Service Consumer Framework
Managing Service Evolution from a Consumer Perspective

George Feuerlicht[1,2] and Hong Thai Tran[2]
[1]*Department of Information Technology, University of Economics, Prague, W. Churchill Sq. 4, Prague, Czech Republic*
[2]*Faculty of Engineering and Information Technology, University of Technology, Sydney, Australia*

Keywords: Service Evolution, Cloud Computing, Service-Oriented Architecture.

Abstract: As the complexity of service-oriented applications grows, it is becoming essential to develop methods to manage service evolution and to ensure that the impact of changes on existing applications is minimized. Service evolution has been the subject of recent research interest, but most of the research on this topic deals with service evolution from the service provider perspective. There is an equal need to consider this problem from the perspective of service consumers and to develop effective methods that protect service consumer applications from changes in externally provided services. In this paper, we describe initial proposal for Service Consumer Framework that attempts to address this problem by providing resilience to changes in external services as these services are evolved or become temporarily unavailable. The framework incorporates a service router and services adaptors and determines runtime behavior of the system based on design-time decisions recorded in the service repository.

1 INTRODUCTION

With growing availability of various types of cloud services organizations are beginning to rely on external cloud providers to deliver a significant part of their IT infrastructure and software services. Cloud computing is associated with a number of well documented benefits that include the elimination of up-front costs, on-demand availability (characterized by up and down scalability and pay-per-use charging model), and a potential for overall cost reduction (Armbrust et al., 2009). In this environment, end user organizations (service consumers) are mainly responsible for service integration and management, with the service provider responsible for most of the other IT related functions. An important challenge, in particular in situations where a large number of cloud providers are involved, relates to dealing with service evolution. In modern business environments characterized by rapid change and technology innovation, software services need to be continuously maintained and upgraded introducing new functionality. Services are often the subject of uncontrolled change as service providers implement functional enhancements and rectify defects (Papazoglou, Andrikopoulos et al. 2011). As the complexity of service-oriented applications grows, it is becoming imperative to develop effective methods to manage service evolution and to ensure that service consumers are protected from service changes and outages. While most service providers attempt to carefully manage version releases and maintain backward compatibility between service versions, in practice changes that result in *breaking* consumer applications are inevitable. In some cases consumers of cloud services may be anonymous (i.e. not known to the service provider) making notification of changes difficult. Importantly, service consumers have no control over the provider service life-cycle and cannot predict when or how services will change. Consequently, service consumers suffer service disruptions and are forced to frequently upgrade their applications to maintain compatibility with new versions of services, resulting in ongoing maintenance costs.

The topic of evolution of software systems has been studied for several decades with Lehman and Belady formulating key principles in 1984 (Lehman, 1984), but today, evolution of software services presents new challenges that arise from the widely distributed nature of services deployed over the Internet. Service evolution has been the subject of recent research interest(Andrikopoulos et al., 2012;

Papazoglou, 2008; Papazoglou et al., 2011; Borovskiy and Zeier, 2008; Romano and Pinzger, 2012; Kajko-Mattsson, 2004; Kajko-Mattsson et al., 2007; Fokaefs et al., 2011; Kajko-Mattsson and Tepczynski, 2005; Eisfeld et al., 2012), however the focus of these efforts has been mainly on developing methodologies and tools that help service providers to manage service evolution. There is a pressing need to develop corresponding consumer-side methodologies and tools to address these issues from a service consumer perspective.

In this paper, we describe a proposal for a Service Consumer Framework (SCF) that attempts to address this problem by providing resilience for client applications to changes in external services as these are evolved or become temporarily unavailable. The SCF framework uses a combination of service adaptors and a service router to protect client applications from external changes. Evolution of services is supported by using service adaptors that transform service request and response messages between internal and external services. Service router determines which external services are evoked at runtime, based on their availability and pre-defined processing rules stored in the Service Repository. In the next section (section 2) we review related literature dealing with service evolution. In the following section (section 3) we describe an example Conference Management System used to illustrate the proposed framework. Section 4 is a description of the proposed Service Consumer Framework, and section 5 contains our conclusions and directions for future work.

2 RELATED WORK

Service evolution has been the subject of extensive recent research interest and a number of methods and tools have been proposed and developed to address the challenges of managing evolution of services. These approaches range from tools that identify changes to service interfaces as services evolve from version to version (Romano and Pinzger, 2012; Fokaefs et al., 2011; Eisfeld et al., 2012), to proposals that describe full life-cycle methods that attempt to address changes that affect multiple services (Papazoglou, 2008). In general, service changes can be classified into changes to functional characteristics (i.e. changes that affect structure of service interfaces, business protocols, policy assertions, and operational behavior) and changes to non-functional characteristics (i.e. quality of service attributes, e.g. security, availability, accessibility, etc.). More specifically, functional characteristics include (Andrikopoulos et al., 2012):

- *Structural Changes,* include changes in message structure and service operations
- *Business Protocol Changes* that affect the interactions between service providers and service consumers, e.g. the sequence of exchanged messages, etc.
- *Policy induced Changes* that include changes in legal requirements, e.g. the terms of international trade contracts, data protection policy, etc.
- *Operational behavior Changes,* that include the cascading effect of changing service operations

Papazoglou et al. (Papazoglou, 2008; Papazoglou et al., 2011) further classify service changes into shallow and deep. Impact of shallow changes is localized to a single service, while deep changes cascade across a number of different services.

Most service systems manage service evolution via controlled releases of service versions that are designed to maintain backward compatibility with older versions of the services. This ensures that applications that use existing versions of the services are not impacted by the release of new versions. In general, addition of new data types and operations (e.g. in a WSDL interface) do not break existing client applications and can be regarded as backward compatible. However, it is difficult to avoid changes that do not preserve version compatibility in practice. Such changes include removal of elements that form the service interface (e.g. operations, complex data types, attributes, etc.) and result in breaking the contract between the service provider and service consumers. Furthermore, while service versioning allows service consumers to decide when and if to upgrade their applications to take advantage of new service features, it also imposes additional complexity on service providers as they need to maintain multiple service versions and ensure that service consumers are notified before old versions of services are decommissioned. Versioning of individual services independently cannot deal with deep changes, and Papazoglou et al. propose a change-oriented service life-cycle to address the issues that arise with changes that cascade across multiple services. The life-cycle starts with the identification of the need for service change and scoping its extent, and then progresses to a service analysis phase that uses the model of the current state of the services (*as-is model*) and the *to-be* service model to perform gap analysis. Following the analysis of the impact of the required changes, decisions are made about how to deal with overlapping and conflicting service functionality. During the final change life-cycle phase new services are aligned, integrated, tested and released into production.

Borovskiy and Zeier (2008) focus on evolution of Web Services and identify two main types of drivers that cause service changes: intrinsic and extrinsic. Intrinsic change drivers include poor design and poor implementation, and extrinsic change drivers include market and business requirements drivers, operational process drivers, legislative and regulatory drivers, and other type of external drivers. Given this classification, the authors discuss versioning and message conversion techniques designed to address changes to Web Service interfaces that involve removing operations and parameters and cardinality mismatches. Fokaefs et al. (2011) present an empirical study of WSDL change analysis of Amazon EC2 service, PayPal service and FedEx services. The paper provides a detail analysis of Amazon EC2 service (18 versions), FedEx Rate service (9 versions), FedEx Package Movement Information Service (3 versions), PayPal SOAP API service (4 versions), and Bing Search service (2 versions). The authors developed a tool (VTracker) based on a tree-alignment algorithm to compare complex WSDL specifications. VTracker calculates the *tree distances* between a pair of operations for two service versions. Service evolution is classified into the following types:

- *Operation Deletions:* This is regarded as a deep change, and existing consumers of the service need to be notified as deleted operations might result in breaking client applications.
- *Inline Type Change:* this is a non-destructive change classified as a shallow change (e.g. changing element type to its parent type). Although this type of change does not impact on existing clients they should be notified.
- *Aggressive Evolution:* this type of change involves removing existing types and introducing new types (for example, in FedEx Rate version 9 more than 50% of existing types were removed, resulting in a significantly different new version of the service).
- *Renaming Variables:* changes in variable names can cause a mismatch between messages generated by the old and new versions of the service.
- *Adding New Types:* this type of change does not normally result in breaking client applications.
- *Changing Input or Output Types:* this type of change affects the service interface and impacts on client applications.

Based on their empirical analysis of Amazon EC2, PayPal and FedEx Web Services the authors conclude that removal of existing elements is relatively rare and that the evolution of services involves mostly adding new elements that do not break existing client applications.Romano and Pinzger (2012) describe the WSDLDiff tool that is used to identify fine-grained changes between versions of a Web Services by comparing WSDL interfaces. WSDLDiff is based on the UMLDiff algorithm of Xing and Stroulia (2005) and identifies most of the frequently occurring types of changes, including changes in XSD elements, attributes, references and enumerations. The authors use the generic Matching Engine (org.eclipse.compare.match) to compute a set of four similarity metrics: type similarity (computes the similarity between types), name similarity (computes the similarity between attribute names), value similarity (computes the similarity between the values of attributes), and relations similarity (computes the similarity based on the relationships). To allow comparison with the results of Fokaefs et al. (2011), Romano and Pinzger (2012)computed metrics for Amazon EC2, PayPal and FedEx Web Services using the WSDLDiff tool. The resulting analysis shows the number of added, deleted, and changed elements for each type of Web Service. The authors identify differences in the evolution of Web Services and suggest that this information can be used to estimate the risk associated with the use of Web Services from a particular provider. The authors also investigated the correlation between the number of interface changes and cohesion metric defined by Perepletchikov et al. (2007). The relationship between the quality of service interface design and maintainability of service-oriented applications has been investigated in the literature (Feuerlicht, 2011; Papazoglou, 2002; Pautasso and Wilde, 2009), and there is a general agreement that maximizing service cohesion and minimizing service coupling localizes the impact of changes and leads to improved maintainability of service-oriented applications. Reliable metrics that can identify poor service design early during system development can significantly reduce maintenance costs (Feuerlicht, 2013), but it is unlikely that the impact of service evolution on client applications can be entirely avoided.

3 MOTIVATING EXAMPLE

To illustrate the need for the Service Consumer Framework (described in section 4) consider the example scenario illustrated in Figure 1. The Conference Management System (CMS) is a simplified scenario based on a *real-world* conference management application. The CMS system supports a number of conference management functions, including enrollment of participants, online payments, and booking of accommodation and transportation. CMS is a service-based system that consumes both internal (on-premise) and external (cloud) services. Internal services (i.e. services supported by on-premise applications) include Financial Management (Finance), and Customer Relationship Management (CRM) services.
Externally provided services include:

- *Payment services:*
  Paypal Payment Gateway (www.paypal.com), OnePay Payment Gateway (www.onepay.vn/), SecurePay (www.securepay.com)
- *Flight tracking services:*
  FlightAware (flightaware.com), live flight tracking maps, flight status, and airport information,
  Flight Explorer (www.flightexplorer.com) - real-time aircraft position display and management tool used for organizing customer pick-up service
- *Address validation services:*
  Google Geocoding API (developers.google.com/maps/documentation)
  QAS Pro Web (www.qas.com)
- *Shipment tracking services:*
  FedEx Express (http://www.fedex.com/us/)

The CMS system operates in an environment where external services continually evolve with providers upgrading their services by adding and removing interface elements and operations. In addition, services are subject to outages and may become temporarily unavailable due to communications failures and site crashes. A key requirement for the CMS system is to maintain operation in this challenging environment. This is facilitated by the SCF Framework that uses adaptors to *shield* internal services from changes in external services. For example, several internal applications may use a payment service that communicates with an external payment service via a payment adaptor (i.e. avoiding direct communication with the external payment service). This avoids the need to modify multiple applications in response to changes in the external payment service as the compatibility with external services is maintained by upgrading the adaptor (see section 4.3 for a detail description of adaptor functionality). Another key requirement for the CMS system is resilience against service outages. This can be achieved by re-routing a payment request to an alternative external service. For example, when the PayPal Payment Gateway becomes unavailable, the request is re-routed to SecurePay or OnePay service (as described in section 4.2).

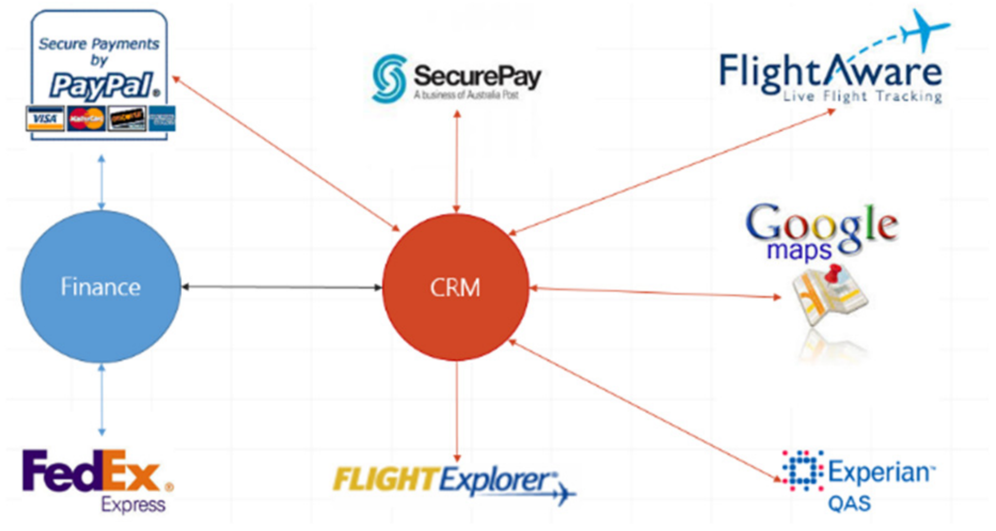

Figure 1: Conference Management System.

4 SERVICE CONSUMER FRAMEWORK

The SCF framework is designed to manage changes in both functional attributes (i.e. changes to message structures, service operations, etc.) and non-functional attributes (i.e. availability, cost, etc.). The SFC Framework is illustrated in Figure 2 and consists of three layers: Process Layer, Adaptor Layer and Service Layer. The Service Layer incorporates both internal and external services and defines their native interfaces. The Adaptor Layer contains adaptors that translate requests between the native services (e.g. PayPal payment service) and the corresponding internal services. The Process Layer defines processes that are implemented using the service router and processing rules stored in the service repository.Service router determines at runtime, which services are evoked based on their availability and pre-defined processing rules. Service repository maintains information about services allowing substitution of services with equivalent functionality to avoid service interruptions. The information held in the service repository also allows replacing external providers in situations where their services become incompatible with existing applications, or in order to optimize a particular parameter (e.g. cost, response time, etc.).

4.1 Service Repository

The function of the service repository is to maintain information about available services and adaptors. Each internal service can be associated with a number of (alternative) external services.Internal service description includes the following information:

- Internal Service Identifier, Service Name, Service Description, and Version Number
- Service Location (URL of the service)

External service description includes the following information:

- External Service Identifier, Service Name, Service Description, and Version Number
- Functional Parameters: WSDL, Service Provider, Service Location (URL), Service Authentication
- Non-functional Parameters: Availability, Response Time, Cost, Security Attributes, etc.
- Service Adaptor: Adaptor Identifier, Adaptor Name, Adaptor Location
- Corresponding Internal Service Identifier

The information held in the service repository is used at design-time to identify suitable services and to define the sequence of service execution. Quality of Service (QoS) attributes stored in the repository can be used to identify external services based on their anticipated availability, response time, cost, or some other QoS attribute, and to define the processing rules that determine the sequence of service execution at run-time.

4.2 Service Router

The function of the router is to control the routing of requests to a provider according to priority rules defined in the service repository. For example, a payment request can be routed to an alternative payment gateway (e.g. SecurePay or OnePay) via corresponding service adaptors if the PayPal service becomes unavailable. The service router uses information in the service repository to execute a service invocation sequence. Figure 3 illustrates the service router sequence for the payment service. An

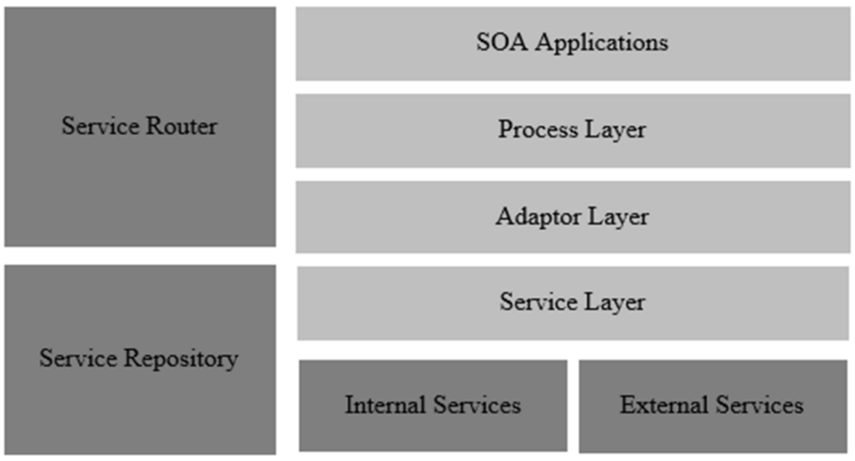

Figure 2: Service Consumer Framework.

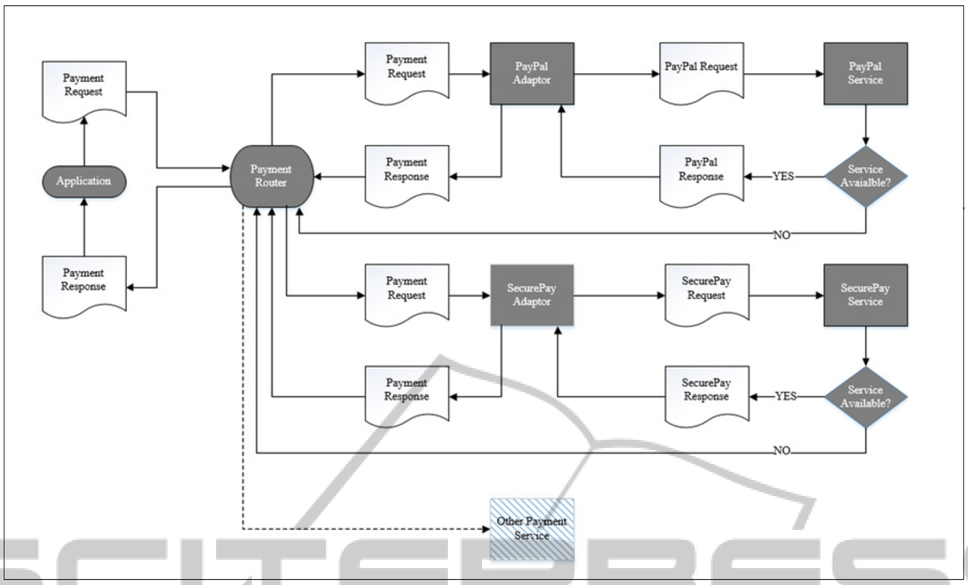

Figure 3: Service Router Sequence for the Payment Service.

application (e.g. the Finance application) passes a payment request to the internal payment service that forwards this request to the external PayPal service via the PayPal adaptor. The external PayPal service response is sent back to the internal payment service via the PayPal adaptor, and then to the Finance application. If the PayPal service fails to respond within a specified time period the request is routed to the next external payment service (SecurePay in our example) via the corresponding adaptor (i.e. SecurePay adaptor). If the SecurePay service fails to respond the request may be re-directed to another payment service, or return an error status to the application. The order of service execution is defined at design-time based on designer preferences. For example, the designer may choose to call the least expensive payment service first, and execute a more expensive service only in the event of failure of the first service. Alternatively, the designer may decide to call the service that gives the best response time first, and only execute alternative services in the event of failure.

4.3 Service Adaptor

The function of a service adaptor is to transform outgoing requests into the format supported by the current version of the corresponding external service, and to ensure that incoming responses maintain compatibility with internal applications. For example, the PayPal adaptor accepts payment requests from the CMS application with the interface containing attributes <Membership_ID, Name, Address, Payment_Type, Card_Type, Card_Holder_Name, Credit_Card_Number, Expiration_Date, Amount, CCV_Number, Note>. The payment request is logged and transformed to a PayPal payment request that has the interface containing attributes <Acct, Expdate, Amt, Comment1, Comment2, Cvv2, Firstname, Lastname, Street, Swipe, Tender, Trxtype, Zip>. Following a successful request execution the PayPal service response is transformed into a message compatible with the CMS application. The response message from the external service indicates success or failure of the request. For example, if the response indicates a communication failure, the service adaptor will mark the transaction as failed and the service router will route the payment request to another adaptor. Alternatively, the response message may indicate that the transaction was declined due to invalid credit card information (e.g. card number or expiration date) and the router may request for the information to be resubmitted.

5 CONCLUSIONS

Growing availability of various types of cloud-based services and their incorporation into enterprise applications greatly increases the dependence of organizations on external service providers. Notwithstanding the efforts of service providers to manage the evolution of services and maintain backward compatibility for service versions, in practice changes that result in breaking consumer applications are inevitable. Moreover, the availability of cloud-based services cannot be fully guaranteed, forcing service consumers to build in redundancy into their applications, so that alternative services can be substituted when required in order to maintain service continuity.

Service evolution has been the subject of recent research interest, but most of the research on this topic deals with service evolution from the service provider perspective. We have argued that there is an equal need to consider this problem from the perspective of service consumers and develop effective methods to protect service consumer applications from changes in external services. In this paper we describe an initial proposal for Service Consumer Framework that attempts to address this problem by providing resilience for consumer applications to changes to external services as these are evolved or become temporarily unavailable. The basic idea of the framework involves the use of service adaptors in combination with a service router that re-directs requests to different service providers based on their availability at runtime. Service adaptors ensure that consumer applications can use services that are currently supported by service providers, and that the timing of upgrades to new service versions is determined by the service consumers, rather than dictated by service providers. Using this framework, application designers can choose from a number of services that provide identical functionality (e.g. payment services) and define the sequence of service execution to optimize the cost and performance of the applications.

We have implemented prototype versions of several service adaptors, and we are currently working on the implementation of service repository and service router using Microsoft .NET technologies. Our current efforts focus on developing a proof-of-concept prototype of the SCF framework and on developing additional adaptors. We will use this prototype to further refine the design of the framework.

ACKNOWLEDGEMENTS

Authors wish to acknowledge support of GAČR (Grant Agency, Czech Republic) grant No. P403/11/0574 and ARC (Australian Research Council) Grant Design of Service Interfaces (2004000242).

REFERENCES

Andrikopoulos, V., Benbernou, S. & Papazoglou, M. P, 2012. On the evolution of services. *IEEE Transactions on Software Engineering,* vol. 38**,** pp. 609-628.

Armbrust, M., et al., 2009. Above the clouds: A berkeley view of cloud computing. *EECS Department, University of California, Berkeley, Technical Report.* UCB/EECS-2009-28, 2009.

Borovskiy, V. & Zeier, A., 2008. Evolution management of enterprise web services. *Advanced Management of Information for Globalized Enterprises, AMIGE 2008. IEEE Symposium on*, pp. 1-5.

Eisfeld, A., McMeekin, D. A. & Karduck, A. P., 2012. Complex environment evolution: Challenges with semantic service infrastructures. *6th IEEE International Conference on Digital Ecosystems Technologies (DEST).*

Feuerlicht, G., 2011. Simple metric for assessing quality of service design. *In Service-oriented computing.* Springer Berlin Heidelberg, (Eds.) Maximilien, E. M., Rossi, G., Yuan, S.-T., Ludwig, H. & Fantinato, M.

Feuerlicht, G., 2013. Evaluation of quality of design for document-centric software services. *Service-Oriented Computing-ICSOC 2012 Workshops*, Springer, pp. 356-367.

Fokaefs, M., Mikhaiel, R., Tsantalis, N., Stroulia, E. & Lau, A., 2011. An empirical study on web service evolution. *Web Services (ICWS), 2011 IEEE International Conference on*, pp. 49-56.

Kajko-Mattsson, M., 2004. Evolution and maintenance of web service applications. *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pp. 492-493.

Kajko-Mattsson, M., Lewis, G. A. & Smith, D. B., 2007. A framework for roles for development, evolution and maintenance of soa-based systems. *Systems Development in SOA Environments, 2007. SDSOA '07: ICSE Workshops 2007. International Workshop on*, pp. 7-7.

Kajko-Mattsson, M. & Tepczynski, M. A., 2005. framework for the evolution and maintenance of web services. *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 665-668.

Lehman, M. M., 1984. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.,* vol. 1**,** pp. 213-221.

Papazoglou, M., Yang, J., 2002. Design methodology for web services and business processes. *Proceedings of the 3rd VLDB-TES Workshop*, Springer, pp. 54-64.

Papazoglou, M. P., 2008. The challenges of service evolution. *Proceedings of the 20th international conference on Advanced Information Systems Engineering.* Springer-Verlag.

Papazoglou, M. P., Andrikopoulos, V. & Benbernou, S., 2011. Managing evolving services. *IEEE Software,* vol. 28**,** pp. 49-55.

Pautasso, C. & Wilde, E., 2009. Why is the web loosely coupled?: A multi-faceted metric for service design.

18th international conference on World wide web, ACM, pp. 911-920.

Perepletchikov, M., Ryan, C. & Frampton, K., 2007. Cohesion metrics for predicting maintainability of service-oriented software. *qsic,* vol.**,** pp. 328-335.

Romano, D. & Pinzger, M., 2012. Analyzing the evolution of web services using fine-grained changes. *IEEE 19th International Conference on Web Services (ICWS).*

Xing, Z. & Stroulia, E., 2005. Umldiff: An algorithm for object-oriented design differencing. *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering.* Long Beach, CA, USA: ACM.