Interface Design for Product Servers

Jan Ortmann
(Dokumentversion 1652)

Introduction

A product server (also known as product engine or product component) provides operational systems, such as policy administration systems or sales and service systems, with product information and other product-related services. These services include, for example, premium computation or validation. Insurance products are defined with a separate product definition system.

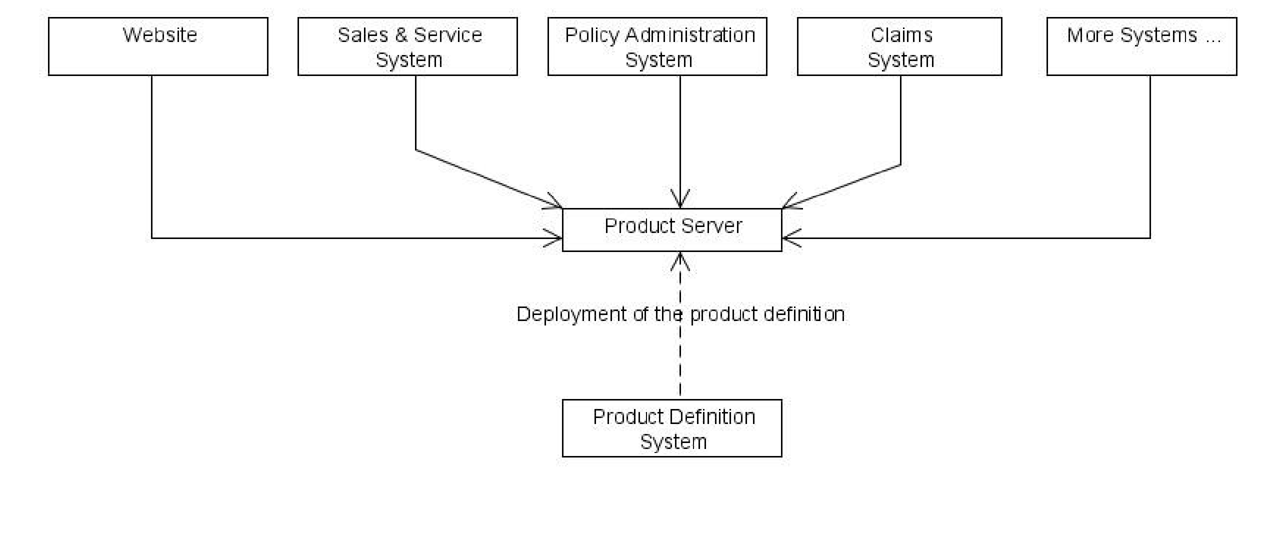
Website
Sales & Service System
Policy Administration System
Claims System
More Systems ...
Product Server
Deployment of the product definition
Product Definition System

Figure 1: Product Server and Product Definition System

This paper will explain which variants there are in constructing the product server's interface, and the pros and cons pertaining to these variants. We assume that the structure of contracts and products within the product server are mapped to a business object model. Proposals will be considered as offered contracts, so they have the same structure as contracts.

In our examples, we assume that the product server makes its services available through Stateless Session Beans. As the services should also be accessible through remote procedure calls, the interface uses Data Transfer Objects (DTOs) [Fowler, PoEAA]. However, the construction principles are technology independent and universally applicable.

Similarly structured Interfaces vs. Flat Interfaces

Many of the services offered by a product server require information about a contract or a proposal. Consider, for example, the computation of insurance premiums or the acceptance check. The method signatures could, for example, look as follows:

```
public ContractDto computePremium(ContractDto contract);

public MessageList isAcceptable(ContractDto contract);
```

The premium computation method will take a contract parameter with empty premium attributes (net premium, gross premium, etc.) and it will return a contract where these attributes contain the computed values.

There are two basic ways to design interface parameters for a product server:

- Interface parameters that are structurally similar to the business object model
- Flat interface parameters

In practice, obviously, there are many possible gradations between these two extremes.

Both variants can best be explained by way of an example. The following figure shows the business object model.

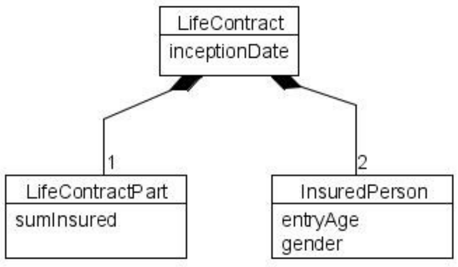

Figure 2: Sample Model 1

As the name suggests, a similarly structured interface would have „similar“ classes with the same attributes and relationships (but without the business methods, of course). The similarly structured interface classes would map exactly to the structure of the business object model, i.e. they would contain all the classes, attributes and relationships of the model.

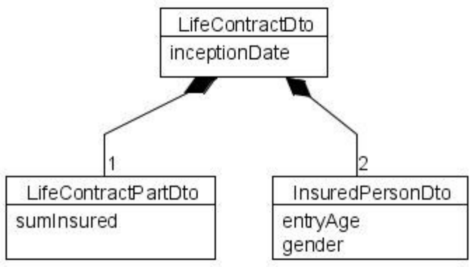

Abbildung 3: Similar Structured Interface Model

A flat interface for a premium computation, however, could consist of just one `LifeInsuranceDto` class with the attributes of all classes of the business object model. The two insured persons would be represented by a postfix to the attributes, such as entryAgeIP1 and entryAgeIP2.

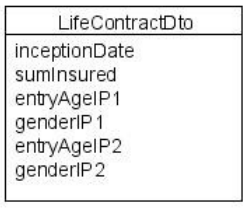

Figure 4: Flat Interface Model

Flexible vs. Product-specific Interfaces

In our discussion we will use the following business object model as an example.

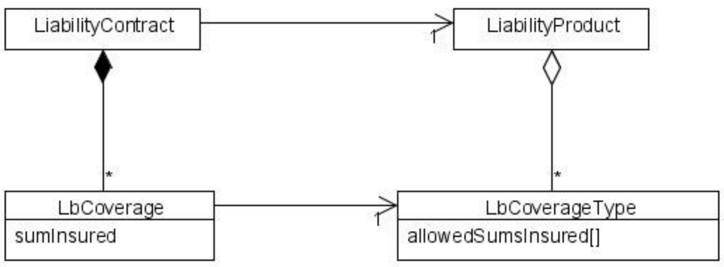

Figure 5: Sample Model2

A liability contract can comprise any number of coverages. Similarly, on the product side, a liability product will comprise any number of coverage types. For each coverage type, the contract can include not more than one coverage. Each coverage has a sum insured and the associated coverage type defines which sums are allowed in a coverage.

The following sample product is defined based on this model.

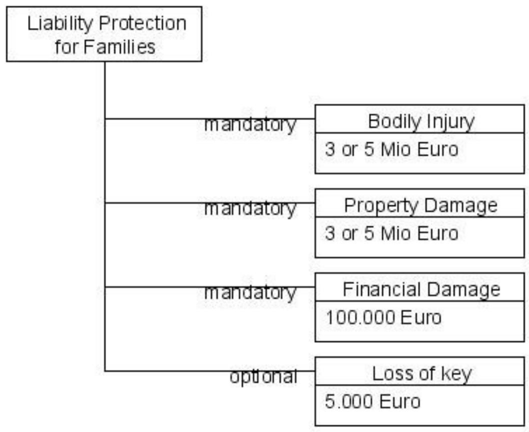

Figure 6: Structure of the Sample Product „Liability Protection for Families“

If you want to develop a premium computation web service for an external partner, you can define a flat interface that takes the product information into account and build it with just the following parameters:

- sumInsuredBodilyInjury
- sumInsuredPropertyDamage
- lossOfKeyIncluded

This implies the following product information:

- There are no more than the above five coverage types
- LossOfKey is the only optional coverage, all others are mandatory
- For LossOfKey and FinancialDamage, respectively, there can only be one sum insured. As a result, the amount of the sum insured need not be transferred for the other coverages.

The obvious consequence is that product changes entail analogous changes in the interface.

Alternatively, the structure of the interface can be designed so as to handle any number of coverages (so that it would basically be similar to the business model). In this structure, the reference of contract classes to the corresponding product classes would be replaced by an ID, like this:

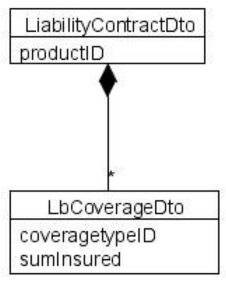

Figure 7: Similarly Structured Interface for Example 2

What consequences would this design have for the calling system? There are two alternatives:

- Product flexibility within the calling system
  The calling system can be built so that it retrieves the product information at runtime. It knows the structure of the business model but ignores the specific product data. Hence, it retrieves the allowable coverage types, asks if these are required or optional, and reads the allowable sums insured. Based on the product information, the system's user interface can dynamically display the coverages including the fields for entering the sums insured. The coverage type ID associated with the actual coverage will be stored in the calling system's database.
- Assumptions about the product structure within the calling system
  An online premium calculator made available by the insurance company could, for instance, display two combo boxes labelled „Sum Insured Bodily Injury“ and „Sum Insured Property Damage,“ respectively, and a „Loss Of Key Y/N“ checkbox. This way the above mentioned product information is coded into the calculator. When the online calculator calls the product server, the calculator's (flat) model has to be mapped to the flexible interface of the product server. To do this, the coverage type IDs have to be included in the mapping code. If, for example, the checkbox is enabled, a coverage with the CoverageTypeID of LossOfKey has to be added to the contract.

To sum up, there are two extremes to the interface construction:

- flat interfaces with assumptions regarding the concrete product structure
- flexible interfaces that exhibit the same (or a very similar) structure as the business model

The latter makes no assumptions as to the product structure that go beyond the representation in the business model.

Assessment

With the introduction of a centralized product server, insurance companies aim to achieve more product flexibility and quicker product development times. To reach this goal, product flexibility first and foremost has to be implemented in the operational systems. This can only be done if the product server offers a reasonably flexible interface. Thus, a product server should expose a flexible interface that corresponds to the business model. That doesn't necessarily mean that business models and interfaces must be 100% structurally congruent. So, for example, the interface model can do without subclasses that differ only in implementation specific ways from their superclass methods.

A drawback of this type of flexible interfaces is, however, that developers of such systems find them difficult to comprehend, when they are working on a different business model. The developer of the premium calculator shown in the example above, has to map the simple (inflexible) model of his calculator to the more flexible model of the interface. To do this, he has to be at least aware of both the flexible model and the individual product IDs.

Sales systems and systems of external partners, as for example broker systems and insurance quotes comparison sites on the Internet, often work with different and simpler models. It makes sense to provide services with a relatively flat interface for external partners on whose systems you have no bearing. This is particularly advisable when only the current product generation has to be supported. If several product generations have to be supported, on the other hand, this can mean that the interface has to provide more flexibility in order to account for all these generations.

Using the Builder Pattern to Create Interface Objects

To facilitate integration for those developers of calling systems who work with simpler models, you can provide a Builder[1] that is able to create the interface objects easily. We could then write a `LbContractDtoBuilder` for the above interface model as follows:

```java
public class LbContractDtoBuilder {

    public final static String PRODUCT_FAMILY_ID = "FAM";
    public final static String BODILY_INJURY_ID = "BI";
    public final static String PROPERTY_DAMAGE_ID = "PD";
    public final static String LOSS_OF_KEY_ID = "LK";

    private LiabilityContractDto contract;

    public void createContract(
            String productId,
            Money siBodilyInjury,
            Money siPropertyDamage) {

        contract = new LiabilityContractDto(productId);

        LbCoverageDto biCoverage = new LiCoverageDto(BODILY_INJURY_ID);
        biCoverage.setSumInsured(siBodilyInjury);
        contract.addCoverage(biCoverage);

        LbCoverageDto pdCoverage = new LiCoverageDto(PROPERTY_DAMAGE_ID);
        pdCoverage.setSumInsured(siPropertyDamage);
        contract.addCoverage(pdCoverage);
    }

    public void addLossOfKey() {
        LbCoverageDto svCoverage = new LiCoverageDto(LOSS_OF_KEY_ID);
        contract.addCoverage(svCoverage);
    }

    public LiabilityContractDto getContract() {
        return contract;
    }
}
```

The following code shows how the premium calculator could now use this builder to create a

1 This usage of the Builder pattern corresponds to [RtoP], rather than to the explanation in [GoF].

LiabilityContractDto object.

```
public LiabilityContractDto createLiabilityContractDto(
    Money siBodilyInjury,
    Money siPropertyDamage,
    boolean lossOfKey) {

    LbContractDtoBuilder builder = new LbContractDtoBuilder();
    builder.createContract(LbContractDtoBuilder.PRODUCT_FAMILY_ID,
        siBodilyInjury, siPropertyDamage);
    if (lossOfKey) {
        builder.addLossOfKey();
    }
    return builder.getContract();
}
```

Thus, the builder can simplify the integration effort for the calling system's developers while eliminating the need to provide a flat interface.

Flexibility with „Dynamic Properties“

Up to this point, we have assumed that the attributes of the interface classes are set at compile time by defining the appropriate getter and setter methods for each attribute. That means that the introduction of new attributes inevitably entails source code changes on the interface. This can be avoided by using the „Dynamic Properties“ pattern. A detailed discussion can be found in [Fowler, Properties].

Instead of defining special getters and setters for every single attribute, you can define generic getter/setter methods that take the attribute name as a parameter (for each interface class or in a base class) as shown in the following code snippet.

```
private Map<String, Object> attributeValues;

public void setAtributeValue(String attributeName, Object value) {
     attributeValues.put(attributeName, value);
}

public Object getAttributeValue(String attributeName) {
    return attributeValues.get(attributeName);
}
```

Dynamic properties are suitable for aspects that are subject to frequent change. For example, new product generations often have additional properties for the insured object or insured person that enable the insurance company to better assess the pertaining risk. Other aspects of the model, however, remain unchanged for a longer period of time. For example, insurance contracts always have an inception date, an expiry date or maturity, a payment mode etc. For these „constant“ properties, no flexibility is required on the interface. In these cases, you should avoid the dynamic property pattern because it would add an unnecessary level of complexity to the integration of the calling systems without offering any advantage. Hence, an interface class should map the „constant“ aspects to explicit methods and at the same time support the dynamic property pattern. The following source code shows how this can be done for the above mentioned example class InsuredPersonDto.

```
public class InsuredPersonDto {

    public final static String ENTRY_AGE = "entryAge";
    public final static String GENDER = "gender";

    private Map<String, Object> attributeValues;

    public void setAtributeValue(String attributeName, Object value) {
         attributeValues.put(attributeName, value);
    }

    public Object getAttributeValue(String attributeName) {
        return attributeValues.get(attributeName);
    }

    public Integer getEntryAge() {
        return (Integer)attributeValues.get(ENTRY_AGE);
    }

    public void setEntryAge(Integer entryAge) {
        attributeValues.put(ENTRY_AGE, entryAge);
    }

    public String getGender() {
        return (String)attributeValues.get(GENDER);
    }

    public void setEntryAge(String gender) {
        attributeValues.put(GENDER, gender);
    }

}
```

Obviously, more flexibility in the introduction of new products by use of dynamic properties can only be achieved if the operational systems themselves offer an appropriate level of flexibility. Hence, the attribute names must not be hard-coded into these systems but rather be retrieved at runtime. The product server has to provide suitable methods that allow clients to get this kind of information. In this respect, it has to be taken into account that under certain circumstances attributes are only used in a specific product generation. So the signature for such a method could look as follows:

```
public String[] getDynamicAttributesForLiabilityContract(String productGenerationId);
```

When designing an interface you should keep in mind that a flexible interface to the product system is not the only thing that is necessary to ensure product flexibility; the entire IT landscape of the insurance company has to be considered. Ultimately, the operational systems must provide the flexibility the company is striving for, from the user interface down to the datatbase and printing functions. Development efforts to design extremely generic systems often fail because the generic user interfaces aren't user friendly and the batch processing performance isn't sufficient. A good compromise between product flexibility, user friendliness, and performance can be achieved by making the operational system flexible in respect to the introduction of new attributes for certain business classes, such as the insured person etc., while defining the overall structure, i.e. the classes and their relationships, at compile time.

Considering Product Changes over Time

Insurance companies change their products periodically. These changes fall into two categories: They can either apply to new policies only or to new and existing policies alike, as for example premium adjustments do. Unfortunately, there are no standardized terms for these two modification types. In this paper, we will use the term „generation“ for changes applying only to new policies, whereas the term „adjustment level“ will be used for changes that equally apply to existing policies[2].

Product Generations

We will first focus on generations and then cover the adjustment levels. A new generation will usually be introduced on a specific effective date. This means that from this date, all contracts for the new product will be concluded based on the new generation.

Example
A new liability insurance generation is launched on 1.1.2010. New liability insurance contracts with an inception date of 1.1.2010 or later are proposed (and concluded) based on this new generation. Insurance contracts with an inception date before 1.1.2010 are concluded based on the terms and conditions of the old generation.

As policy administration systems manage contracts of different product generations, they are able to pass the ID of the applicable product generation to the product server's interface with relative ease. By contrast it is not advisable to determine the product generation from the inception date at this point, because several different product generations might be on the market at any one time. When creating new contracts, the product generation is either provided by the sales systems or entered by the back office staff. In the latter case, of course, you can as well use a product service that returns all generations that are sellable at a given inception date.

To do this, the policy administration system obviously has to store the generation ID in its database. This imposes no difficulty, for the ID – as its name suggests – identifies the generation. (Of course, a generation's ID that has once been used must remain unchanged within the product system. After all, it's an ID, as previously stated.)

Sales systems, by contrast, are often built so that they can work with only one product generation. This means that the launch date of the new product generation in a sales system will be determined by the release change to a new version of the software. Sales systems often limit the inception dates of new proposals to the 1st (and sometimes the 15th) day of a month. This way, the new software version can easily be put into operation the month before the launch of the new product generation. In our previous example, this would correspond to December 2009. All proposals generated by the new software version will automatically stipulate an inception date later than 1.1.2010 and must use the new generation. Proposals whose inception date would fall into the scope of the old generation can no longer be created, provided that in this case the software doesn't allow proposals with an inception date that lies in the past.

As the product server supports all product generations, the applicable generation will have to be determined upon calling by the sales system. This can be done in several different ways.

2 The Association of German Insurers uses the term „version“ for changes concerning only new business and „Generation“ for changes that apply to new and existing contracts. A detailed description can be found in the Faktor-IPS Tutorial.

Determining the generation from the inception date

A product generation has an earliest possible inception date within the product server (1.1.2010 in the example ababove), so the current generation for a given inception date can be determined. But this solution doesn't hold when the product server is installed independently from the sales system. The reason for this can be derived from the following scenarios:

- Scenario A
  The new liability product generation in our example requires us to enter a new property. The product server is installed on a central server on 15.12.2009 but the sales system is not installed until 20.12.2009. In the meantime, no proposals with an inception date that lies in the year 2010 can be created, because the sales software doesn't yet prompt for the new property whereas the product server already stipulates the new generation as valid. :-(
- Scenario B
  The structure of the product generation remains unchanged; only the premium changes. In this case there would be no problem as regards software technology. On the business side, however, the launch of the new product generation in the sales system would now be dependent on the installation of the new product server version.

Using the inception date to determine the generation and synchronize the installation

In addition to the procedure described above, a scenario might be conceived where the product server and the sales system are always installed at the same time. In theory, this method may hold in such a way that always the correct generation is used, however, in practise it will not work because it implies an unnecessary coupling of systems that would impair operation. The level of complexity would increase with the number of systems that use the product server.

This method can only be applied if the product sever isn't installed on central servers but is, for example, provided as part of an offline application.

Using inception date and application date

Instead of synchronizing the installation, the current product generation could be determined by the inception date and application date of the insurance. To do this, the product server would have to maintain not just an inception date but also a launch date for each generation.In our example we would define that the new liability insurance generation would have an earliest inception date of 1.1.2010 and can be sold starting from 20.12.2009. Then, if the sales system computed a proposal with an inception date of 1.1.2010 and an application date of 19.12.2009, the product server would choose the old generation, whereas with an application date of 20.12.2009 it would choose the new generation.

The downside to this solution is the fact that the launch date depends on the respective sales system. A website release upgrade usually occurs independently from a release upgrade of the sales system. As a result, the product server would have to maintain a launch date for each system that uses it. For each system, this launch date would have to be synchronized with the installation time, as these systems support only one generation. If the installation has to be deferred, the launch date within the product server would have to be adjusted – an awkward dependency.

Determinig the generation through the calling system

In this scenario the calling system passes the applicable generation's ID to the product server. Thereby the launch date of the product and the release update of the sales software are synchronized. However, each sales system needs to know the IDs of the current product generations. Moreover, when switching to a new product generation, the ID of the applicable generation has to be adjusted in the sales software. But as the introduction of a new product generation is always associated with changes in the software and an additional system test, there is little disadvantage to this solution.

Recommendation for the Construction of an Interface to the Product Server

Rather than having the product server manage a launch date for each system, you should explicitly pass the ID of the applicable generation to the product server. The sales software can also use the current generation pertaining to the inception date by setting this ID to NULL. It is up to the developers of the sales software to decide which method will be used.

Adjustment Levels

As a product adjustment affects all existing contracts, the structure of the model usually doesn't change. For example, no new rate factors will be introduced, as those values are not defined for the old contracts. As a result, operational systems can generally support new adjustment levels without any modifications of the software being necessary. For this reason, the product server will determine the appropriate adjustment level according to an effective date (inception date, last premium due date). This way the product modifications come into force when the new product server version is installed. Contracts that exist in policy administration systems will usually be adjusted by a batch job.

Summary

In this paper we have discussed the basic construction principles of product server interfaces. Specifically, we have explained how product information is implicitly contained in (flat) interfaces. For this reason, product modifications often call for modifications in the interface as well. Product flexibility can be achieved by using interfaces that are structurally similar to the business model and support the introduction of new rate factors through „Dynamic Properties“. The challenge is to provide this product flexibility not only in the product server but to support it within the operational systems as well.

References

[Fowler, Properties]	Fowler, Martin. Dealing With Properties, 1997
[Fowler, PoEAA]	Fowler, Martin. Pattern of Enterprise Application Architecture, Addison Wesley 2003
[RtoP]	Kerievsky, Joshua. Refactoring to Patterns, Addision-Wesley 2004
[GoF]	Gamma et al. Design Patterns - Elements of Reusable Object Oriented Software. Addision-Wesley 1995.