this new style of development. An outline of the *development process model* for the *target platform*, which summarises the description, discovery, architectural design, and assembly activities, is presented in Figure 1. It consists essentially of two core activities that can be supported by ontologies:

- *Discovery* – the lower part of the diagram that illustrates the marketplace idea where providers and clients (users) meet.
- *Assembly* – the upper part of the diagram that illustrates the assembly of service components to interacting systems.

3 The Case Study – an E-Learning System

Today, the Web is the predominant platform for *computer-supported teaching and learning*. Recent developments have seen more interactive media among the Web resources, allowing interactions between human users and provided services and also interactions between the services themselves.

We have been involved in the development and extension of a Web-based teaching and learning environment called IDLE [7] – the *Interactive Database*

Learning Environment. Recently, we started to convert the architecture of the system to a Web services-based platform [14]. Our report here is based on our experience with a prototype that we developed in order to investigate the re-engineering of systems for the WSF platform.

3.1 The Interactive Database Learning Environment IDLE

IDLE is a Web-based integrated learning and training environment for an undergraduate course programme. The system provides a wide range of educational features from audio-supported lectures and animation-based tutorials to active learning features for authentic database development. Some of the components are: audio server and audio player, animation and simulation features, a graphical modelling tool, an execution tool for a database language, a workspace feature for student projects, a content repository, a feedback feature, an evaluation tool, and a Web server and other delivery software. The complexity and diversity of the components requires architectural support in order to develop, extend, and maintain the system. Explicit interfaces between components have in the past supported extensions and maintenance of the system [15].

3.2 Learning Technology Standards – LTSA and LOM

Two *standards* are of relevance for the re-engineering process, addressing *architecture* and *description* aspects.

The *Learning Technology Standard Architecture* LTSA [8] has been developed by the IEEE Learning Technology Standards Committee LTSC in order to provide a framework for the development, evaluation and discovery of learning technology systems. It provides a basic architecture consisting of process and storage components and the interactions between them. The LTSA identifies *process components* such as coach, learner entity, evaluation, and delivery, and *storage components* such as learning resources or learner records. The objective is to enable sharing and reuse of learning technology components.

Sharing and reuse of educational resources is also the motivation behind the *Learning Object Metadata* LOM standard [16]. LOM is a metadata framework for learning objects that allows these objects to be annotated with basic properties that are important for instructors and learners, such as content information, learning goals, target audience and level, etc. Objects can then be made available to potential users (both learners and instructors). These objects will be discovered based on their metadata annotations.

3.3 Learning Objects and Architectures

Our objective here is to explore architectural aspects of re-engineering – illustrated in the context of Web-based teaching and learning environment (TLE) design and development. A service-oriented architecture (SOA) shall form the backbone of the re-engineered system.

A notion of *learning objects* is central in the case study context. It comprises content units but also functional components such as delivery or storage.

- *Domain-specific descriptions* address for instance the educational aspects relating to the learning object usage in the case study domain. This form of information supports the user (learner or instructor) in discovering and retrieving services (learning objects) within the terminology of the domain.
- *Infrastructure* and other *software-oriented descriptions* address the technical aspects relating to the service assembly within an architecture. This form of information supports the developer for instance in integrating a learning object into a TLE.

In the remainder of this section, we will focus on the first, domain-specific layer. The technical specifications are important in the context of architectural composition and transformation. We will revisit them in Section 5.

4.2 Domain-specific Description and Discovery

Retrieval and discovery is usually supported by a description and a query language, and accessible repositories for descriptions and possibly also the services themselves. We can distinguish closed, database-like repositories and open, e.g. Web-like repositories. Both differ in their navigation and search support.

The WSF supports these aspects, whereas LOM only provides a description notation (for learning objects in the case study domain). LOM, however, is an XML-based, *domain-specific metadata annotation format* addressing educational properties of learning objects. We envisage LOM-like, domain-specific annotations as the basis of discovery [18].

LOM defines attributes required to describe a learning object. It classifies attributes into nine categories addressing for example general, technical, educational, and lifecycle aspects. The provider of a service (a learning object) describes the service in terms of domain and infrastructure properties. A potential user – learner or instructor – then uses a related query language (or just a Web search engine) to formulate requirements in terms of the properties described.

4.3 Re-engineering for Discovery

Target services of the transformation need to be publishable services. Their description based on a domain ontology, such as LOM, is essential.

We illustrate the LOM standard briefly using the IDLE system to indicate the type of annotation that needs to be generated in the re-engineering process:

- General attributes such as `title='Introduction to Databases'`, `language='en'`, or `description='an introductory course for computing students addressing principles, models and languages for database systems'` can be used.
- Technical attributes include `format='text/html'` or `format='audio/mp3'` for instance. Other technical attributes are size and location.
- Educational attributes include the interactivity `type='active-simulation'` or `'active-exercise'` or `'expositive-audio'`. Other educational attributes are learning resource type, interactivity level, or semantic density.

The discovery of suitable objects is based on this domain-specific annotation layer, even when a Web services-based architecture is developed. Web services descriptions (in WSDL; cf. Fig. 3) are only relevant from a technical perspective when services have to be integrated into applications. Content of learning objects is the primary search criterion for a potential user. WSDL specifications address interface and messaging aspects. Consequently, their information is only relevant if a preselection has been made and a system needs to be assembled.

We have chosen LOM as one possible metadata format. Other standards, or adaptations to Semantic Web technology, might also be suitable. It is essential to consider this abstract, domain-specific layer in addition to WSDL descriptions to support the current trend in semantic Web services discovery.

5 Architecture

Modern object-oriented software systems are usually not monolithic. In the case study domain, various content objects might be used within a course; a variety of functions (such as storage or evaluation) support the content objects. Therefore, consideration of assembly and integration of these components in the re-engineering process in terms of the target architectural requirements is a central task. We will focus here on these more recent systems, ignoring unstructured and monolithic systems of the past. The re-engineering of the latter has been dealt with extensively in the literature.

5.1 Coarse-grained Architecture

Architectural Topologies and Reference Architectures. Service-oriented architectures are usually connected to a centralised control unit. However, peer-to-peer architectures are also possible. These *architectural topologies* and more specifically *reference architectures* strongly influence the *coarse-grained architecture* of a service-based system. For instance, the LTSA standard needs to be considered when defining architectures for Web services supported TLEs.

The LTSA exhibits the characteristics of a services-oriented architecture. In our terminology, both LTSA components and LTSA storage elements are learning objects with service character. They are defined in terms of interactions with their environment, i.e. the LTSA components can be provided in form of services. The LTSA defines a reference architecture for TLEs that will provide us with a first re-engineering tool in the architectural transformation.

Object/Service Interface Descriptions. The starting point for the architectural transformation are interface descriptions for both the legacy and the re-engineered system. Services in general are described in terms of two different aspects: domains-specific and software-oriented. The second aspect describes the infrastructural aspect. This aspect arises if services such as learning objects are considered as interacting computational entities. An interface description defines how to access the services provided by the object and how to interact with it.

```

<message name="LocatorInput">
  <part name="body" element="xsd1:Locator"/>          </message>
<message name="MultimediaOutput">
  <part name="body" element="xsd1:Content"/>        </message>
<message name="InteractionContentOutput">
  <part name="body" element="xsd1:InteractionContent"/> </message>

<portType name="Delivery">
  <operation name="GetContent">
    <documentation> Interaction with Learner Entity      </documentation>
    <input  message="tns:LocatorInput"/>
    <output message="tns:MultimediaContentOutput"/>
  </operation>
  <operation name="GetEvalData">
    <documentation> Interaction with Evaluation component </documentation>
    <input  message="tns:LocatorInput"/>
    <output message="tns:InteractionContentOutput"/>
  </operation>
</portType>

```

Fig. 3. A WSDL Specification of an LTSA Component (excerpts).

The WSF [1] provides a description language: the Web Services Description Language WSDL. This XML-based notation provides features to express

- the functionality of services in abstract terms: services are provided through connection points, called ports, which allow a user application to interact (send/receive messages to/from) different operations,
- the location and the protocols supported by the service, allowing a user to actually connect to the service.

LTSA components can be defined in terms of the WSDL. An excerpt from an LTSA service interface definition in WSDL can be found in Figure 3. It aims to demonstrate the closeness of the LTSA to component-oriented architectures.

Architectural Transformation. Our aim in the case study is to embed learning objects from the legacy system into a Web services architecture. LOM annotations can contain technical aspects, but they do not give any guideline on how to integrate and assemble service objects into larger systems. The LTSA provides a first top-level outline of a services-oriented architecture for TLEs.

We have outlined the main IDLE components above. The architecture of the system can be described in a number of ways. The architecture of IDLE system can be presented in the three tiers interface, server, and database backend. However, since several components fall into each tier, a refined architecture is necessary. In terms of the LTSA, we can associate IDLE components to LTSA components. Examples are the evaluation or delivery components.

The following activities support the *architectural transformation*:

1. *Legacy Architecture Standardisation*: In order to support architectural transformation, we have captured the IDLE architecture in terms of the LTSA reference architecture. This is a *modularisation* and *standardisation* step.
2. *Architectural Invariant Definition*: As pointed out, the LTSA exhibits service character. Therefore, the LTSA-imposed architecture forms an *invariant* in the transformation process that guarantees evolutionary stability.
3. *Target Architecture Representation*: LTSA components and storage features can be described and, if implemented, be made available as services in a Web services architecture; cf. Fig. 3.

Fig. 3 shows how the LTSA can be mapped to the WSF. The LTSA defines services (on an abstract level) that can be mapped to WSDL port types. Access to static components, called stores in LTSA, can also be encapsulated by service interfaces. However, in the case study it becomes clear here that the LTSA is only a reference architecture, identifying no more than component clusters. Several IDLE components are part of one LTSA element. In order to support the development of complex systems, a more fine-granular architectural support than provided by e.g. the LTSA is needed. Each of the LTSA components is often implemented through several objects or services.

5.2 Fine-grained Architectures

A more *fine-granular approach to architectural transformation* is required that gives a developer more support:

- Firstly, to identify *architectural patterns* [19] that are suitable in the architectural transformation of a services-based system. We will discuss different architectural patterns. Since the focus in Web-based systems is often on the user interaction with content, the Model-View-Controller MVC paradigm can be applied.
- Secondly, a *Web services description* of the service interfaces that supports the actual assembly and implementation of the system. SOAP toolkits, e.g. Apache Axis, can then be used to coordinate the service interactions.

Architectural Patterns. Software systems often follow common recurring architectural structures – called *design patterns* [19]. Patterns provide solutions to reoccurring problems that occur in object-oriented software development in order to make these maintainable, self-documenting, and reusable through component-based architectures with clearly defined interfaces. Patterns are usually described in terms of their underlying architectural structure, but also informal descriptions of their expected usage.

Patterns can guide the *architectural transformation on a fine-granular level*.

- A typical logical architecture is known as the *three-tiered architecture* comprising an interface component, a server component, and a database backend. This architecture often needs refinements in the presence of several interface, server, or backend components.

- An architectural design pattern, focusing on the functionality of components in user interface design, is the *Model-View-Controller* MVC paradigm. MVC is often also referred to as a paradigm or a framework as it is more abstract than a pattern. The MVC paradigm defines an architectural pattern that provides for clear separation of concerns within the architecture. Using such a pattern shields the developer from architectural/design decisions; instead they can concentrate on the domain problem (e.g. learning aspects) at hand.
- A number of other, *small-scale patterns* can be identified that are relevant in the context of educational systems, such as the factory pattern or the proxy pattern. These also explain structures within larger architectural patterns.

A central objective of a re-engineering method focusing on architectural aspects is to *identify domain-specific patterns* that act as targets for the architectural transformation. In our case study, we focus on patterns that explain structures and interactions resulting from the LTSA. We have already mentioned the three-tiered architecture and the MVC paradigm. These are particularly important in the educational context where the user is central and needs to be integrated in complex learning processes supported by the architecture.

These patterns provide *fine-grained targets* for the architectural transformation in the re-engineering process. The identification of these pattern in the legacy system and the definition of the target architecture in terms of these patterns is a central transformation technique. Patterns can become invariants of the transformation.

Core Patterns. In the following, we introduce a number of central patterns – called *core patterns*. We found them relevant for the educational domain, but, due to their structural properties they can be applied in a wider range of Web services-based systems. These patterns range over different *pattern categories* such as creational, structural, and behavioural; see [19] for more details on these categories. We will relate these patterns to aspects of the IDLE system.

The following three patterns turned out to be useful to address LTSA-specific structures:

- The *factory pattern* is a creational pattern that provides an interface for creating related or dependent objects without specifying their concrete classes.

The factory pattern can be applied for manipulating a variety of related persistent stores such as the learners records or adding/retrieving learning object to/from a databases. The Java database connectivity approach JDBC for example is based on the factory pattern.

- The *proxy pattern* is a structural pattern that provides a placeholder for another object to control access to it.

The implementation of a learner entity can be based on the proxy pattern to access generic learning components across the Internet using Web services toolkits. This pattern provides for the use of different languages and platforms on client and server. The proxy pattern abstracts the user from the creation and decoding of XML messages used in SOAP interactions.

- The *serializer pattern* allows a developer to efficiently stream objects into data structures as well as create objects from such data structures.

The serializer pattern matches requirements of the coach and learner entity component to maintain state over time when the learner wishes to pause the learning process. The serialiser pattern can convert the appropriate object into a data stream and store them in a persistent storage area such as in learner records.

The following two patterns were used to support the MVC paradigm:

- The *observer pattern* is a behavioural pattern that defines a one-to-many dependency between objects so that when one object changes its state, all dependents are notified/updated automatically.

The observer pattern can be used to notify view and controller of updates in the model, thus de-coupling the individual components. This does however introduce a new form of coupling in that the view/controller must query the model state or register interest in events to be broadcast by the model. Both the view/controller object roles and the model would need to be modified implying a lack of separation of concerns between the domain model and the GUI. Veit and Hermann [20] address these issues by using aspect-oriented programming to realise the ideals of the MVC paradigm.

- The *composite pattern* is a structural pattern that composes objects into tree structures representing part-whole hierarchies.

The composite pattern could be used to combine many simple view objects to create a complex view object. The reuse of the simple view objects supported by the MVC paradigm should make the assembly of additional view objects much quicker and simpler.

Patterns and Service-oriented Architectures. Even though patterns have been formulated based on an analysis of object-oriented systems [19], we have applied the pattern concept here to a services-oriented context. Principles of encapsulation, abstract interfaces, interaction through message passing that characterise object-orientedness are still valid in the services-oriented view.

Some observations are important in a context where, firstly, the user (e.g. the learner) is of paramount importance and, secondly, the access to material is provided through navigational features (the Web):

- The separation of the model from navigational and interface concerns is important. The MVC paradigm does not consider the separation of the navigational structures of an application from the view. A *navigational layer* is necessary as proposed in [21]. The LTSA separation into the LTSA components Delivery and Learner Entity illustrates this issue.
- *Navigational contexts* should be defined. These contexts refer to other related contexts and the nodes that should appear when they are navigated to.
- All *content* contained in a learning object should be completely *independent* from the interface, which will display it. This allows the content information to be displayed in many different and diverse ways [22].

One of the essential extensions we have made to the MVC pattern in our prototype is to introduce a *navigational layer* as an additional element in the transformation process.

Secondary Patterns. The LTSA defines a reference architecture, which supports a developer in the first steps of a *top-down design process*, which can be extended by the consideration of core patterns. Often, however, a *bottom-up development style* is required. More *fine-granular secondary patterns* can support this style more adequately.

- XML is an ideal notation to markup metadata, but also the content itself. The composite and *iterator pattern* exists in an XML parser as it provides generic access to elements of a hierarchical nature.
- The *strategy pattern* can be used to switch between various methods of evaluation in the LTSA evaluation process. For example, the developer can specify in an XML file or hotspot which algorithm to use (assuming the algorithms are prewritten) for student data evaluation. The strategy pattern can also be used between view and controller object roles. The controller object role uses the strategy pattern to decide at design or run time which view to use. This allows the controller to change the way it handles a request dynamically.
- The *template method* strategy may also be used to implement the switching behaviour described in the previous paragraph whereby the hotspot is coded in the subclass to override the implementation in the superclass to suit the specific domain model. The use of patterns should provide a framework whereby hotspots are identified as the location of model specific behaviour. These hotspots must then be configured to implement the domain model.

We found these pattern to be less central in service-oriented architectures than those that we classified as core patterns. However, they turned out to be useful in more detailed architectural designs.

5.3 Quality of Service

We have started the transformation of the original architecture onto a Web services platform by developing a prototype. Our aim was to investigate architectural transformation issues before addressing the full IDLE system. Several architectural patterns seem suitable. Nonetheless, a closer investigation and a possibly refined framework are necessary. Our analysis of patterns such as factory, proxy, serialiser, observer, etc. was a first step to determine a fine-granular architecture definition based on reusable structures.

The WSF is often promised as a Web middleware platform for the integration of various applications. So far, however, *quality of service* QoS aspects such as performance of the assembled system have not been addressed in enough detail [23]. Adding additional layers and discovery and integration mechanisms increases reusability and improves maintenance aspects, but might affect characteristics such as performance negatively. Consequently, QoS has to be part of a re-engineering strategy.

Although we have not carried out extensive tests on the re-engineered system, a number of *performance-related issues* are clear. The encoding and decoding followed by the subsequent transportation of verbose SOAP messages over HTTP is a considerable overhead when compared to binary RPC mechanisms such as CORBA. This latency may or may not hinder the user acceptance of the system. It is interesting to note, however, that some architectures such as Vinci [24] have tackled the high latency issues of SOAP over HTTP by using an approach based on a semi-parsed, pseudo-binary representations of XML documents – called XTalk. The performance of Vinci is comparable with RPC binary mechanisms based on their test data. This performance does however come at a price, as unlike a pure Web service-based implementation, Vinci requires a gateway to process the XTalk messages communicated across the network. Caching of learning objects, which have a low update frequency, should help improve the response time from the Web service components. This issue is also currently being addressed by the World-Wide Web Committee W3C. These considerations will impact architectural transformation strategies.

This discussion is not yet complete. An open distributed platform such as the Web is open to failure attacks. *Reliability* and *fault tolerance* of components in assemblies and in particular *security* are important in Web-based contexts. Only using secure protocols for transport is not sufficient. A secure architecture would be needed to obtain secure systems. A detailed discussion of these issues is, however, beyond the scope of this paper.

6 Conclusions

Our investigation of *re-engineering techniques* for services-oriented architectures has identified a number of central requirements:

- *coarse- and fine-granular architectural transformation support* through reference architectures and architectural patterns, respectively,
- *re-engineering for deployment and also development* – providing services for use is central in the Web services framework,
- consideration of *domain-specific standards* for description and architecture,
- *quality of service* aspects such as performance.

We have addressed units of software that have both service and component character. They can be annotated with metadata and discovered through searches based on this metadata. These units are also units of composition in the development of architectures for Web services-based systems. The notion of learning objects has captured this type of software unit in the case study context.

Two interface *description dimensions* reflect the development/deployment duality. Metadata annotations reflect content-related aspects of a service resource. Technical interfaces reflect the interaction and connection properties of these services as computational entities. We have illustrated that a *services-oriented architecture* for Web-services based systems is ideal (supporting flexible composition, maintenance, extensibility, etc.) if the new paradigm of sharing

and reusing educational resources is adopted. We have investigated and presented architectural patterns that can support *architectural transformation* in complex application domains. Central in this endeavour is a new *target development process model* for these systems focusing on sharing and reuse.

Our intention was to outline central aspects of a re-engineering method for the Web services platform. Architectures – coarse and fine-granular architectural transformation – and preparing software components as services that can be provided – description and development process – have turned out to be central.

We have presented architectural patterns that match system requirements. We plan to invest more time into the investigation of styles and patterns. We intend to develop domain-specific patterns, which would simplify the re-engineering process and increase the reusability of the framework even further. More experience is necessary with respect to management issues such as maintenance and reuse. Some questions in relation to patterns have remained open. For instance, aspect-oriented design and programming seem to suggest solutions to particular problems in separating concerns.

The description framework that we have introduced can be seen as a simple *two-layered ontology*. The *upper layer* describes e.g. a learning object in terms of educational properties. This layer would normally be used by a learner or instructor to discover and retrieve a suitable object satisfying particular needs. The *lower layer* describes a service (learning object) in terms of the properties relevant to integrate this object as a service into a Web services-based architecture. We have not pursued this direction here, developing it into an ontological framework that supports the re-engineering process, as we feel that a number of practical architectural issues have to be addressed before a more elaborate ontology framework can and should be investigated. Consequently, we have turned our focus on architectural aspects.

Acknowledgements

Our work was supported by the Dublin City University Teaching and Learning Fund DCU-TLF and the Irish Research Council for Science, Engineering and Technology IRCSET.

References

1. World Wide Web Consortium. *Web Services Framework*. <http://www.w3.org/2002/ws>, 2003.
2. World Wide Web Consortium. *Web Services Architecture Definition Document*. <http://www.w3.org/2002/ws/arch>, 2003.
3. L. Anido, M. Llamas, M.J. Fernandez, J. Rodriguez, M. Caeiro, and J. Santos. A Standards-driven Open Architecture for Learning Systems. In *Proc. International Conference on Advanced Learning Technologies ICALT01*. IEEE, 2001.
4. D. Sampson, C. Karagiannidis, and F. Cardinali. An Architecture for Web-based e-Learning Promoting Re-usable Adaptive Educational e-Content. *Educational Technology and Society*, 5(2), 2002.

5. R.S. Arnold, editor. *Software Re-engineering*. IEEE Tutorial. IEEE, 1994.
6. I. Sommerville. *Software Engineering - 6th Edition*. Addison Wesley, 2001.
7. C. Pahl, R. Barrett, and C. Kenny. Supporting Active Database Learning and Training through Interactive Multimedia. In *Proc. Intl. Conf. on Innovation and Technology in Computer Science Education ITiCSE'04*. ACM, 2004.
8. IEEE Learning Technology Standards Committee LTSC. *IEEE P1484.1/D8. Draft Standard for Learning Technology - Learning Technology Systems Architecture LTSA*. IEEE Computer Society, 2001.
9. N. Medvidovic, D.S. Rosenblum, and R.N. Taylor. An Architecture-Based Approach to Software Evolution. In *ICSE'98 Intl. Workshop on the Principles of Software Evolution*, pages 11–15. 1998.
10. J. Williams and J. Baty. Building a Loosely Coupled Infrastructure for Web Services. In *Proc. International Conference in Web Services ICWS'2003*. 2003.
11. J.F. Sowa. *Knowledge Representation - Logical, Philosophical, and Computational Foundations*. Brooks/Cole, 2000.
12. W3C Semantic Web Activity. Semantic Web Activity Statement, 2002. <http://www.w3.org/sw>.
13. E. Motta, J. Domingue, L. Cabral, and M. Gaspari. IRSII: A Framework and Infrastructure for Semantic Web Services. In D. Fensel, K.P. Sycara, and J. Mylopoulos, editors, *Proc. International Semantic Web Conference ISWC'2003*, pages 306–318. Springer-Verlag, LNCS 2870, 2003.
14. C. Pahl and R. Barrett. A Web Services Architecture for Learning Object Discovery and Assembly. In *Proc. of the 13th International World Wide Web Conference (Poster)*. 2004.
15. C. Pahl. Managing evolution and change in web-based teaching and learning environments. *Computers and Education*, 40(1):99–114, 2003.
16. IEEE Learning Technology Standards Committee LTSC. *IEEE P1484.12/D4.0 Draft Standard for Learning Object Metadata (LOM)*. IEEE Computer Society, 2002.
17. C. Szyperski. *Component Software: Beyond Object-Oriented Programming - 2nd Ed.* Addison-Wesley, 2002.
18. D. Fensel and C. Bussler. The Web Services Modeling Framework. Technical report, Vrije Universiteit Amsterdam, 2002.
19. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Design*. Addison Wesley, 1995.
20. M. Veit and S. Herrmann. Model-View-Controller and Object Teams: A Perfect Match Of Paradigms. In *Proc. of the 2nd International Conference on Aspect-Oriented Software Development, AOSD 2003*, pages 140–149. ACM, 2003.
21. M.D. Jacyntho, D. Schwabe, and G. Rossi. A Software Architecture for Structuring Complex Web Applications. In *Proc. of the 11th International World Wide Web Conference*. 2002.
22. F. Lyardet, G. Rossi, and D. Schwabe. Using Design Patterns in Educational Multimedia applications. In *Proceedings of ED-Media'98 World Conference on Educational Multimedia and Hypermedia*. 1998.
23. S. Ran. A Framework for Discovering Web Services with Desired Quality of Services Attributes. In *Proc. International Conference on Web Services ICWS'2003*. 2003.
24. R. Agrawal, R. Bayardo, D. Gruhl, and S. Papadimitriou. Vinci: A service-oriented architecture for rapid development of web applications. In *Proc. of the Tenth International World Wide Web Conference*. 2001.