Some Patterns for Insurance Systems

Wolfgang Keller
c/o EA Generali, Neusetzgasse 1, A1100 Wien, Austria
Email: 100655.566@compuserve.com
http://www.objectarchitects.de/

Abstract

Building flexible insurance systems that allow short product release cycles is a challenge for many insurance firms. Despite the huge market and the many projects that do near identical things these projects are still a little bit like black magic. There's virtually no literature around on the topic. This paper provides a collection of patterns that help explain the basic forces and solutions for the design of product driven insurance systems.

Introduction

With the arrival of deregulation and more competition in the insurance market, shorter product innovation cycles may give you a leading edge in the very competitive financial services market. Service providers try to expand to new product fields, allowing cross sales. They try to offer individual products in order to serve niche markets and in order to serve their existing customers better. At the same time the need to reduce costs calls for flexible systems that allow achieving the above goals with acceptable maintenance cost.

Systems used in the financial industry to achieve this set of goals show great similarities. The so called product driven approach can be found in the banking as well as in the insurance industry. This approach is borrowed from traditional manufacturing industries [Sch+96]. The significant difference is that you produce your article (an insurance) the very moment you have entered the proposal into your system. Financial products are intangibles.

Analogies in Other Industries

This paper provides a collection of patterns for flexible, product driven insurance systems that can be found in many systems in the insurance and financial industry. It is very likely that many of the patterns apply identically in other industries - but as the patterns here were mined in insurance projects we will not speculate how they can be applied elsewhere without practical project knowledge in those other domains. If somebody writes up similar patterns from other industries we will be able to compare and match them.

The core pattern Product Tree - the idea to represent products as a tree structure - stems from the manufacturing industry. It has only recently moved to the insurance industry (some 10 years - see [Sch+96]). The idea to configure individual products the way the customer wants them using a set of predefined building blocks can also be found in the manufacturing and other industries. There

are several approaches called mass customization that heavily rely on information technology to deliver mass customized products [And+97]. Popular examples can be found in the computer industry - personal computers made to order e.g. by DELL or VOBIS and can also be found in the clothing industry (Levy Strauss and others).

Individual immaterial products can also be found in the telecommunication industry. Billing plans can also be configured from predefined building blocks using technologies very similar to product tree and product server[Joh98b].

This paper is not a full pattern language. There are many ways to model a product [Sch+96], where we only present one of them (Object Event Indemnity).

Existing Reference Models for the Insurance Industry

Many readers who are interested in the field might know IAA - IBM's Insurance Application Architecture - and might wonder how the material presented in this paper is related to it. As IAA is proprietary material with a solid price tag, and as I don't have access to it - except secondary sources - I cannot discuss the relation of the material presented here to IAA in any detail. Sorry about that, but not my fault. So we can only refer to VAA [VAA95], a public domain reference model by the German insurance industry, wherever that is possible.

General Forces Driving Insurance Application Design

The set of forces that drives the patterns to be presented is mostly identical. They are:

Time to Market and Flexibility: In Europe the insurance market has been heavily regulated by government agencies until recent times. With the advent of deregulation, global markets and a European market, insurance companies find themselves in a new dimension of competition: New competitors include banks and other providers of financial services, insurance companies from neighboring countries and many others. In fierce competition *time to market* plays a dominant role. Companies that are able to design and market more products faster and are also able to offer individual products have a competitive advantage over companies that need years to bring a new product to the marketplace.

New and individual products: Michael Porter distinguishes between several competitive strategies [Por85]. You can become a cost leader, a quality leader or a niche player. If you want to open new market segments, or if you want to become a leader in customer perceived quality, it is helpful to be able to have many individual products for many customer groups. In times of early industrialization this has been a contradicting goal to cost effectiveness. With the advent of computers, individual products at low costs become feasible. You need to combine elementary standardized building blocks to receive individual products. Companies who manage to provide individual products have a serious competitive advantage.

Point of Service and Cost Cutting: There are several ways to react to more competition. One way is to provide service closer to the customer. Many insurance systems today are still pure back

office systems with a lot of paper flowing between several layers of regional offices and the sales representative. Cost cutting strategies aim at this paper flow and also aim at reducing process steps. Insurance systems are deployed closer to the customer, calling for Internet and offline PC architectures and the sales representatives' laptop becomes closely integrated with the central transaction system.

Development Cost: The insurance business provides pure immaterial services – no hardware – no product you can touch. An insurance company that can slash data processing costs is like a manufacturing company that reduces its labor and material costs. The result is a competitive advantage. Therefore total relative development costs for insurance systems need to be lowered while providing more functionality and flexibility at the same time.

Traditional Structure of an Insurance Company versus Product Bundles and Development Costs: So far we did only talk about the new world of insurance systems. Insurance companies are in business for quite some time and belong to the early adopters of data processing technology. Legacy systems and organizations in insurance companies are typically built around product categories. You will find for example separate life insurance, health insurance or property insurance systems. This reflects the organization and the distribution of know-how within the companies. You will typically find a health insurance department, a property insurance department and so on. The resulting systems tend to having a lot of redundant functionality. For example the claims processing systems for property insurance and auto insurance have large parts in common. Health insurance and life insurance share the same insured object: The human body. Developing systems by product category will result in higher than necessary development costs. On the other hand it may become very hard to develop systems that work for all product categories at a time if most domain experts grew up in the old world and have mostly good knowledge of a few product categories but seldom good knowledge of all product categories. Often there is also political disturbances between departments who are not interested in collaborating to construct systems for all product categories at a time.

The Patterns

All the patterns to be presented have strong relations and can be grouped into chapters representing the design field in which you will apply them.

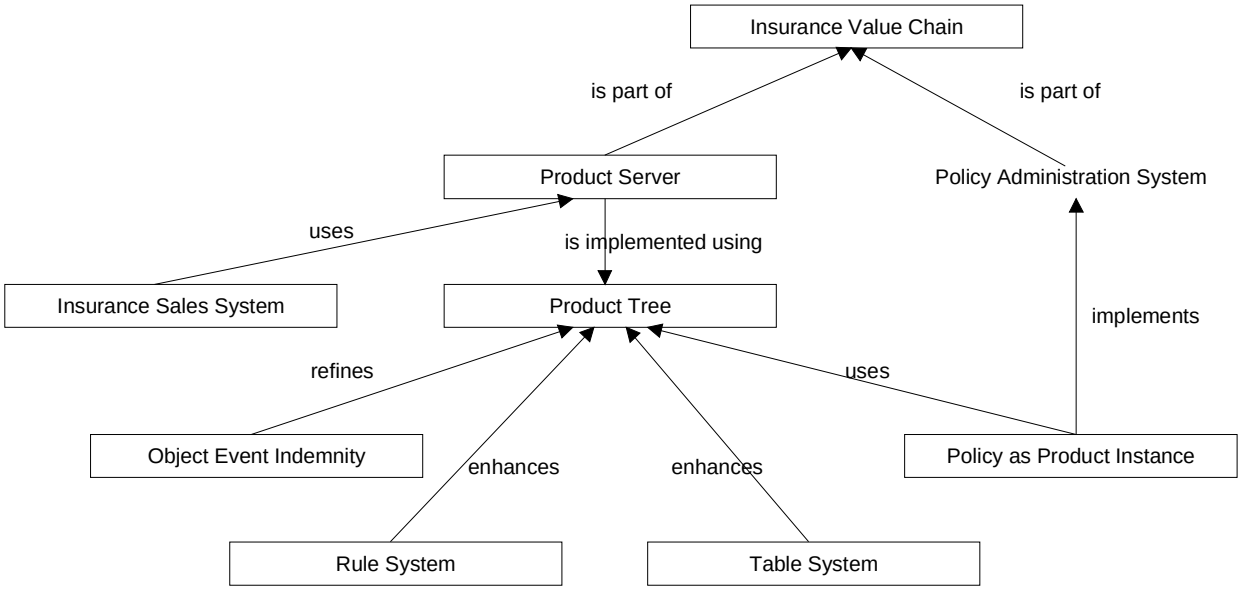


Figure 1: Roadmap of the Pattern Collection

The patterns seen here are a random subset of the set of all patterns that may possibly be mined in the insurance domain. Nevertheless they are easier to understand if they are grouped into some fields of design decisions. Therefore we have arranged them in the following chapters:

***Top Level Structure of an Insurance System*:** Deals with the top level subsystems that you will find in most modern insurance systems. The patterns in this chapter are: Insurance Value Chain, Product Server and Rule System as a typical subsystem of a Product Server.

***Insurance Product Models*:** Contains a few patterns that are useful if you want to model insurance products for use in a product server. The patterns presented in this chapter are: Product Tree and Object Event Indemnity.

Policies: This chapter deals with how to implement insurance policies. The only pattern in this chapter so far is Policy as Product Instance.

Distributing Insurance Systems: Some subsystems that you will find in a typical insurance system would not be necessary if we had a virtual supercomputer at the hands of everyone in the insurance organization with zero time for data access and unlimited wide area network bandwidth at no cost. Unfortunately such systems are yet to be developed. Until then we will find patterns that deal with distribution like the Insurance Sales System or a Table System.

Top Level Structure of an Insurance System

In this chapter we will present the top level subsystems that you will find in most modern insurance systems according to the Insurance Value Chain. A very important part of a product driven insurance system is the so called Product Server. A product server typically has a system called Rule System that deals with product rules.

Pattern: Insurance Value Chain

Example

You are assigned the task of proposing a structure for a new insurance back office system. What you find is a bunch of existing systems for each product category like life insurance, property insurance, auto insurance. You find that there's a lot of redundant functionality across those existing systems, especially in claims processing and handling of policies. You also find that adding new products is a tedious task that is performed at a rate of two new products per year and product category. This comes from the fact that product knowledge is cluttered across the system that manages policies and the system part that manages claims processing. Some more product knowledge can be found in other system parts.

Problem

What is a good domain architecture for the mass of business functionality and business objects that occurs in an insurance back office system?

Forces

The system should be structured in a way that it is able to support your competitive strategy. We have already discussed the market forces at work.

Additional forces come from good software design like clear modularity that goes with good maintainability and management of the complexity of large insurance systems that have a typical size of far more than 10,000 function points.

Solution

Structure the domain architecture according to the value chain of an insurance. That is *Product - Policies – Claims Processing - Sales & Marketing - Customer Service* plus helper systems like a partner subsystem, a subsystem for insured objects and a subsystem that handles in- and outgoing payments.

Structure

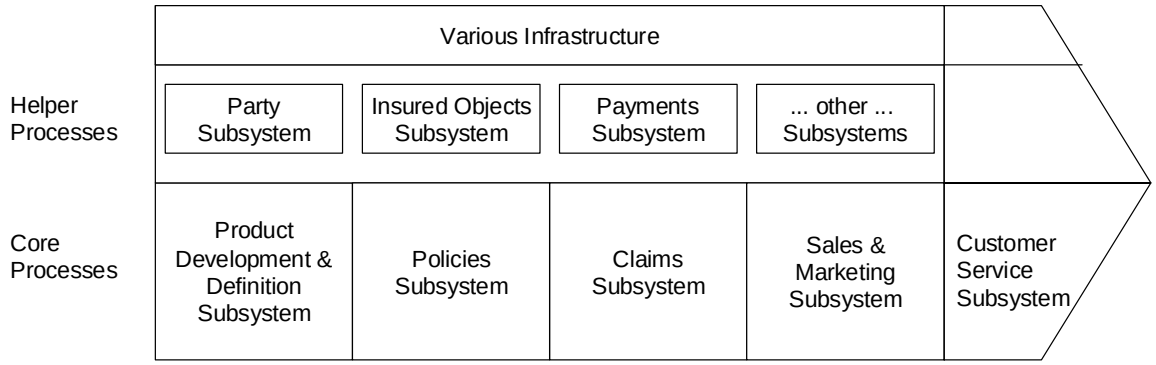


Figure 2: Structure of an Insurance System following the Insurance Value Chain. Adapted to the Insurance Industry analogous to Porter's Value Chain Models [Por85]

The value chain is modeled by the subsystems:

- *Product Development & Definition Subsystem*: Is responsible for supplying the other system parts with product definitions that are interpreted by the other parts of the system. Before these can be used by other subsystems they need to be developed and defined.

- *Policies Subsystem*: Is responsible to store the policies and support all use cases that perform business actions on them.

- *Claims Subsystem*: Concentrates claims processing for all product categories. Main business objects are events, claims, involved parties and more.

- *Sales & marketing Subsystem*: Deals with how to sell the products to the customer.

- *Customer Service Subsystem*: Deals with additional services that you might want to offer your customers besides what you are obliged to support in the claims system anyway.

To properly model all business functionality you need a few helper subsystems like:

- *Partner (Party) Subsystem:* that encapsulates knowledge about all parties the company deals with.

- *Insured Objects Subsystem:* contains information about insured or damaged objects.

- *Payments Subsystem:* handles in- and outgoing cash flows.

You will typically find a few more subsystems that are not specific to the insurance business like general bookkeeping, management information systems and data warehouses, human resources systems and the like. We will not treat these any further as they are no way specific for the insurance business.

Consequences

Using the insurance value chain model does not necessarily imply that you will end up with a flexible system and short time to market. You need to use a few more patterns like product server and policy as product instance to lay the ground for product flexibility.

The pattern enables you to build one system for all product categories but mostly you will start with a few product categories and expand the system in order to migrate from your legacy systems. Doing this you have the chance to cut development costs by avoiding redundant functionality.

The structure is a structure for a back office system. Alone it does not affect your presence at the point of service.

This structure alone also does not imply that you diminish layers of communication and cut costs. To do this you need to start a business process reengineering effort and use a workflow system.

Implementation

Insurance Value Chain is a domain architecture pattern. To implement a system you need an application architecture. Most systems in the domain today are built using a Three Layer Architecture [Ren+97]. A workflow system is also used pretty often. You will typically not be able to implement the system as a whole at a single site or as a virtual single system. Today's performance and bandwidth considerations for distributed systems will force you to spilt your system up into a back office System and an Insurance Sales System.

Variants

You will often find differences in the way the product and the policy subsystems work together. The product server approach to be discussed, decouples the two while other frameworks like the UDP framework [Joh+98] do not provide a hard subsystem border between the two subsystems but see policy as product instance and closely couple the two systems.

Related Patterns

Typically, the product subsystem will be implemented as a product server.

A policy subsystem will be implemented using a framework like the User Defined Products Framework[Joh+98], making heavy use of the composite [GOF95] and type object patterns [Joh+98b]. You will typically treat a policy as product instance.

Known Uses

The above architecture is described in more detail in several domain architectures for the insurance industry like the Phoenix Architecture [Ald+98], IAA - IBM's Insurance Application Architecture or also VAA [VAA95]

Further Reading

Literature on insurance systems is somewhat scarce despite the huge market – but other markets are even bigger. For the basic ideas behind the value chain based architecture and the analogies to production of material goods see "Innovative Gestaltung von Versicherungsprodukten" [Sch+96]. The above known uses are mostly large documents describing very similar domain architectures. IAA from IBM is also a large domain architecture but is a licensed product that is not accessible to the public.

Pattern: Product Server

Example

Imagine you want to build a product driven insurance system. Typically you have a back office system that processes proposals, manages policies and handles claims[1]. On the other side you need the same product definitions for your sales PCs and maybe also for other systems like internet offerings or systems that support insurance brokers.

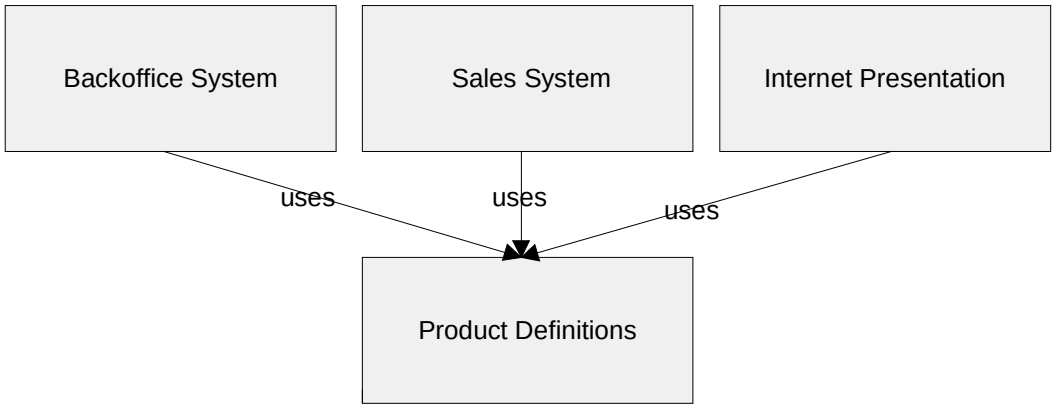


Figure 3: Various Systems need to use identical product definitions

Problem

Where do you define your product knowledge and how do you distribute it?

[1] In the US claims processing would be outsourced in most cases – in Europe claims processing is mostly done by the insurance itself

Forces

Besides the forces mentioned above there are a few more forces to be taken into account here:

Platform independence: Back office systems typically run on OS/390 machines. Sales PCs typically run Win95 or other PC operating systems. Internet Clients may run in Java or may run as ultra thin HTML clients with a server written in some language. Your product definitions must be accessible on different platforms.

Interface design: Broad interfaces tend to result in bad maintainability and monolithic software structures. Narrow interfaces are better if you want a subsystem for a large number of clients on different platforms.

Encapsulation of product knowledge versus optimal user interface design: On the one hand you want to encapsulate all of your product knowledge. The advantage of this would be that for example all dialogs only interpret a product definition and are able to automatically adapt themselves to new product definitions. The downside of this approach is, that "automatic" user interface are seldom beautiful. The approach might work for back office dialogs. It is very unlikely it will work for sales dialogs or multimedia product presentations on a sales laptop.

Different Requirements: Even though back office systems as well as sales systems need to use the same product definitions, their views on the product definitions may slightly differ. For the back office system that stores all policies you need old as well as actual product definitions. For your sales system you only need the product program that you actually sell. A product definition systems knows more products – those that are still being developed and are not in the sale program yet. You need to provide different views for these systems.

Solution

Encapsulate your product knowledge in a product server. At runtime encapsulate it in a portable product runtime system and provide views using facades.

Structure

In a typical product server you will find the following components:

- *Product Editor:* Provides an easy to use interface for the person who defines products. The product definitions may be persisted in a product model.

- *Persistent Product Definitions*: are a set of persistent business objects representing your products. How this is best organized will be discussed in Product Tree and Object Event Indemnity.

These first two components may be organized in an ordinary object-oriented Three Layer Architecture.

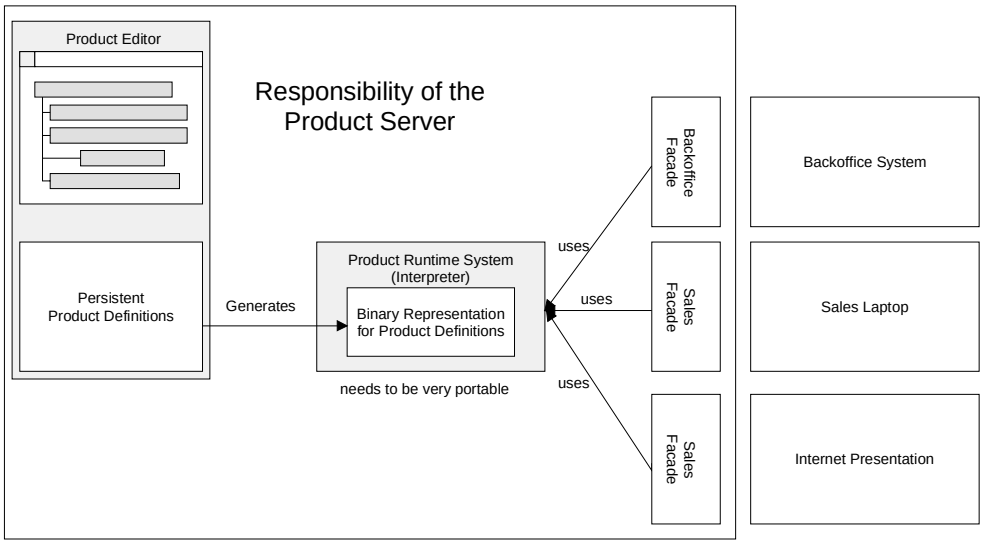


Figure 4: Structure of a Product Server

The business object representation of the persistent product definitions will seldom ever be portable. Therefore we need more components

- *Generator:* responsible to produce some portable byte code from the persistent product definitions.

- *Product Runtime System*: interprets this byte code and is able to run on OS/390 as well as windows platforms. Some systems use ANSI C for the Product Runtime System to ensure this portability.

As you want to provide a very narrow interface on your product runtime system and as you may not want to deal with too primitive operations of such a narrow interface you may implement

- *Facades*: that provide different views on your product definitions.

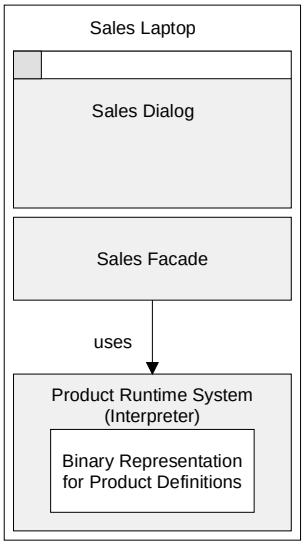


Figure 5: Sales Laptop

The fact that the so called product server contains all of thes components does not indicate that all of these components wi necessarily run on a single machine or that the product server is server in the sense of a client/server system.

To get the idea have a look at Figure 5: The configuration for a sale system. The sales dialog uses the product runtime system via facade. The rest of the so called product server, the product editor, the product business objects, the generator may run on differer machines under totally different operating systems and technic infrastructures.

Consequences

An insurance system design using a product server provides product flexibility and platform independence. Compared to approaches using an active object model only, you can use the same product definitions with different views to drive your sales system, your internet presentations and the like.

The downside is the generation process. Systems using a generator have some downsides compared to systems using active object models and reflection [Bus+96]. On the other hand the downsides are partly compensated by the fact that you usually do not want your changes to the product model to be visible everywhere at the same moment you enter them in a product editor. Insurance Products, like code need to be tested, released and versioned. A system using a single active object model and no separated spaces for product development, product test and productive product can easily bring you into trouble.

The other consequences are:

- *Platform independence:* is achieved by using a portable product runtime system.
- *Good Interface design:* may be achieved by defining a very narrow interface for the product runtime system.
- *The Optimal User Interface Design:* will contain some product know how of it's own – containing this tendency is a nontrivial design issue.
- *Different Requirements*: can be incorporated using different facades on the product runtime system.

Implementation

You can implement the product editor as a conventional object oriented application using an active object model in a Three Layer Architecture.

Related Patterns

To implement the product editor you should use product trees organized by the object event indemnity pattern. You will also incorporate a table system and a rule system. The runtime system can be implemented using pattern for virtual machines [Jac+96]. Facades [GOF95] may be used to provide different views on product definitions.

To implement the product business model for the product editor use active object models and reflection.

Known Uses

The basic idea to have a product server can be found in many insurance system. Parts of the approach can be found in VP/MS, a product definition system by CAF [CAF97]. Two projects by EA Generali also contain product server ideas: KPS/S an ongoing project for property insurance and Phoenix, a nearly finished project for life insurance [Ald+98].

Pattern: Rule System

Example

Once you have built a product editor that allows you to define product structures you need a mechanism to derive attributes of tree elements from other attributes for example in order to rate a policy.

Problem

How do you code the functionality that you need to perform plausibility checks or policy rating on product trees?

Forces

Usability by application administrators: If you want your application administrators to be able to define new products without coding, you need to define a scripting language that allows linking of product attributes, performing computations and doing table lookups. Once you start this, you easily end up with a complete new programming language. For a deeper discussion see also the CustomizationViaProgramming anti pattern on WikiWiki-Web (http://c2.com). This discussion is a variant of the discussion on *Hard coding versus scripting languages*: Often it is easier to hardcode something (especially in Smalltalk) than trying to build a second Smalltalk on top of Smalltalk.

Cost: Developing a rule system easily becomes an expensive adventure. It can often be cheaper to educate a few application administrators in Smalltalk than developing a custom programming language for them. On the other hand we know examples of an Excel like approach (formula interpreter) at reasonable cost that is well accepted with application administrators.

Completeness of derivation mechanism: If you build a rule system it should be suited to perform all attribute derivations you need for product definition. Otherwise you have to program anyway.

Flexibility: For compiled languages like C++ a Excel Sheet like approach is more flexible for application administrators than an edit/compile/link cycle.

Testing and Debugging Products: If you use a code attribute derivation mechanism in you product definitions that behaves like a scripting language you will need additional tools like in any language environment: A custom debugger and

Solution

Build an attribute derivation mechanism similar to semantic functions in a parse tree [Aho+86] or cell calculations in an excel sheet. Have this interpreted at runtime.

Structure

See Figure 6 : Insurance Product as Tree: Like in Compiler Building you can program your semantic functions in a programming language (the Lex/Yacc approach – see Aho+86) or you can also define a closed tool set with a separately invented scripting language.

Note that this does not have too much to do with Rule Based Systems like the one's you find to do risk assessment e.g. for life insurance. There you often find expert systems (for example implemented in Prolog). The Rule System we talk about here implements business rule but on a pretty deterministic and predictable basis.

Consequences

The consequences are as manifold as the implementation variants and depend on how serious you take the discussion in the forces section. The issues involved are completely analogous to programming language design. Here are some examples of yogurt that may happen and actually happened in systems we know:

Incomplete rule definition language: In this case you will end up with coding parts of the attribute derivations and defining some, resulting in a system that is too expensive compared to the flexibility and customizability it offers.

Overcomplete rule definition language: Such a language is complex, and might be error prone and expensive. In this case you may as well use an interpreted programming language instead of a so called rule system.

Related Patterns

Something like the pattern is needed to implement Product Trees. Interpreter[GOF95] or Virtual Machine[Jac+96] is often used to implement a rule system.

Known Uses

The UDP paper sketches how to implement a rule system in Smalltalk [Joh+98]. VP/MS uses a version that looks more like a spreadsheet [CAF97]. Phoenix uses a mixture of hard coded product logic and rules [Ald+98].

Insurance Product Models

This chapter contains a few patterns that are useful if you want to model insurance products for use in a product server. The patterns presented in this chapter are: Product Tree and Object Event Indemnity. There are far more ways to model products than just Object Event Indemnity. See [Sch+96] for a few ideas. Maybe we'll mine a few more in future work.

Pattern: Product Tree

Example

You have followed the insurance value chain approach and decided to build a product server. Before designing the product server you need an idea, how to model insurance products.

Problem

What is a good representation for insurance products?

Forces

Besides the general forces that drive insurance application design the following should be taken into account:

User involvement vs. flexibility and generality: You need a product model your users will understand. On the other hand the industry has seen very flexible approaches based on abstract data models that are very hard to understand[2].

[2] If you're in the business you know what I mean and you also know why I dont name it ☺

Reuse and flexibility: A good approach would allow for reuse at a building block level. What would be desirable are libraries of product building blocks that can be reused and recombined to form new products like in the manufacturing industry.

Solution

Model your insurance products as a tree like you would model conventional products like bicycles.

Structure

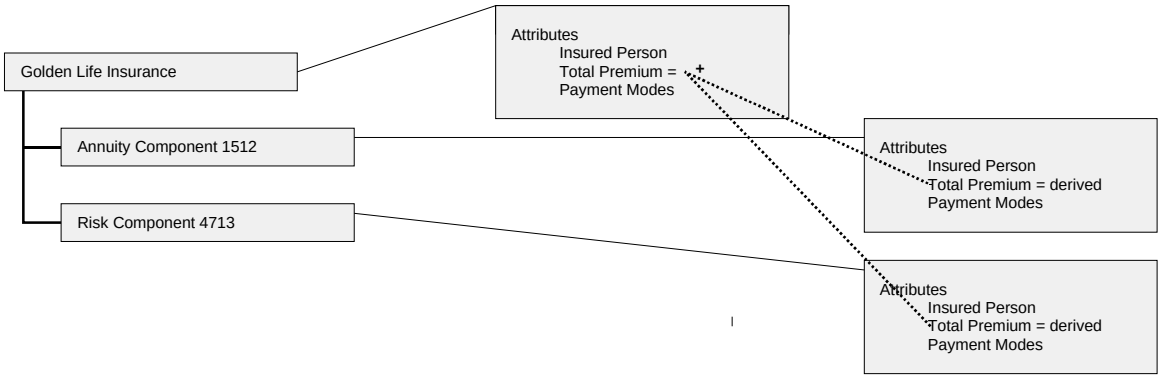


Figure 6 : Insurance Product as Tree

- You model products and their components as nodes of a tree.
- You add *attributes* to each node describing it.
- You add functionality like the calculation of a premium as derivation rules, analogous to the semantic functions in a compilers parse tree [Aho+86] or you can also write your own rule system (see also Ralph Johnson's UDP paper [Joh+98]).

Consequences

Product Flexibility: Once you have identified the elementary building blocks of insurance products, designing and modeling new products becomes much easier and straightforward.

User involvement: Practical work with product servers shows that domain experts with some additional education are able to model products as trees.

Reuse: Once you have defined your building blocks you can reuse them all over in new and existing products.

Implementation

To seriously implement the scheme you need quite a few more design considerations. Modeling products as trees does not say anything on how to best structure these trees and on how to best design product models. See object event indemnity for a few hints.

You will typically implement the tree using composite and the type object pattern to model an insurance policy as product instance . Combining this with derived attributes leads to the design of a whole framework as described by Ralph Johnson and Jeff Oakes in their ongoing work [Joh+98].

There are two very fundamental variants on how to design the runtime system for product trees

- The first variant assumes that the product editor works on the same database and models as the active implementation in the productive system. This may lead to an active object model as described by Ralph Johnson and Jeff Oakes [Joh+98]

- The second variant assumes that the development space and the production space are strictly separated. This is more the product server approach.

Variants

Often building blocks share identical attributes. If you take for example two elementary products: An annuity component and a risk life insurance you will find an insured person as an attribute (related object) in both cases. The combination of the two in a product bundle calls for identifying the two insured person attributes. This may lead to a DAG representation (directed acyclic graph) instead of a tree representation.

Related Patterns

The pattern is the key to implementing systems according to the insurance value chain. The pattern you need to implement the scheme have already been discussed in the Implementation Section above.

Know Uses

The approach is quite common in the insurance industry by now. The approach can be found in VP/MS, a product definition system by CAF [CAF97]. Two projects by EA Generali also contain product server ideas: KPS/S an ongoing project for property insurance and Phoenix, a nearly finished project for life insurance [Ald+98]. We know that other insurance companies work on similar approaches or already use them.

Further Reading

The idea to build insurance systems analogous to discrete parts manufacturing systems has been discussed by Paul Schönsleben and Ruth Leuzinger [Sch+96] but on a relational level.

Ralph Johnson and Jeff Oakes [Joh+98] provide a discussion on how to implement such a system in an object oriented style.

Pattern: Object Event Indemnity

Example

Now that you have decided to model your insurance products as trees you start to define products. You do it, and your colleagues do it, and after trying to join two models that were built by different people you notice that the tree paradigm alone does not allow you to arbitrarily recombine building blocks.

Problem

What is a good way to model insurance products, using a tree structure? What are good abstractions to model insurance products?

Forces

Quality and understandability: As with many programming languages, the mere paradigm to implement insurance products as a product tree allows many models that are ill structured, not good to understand and will later on cause a maintenance headache. Similar observations hold for many domains with a very high level and abstract modeling paradigm, like object modeling, entity relationship modeling and the like: You need some rules or domain analysis patterns (see e.g. [Fow97]) to narrow the space of possible solutions and some conventions to ease modeling.

Reuse: also calls for some modeling conventions. The more uniform your insurance product components are modeled, the easier they can be reused in other products.

Solution

Model you insurance products in terms of OEI: Object – Event – Indemnity. Each insurance product insures one or more insured objects. If a certain event (e.g. damage) happens to that object, the client has the right to claim a certain indemnity.

Structure

Your product trees will then look like the following:

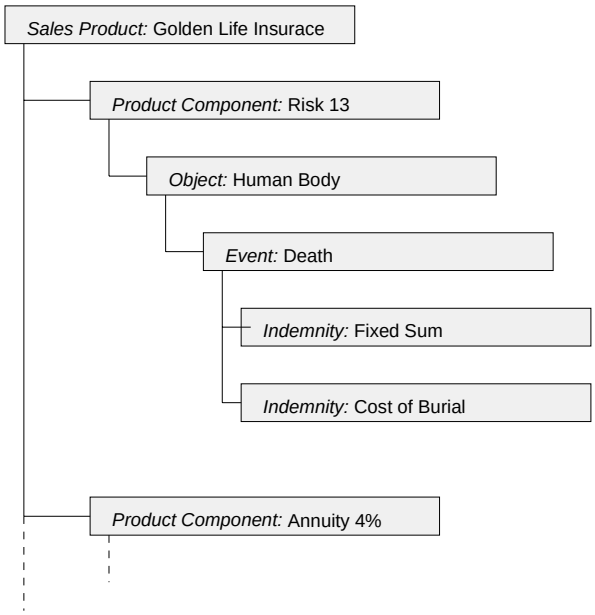


Figure 7: Product Tree Organized using the Object Event Indemnity Pattern

Each term may be used recursively: Objects may contain other objects, Products my contain other products and so on. But the basic idea behind the structure is pretty stable.

Consequences

Productivity and reuse: If everybody adheres to this or a similar modeling pattern you will have a common understanding about product structures. You will have higher productivity and better reuse.

Related Patterns

The pattern refines product tree. But Object Event Indemnity describes only the very top level part of a product structure. There must be further patterns for the definition of attributes, semantic rules, use of table lookups and the like that are still waiting to be mined. To mine them you need access to a set of product models modeled using similar tools by different people or better different companies. As product models are competition critical knowledge, we will wait quite a while for such patterns to be published ☺.

Further Reading

There are no books or articles yet on the subject. But some product providers like CAF offer training sessions and some course materials [CAF97].

Policies

This chapter deals with how to implement insurance policies. The only pattern in this chapter so far is Policy as Product Instance.

Pattern: Policy as Product Instance

Also known as

Mirrored Parts List

Example

You are proud of having just finished your product editor. You have implemented it using composite for the model and some hierarchical list box for the GUI. You have also typed your tree nodes as products, or product components, and object event indemnity plus some more.

Problem

How can you represent an insurance policy?

Forces

General Forces: The strongest forces are the general forces driving insurance application design (see above). The key word is flexibility. The best product server is near to useless as a competitive weapon if you cannot handle the new products in your back office system. Therefore the system that handles policies needs to be as flexible as a product definition tool.

Solution

Make the policies instances of products. Use the type object pattern to do this. The policy then mirrors the tree structured product definitions.

Structure

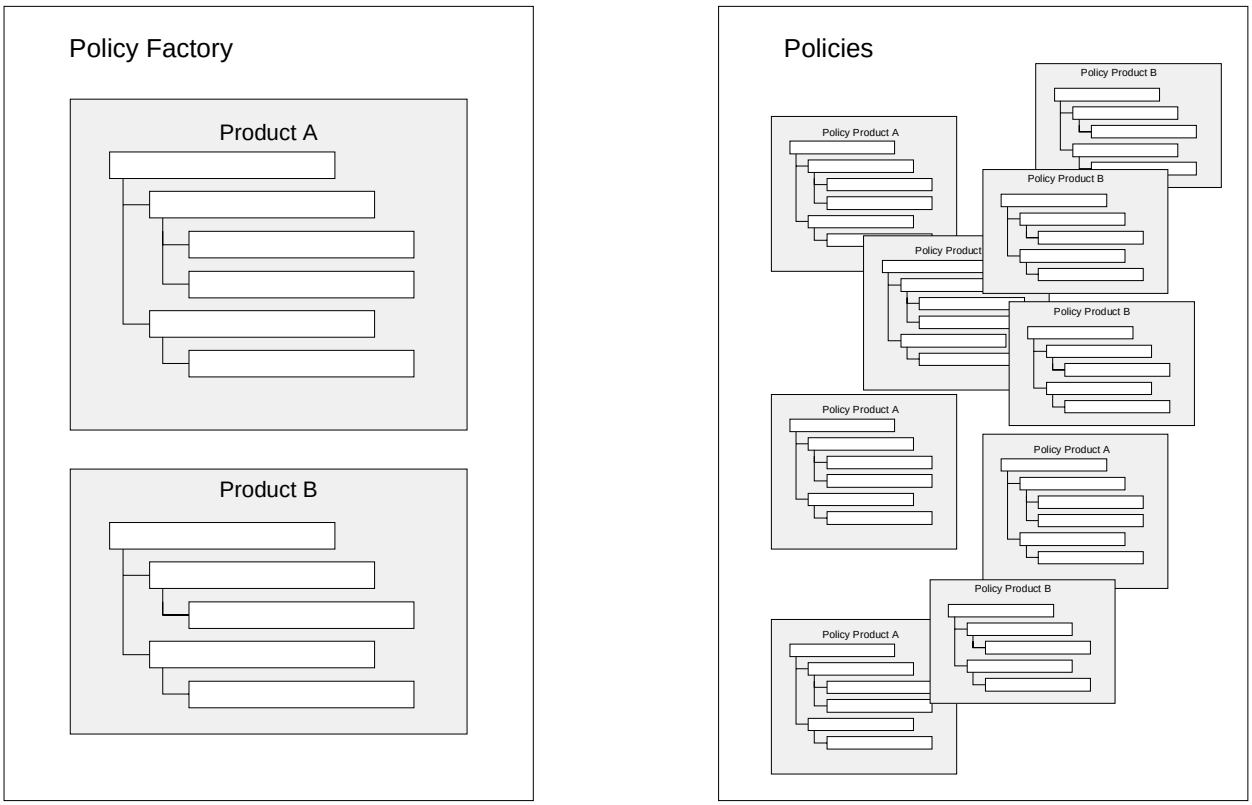


Figure 8: Policies as Instances of Products

Some product instances are kept as prototypes[GOF95] in a policy factory[GOF95]. The prototypes are instantiated with real life data to form a policy.

Consequences

Flexibility: Like most reflective systems (see reflection [Bus+96]) you will have a very flexible system that you can keep in sync with your product definitions.

Portability: You have to take special precautions to keep your system portable. If you use the pattern to implement an active object model on a host computer you might have a hard time implementing the same thing on your sales persons' laptop computers and vice versa.

Performance: As with most reflective systems performance may become poor if you do not keep an eye on it constantly. If you have for example a product tree with a depth of 6 and a naive database mapping you need to retrieve let's say around 100 database records (using joins) until you can work on the policy that mirrors the product tree. To prevent this you need to come up with a clever database mapping [Kel97].

Variants

In practice you can find two main variants of the pattern.

***Single database system*:** In case you implement your product definition system and policy component in one single database you can use the product editor to produce new prototypes in the policy factory. The policy factory in this case is also the data pool for the product definition system.

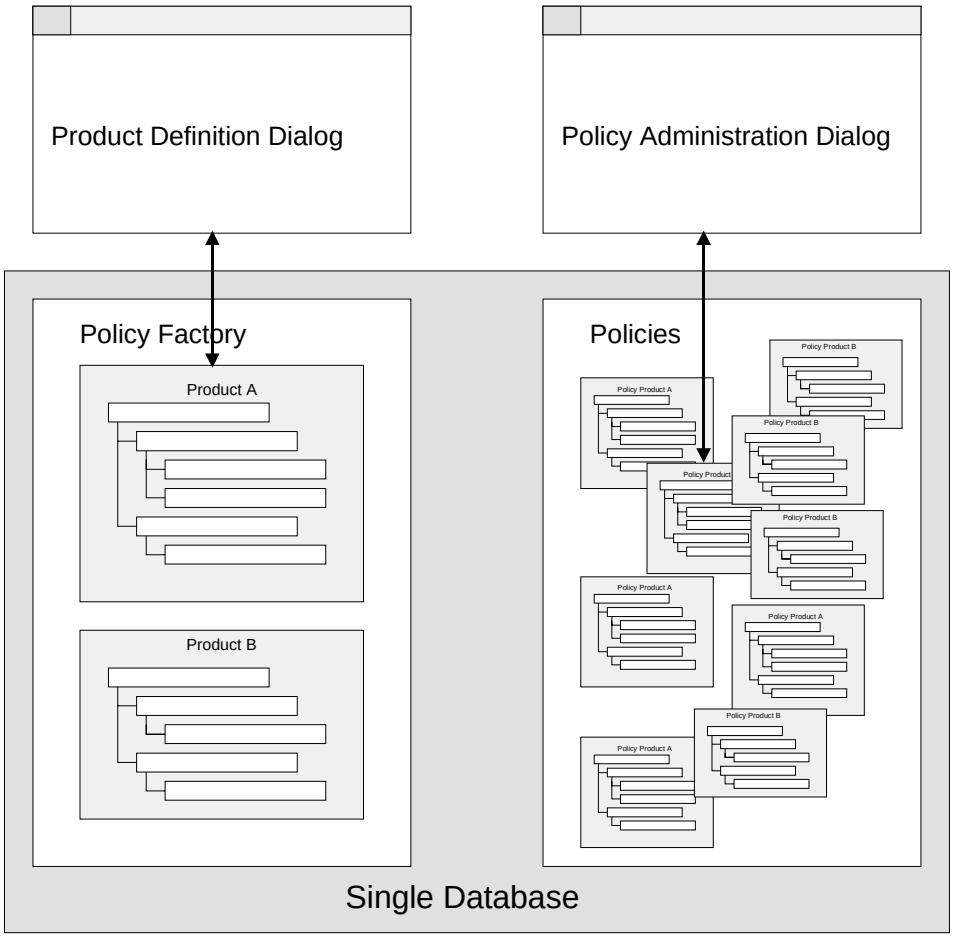


Figure 9: Single Database Product Definition and Policy Administration System

The downsides of this approach are:

- You closely couple spaces for product development and the actual productive products. The advantage is that each product becomes available in the production system the very moment you define it (which is actually a downside as you have to test it and release it).

- In most cases you have no way to use the product definitions single source on a sales laptop.

- If you persist the resulting business objects in a naive way you'll be in big performance trouble.

Decoupled approach: A product server for the central policy administration system and the insurance sales systems: You use the product server approach (see Figure 4) and make the Policy Factory (see Figure 9) a client of the Product Runtime System and remove the product definition dialog from Figure 9.

- *The downsides of this approach:* are it is more stuff to implement, less straightforward and comprises some of redundancy.

- *The advantages*: are the product definition system is decoupled from policy administration. You have more possibilities to tune performance. You can also have other clients, for example insurance sales systems.

Related Patterns

The pattern is a recursive use of composite and type object in a specific context. Another term for this special kind of system is active object model which is a variant of reflection.

The first time you use type object is when you assign types to your tree components [Joh+98] in the product definition system. The second time you use type object is when you take a complex tree instance (a product tree instance) as the (proto)type for an insurance policy (see Figure 8).

You use composite to build your product trees as well as to build your policy trees. A concrete policy tree is a type of a product tree.

Known Uses

PVS, a system by Generali Munich uses the single database approach on a host system. The UDP[Joh+98] does not state explicitly which variant it uses but it looks a lot like the single database variant. The Phoenix Project [Ald+98] at Generali also uses a single database variant.

We have not yet seen the decoupled approach – but will most likely implement it in one of our next projects.

Further Reading

If it comes to the discussion of detailed object oriented design considerations on how to build the product tree, we'll leave the field to Ralph Johnson and Jeff Oakes for the moment, who provide a pattern based in depth design discussion of the subject [Joh+98]. This paper also contains a deeper discussion of how to use interpreter to calculate derived attributes.

Distributing Insurance Systems

Some subsystems that you will find in a typical insurance system would not be necessary if we had a virtual supercomputer at the hands of everyone in the insurance organization with zero time for data access and unlimited wide area network bandwidth at no cost. Unfortunately such systems are yet to be developed. Until then we will find patterns that deal with distribution like the Insurance Sales System or a Table System.

Pattern: Insurance Sales System

Example

Now that you have a flexible back office system you want a laptop front office system for your sales representatives that helps you sell the products you have defined using your product server.

Problem

How do you structure the sales system on the laptops of your sales representatives?

Forces

Besides the general forces above you can observe a few more specific forces here:

System Distribution: The ideal insurance system would be a distributed virtual mainframe with a GUI that provides your sales force with all the available information about your company at a fingertip. This is prohibitively expensive yet as we have differences in performance between memory access, and file or database I/O, and networked I/O. There is no thing such as free unlimited bandwidth yet.

Low cost versus individual market appearance: Today you can buy low cost me too systems that sell standard financial products at low prices. But that's not what you want. You want an individual market appearance, you want non standard products, you want these quick. The section on general forces above has discussed this in more detail.

Flexibility: Short product innovation cycles call for a very flexible system.

Reuse and Single Source: To keep costs contained and to avoid sources of problems like redundancy you want to use a single code base and maximum reuse for both your front office and back office systems.

Innovative Aspects: You will not attract your customer by making "me too" offers. You need a place to plug in innovative parts into your application.

Solution

First of all put those things on the laptop that you need to sell your products or provide service. The more you can afford the better. Then divide the functionality into a common shell that's not specific to any insurance company, a partner (party) subsystem, a sales record, a contact subsystem, and the core sales part that converts proposals into policies. Add a multimedia sales part at your liking. Link the system via an offline connection to your central processing unit in order to process proposals and exchange customer data.

Structure

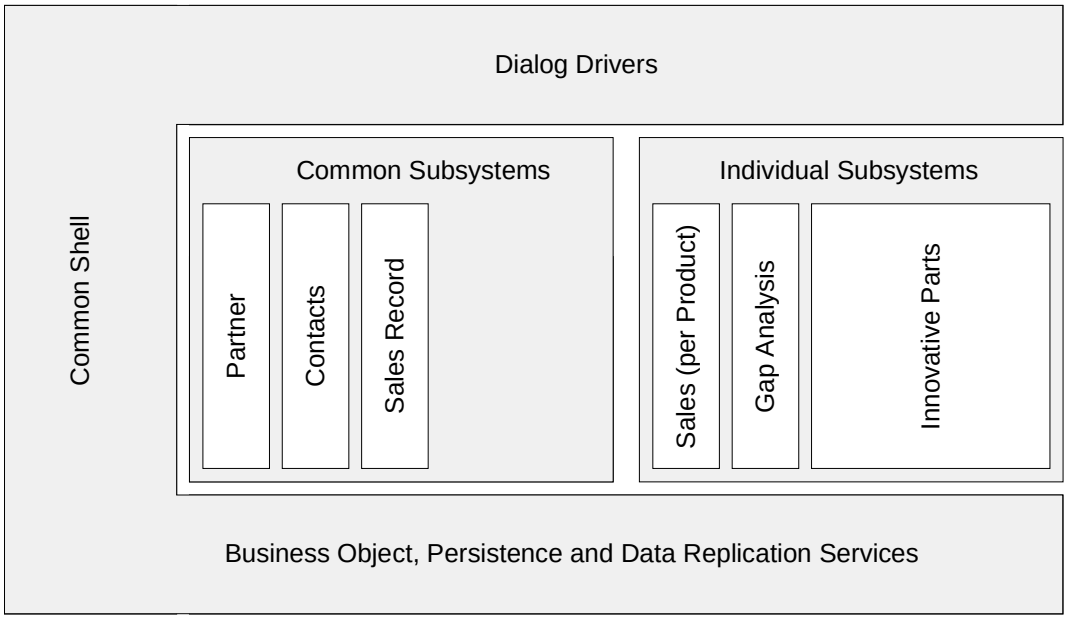


Figure 10: Structure of an Insurance Sales System Application

The components of your system have the following responsibilities:

- *Common Shell:* Provides all the technical infrastructure that you need for a PC based object-oriented solution like the MVC frameworks, persistence frameworks, data replication plus a generic application driver that allows plugging in new applications easily.

- *Partner Subsystem:* Allows your sales representative to collect information about her customers, their financial circumstances, their hobbies and the like. All the facts that smoothen sales and allow selection of target groups for sales action. A partner subsystem for a sales person typically offers much more functionality than a back office partner subsystem.

- *Sales Record:* Allows you to keep track of the products your partner has already bought from you or other vendors. The extent of individuality that you need also depends on Quality of the *Gap Analysis Component* that you want to provide.

- *Contact Subsystem:* Is a component that allows you to keep track of your appointments, to do route planning and the like.

- *Sales:* Offers dialogs that allow you to formulate and calculate proposals together with your customer. These dialogs are product oriented, typically using your product server.

- *Gap Analysis:* Is a typical individual part that gathers data about your customer and compares the financial products she has with her total need of financial products resulting in a list of products she might buy from you.

- *Innovative Parts:* Anything that helps you sell your products like additional multimedia product information, benefits analysis and the like.

Example Resolved

Having heard about the shell approach, you contact a vendor and start a project that fills the shell with your products and innovative sales ideas.

Consequences

Performance over the network: With an offline system you are on the safe side if it comes to performance but on the poor side if it comes to access to all enterprise data. What you need to sell your products is not what you want as the maximum functionality you might imagine at the POS (point of service, not only sales).

Functionality: The pattern describes the kind of systems that you see today at the point of service. The pattern does not describe the systems that we will see in the near future - we will see far more functions moved to the POS. So sorry, but this is a patter and not a look into the future.

Low cost versus individual market appearance: The pattern allows for an individual market appearance (products and innovative parts) at comparably low costs. The pattern also leaves enough space for *Innovative Aspects* by seeing these as “plug ins”.

Flexibility: Short product innovation cycles are supported by a product server approach in the sales per product part.

Reuse and Single Source: Reuse can be achieved by a series of measures: First by using the pattern as you will not write a separate partner or contact application. Second by using the product server approach. Another idea to promote reuse is at the widget level by using component concepts or at the business object level also by using component concepts.

Related Patterns

An Insurance Sales System often uses a product server runtime component, and uses a table system and a rule system.

Known Uses

There are several examples of customizable products that support this pattern's approach. For example the FINAS system by NSE (www.nse.de), the SILVA system by CAF (www.caf.de). Some individual products like KUBIS by EA-Generali or AdiPlus by Interunfall are similar.

Pattern: Table System

Example

Insurance systems (especially those that deal with products) need many tables, to provide valid values for keys, to provide mappings from e.g. regions to data relevant for rating auto insurance policies.

Problem

How do you provide flexible but yet performing access to data that need to be accessed mostly read only, can be organized in table form and need to be updated from time to time by domain experts?

Forces

Performance versus redundant functionality: You implement a redundant system if you implement a second database system with better performance than a relational database system but with less functionality. On the other hand, a second local read only database is so much faster that the price you pay in terms of maintainability is worth the price of a duplicate component.

Costs and effort of data distribution: In a central database you have no data replication problems. If you have a second database using flat files at passivation time and main storage at the time your application is up you get a replication problem. Therefore you have a tradeoff between data distribution and data access performance.

Testing and delivery of new products: Definition data behave like code. Table data in an insurance application are definition data and therefore have to be treated like code: They need versioning, testing and release procedures. If you want to provide this functionality in a central database system you have to invent some additional procedures anyway.

Need for historical data: Insurance systems rely heavily on historical data. For reasons of internal audit procedures you need to be able to reproduce any old state of a policy. To make things worse, policies may be changed and you need to be able to reproduce the actual and historic states of any relevant insurance contract at any time. This leads to so called two dimensional histories [VAA97, Fow97 page 305]. There are virtually no database systems that natively support such functionality.

Solution

Use a table system which provides Tables as the only abstraction. These tables reside in main memory at runtime.

Structure

To a client a table system provides a set of tables as the central abstraction.

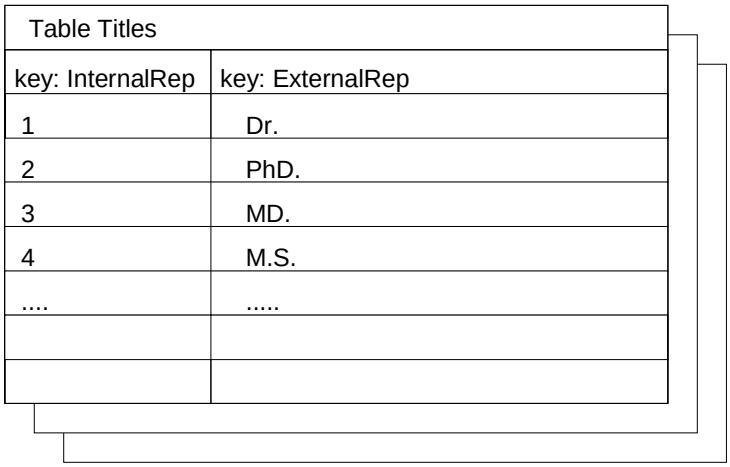
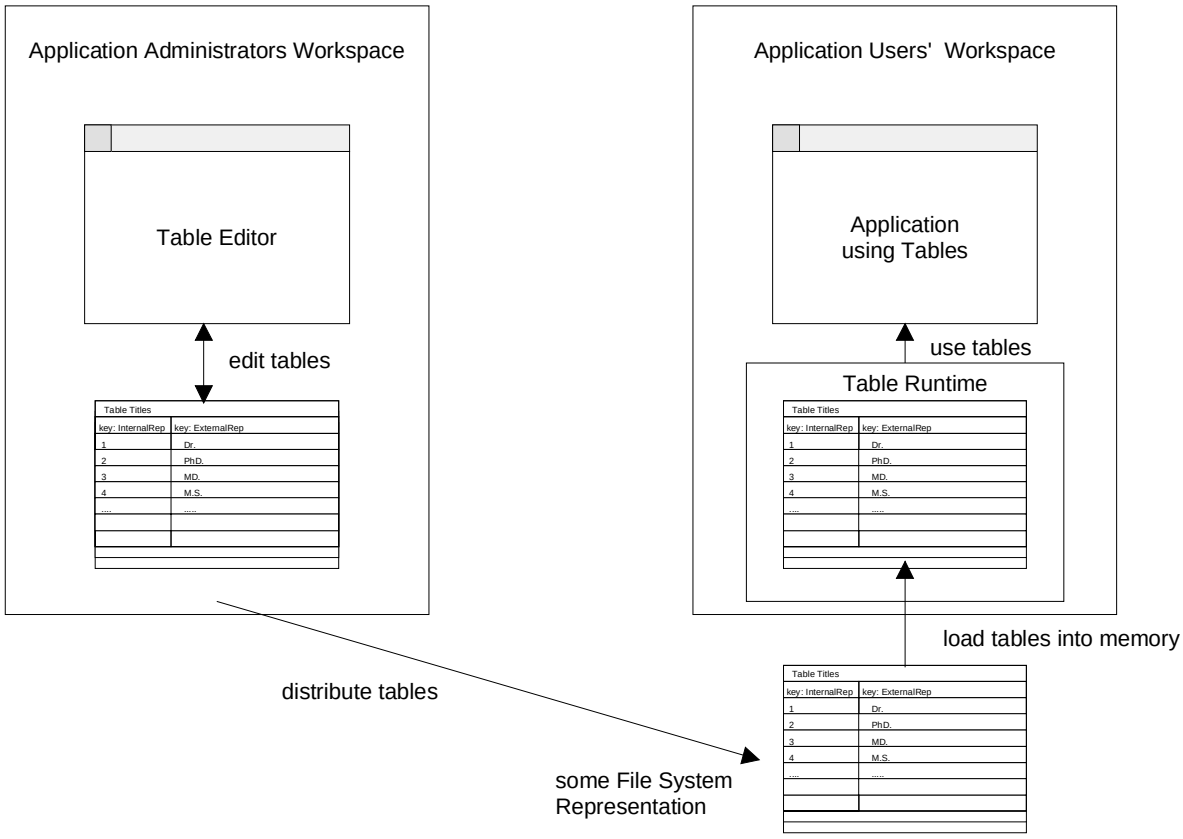


Figure 11: Table containing the valid set of titles for an insurance application

For a table system you need some infrastructure



- A *table editor* allows your application administrators to enter valid data for tables.

- The tables then need to be *distributed* to all client sites. They will typically reside in one or more flat files or some other *file system representation*.

- When the client system starts up it might load all the tables into memory. If the client system does not have enough memory to do that it may also use a caching algorithm to read tables by need and swap out unused tables.

Consequences

Performance: Accessing a table system is several orders of magnitude faster than a database access (nanoseconds instead of milliseconds).

Redundancy and cost: The downside is that you install a second stripped database in your main memory and that you have to develop the procedures for data distribution, data testing and data editing. A table system is not cheap.

Variants

On the market you will find table system with and without the ability to treat historical data.

Related Patterns

For patterns of historical data see Fowlers Analysis Patterns: Historic Mapping and Two Dimensional History. To really implement a system the level of detail provided is not sufficient. Our own Phoenix Framework [Pho98] contains an implementation, but ONE design is not a pattern and we do know only one other implementation [Joh+98] which is very different.

Known Uses

Tabex/2[3] is a product common for host and client systems in the European insurance market. VP/MS [CAF97] contains a table system as a component of the product server (but without historical data). Both systems are used in many insurance systems. VAA contains an own specification for a table system to be used in the insurance business [VAA97].

Frequently Used Patterns and Strategies

The following is an account of patterns that have been frequently referenced in this paper.

Active Object Model

You can adapt an Active Object Model [Joh98] without programming. Other terms that are used for the same thing are meta system or reflection [Bus+96].

Business Process Reengineering

You will often combine a new implementation of an insurance system with a business process reengineering effort. For a discussion of BRR in pattern form see [Bee97].

Composite

Whenever you want to model a product tree, this calls for the use of the composite pattern [GOF95].

Interpreter

To implement a rule system, you will create you own little programming language, mostly using the interpreter pattern [GOF95].

[3] by BOI Software, Linz

Reflection

The reflection pattern [Bus+96] describes meta systems in pattern form. You need to use it to implement product flexibility, using Active Object Models [Joh98].

Type Object

The type object pattern may be used to connect policies and products by making policies instances of products. The recursive structure of product trees is mostly implemented using composite.

Virtual Machine

If you build a product server, you somehow need to interpret your product definitions. You may use a virtual machine to decouple product definition (programming) from running products (product instances are policies).

Whole Part

The composite pattern is a variant of the whole part[Bus+96] pattern. Therefore you use whole part, whenever you use composite to model product trees.

Acknowledgments

I'd like to express thanks to my PLoP shepherd Richard Helm for many useful hints and comments.

References

[Aho+86] **Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman**: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley 1986.

[And+97] **David M. Anderson, B. Joseph Pine**: *Agile Product Development for Mass Customization: How to Develop and Deliver Products for Mass Customization, Niche Markets, JIT, Build-to-Order, and Flexible Manufacturing*; McGraw-Hill, 1997.

[Ald+98] **Robert Aldrup, Jörg Baumann, Christian Weitzel:** *Die Phoenix Facharchitektur*, agens & EA Generali AG, 1995-1998. http://www.agens.com/ - Look for Phoenix.

[Bee97] **Michael A. Beedle:** *cOOherentBPR- A pattern language to build agile organizations*, Proceedings PloP 97, http://st-www.cs.uiuc.edu/users/hanmer/PLoP-97/Workshops.html

[Bus+96] **Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal:** *Pattern Oriented Software Architecture - A System of Patterns*, Wiley 1996.

[CAF97] **CAF GmbH**: *VP/MS, Versicherungsprodukt-Modellierungssystem*, CAF GmbH, Gilching 1996, 1997, 1998. http://www.caf.de/.

[Fow97] **Martin Fowler:** Analysis Patterns; Addison Wesley Longman, 1997.

[GOF95] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**: *Design Patterns, Elements of Reusable Object-oriented Software*, Addison-Wesley 1995.

[Jac+96] **Eydun Eli Jacobsen, Palle Nowak:** A Pattern Language for Building Virtual Machines, Procedding EuroPLoP 1996; http://www.cs.wustl.edu/~schmidt/europlop-96/ww1-papers.html

[Joh98a] **Ralph Johnson:** *ActiveObjectModel*; Wiki Wiki Web; http://c2.com/wiki?ActiveObjectModel

[Joh98b] **Ralph Johnson:** *Personal Communication*, 1998

[Joh+98a] **Ralph Johnson, Jeff Oakes:** *The User-Defined Product Framework*; Work in progress, available via the author johnson@cs.uiuc.edu.

[Joh+98b] **Ralph Johnson, Bobby Woolf:** *Type Object*, in Robert Martin, Dirk Riehle, Frank Buschmann (Eds.): Pattern Languages of Program Design 3. Addison-Wesley 1998.

[Kel97] **Wolfgang Keller**: *Mapping Objects to Tables: A Pattern Language*, in „Proceedings of the 1997 European Pattern Languages of Programming Conference, Irrsee, Germany, Siemens Technical Report 120/SW1/FB 1997.

[Pho98] **Jörg Kröger, Wolfgang Keller:** *Phoenix Business Model Framework, User's Guide*, Internal Technical Document, EA-Generali AG 1998.

[Por85] **Michael E. Porter:** *Competitive Advantage*; The Free Press 1985.

[Ren+97] **Klaus Renzel, Wolfgang Keller:** *Three Layer Architecture* **in Manfred Broy, Ernst Denert, Klaus Renzel, Monika Schmidt (Eds.)** *Software Architectures and Design Patterns in Business Applications*, Technical Report TUM-I9746, Technische Universität München, 1997.

[Sch+96] **Paul Schönsleben, Ruth Leuzinger**: *Innovative Gestaltung von Versicherungsprodukten.; Flexible Industriekonzepte in der Assekuranz*, Gabler, 1996

[VAA95] **GDV**: *VAA - Die Versicherungs-Anwendungs-Architektur*, 1. Auflage, GDV, Bonn 1995.

[VAA97] **GDV**: *VAA - Die Versicherungs-Anwendungs-Architektur*, 2., überarbeitete Auflage, GDV, Bonn 1997.

Table of Contents