Software Product Lines: Source Code Organization for 3-tier OLTP Architecture Systems

Zdravko Roško
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
zdravko.rosko@zd.t-com.hr

Author(s) Name(s)
Author Affiliation(s)
Department/Institute
Full Address(es)
E-mail(s)

Abstract. *This paper proposes the use of 3-tier class of system as core software architecture for building Software Product Lines (SPL). A base for building a SPL is commonality among software products based on 3-tier architecture, use of available industry frameworks, development organization's "glue" code and developed services needed by most of the software products within the proposed SPL. A software development organization specialized to develop 3-tier (logical tiers) client-server applications for differnet kind of industries such as: banking, tourism, telecommunictions, etc., can benefit from adopting SPL principles. SPL assumes management of commonality and variability among applications belonging to 3-tier SPL. This paper focuses on benefits from well structured source code organization architecture for 3-tier class of system which is based on SPL principles. To illustrate the approach, the paper presents a usage of 3-tier for On Line Transaction Processing (OLTP) software product line, based on Java technology which serves as the implementation platform.*

Keywords. OLTP, SPL, Java, framework, domain, platform

1 Introduction

The key difference between traditional single system development and software product line engineering is a fundamental shift of focus: from the individual system and project to the product line. As opposed to many other reuse approaches that focus on code assets, the product line infrastructure includes all assets that are relevant throughout the software development life-cycle [1]. SPL can be divided to: *domain engineering* (development for reuse) and *application engineering* (development with reuse) that builds the final products. Successful product lines have enabled organizations to capitalize on systematic reuse to achieve business goals and desired software benefits such as productivity gains, decreased development costs, improved time to market, higher reliability, and competitive advantage [2, 3].

Software development organizations that develop systems or products, software consultant organizations developing software on a project basis for other organizations in different business domains, or IT departments that develop IT support systems, can benefit from well structured source code organization architecture which is based on SPL principles.

An organization involved to many projects, on one project can be developing a "core banking" application for a local bank. The same organization can be working on a project developing an Operation Support System (OSS) inventory for a telecommunication company. The third project for the same organization could be a development of an on-line reservation system for an international tour operator. One responsible for all these projects would benefit if looking for a common assets among these projects. Is there 20% or even 80% of the common reusable assets among the projects, could be an important question to answer for anyone who wants to achieve better productivity, decreased development cost, improved time to market, higher reliability and maintenance cost. One could argue that there are already existing industry frameworks, standards for project management, test case and architecture templates that cover commonality among these projects, but our experience prove that building of core SPL assets for reuse among different projects is a benefit.

SPL is mostly used by organizations that develop software for mobile phones, cars, electronic instruments,

while information systems domain, mostly implemented as a 3-tier class of system, is not often considered as a potential base for developing SPL.

OLTP is a one of the most typical 3-tier system that facilitates and manages transaction-oriented applications, typically for data entry and retrieval transactions in a number of industries, including banking, airlines, travel, supermarkets, manufacturers, telecommunications, and others, that shares common set of components and architecture that can be used for subsequent projects. OLTP as a way of designing, developing, deploying and managing software systems has these characteristics:

- It provides reusable transactions (Logical Unit of Work) to multiple users.
- Applications or other transaction consumers are built using functionality from reusable transactions.
- Transactions performance time is critical to the end user.
- Transactions are predominantly, but not exclusively, short-running processes.

A specific object-oriented framework for 3-tier OLTP systems as a domain-independent layer of commonality between different domains, on one level, and between different software products of the same domain on another level, can be defined as a SPL *platform* to be used for building products in more than one business domain. Specific business domain such as banking can benefit from inheriting a domain independent platform, and building a domain specific layer of components and services to be used by more domain specific products.
Specific 3-tier object-oriented framework (platform) has high level of configurability which allows for the easy configuration of components for individual product in a specific business domain.

A number of authors have suggested relations between SPL and other technologies [4, 5, 6, 7, and 8]. These works did not address 3-tier or OLTP as core software architecture for building SPL. The goal of this paper is to identify the key components for 3-tier product line and to propose the organization architecture of the product line source code.

In this paper the technologies supporting 3-tier class of system (EJB, JCA, JDBC, etc.) are viewed as an adaptable layer of inter-organizational reusable components connected to the domain independent object-oriented framework. SPL allows a configuration or a substitution of the technologies supporting the 3-tier class of system. Many of the principles of SPL apply to 3-tier software products which can be developed in more than one business domain. SPL platform components do not necessarily implement all of domain-specific requirements nor do the domain specific components implement all of product-specific requirements. The subset of the requirements that is not fulfilled by the SPL platform must be implemented as domain specific components and the subset of the requirements that is not fulfilled by domain specific components must be implemented by product-specific software. The components that are not part of the shared SPL platform assets are only included in the source code for the specific domain. Also, components that are not part of the SPL platform and specific domain assets are only included in the source code for the specific domain product.

2 SPL Concepts

The SPL involves *core asset development* and *product development* using the core assets, both under the aegis of technical and organizational *management* [2]. Core asset development has also been called domain engineering. Product development from core assets is often called application engineering. Besides components, requirements, architecture, modeling and analysis, test cases, test data, test plans, documentation templates and other software engineering artifacts were also expected to be reusable while working on a new SPL project or product.
Product line scope is a description of the products that will constitute product line [2]. For the purpose of this paper we define the scope of the 3-tier OLTP product line to include any software product for domain independent transactional system which is expected to follow 3-tier concept and principles.

3 OLTP source code organization architecture

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationship among them [9]. Source code structure, as one of them, reflects the level of compliance to the architecture.

3.1 Layered Style

Recurring forms have been widely observed, even if written for completely different systems. We call these forms architecture styles. An architecture style is a specialization of element and relation types, together with a set of constraints on how they can be used [10].

In this paper a *layered style* where the layers are allowed to use only the facilities of lower layer, is dominantly applied to the SPL for 3-tier class of system design. We arrange the modules into useful units (layers and layer segments) by restricting what each one is allowed to use “Fig. 1”. SPL for 3-tier module structure determine how changes to one part of a system might affect other parts, and its ability to support modifiability and reuse. SPL for 3-tier is composed of three layers: presentation, business logic, data access logic which in

turn is using other services components such as security, transaction, logging, and etc.
At the run time, the system can be configured to run within one or two processes where some modules (services) are used by more than one system layer.

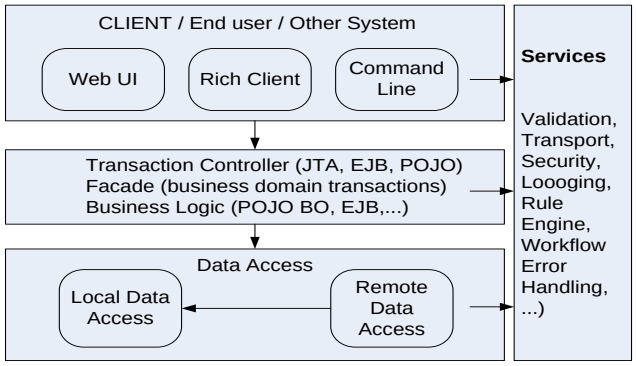

Figure1. SPL for 3-tier Layers

3.2 Platform components

Software product line engineering relies on a common product line architecture (also called reference architecture). The central role of a common architecture is a major ingredient of the success of product line engineering compared to other reuse approaches [1]. We propose development of the 3-tier OLTP specific product line platform (framework), which is composed of the technologies such as EJB, JMS, JTA, JCA, Spring, Struts, Hibernate, TopLink, and 3-tier OLTP product line specific components, both considered a variations ponits of the 3-tier product line "Fig. 2". Technologies used to build the platform are abstracted and integrated into the platform, and are viewed as the interorganizational reusable components while 3-tier OLTP product line specific components such as error handling, caching and validation are viewed as the intraorganizational reusable parts.
Product line platform abstracts the business domain-independent functionality where its variation points can be used to build products for different business specific domains. Having developed SPL platform for a 3-tier product line one can implement additional services, typically needed in 3-tier products such as: validation service, rule engine, exception handling, and may also abstract the technology services such as transaction, connection pool, logging, security and etc. The SPL platform can be viewed as a „glue" code which encapsulates technology domain independent functionality and also implements a new services which are not available from the standard technologies. SPL for 3-tier assumes that organizations does not start from scratch on each new project or product, but rather inherit a tested and documented variability points enabling application developer to select among the available variations. SPL for 3-tier OLTP class of system is composed from a three types of packages (modules): client, common, server "Fig. 3".

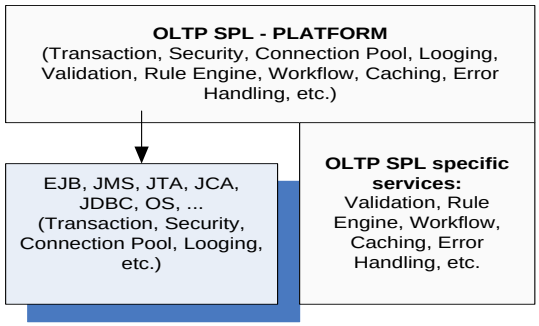

Figure 2. 3-tier OLTP SPL Platform

Common package is used both by client and by server, and contains value objects, utility classes and domain interfaces used by client while calling the server components.

Server is used to name the group of classes which contain business logic and data access logic. It does not necessary means that server process is separated from the client. *Server* also means a logical server which can be used to compose a "*fat client*" product and run within the same process where the client is running. The *common* package of SPL platform contains the transport classes used by the client to send a message (data transfer object) to the logical server. The transport is using a "*protocol plug-in*" design pattern to implement *local*, RMI, IIOP, SOAP, and other transport protocols. *Local* transport can be used within an IDE for unit or integration test purposes, and also in the case the "*fat client*" architecture is an acceptable variation, used when deploying the application.

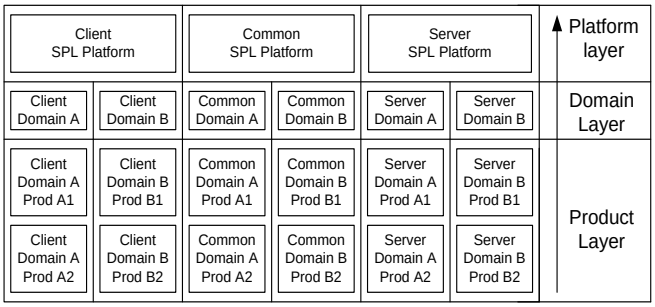

Figure 3. SPL domain independent/dependent layers

3.3 Domain components

Having developed, tested and documented a domain independent platform for SPL, domain specific components can be built on top of the SPL platform. All common and repeating components, specific for a domain, may be organized as a layer between the platform and domain specific applications (products). A common domain classes such as: value objects, user interface parts, desktop, component facades, business objects, data access objects, which may be used by many domain specific applications as a variation points at the time of building the application, may also be considered by domain engineering group to be a part of the domain-specific layer components.

3.4 Product (application)

Any domain-specific application includes the components from platform (*client, common, and server*) and components from domain (*client, common, and server*) beside its own components "Fig. 4". Applications share a common business domain assets and SPL platform assets to satisfy its requirements. All potential reusable components done by application engineering team may be candidates for platform or domain-specific framework.

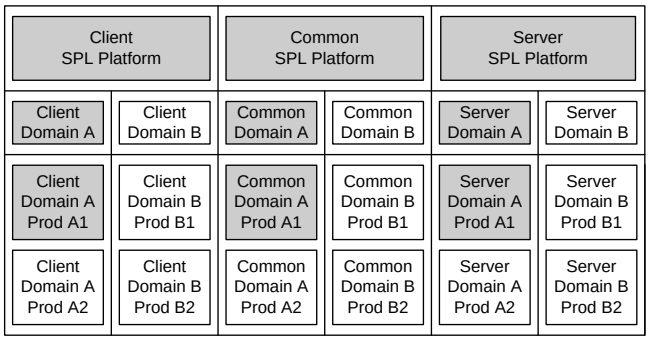

Figure 4. Domain specific product

3.5 Client (Models, Views, Controllers)

Client layer contains all product specific parts needed to represent the application to the end user. It includes abstracted components such as charts, attachment handling, interfaces to Excel, Word, PDF, and etc. Client application may be implemented by using different technologies such as GWT, Servlet, JSP, JSF, SWT or SWING. The term *client* means a logical client which is initiating a communication to the business logic implemented by business objects which in turn call the data access logic layer to get the data from a data source. Deployment of the client assumes packaging the required components as shown "Fig. 5".

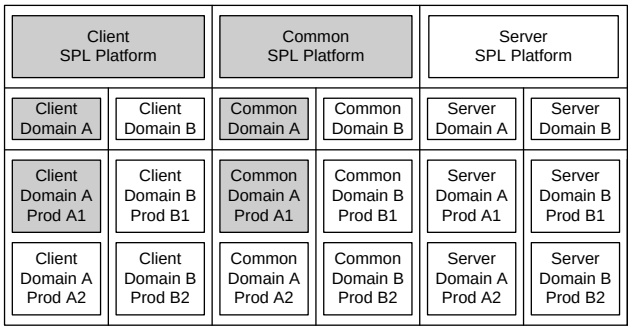

Figure 5. Client packages

3.6 Common (Value Objects, Façade proxies)

Common layer contains utility classes, exception handling, security, session, data caching, transport classes, data transfer objects, domain interfaces (façade proxies) which are packaged together and used by a domain applications. Utility classes include the components for processing XML, email, File, String formatting, and etc. In the case some or all of the functionality provided by product line is also implemented by abstracted technologies, the variability management allows using them if required.

3.7 Server (Façade components, BO, DAO)

Server layer assumes all classes implementing the business logic and data access logic, transaction handling components, data source connection pool, value list handler, security, transport, and other services components. Deployment of the server assumes packaging the required components as shown "Fig. 6". Packaged server can be deployed to an EJB container or to a *Servlet* container, depending on specific application requirements. In case the server is deployed as an EJB component the variability has to be handled properly to enable all required variation points needed by the used container. SPL can have an option to use Plain Old Java Objects (POJO) or EJB entities to handle application specific business logic. SPL transaction management is not tied to JTA or any other technology and can work with different transaction strategies.

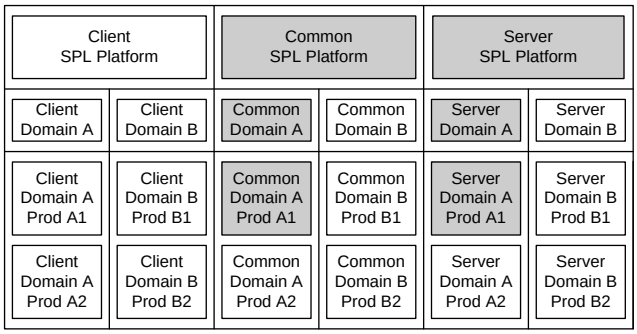

Figure 6. Server packages

3.8 Design Structure Matrix (DSM)

Hierarchical relationships and interdependencies among design parameters can be formally mapped using a tool called the Design Structure Matrix [17]. The mapping procedure was invented by Donald Steward [13], and has been extended and refined by Steven Eppinger [15]. The distinction between acceptable and unacceptable SPL for 3-tier module dependencies is expressed using design rules, which are entered in the DSM matrix table "Fig. 7". *Design rules* come in two forms *Component*$_1$ *can use* *Component*$_2$ and *Component*$_1$ *cannot use* *Component*$_2$ indicating that *Component*$_1$ can and cannot depend on *Component*$_2$. Each row and each column of the DSM corresponds to a module, and each dependency is denoted by a mark "**1**" in the column corresponding to the dependent module and the row corresponding to the depended upon module. DSM is used to spot circular dependencies since they are immediately visible as marked cells on both sides of the matrix's diagonal.

The three layers of the 3-tier OLTP product line system: *product*, *domain* and *platform* may be documented using the DSM. The product line DSM, shows dependencies above the matrix's diagonal but it does not violate the circular dependencies rules since *common* modules are not representing a layer but rather a common set of utility or data encapsulating classes used on presentation layer as MVC models and on business logic and data access layer as *data transfer objects*.

using / used	PROD Client	PROD Common	PROD Server	DOM Client	DOM Common	DOM Server	PLAT Client	PLAT Common	PLAT Server
PROD Client	0	0	0	0	0	0	0	0	0
PROD Common	**1**	0	**1**	0	0	0	0	0	0
PROD Server	0	0	0	0	0	0	0	0	0
DOM Client	1	0	0	0	0	0	0	0	0
DOM Common	**1**	**1**	**1**	**1**	0	**1**	0	0	0
DOM Server	0	0	**1**	0	0	0	0	0	0
PLAT Client	**1**	0	0	**1**	0	0	0	0	0
PLAT Common	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	**1**
PLAT Server	0	0	**1**	0	0	**1**	0	0	0

Figure 7. DSM for layered design of OLTP product line

3.9 Use Cases

SPL source code is the core artifact for developers "Fig. 8", code reviewers and tester, which suggests the high importance of its organization structure. The structure can improve the productivity, maintainability, testing processes, architecture compliance, documentation and other source code related activities.

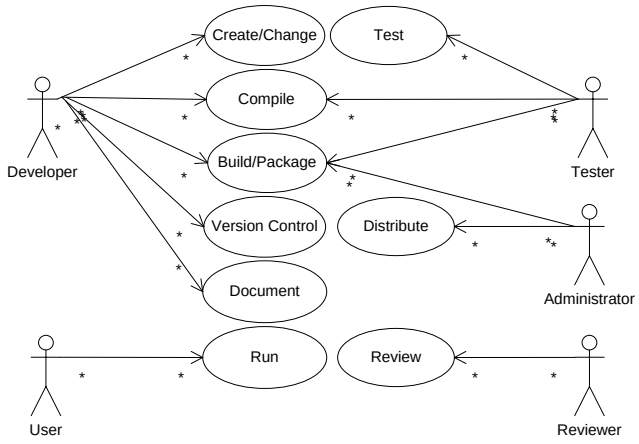

Figure 8. SPL Source code Use Cases

Developer who are able to make changes, check-in, check-out, do builds, document a do compile of the source code independently from other developers, may be more productive in case the source code is properly structured. Application engineering, where one can instantiate the environment, specific for a product, with no collision with other applications, have less interactions and saves the resources while programming. Testers performing integration, regression, stress test and other

tests, which require direct interaction with the source code, can benefit from well structure code. Having the option to merge client and server to a "fact client" and at the same time the option to split them, in order to run the application within two separate processes is also a benefit "Table 1". SPL platform's component named *transport*, which connects client and server programming logic, needs to be tested just once during the SPL platform test, and later used by all applications. The development within an IDE and most of the tests can be performed using *local* transport. Code review, where source code is organized as three units: client, common, and server within all three layers: application, domain and platform, makes it easier to spot a potential noncompliance with the reference architecture.

Table 1. SPL benefits

Actor	**Source code Use Cases**	
	Activity	***Benefit***
Developer	Create/Change/ Compile/Build/ Package/Version Control/ Document	Develop more independently, less check-in/check-out actions since the code is divided among domain and application engineering developers for each business domain specific products.
Tester	Test/Compile/Build/ Package	Perform unit and integration test in local development environment using "local" transport mechanism between client and server (no need to have all system set-up).
Reviewer	Review	Easier to spot non-compliance.
Administrator	Build/Package/ Distribute	Adaptable to build tools for easier packaging.
User	Run	The size of the executable program, performance, encapsulated platform errors.

4 Variability management

3-tier OLTP product line aims at supporting a range of products from different business domains such as banking, telecommunications, travel and etc. These products may also support different individual customers within a business domain. Management of variability points within the platform, domain or application is a key to product line success.

4.1 Domain variability

Assuming the inheritance of the platform components by business domain-specific components or services, the variability can be achieved by configuration, parameters, use of a system including the functionality of other system, reflection, dynamic class loading, overloading or inheritance of other classes from the platform layer.

4.2 Product variability

Product-specific characteristics as a part of the 3-tier product line variability, often required by specific product needs, are handled mostly in application engineering while inheriting domain and platform components.

5 Conclusions

The implementation of software product line for 3-tier OLTP products using domain independent platform, domain specific and application specific components, has significant potential for organizations developing software products, consulting companies and IT departments. The available technologies that support 3-tier are viewed as inter-organizational reusable components which still need to be integrated, tested, documented and extended in the form of 3-tier OLTP product line platform to be used while building domain-specific and customer-specific applications.

A building of 3-tier platform component is part of domain engineering, which sets up the common product line infrastructure. Business domain specific engineering is also part of domain engineering while development of products is considered as part of application engineering. By partitioning the typical application into layers and programming using SPL, the third party technology used for each application layer can be replaced. Source code organization architecture for SPL is of significant importance in order to achieve the high productivity and other benefits from adopting SPL principles.

References

[1] Frank J. van der Linden (Author), Klaus Schmid (Author), Eelco Rommes, "Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering", Springer; 1st Edition. edition , 2010, pp. 6, 14.

[2] Clements, P. & Northrop, L. M. „Software Product Lines: Practices and Patterns", Addison-Wesley, 2001, pp. 17, 29, 31.

[3] Kyo C. Kang, Vijayan Sugumaran, Sooyong Park, "Applied Software Product Line Engeenering", Taylor and Francis Group, 2011, pp. 6.

[4] Cohen, Sholom & Krut, Robert. Proceedings of the First Workshop on Service-Oriented Architectures and Product Lines (CMU/SEI-2008-SR-006). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2008.

[5] Cohen, Sholom & Krut, Robert, „Managing Variation in Services from a Software Product Line Context", Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2009 - forthcoming.

[6] Ralph Mietzner, Andreas Metzger, Frank Leymann and Klaus Pohl, „Variability Modeling to Support Customization and Deployment of Multi-Tenant-Aware Software as a Service Applications". In

Proceedings of PESOS Workshop, ICSE, Vancouver, Canada, May 18-19, 2009.

[7] Tarr. P. Technologies for Software Product Line *Development*. https://www-950.ibm.com/events/wwe/grp/grp004.nsf/vLookupPDFs/tarr-product-lines-033009-slides/$file/tarrproduct-lines-033009-slides.pdf

[8] S. G¨unther and T. Berger, „Service-oriented product lines: Towards a development process and feature management model for web services“, in SPLC ’08: 12th International Software Product *Line Conference*, pp. 131–136, 2008.

[9] J. Bosch, “Design and Use of Software Arhitectures,” Addison-Wesley Professional, May 2000, pp. 11, 169–170.

[10] Paul Clements, Felix Bachmann, Len Bass, Davig Garlan, “Documenting Software Architecture”, Addison Wesley, 2011, pp. 25.

[11] Northrop, L. & Clements, P. “A Framework for Software Product Line Practice”, Version 5.0 http://www.sei.cmu.edu/productlines/framework.html (2009).

[12] Neeraj Sangal, Ev Jordan, Vineet Sinha, Daniel Jackson, “Using Dependency Models to Manage Complex Software Architecture”, OOPSLA '05 Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.

[13] Steward, D. V. “The Design Structure System: A Method for Managing the Design of Complex Systems.” *IEEE Trans. on Eng. Mgmt* EM-28.3: 71-74, Aug, 1981.

[14] Baldwin, C. Y. and Clark, K. B.. Design Rules, Vol. 1: “The Power of Modularity”. The MIT Press, 2000.

[15] Eppinger, S. D. “Model-based approaches to managing concurrent engineering,” *Journal of Engineering Design*, 2(4):283–290, 1991.

[16] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. 2001. “The structure and value of modularity in software design”. In Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-9). ACM, New York, NY, USA, 99-108.

[17] LaMantia, M.J.Yuanfang Cai, MacCormack, A.D.Rusnak,J., "Analyzing the Evolution of Large-Scale Software Systems Using Design Structure Matrices and Design Rule Theory: Two Exploratory Cases", Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference, 2008, pp. 83-92.

[18] Bass, Len; Clements, Paul; & Kazman, Rick. „Software Architecture in Practice“, 2nd ed. Boston, MA: Addison-Wesley, 2003.

[19] Dennis Smith and Grace Lewis, "Service-Oriented Architecture (SOA) and Software Product Lines: Pre-Implementation Decisions", SPLC 2009, pp. 310.