Identification of Software Product Line Component Services

Martin Assmann[1], Gregor Engels[1]
Thomas von der Massen[2] and Andreas Wübbeke[1,2]

[1] Dept. of Computer Science, University of Paderborn
Warburger Straße 100, 33098 Paderborn, Germany

[2] DC Application Development, arvato services
An der Autobahn 33310 Gütersloh, Germany

Abstract. Software Product Line (SPL) approaches do not centrally improve the maintenance of software products of a SPL. This paper presents an approach for reducing maintenance costs of SPL products by using the concept Software as a Service (SaaS). The SPL-SaaS approach was developed with the experiences of arvato services integrating the SPL concept since years. It shows up the advantageous and disadvantageous characteristics of components that play a role for the concept combination. The main goal is to enable an IT-architect to identify adequate components. Therefore criteria for the identification of software components suitable for the approach are derived from these characteristics. Furthermore the requirements of the potential service users are examined and categorized concerning their effects on the system architecture. Special requirements of customers often lead to architectural constraints that are not compatible with the approach. If both, the criteria are met and the architectural constraints are compatible, the SPL-SaaS approach can be applied to a component. The whole approach is applied on an example of arvato services.

1 Introduction

SPLs were becoming an important development paradigm over the past years. The idea is to develop similar software products on a common basis, called platform. This paper presents the idea to extend the SPL development process with the maintenance process and illustrates this with the example of an address validation service. The goal is to reduce costs for maintaining software products by using the concepts of Service-oriented Computing (SoC) and SaaS.

The domain we are developing software for, deals with customer specific solutions in heterogeneous system and user landscapes. We learned from it, that it is hardly possible to satisfy the different customer needs and at the same time convert our SPL approach to a completely service oriented SPL. Often the customers are against giving away the sovereignty of their systems. Further on, things like non functional requirements (e. g. performance and security) and the increased complexity of the systems and their infrastructure impede the realization of a fully service oriented ap-

Assmann M., Engels G., von der Massen T. and Wübbeke A. (2009).
Identification of Software Product Line Component Services.
In *Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering - Evaluation of Novel Approaches to Software Engineering*, pages 45-56
DOI: 10.5220/0001961300450056

proach. Although we try to identify single (functional) components, which can be exposed as a service without provoking the above mentioned problems.

At first, this contribution presents the idea of reducing maintenance costs by deploying common software components as central services that are used for different software products at the same time. We point out the important characteristics of software components, which make them reusable in the way our approach proposes.

The remainder of this paper is structured as follows: In the 2nd section foundations on SPLs and the concept SaaS are introduced. In the 3rd section the potential of extending the SPL development process is described. In the 4th section the characteristics of our approach with its advantages and disadvantages are figured out. In section 5 we present an exemplary address validation service focusing its basic architecture. Related to the characteristics from section 4 and with the experience from the address validation service we present criteria to find fitting software components for our SPL-SaaS approach in section 6. Section 7 provides the related work in this area. The last section summarizes the results and points out further research topics in this field.

2 SPL, SoC and SaaS

The SPL approach deals with the development of similar Software Products based on a common platform. Thereby in all phases of the development process reuse of different artifacts is the main aim. In this context the platform provides different types of artifacts: Artifacts can be common to all or to some products developed within the SPL. The SPL development process proposed in literature (compare [2] and [3]) contains the phases requirements engineering, design, implementation (realization) and testing. In these phases different techniques (e. g. variability modeling) provide the possibility of reuse parts of the platform in different products of the SPL.

The second major concept that is part of this paper is *SoC*. A relatively similar concept is SaaS. Both utilize the service notion and both relate to software that is offered as a service. The vision of SaaS is to change the basic paradigm for development and maintenance of software systems, which is discussed in detail in [4] by Turner. He proposes to deliver software as a service rather than a licensed product. The main idea of the service is that it is not deployed where it is used but somewhere centrally. Users bind services needed at compile time or as preferred by Turner at runtime. Therefore the software services must have adequate descriptions, must be discoverable and should be composable, i.e. services can be created by combining other services. Service-oriented Computing, as described by Papazoglou in [5] and [6], builds the foundation for Service Oriented Architectures (SOA).

While SaaS aims on providing complete applications as services, SoC wants to provide business functions of finer granularity. Though, both approaches aim at advantages like lowered maintenance costs, higher degree of reuse as well as a different business model. It allows paying software per usage instead of buying licenses inflicting high fix costs. In the following we will point out how the software product line approach can benefit from these advantages.

3 Exploiting Potentials in the SPL Development Process

Regarding to software development processes like the Rational Unified Process (RUP) or the waterfall model the software lifecycle usually contains a phase for maintenance and operations. Within the RUP this phase is called transition [7]. SPL covers mainly the first three phases of the RUP. We believe that there are potential cost savings in the maintenance and operations phase being not utilized.

The SPL approach primarily decreases the development effort for software products but hardly addresses reduction of their maintenance and operation costs. Deploying and maintaining products separately at customer sites, makes it hard to exploit commonalities of the products. This means that the common components of a platform are only reused until software product assembly. Afterwards, deployed common components exist in separate system environments and are maintained individually. Therefore their maintenance is as expensive as the maintenance of single software products. As similarities between components exist but are not exploited at all regarding maintenance we think that there is a high cost saving potential in the transition phase. However, SPL domains that do not allow remote communication within their software components cannot benefit from our idea, for example SPLs for embedded systems like cell phone software.

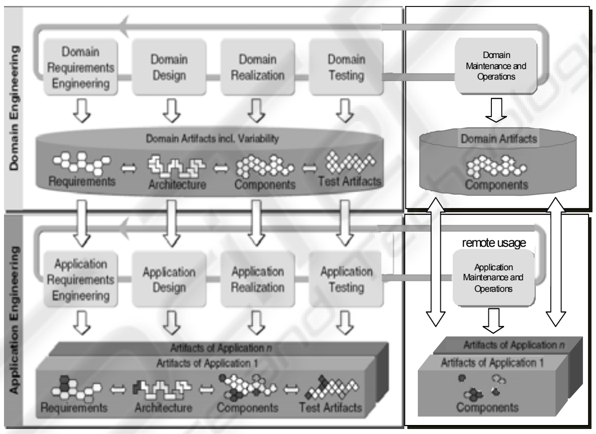

Fig. 1. Extension of the SPL development process modified from [2].

Fig. 1 shows the extension of Pohl's SPL development process [2] by a new pair of sub-processes called domain maintenance and operations, and application maintenance and operations. As the figure illustrates, the common components are held on the domain level (maintained and operated by the SPL platform provider). Only the product specific components are maintained and operated by the customer. The communication between the two levels is realized remotely. As shown the lifecycle process is extended to the maintenance sub-processes, because change requests concerning the components take effect on the maintenance.

Fig. 2 illustrates the changes from the usual situation in SPL architectures (upper part) to the situation that our approach suggests (lower part). The upper part of the picture depicts the architecture of two customers with individual applications. In the realization phase both have been assigned a bonus system component and a customer management component. The blue background of the bonus system component indicates that this is a custom component that usually has some customer specific modifications. The customer management component is deployed with little or no modifications.

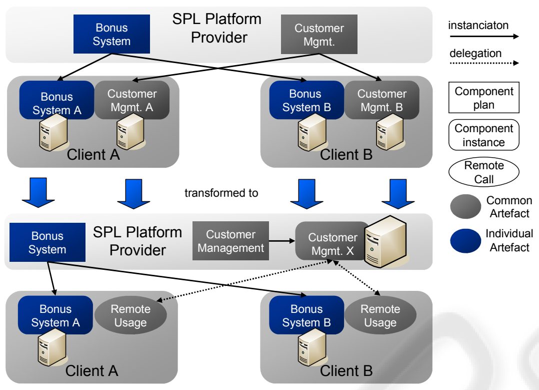

Fig. 2. Changes in the component deployment.

The transformation addresses the deployment of the customer management component. Instead of deploying it once for every customer it is deployed as part of the SPL Platform. The customer application now remotely uses the component as a software service. This means that the component is reused and has to be maintained and deployed only once. Major question arising when examining **Fig. 2** are: “What makes a common component in a SPL that is suitable for the deployment as a service?” and “Are there any other advantages and which disadvantages come with the approach?” We will answer both questions in the sections 4 and 6 in reversed order.

4 Characteristics of the Combined Approach

For our approach we want to answer two important questions. Firstly, what are the advantages/disadvantages? Secondly, when is a software component suitable for the approach? In the following we list the characteristics. With each characteristic we try to identify influencing factors of software components that can intensify/weaken the advantage/disadvantage. With this information the software architect can determine software components with characteristics that minimize disadvantages and maximize advantages of the approach. Regarding SPLs there are always two interesting aspects, the variation points and their binding time. There are two major categories of variation points. Firstly, the deployment location of a service has to be settled during design time. Secondly, there are minor variations for centrally deployed services. These variations reflect the range of the customers’ requirements for the service. After the range is anticipated the service is designed accordingly. By this the variability can be bound at design time. The variation points are described in section 6.

4.1 Advantageous Characteristics

First of all several issues can be consolidated. The first four of the following points address this topic. Firstly, hardware (e. g. a server system) can be used for several products. This means to deploy a common software component only once on a central server system and share it via remote connection.

Secondly, by sharing a server system providing services of common components deployed on it, several maintenance and operation aspects can be improved: availability and backup solution only has to be treated once, i.e. before we had several servers and every of them had to have availability and backup mechanisms like idle stand-by servers and mirrored hard disks. Functional extensions and updates to a common component have to be made only once. The same applies to the correction of faults. These changes always concern one component and thus the distribution effort is reduced, because the component is deployed only once on the shared server system.

Thirdly, the overall operation costs for hardware resources are reduced because as a single instance of a component requires less resources than several instances. The mentioned advantages all lead to less effort concerning maintenance and hence reduce the costs for it. According to this advantage suitable components should underlie frequent changes for updates. Additionally they should have high availability and backup requirements.

Fourthly, consolidation comes with an additional advantage concerning load-balancing. Usually a customer with its own systems can hardly afford to cover peak loads so his systems will just cover average load. Even if the centralized service is only able to cover average load of all its consumers, a single consumer causing a performance peak will not cause heavy performance losses on the central system, because his peak load has to be put into relation with computing power of the system designed for several customers. For the suitability of a component we can derive that it is increased if the component causes critical peak loads with relatively low average load.

Furthermore our approach of SPL component services opens up the possibility for new cost models, which are addressed by the next two paragraphs. In this context the maintenance and operations costs for the common parts of all products of the SPL can be shifted from a model with high fix investment cost (e. g. license, hardware) combined with effort for human resources to a usage cost model. Now there are mainly two new options for payment. Firstly, the customer can buy the service, its maintenance, and operations for a fixed amount of money per period. The second possibility derives from the SaaS approach. It means that the maintained and operated service is paid per usage only (variable costs). The latter kind of cost model has definitely less fix costs, which reduces the financial risks of the customer. Hence suitable software components are expensive with a high investment risk for the customer.

Furthermore, the new cost model provides the possibility of outsourcing the maintenance of the common parts of the product line. Firstly, this leads to the possibility for the customer to save money, because the SPL platform provider is able to be more efficient in maintenance and thus cheaper. Secondly, the customer can outsource functions that do not belong to his core business. Especially in combination with the pay-per-usage cost model he gains flexibility, i.e. he can dispose of the product respectively the service easily. From this we can conclude that suitability is influenced in a positive way if software components do not provide core business functions of the customer.

4.2 Disadvantageous Characteristics

In the following we point out disadvantages and as in the section before also the factors regarding software components that influence the disadvantages.

The first three points address drawbacks that arise from the centralization of components. Firstly, all shared software services have to be client-capable unless they are stateless. That means customer data is stored by the component this has to be taken into account designing the component. If every customer runs its own instance, then data and also access rights are divided without extra effort. Suitability of software components is increased by statelessness and public access. Otherwise the implementation of client-capability is inevitable.

The second point is closely related to the last one. Data sovereignty is transferred from the customer to the service provider. The requester has to trust the provider that handles the data with adequate security mechanisms. Suitability of software components will be decreased if the component stores confidential data and the customer does not trust the service provider at the same time.

Thirdly, virtual reuse means that several completely independent clients use a single server solution. A single point of failure is created by this. If the central component crashes, then every customer is affected. Nowadays, this should only be a question of costs as high-availability server solutions are on the market. The single point of failure is a performance bottleneck at the same time. The load that was distributed over many systems is now concentrated on one. Therefore a high performance system may be required. On the one hand, in peak load situations again all customers are affected, even those that are not responsible for the peak load situation. On the other hand a load balancing between all the requesters is given now. Peak loads of single requesters do not lead to performance problems as their single server would have encountered. High availability and high performance server solutions are required unless performance and availability of the centralized components are not critical. For the suitability of a component this means that high average loads or simultaneous peak loads of several users are negative.

Fourthly, communication via internet is sheer unavoidable. This has several consequences: Secure transmission has to be implemented. This might not have been necessary in decentralized solutions with data transfer in secure subnets only. Secure communication usually leads to higher communication overhead. Furthermore, the reliability of internet connections usually cannot be guaranteed. Last but not least it is more difficult to implement communication that can be initiated by both, remote and local, components. Suitability of components is decreased by data confidentiality as well as high communication effort and the ability to initiate communication. Higher reliability demands than the internet can offer prohibit the service approach.

The fifth point concerns change requests. Individual change requests have a negative effect on the approach. Too much individuality caused by change requests leads to software that is hardly maintainable and can decrease all the benefits of the approach. However, any change request leads to a down time if the software component is not runtime reconfigurable. All customers are affected from the downtime but all except from one have no benefit from the down time. Only components with little individual change requests are suitable, but these should be identified in the SPL platform anyways.

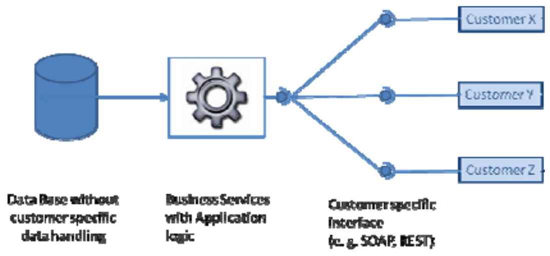
Customer X
Customer Y
Customer Z
Data Base without customer specific data handling
Business Services with Application logic
Customer specific Interface (e. g. SOAP, REST)

Fig. 3. Basic service architecture.

5 The Address Validation Service

Arvato services is an enterprise specialized on customized software solutions and services for outsourcing solutions. To achieve a high degree of efficiency by using synergies among different projects, arvato services is on the way to implement a SPL approach as depicted in **Fig. 1**.

The address validation service (AVS) is a service allowing access on an address database. In general the requester sends an address to the service and the result is whether the address is valid or not. The validity check of an address contains several sub checks that are partly optional.

Address validation is useful for many applications that can be (and some also were) produced with a SPL approach. Therefore it was recognized as a reusable component and as this it is used in different products.

If deployed at the customers IT site this component causes high maintenance costs. First of all it has to be regularly updated with new customer data and secondly optimization of algorithms is steadily done due to the high and still growing number of data entries. Furthermore, the component requires relatively expensive hardware, so that performance can be guaranteed. These are the main reasons why the address validation component was designed to be a centrally hosted service. It is remotely connected to customer systems that are produced with the product line and can also be used as a stand alone service.

The address validation is an example that can realize many of the advantageous characteristics while avoiding many of the disadvantageous characteristics of the SPL-SaaS approach..

The very basic system architecture of the address validation service is depicted in **Fig. 3**. In the back end there is a data base containing all the relevant data for the validity checks of the addresses. Its content is steadily kept up-to-date. Furthermore there is a business service that contains all the business logic to serve customers' validation requests. It has exactly one well defined interface offering all possible operations.

As customers may have their requirements concerning the interface, adapters offering different interface technologies like SOAP and REST are created. In addition the adapters may hide parts of the functionality, reducing the complexity for the customer as well as strictly enforcing access rights.

We have great cost-reducing effects from embedding the service in customized software solutions that are created with the help of our SPL approach.

6 Characteristics of Suitable Software Components

To identify suitable components we derive two categories of characteristics. Firstly, the high level characteristics delivering service candidates that are worth the effort of being analyzed on the architectural level. If the analysis on the architectural level is also positive, the realization of the component as a centrally deployed service is assumed as economically recommendable.

6.1 High Level Characteristics

As already mentioned our SPL-SaaS approach has several advantages and drawbacks that are dependant from the choice of the software components centrally maintained and delivered as a service. With the experience gathered from the address validation service in mind, we analyzed advantages and drawbacks mentioned in section 4. We derived adequate high level criteria to evaluate suitability of components for our SPL-SaaS approach. We identified positive as well as negative criteria. If components mainly fulfill the positive and mainly avoid fulfilling the negative criteria, they are adequate service candidates having enough potential for achieving a reduction of maintenance costs. The positive criteria maximizing the benefit of the approach are:

- High usage degree provides high reuse potential
- Frequent changes (functional extensions, fault corrections)
- High availability and backup requirements
- Components cause critical peak loads but have relatively low average load
- A high price which means a high investment risk (due to fix costs) for the customer

On the other hand there are negative criteria that should be minimized with the choice of suitable components to increase benefit:

- Providing core business functions of the customer
- Statefulness of the service and non-public access
- Storing confidential data while the customer does not trust the service provider
- High average loads or simultaneous peak loads of several users
- High communication effort
- Bi-directional initiation of communication
- Higher reliability demands (higher than the internet can provide)
- Individual change requests
- High performance requirements

6.2 Architectural Level Characteristics

If there is a service that fits to the criteria listed in the previous subsection, it is not yet clear if it is adequate for our approach. This is because of the architectural constraints that will arise due to the requesters' requirements. In general the abstract system architecture would be as depicted in **Fig. 3**. There is a database that is accessed by a service and a business service implementing the business logic used by the database. In addition there are several customer specific interfaces, which access the business service interface.

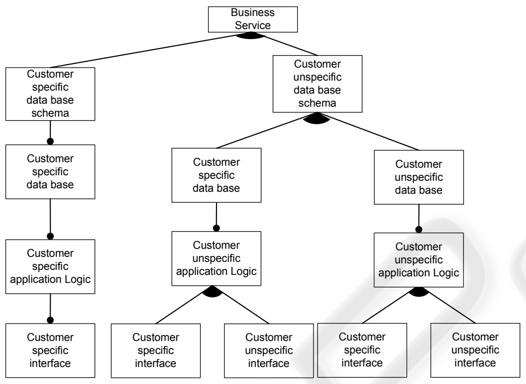
Business Service
Customer specific data base schema
Customer unspecific data base schema
Customer specific data base
Customer specific data base
Customer unspecific data base
Customer specific application Logic
Customer unspecific application Logic
Customer unspecific application Logic
Customer specific interface
Customer specific interface
Customer unspecific interface
Customer specific interface
Customer unspecific interface

Fig. 4. Service architecture feature diagram.

But there can be differences in the architecture according to varying customers' requirements. We have depicted these differences that have to be covered by the service in a feature diagram in Fig. 4. To any given service candidate the range of possible customer requirements being supported has to be anticipated. This results in a set of variation paths that have to be covered by the service solution. Every path indicates a certain complexity concerning the service realization and operation. Within the figure the complexity is generally decreasing for leaves from left to right. The variation points from the feature diagram are application logic, data base schema, data base content and interface. Any of them can be required to be customer specific or not, but not every combination makes sense. The feature diagram lists the five reasonable variation possibilities. The variation points concern variants on: *Data base schema, Data base content, Application logic,* and *Interface.*

First of the variation points is the data base schema. It may happen that customers need to have different data base schemas within a service. This requirement makes it quite hard to realize the service economically as it implies that each of the following presented variation points is forced to the customer specific choice. An example is a master data management service. Its purpose is to allow customers to store, update and request their master data. This is useful as else wise master data would be scattered over several databases. But for our case this would imply that we have to maintain a data base schema for each customer. Customer specific business logic and interfaces would also be required. For our address validation service, there exists only one data base schema serving all customers.

Second variation point is the data base content. If customers can be served with a service that has only one data base schema it still may occur that each customer has its own content for the data base. Furthermore the customers do not want that their data is accessible by other service users. An example for customer specific data base content is a schedule service. It offers a calendar and participants known to the system can be invited to meetings. The data base schema is the same for every customer, client, but every customer wants to be sure that only his employees can see their own appointments. To this end, data records could be marked with their owners name so that access control can be guaranteed. In the end it means less effort to maintain a database with a single schema instead of maintaining multiple schemas. In the case of our address validation service all customers access the same data base content – the addresses plus some extra validation content.

Third variation point is the application logic. It is possible that not all customers want exactly the same operations on the data. If that is the case, usually the data base schema and the thus also the data base content will be customer specific. This is the worst case scenario because it means that every customer has its own specific service. This makes it hard to realize cost efficiency with our SPL-SaaS approach and is generally not encouraged. If there is only customer unspecific application logic, then we assume that there are only customer unspecific data base schemas. Executing the same operations on different data schemas is generally not advisable. The address validation service offers the same application logic for all customers.

Fourth variation point is the interface towards the customer. Even if the service is always part of products built with the SPL and the customer does not access it directly, it is helpful to have different interfaces. For example, if variability within the SPL platform allows products based on different technologies, several interfaces differing in technology are useful. The different interfaces are only adapters to the interface offered by the business component. Adapters could also be located on the client side, but if other customers want to access the service directly, it is an advantage being able to offer a plethora of interfaces to the service. A customer specific interface can also hide some functionality if the complexity of functions shall be reduced towards the customer. Additionally it can be used to restrict the access to certain functions by simply not adapting them. The adapters can restrict the functionality of the original interface but do not change it in other ways. Different point-to-point security variations, like REST over https, can also be covered with adapters. The address validation service offers different interfaces like SOAP over JML and REST over http.

We have seen that not all combinations of features make sense. According to **Fig. 4** there are five combinations. Each combination implies a different complexity for the centralized service.

The left most leaf shows the most inadequate case. Customer specific data base schemas require higher development, test and maintenance efforts. Therefore the realization of a service that has only customer specific properties is discouraged.

The right most leaf has only customer unspecific variants. This means that there is no variability that has to be bound. The service provider is in the lucky position to design one service and to have multiple service users. In this case efforts for design, test and maintenance do not scale with the number of users. This is the good situation as the savings with the centralized service approach are very high.

Customer specific interfaces are relatively easy to develop and maintain. Testing new customer specific interfaces can be reduced to testing the sole interface instead of testing the whole service. Our address validation service is an example for a service categorized like the second leaf from the right. It has different interfaces and we have experienced that each interface causes only little effort overall. Furthermore an existing interface might be reused for a new customer. The more interfaces there are the higher is the probability to be able to reuse an interface.

The two remaining leaves in **Fig. 4** cause more design, test and maintenance effort, but are still considered as suitable service candidates.

7 Related Work

The idea of combining the SPL and SaaS approach is addressed by several contributions. The importance of the topic is brought to life by [1] and [8].

In [9] the authors describe the idea of a web based product line. In this case technological issues on building such product lines are discussed. Compared to our approach the contribution concentrates on building product lines completely from web services. [10] concentrates on variability in web service flows. Some of the described variability points like *protocol* are also interesting for our approach, but the aim of our contribution is not centrally to handle variability in the flow of web services.

Chang and Kim also recognized the common reuse potentials in SPL and SOA [11], but they consider everything as a service. Thus, they identify variation points on process level, which is not applicable for our domain, because of the previously mentions drawbacks (see also section 2).

An interesting approach combining SOA and SPL concepts for creating business process families is given in [12]. Though, this approach does not cover the deployment phase.

In [13] the authors describe how to manage variability in service centric systems with technologies from the SPL approach. Our approach works the other way round and provides service technology for SPLs.

8 Conclusions and Future Work

In this paper we presented an approach to reduce maintenance costs of software products developed from a SPL. Therefore the SPL process is extended with the maintenance process. This means to identify and deploy common components centrally and offering them as a service. The service then is used for different products of the SPL. This concept is called SaaS as the common components can be seen as a service provided for different products. The presented approach holds several advantageous and disadvantageous characteristics ready which have to be taken into account for selecting adequate candidates for common services. From these characteristics several criteria for identifying suitable components have been derived. Additionally we have developed a service architecture feature diagram, which provides the possibility to evaluate components concerning their adequacy for our SPL-SaaS approach. Suitable components have a high reuse potential while causing little service development costs. We presented an example of such a component from the arvato environment to show the practical need of our idea.

The presented criteria are a first step for a detailed evaluation catalogue for assessing common components concerning to their reuse potential. In future research we are going to built up this evaluation catalogue to evaluate the reuse potential in reference to maintainability. Afterwards the catalogue is going to be used and evaluated by analyzing the arvato SPL to show its workability.

References

1. Helferich, A.; Herzwurm, G; Jesse, S.; Mikusz, M.: Software Product Lines, Service-Oriented Architecture and Frameworks: Worlds Apart or Ideal Partners? Lecture Notes in Computer Science, pp. 187—201. Springer (2006)
2. Pohl, K.; Böckle, G., van der Linden, F. J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer (2005)
3. Clements, P.; Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley Professional (2001)
4. M. Turner, D. Budgen, and P. Brereton: “Turning Software into a Service”, Computer, vol. 36, no. 10, pp. 38–44, (2003)
5. M.P. Papazoglou and D. Georgakopoulos: “Service Oriented Computing”; Comm. ACM, vol. 46, no. 10, pp. 25–28, (2003)
6. M. P. Papazoglou: “Service-Oriented Computing: Concepts, Characteristics and Directions”, 4th International Conference on Web Information Systems Engineering (WISE'03), Rome, Italy, (2003)
7. Jacobson I.; Booch, G.; Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley Professional (1999)
8. C. Wienands. “Synergies Between Service-Oriented Architecture and Software Product Lines,” 2006. Sie-mens Corporate Research. Princeton, NJ.
9. A. Sillitti, T. Vernazza, G. Succi: “Service based Product Lines”, Proceedings of the 3rd International Workshop on Software Product Lines: Economics, Architectures, and Implications (ICSE 2002)
10. S. Segura, D. Benavides, A. Ruiz-Cortés. P. Trinidad. “A Taxonomy of Variability in Web Service Flows Service Oriented Architectures and Product Lines” SOAPL - 07. SPLC'07. Kyoto, Japan. 2007
11. S. H. Chang, S. D. Kim, “A Variability Modeling Method for Adaptable Services in Service-Oriented Computing”, Software Product Line Conference, 2007 (SPLC 2007), pp. 261-268
12. E. Ye, M. Moon, Y. Kim, K. Yeom. “An Approach to Designing Service-Oriented Product-Line Architecture for Business Process Families” Proceedings of the 9th International Conference on Advanced Communication Technology, pp. 999-1002. Phoenix Park, Republic of Korea (2007)
13. J. Lee, D. Muthig, M. Kim, S. Park “Identifying and Specifying Reusable Services of Service Centric Systems Through Product Line Technology”, Proceedings of the First Workshop on Service-Oriented Architectures and Product Lines (SOAPL 07), pp.57-67