ENTERPRISE SERVICES ON THE CLOUD BY A TECHNIQUE TO MAINTAIN VARIABILITY

J.Vinothkumar
Assistant Professor
Department of computer Application
Rajiv Gandhi College of Engineering and Technology
Pondicherry
jaivinothkumar.mca@gmail.com

V.Ramsundhar
School of computing
Sastra University
Thanjavur
sundar_varathu@yahoo.com

V.Raghavan
Assistant Professor
Department of computer Application
Rajiv Gandhi College of Engineering and Technology
Pondicherry

Abstract
These variability concerns affect both the service interface and the service provider implementation and hence are crosscutting in nature. In this paper, we use principles of aspect-oriented software development to modularize these variability concerns. We also provide an aspect specification scheme to specify these concerns. We propose an approach to create heavyweight service variants centered on a Service Kernel, which forms a common service core across tenants. Heavyweight service variants are created by weaving aspects into the service kernel. Our approach provides improved governance for the provider while offering maximum flexibility for the consumers.

1. Introduction
Web-business platforms are hosted software platforms that provide business processes to support core businesses of customers based on a software-as-a-service (SaaS) model e.g. SalesForce.com®, SAP® Business By Design. Web-business platforms are typically hosted on a multi-tenant cloud infrastructure. They deliver business capabilities as enterprise services (e.g. PAYROLLRUNSERVICE – a service to run employee payroll). These enterprise services are based on Service-Oriented Architecture (SOA) [1]. Every enterprise service has two parts – the ‘capability on offer’, the business capability that the service represents and the ‘terms of offer’, the terms at which the service is offered to customers (e.g. pricing, quality of service etc.).

Technically, the ‘capability on offer’ is described by the Service Interface, which is realized by an underlying service provider implementation. Enterprise service is offered to different customers on different terms of offer, leading to various flavors of the same service – we call these Service Flavors [2]. For example, a SHIPPINGSERVICE – a service to ship consignments from one place to another – can be offered at different pricing and payment terms to a subscription customer than to a regular customer. Service Flavors represent lightweight variants of an enterprise service where the capability on offer remains the same, but the terms at which the service is offered is varied i.e. service interface and service provider implementation remain unaffected across service flavors. The challenge of heavyweight service variants becomes complicated in a hosted SaaS model. Present day SaaS providers employ a multi-tenant cloud infrastructure [3] to host services. SaaS customers could be from different geographies, belong to different industry verticals and seek business-specific variants of the hosted enterprise services. In multi-tenant environments a single shared instance of an enterprise service must support customer (tenant) specific variations. Multi-tenancy imposes runtime governance issues such as runtime configurability and sharing of runtime instances. Supporting service variability in such an environment is an important requirement for hosted enterprise services.

Our key goal in this paper is to support heavyweight variants of enterprise services, especially those hosted in multi-tenant cloud environments. Our approach is based on aspect-oriented software development [4] principles that allow separation of concerns through systematic
identification, modularization, and composition of concerns. Our contributions are: the notion of heavyweight service variants, the approach to modularize service variability factors as aspects, and the approach to host tenant-specific service variants on the cloud.

Fig 1.b shows the service provider implementation, the Payroll Provider class with language specific data types.

1.1. Rousing Example

This is to arouse from slumber, apathy, or depression. In this paper, we use Pay Cloud, a fictitious multi-country outsourced payroll provider as an example. Pay Cloud offers a multi-tenant “payroll on the cloud” service to small and mid-size companies to handle employee payroll and benefits. We chose this example because payroll is a commonly outsourced service and is a complex service to be delivered on the cloud from a provider perspective. Multi-country payroll service is a specialized service with several variations. Firstly, there are significant differences in tax computation rules, especially tax rates and tax slabs for different countries. Legal and regulatory compliance obligations also significantly vary across different geographies, e.g. maternity and paternity benefits in different countries. Regulatory obligations may also vary across industry sectors, impacting payroll. Secondly, companies can have different employee compensation schemes and company policies (e.g. car lease policy, personal loan policy), which have to be supported by the outsourced payroll provider. We consider two use-cases that highlight variations in the payroll service offered by Pay Cloud.

1: Run payroll for a multi-national company

Mull over a multinational company having employees across sales offices in New York and Singapore and manufacturing facilities in India and China. Pay Cloud’s payroll service must now run employee payroll differently for four different countries considering variations in tax rates and slabs of these countries. Additionally, compliance to local laws and statutory regulations has to be ensured. Technically, apart from the variations in the payroll calculation procedure, even the payroll data needed by Pay Cloud to process employee payroll can be different across country versions.

2: Direct Deposit of Employee Salary

A number of customers prefer direct deposit of salary to employee bank accounts rather than issuing pay cheques to employees. Pay Cloud must support such variations in the payroll process. In case of direct deposit, the customer must additionally provide employee bank details to Pay Cloud. We provide a simplified white-box view of the PAYROLLRUNSERVICE offered by Pay Cloud showing both the abstract service interface and service provider implementation classes. Fig 1.a shows the service interface Employee Payroll Run Interface with the request message data type (MDT) Payroll Run Request Message.
The remainder of the paper is organized as follows: section 2 describes the approach to support heavyweight service variants using aspects, a simple join-point model for service interface and details about aspect weaving; section 3 deals with specifying tenant specific variations using aspects; section 4 discusses related work; section 5 provides future work and conclusions.

2. Underneath long-lasting Variants Using Aspects

Heavyweight variants of an enterprise service result due to the inherent variations in the business process, business rules, globalization concerns, industry-requirements as well as customer (tenant) specific requirements. Usually these variations may not affect the service capability or the goal of the service consumer.

However following variations are possible:

- Interface variations where the input and output messages need to be enhanced with additional data types. For example, additional employee bank details need to be sent to the payroll provider in case of UC#2, resulting in the enhancement of Payroll Run Request Message.

- Implementation variations where the service provider implementation is varied. For example, applying different tax computation rules for different country versions in case of UC#1, resulting in changes to Calculate Gross Payroll () method.

The concerns that necessitate heavyweight variants can be categorized as domain-specific (e.g. globalization, business rules, and compliance) or as business-process-specific (e.g. direct deposit of employee salary) concerns. In either case, the semantics of these concerns relate to the problem domain. These concerns are crosscutting in nature – they may impact both the service interface and the implementation (Fig 2). Thus they imply a change of service contract: each variant offers an enhanced contract with its corresponding behavioral modification at the implementation. We identify these concerns and modularize them as aspects, following the tenets of aspect-oriented software development (AOSD). By doing so, we can later combine these aspects with the core service capability to create heavyweight service variants.
For instance, if writing an application for handling medical records, the indexing of such records is a core concern, while logging a history of changes to the record database or user database, or an authentication system, would be cross-cutting concerns since they touch more parts of the program.

Cross-cutting concerns are parts of a program that rely on or must affect many other parts of the system. They form the basis for the development of aspects. Such cross-cutting concerns do not fit cleanly into object-oriented programming. During early service design, while defining the scope of the service (i.e. the service cut) the core service capability has to be identified based on a minimality criteria on the capability on offer. While the capability on offer is derived from consumer goals, the minimality criteria are driven by Parnassian goals – change and independence. Those concerns that are likely to change together (highly cohesive) must be grouped together and those concerns that are likely to change independently (loosely coupled) must be separated. Out of these concern groups, a primary group is identified such that it identifies a primary concern – without which all other concerns are irrelevant. For instance, custom computations for tax rules and the direct deposit feature would be redundant without the notion of a module responsible for basic payroll calculation. We call this least common denominator that we derive as the minimal service core – the Service Kernel based on the Microkernel architecture pattern [5]. However, in the microkernel architecture the kernel is independent of the adapters i.e. using adapters help to derive variations of the kernel but the behavior of the kernel is not altered. In our approach, the behavior of the service kernel may be altered by the other concerns. Therefore, we modularize these concerns as aspects and weave them into the service kernel. In AOSD, crosscutting concerns are identified as aspects. These crosscutting concerns are composed with the primary decomposition i.e. base module. In practice, the most popular crosscutting concerns such as logging and security are technical concerns. Though these technical aspects impact the runtime execution of the base module, they do not alter the behavioral semantics or the contract of the module. In our approach, the aspects are problem domain crosscutting concerns. These concerns impact both the contract (interface) and the implementation.

For example, the PAYROLLRunSERVICE is designed as a service kernel with the request message containing basic employee information such as employment, position, compensation and time recording and the provider implementation supporting basic gross payroll, net payroll calculation as well as payroll results. Concerns such as globalization, business rules and compliance that lead to heavyweight variations are identified and modularized as aspects. Methodologies for concern identification and modularization are beyond the scope of this paper and are dealt in detail elsewhere [6-9]. However we do provide a few pragmatic guidelines to achieve flexibility and separation of concerns:

I) Service kernel has value proposition as an independent business capability. This may not necessarily be true for other concerns. This establishes the notion of a service and its variants i.e. all variants are dependent on a common service kernel.

II) Service kernel is usually arrived at as an intersection of consumers’ requirements and provider’s capabilities. Factors that distinguish the core capability from the crosscutting concerns are often specific to a particular application scenario. Thus it is possible that a service and its variants may be specified and designed together but it is often likely that variants evolve depending on tenant requirements.

III) Service kernel must represent a core business capability of the provider – in contrast, crosscutting concerns may be modularized (i.e. packaged) as value additions and may even be provided by the partner eco-system.

IV) A concern need not be part of the service kernel unless the provider determines that a concern will be packaged and provided to all the consumers.

For our purposes, we assume that crosscutting concerns identified are orthogonal – not only to the service kernel but also to each other. In our example, the globalization concern is orthogonal to the direct deposit concerns.

2.1. Aspect Definition

Once aspects are identified, they need to be specified in a machine-readable format to be used by an aspect weaver [10], which composes these aspects with the service kernel. Aspect specification involves specification of point cuts and advices based on a join-point model [11]. In the case of enterprise services with variants, we need ways to specify aspects that crosscut the service implementation as well as the service interface. Aspect specification at the implementation-level has matured and language specific support is available – e.g. AspectJ, AspectC++, Seaser.NET etc. The AspectJ [12] join-point model (JPM) is one of the most popular join-point models for quantifying aspects. Since service provider implementations could be in different platforms and languages, we prefer a language-independent aspect specification scheme, for instance [13]. We use a language-independent aspect specification that is an XML schema, which is a minor variation of the schema used in Aspect Werkz platform [14]. The aspects specified using this XML schema can be translated to language specific constructs.
Our aspect specification scheme supports the usual specification for provider implementation with point cut, point cut expressions (to refer to dynamic join points), and advices. A regular aspect weaver would be able to apply the advice on the service provider implementation based on the aspect specification. Our aspect specification scheme supports simple enhancements of the service interface. We have modified the Aspect Werkz schema to specify our service interface enhancement model, which supports enhancement of service interfaces as well as message data types (MDTs) (Fig. 3.a).

The input and output messages of a service interface are MDTs. The MDTs are constructed out of core data types (CDT). Our service interface enhancement model distinguishes between enhancement of message data types (MDTs) and enhancement of core data types (CDTs). Enhancement of MDTs occurs in the context of the service interface to support reuse of MDTs across service interfaces. Enhancement of CDTs is independent of service interfaces (Fig 3.b). An aspect weaver will apply these data type enhancements at the interface level using the XML extension mechanism. At the implementation level, these XML data types have language-specific data types. We use the static join-point model and inter-type definitions to enhance these language-specific data types.

Fig 3.b: Service Interface: Data Type Enhancements

We illustrate our aspect specification for UC#2: Direct Deposit of Employee Salary. We treat direct salary deposit as a separate concern due to variation in the underlying payroll business process. Fig 4.a presents a schematic view of the direct salary deposit aspect. When a customer chooses this feature, this concern would be composed with the service kernel. Fig 4.b shows the Direct Salary Deposit aspect specification using XML.

Fig 4.a: Aspect: Direct Salary Deposit

Fig 4.b: Direct Salary Deposit Aspect Specification

When the Direct Salary Deposit is weaved into the service kernel – the PAYROLL RUN SERVICE, the service interface in enhanced by enhancing the Payroll Run Request Message MDT with Employee Bank Details Info data type. The provider implementation is also affected by the direct Deposit () advice at the process Results point cut. The process Results point cut refers to a dynamic join point – the call of deliver Payroll Results () method of the Payroll Provider implementation. The direct Deposit () advice will provide to implementation to support the direct deposit feature of the payroll process.
2.2. Weaving Aspects into the Service Kernel

Aspect weaving is a process by which crosscutting concerns, which are expressed as aspects are composed with the primary decomposition – the Service Kernel in our case. Aspect weavers could support weaving either at compile-time (static) or at run-time (dynamic) [15-17]. We discuss the weaving issues and implications for our use cases. Whenever a heavyweight variant needs to be created the aspects are designed and statically weaved into the service kernel. But in a cloud environment, it may not be practical to stop and restart provisioned services. In such cases a hot fix approach to aspect weaving is required. Several approaches for hot-fixing have been proposed in the context of aspects [18]. Specifically [19] discusses an approach for dynamic provisioning of web services using aspects. We have assumed the concerns identified are orthogonal. Ensuring orthogonality among concerns is beyond the scope of this paper. But tools and techniques for detecting and resolving conflicts – such as precedence issues - between aspects have been considered extensively in the literature [20, 21] and are more than adequate to handle conflicts among aspects.

3. Tenant-Specific Variations with Aspects

In the multi-tenant scenario, consumer requirements vary across tenants and evolve over time. This presents a serious conflict between governance and flexibility:

- offering a single service supporting all the required features reduces the governance efforts for the provider but constrains all consumers to a single interface that cannot be changed easily over time

- offering separate services for each consumer ensures maximum flexibility for consumers but increases the governance efforts for the provider

We propose the use of aspects to derive heavyweight service variants such that a single instance of the service kernel can serve multiple tenants. In our approach each service variant is associated with a tenant. The methodology outlined in Section 2 can be used for architecting and managing the entire system as a collection of service variants. (Fig.5.a)

![Fig. 5.a: A system of Service Variants for Multiple Tenants](image)

The heavyweight variants are defined and implemented around a service kernel that is identified to be - and is likely to remain – the common component across all tenants. A service variant is then realized via a composition of the service kernel with a tenant-specific set of aspects. We provide a tenant-specific configuration as an XML schema (Fig. 5.b), which can used by an aspect weaver to provision heavyweight service variants for tenants. Our approach ensures maximum reusability (and hence minimal governance) for the service provider while at the same time offering maximum variability for the service consumers.

![Fig. 5.b: XML Schema for Tenant-Specific Configuration](image)

4. Related Work

Even though there are a variety of tools to handle conflicts among concerns we plan to address the orthogonality issue at the specification level. In the minimum our specification must be enhanced to support orthogonality constraints .Over the last few years, there has been a flurry of activity in the research community addressing variability in software. Most of these efforts – for instance, [22, 23] – address various facets of software governance such as re-use, configurability, and maintainability. Many such efforts are specifically targeted at differentiation in product lines and process families [22-26]. Despite the seeming differences in the approaches, the primary themes of these efforts was identifying variability factors at the software implementation level and addressing them through
techniques at the modeling and design level.

More recently focus has shifted to addressing variability in service-oriented systems. Variation-Oriented Analysis and Design, a paradigm proposed in [27-29] outlines specific techniques for modeling and designing services in such a way that variability requirements are well specified, analyzed, and categorized, and the solution is designed and architected to capture the variability requirements at different levels of the system. On the other hand, [30] clearly identifies and delineates the types of variability that may occur in a service-oriented system. Our work is supplementary to both of these efforts in that we address variability in the context of a service provider hosting a multi-tenant enterprise service on the cloud. Our approach uses aspect-oriented techniques for designing and implementing heavyweight variants of a service. While [29] briefly mentions the use of aspect-oriented tools for handling legacy components at the implementation level, we propose the use of aspect-oriented approach for concern identification, design and implementation of service variants with a common kernel. Aspect-Oriented Programming was proposed as a solution for modularizing cross-cutting concerns at the implementation (i.e. systems) level [11]. Aspect Oriented Modeling [31] was later proposed to address early design issues in specifying crosscutting concerns and their composition with primary concerns. Later works on aspects has contributed to identification of concerns at different levels – not just at systems level – that can be modularized as aspects [32, 33]. Lightweight service variants or service flavors are proposed in [2]. In this paper, we propose the use of aspect-orientation for supporting service variants by modularizing concerns as aspects irrespective of the nature (functional or non-functional), level (domain, service, or system) or granularity (lightweight or heavyweight) of the concern.

5. Conclusion and Future Work

We proposed the use of aspect-orientation for modularizing variability factors in enterprise services as crosscutting concerns or aspects. We also proposed a way to compose these concerns with the service kernel – a minimal service core – to create heavyweight service variants. The use of aspect-orientation for deriving heavyweight service variants is novel in our approach.

We also proposed a method to improve governance for a provider offering enterprise services on a multi-tenant cloud infrastructure by treating a tenant as a system of heavyweight service variants. Our approach offers maximum flexibility for tenants while ensuring easy configurability and reuse for the provider. Flexibility for tenants is offered by providing variability through heavyweight service variants. Easy configurability and reuse for the provider are achieved by constructing the variants around a common kernel.

References


