Service Provider: A Domain Pattern and its Business Framework Implementation

Ali Arsanjani[1]

1. Introduction

The Service Provider domain pattern is the result of trying to unearth commonalties and mine patterns across several industrial projects including telecommunications (customer care, order entry, sales leads, sales, billing, marketing) , automotive (design of new car features, marketing regions, engineering integration, etc.), higher education (university registration system, distance education, etc.), packaging and healthcare (patient management system).

Rather than discovering patterns that were unique to each specific domain, it was observed that, below the surface level, there lay deeper patterns that *spanned* domains. These cross-domain patterns were not restricted to GUI, persistence or any other infrastructure-specific commonalty. This domain spanning is not at a design level as in, for example, the Composite design pattern [GHJV95]. It is, rather, at a business level. In trying to implement the meta-level concepts across a number of business application domains, it was observed that creating a business framework from the vantage point of a *domain pattern* provided exceptional generative capabilities.

In writing business applications, do we need to start from scratch for each new application in another domain? Or can we build a generic business framework that captures most key abstractions and their collaborations that can be reused and customized for the specific idiosyncrasies of a given domain for which we are writing applications?

The result of this effort was what was named the Java Business Frameworks (JBF). The aim was to be able to respond to new application development needs in a timely manner by avoiding code re-writing; but instead customizing a framework. JBF has been evolving during the past two years. Its foundation lies on the meta-domain pattern called Service Provider. Since “meta” is a strong prefix, we will use this term sparingly and classify the pattern as a “domain pattern” in the taxonomy of patterns.

Therefore, in this paper we will concentrate on the application (business logic) layer. The Service Provider domain pattern generates a non-domain specific architecture that can be customized for many business domains in order to provide meaningful default behavior and structure. Developers can then customize the framework according to the actual business domain they are designing and implementing. Being a domain-pattern, Service Provider contains many smaller patterns that we will not be discussing in depth. These sub-patterns pertain to the user-interface, business logic, communications, database and middle-ware layers of the application systems being used for pattern mining and have found their way into the framework (some appear in the appendix).

Many of the terms you will read about such as (Party/ Client/ Service User/ Service Subscriber) are used interchangeably, since one or the other seems to be the natural one to use in each given context, and as you change contexts, you might tend to use the other term, synonymously. For example, a Service Subscriber will subscribe to a cellular phone service offered by a telecommunications company. A Client may be used in the context of a software services company, or in a retail scenario. Whereas a Party may be used to denote the party we are doing business with in a financial institution.

Also, we have tried to limit the running examples to two domains: course registration at a university or commercial training center and subscription to telecommunications services. Other examples were initially

[1] Ali Arsanjani is a Consulting I/T Architect at IBM (Enterprise Java Services National Practice). He was formerly Chief Architect and CIO at Object-Oriented Technologies, Inc. He is an adjunct faculty of computer science at Maharishi University of Management. He can be reached at arsanjan@us.ibm.com .

used, but this was thought to put too much emphasis on the examples and detract from the underlying theme and were thus omitted.

2. Common Themes in Business Applications

Although different in behavior and domain, typical business applications share a lot more than meets the eye. These "undercurrents of order" underlying the way business is conducted can be identified and managed (as in its implementation in an information system) at several levels. First, consider your typical self-care e-business order-entry application. At some point, a form or page will appear showing a list of available services that are being offered. This allows the user (or customer service representative) to select a service and "subscribe" (telecom service) or "register" for it (e.g., course at a university).

After the initial selection of services and the entry of customer information, certain pre-requisites need to be checked: is this an existing customer, if so, retrieve their records. If not, create the customer information, payment information, demographics, etc. Once the user subscribes to a service, they are in reality agreeing to certain explicit Terms of Agreement. These Terms of Agreement, exist for the system as a whole and perhaps for each individual service a service user subscribes to or products that they order. These may include rules pertaining to the fact that only certain services may be combined in package or to be subscribed to simultaneously. They may also pertain to how the user may use the product, services; whether the user has the right privileges to do something, to check statements online, verify correctness of payments, etc.

Note that there is not much difference in the order-entry process between subscribing to services or purchasing a product. Also, services and products are usually packaged and grouped for marketing or categorization purposes.

Here are specific areas of commonalty that are more frequently encountered that have *parts* that are potential candidates for creating components or of creating a white-box framework to handle its interactions and common structure.

1. **The User Interface layer**. Among the plethora of different GUI specifications, we can zoom in on a common theme. Frequently, within this recurring theme, the user needs to have the following capabilities.
    1.1. **Log-on** (Authentication, Authorization)
        1.1.1. Log-on layout, which is usually the first layout, the user sees.
    1.2. **Fabrication** : The subsequent user-interfaces screens to be displayed for this user are governed by his access rights, which depends on his Role (identified by Authorization; the level of access and privileges he has been authorized for as part of a System Administration Setup ; which is a set of screens of its own).
    1.3. **Workflow**: The user then makes a choice using a menu-based system and enters into one of several possible workflow; each with their own set of activities (steps in the workflow) and interactions (screens displayed for each step/activity along with the variety of navigational options and data/maintenance options the user will have)
        1.3.1. These workflow panels often allow some degree of navigability. They may allow the user to go back to a previous step in the flow, to terminate the flow or in some rare cases to by-pass a number of steps in the workflow and go directly (as in a tabbed pane scenario) to a given step in the workflow.
        1.3.2. What the user will see as the "next" panel or screen will be determined by a number of factors.
            1.3.2.1. **User Role**: These may consist of his Role (access rights and privileges), (**Locale**) language/locale he is using, step in the workflow (**Workflow Step**), state of the activity (**Workflow State**) he is in (has he completed all mandatory data items to be entered on the previous screen? Has his credit card been approved? Etc.) This will also be affected by data returning from a back-end system. For example if the customer already exists in the database; or if his subscription has expired. These Notification Events impact the Workflow State and the Rules governing the flow of

the Workflow. (See [Arsanjani99;2])

1.4. **Search/Navigate(List)/Maintain**. At some point the user will want to maintain or add information. He may begin by conducting a search (next bullet point) , or going through a navigation layout that displays current working (or the saved record set previously worked with) records. This may be as simple as a list of customers the customer service representative found in his last interaction with the system; or one that he saved and has now loaded (Maintain Navigation Layout). It may be a more complex tree structure of all services this customer account includes, etc. In short the Navigation Layout may be a Table structure or a Tree structure.

1.4.1. **Search**. When the user wants to search for a record set he will use the find or search layout that searches for some specific records in the database according to a criteria entered . Frequently, the Search Layout will pass the result set back to the Navigation Layout in order to have the result displayed.

1.4.2. **Maintain (Data-entry)**: Ultimately, the user will want to narrow his search and get to a desired form or record to maintain. This may be Viewing, Editing, Deleting or Adding data in a Data-entry Layout .

2. **The Database or Persistence Layer**. This layer or tier in a multi-tier system will be responsible for handling a persistent object's state. An object may have save/load/update operations that will be delegated to this tier. Alternatively, the object may not know about persistence. Example: Enterprise Java Beans. The Container may handle persistence along with other infrastructure issues including life-cycle, transactions and concurrency. In this case the Persistence Layer will handle saving and loading of streams or objects to and from back-end database systems (relational or object-oriented).

   The issue of connection pooling arises and thus the Database may also need to have a connection pool to handle different kinds of object requests, possibly for different database types (such as relational databases and Object databases) and structures (IBM DB2, MS-SQL Server, Oracle, etc.).

3. **The Business Logic or Application layer or tier**. This tier will contain the objects, primitive data types and the rules, which are available to the objects. Business Processes are frequently implemented in this layer or tier. This allows a common ground and basis for all enterprise wide or global (in the case of e-business) systems to converge upon this layer of business logic and rules before a workflow or business process is deemed to be complete. The rules of the business will be checked at this tier, and all applications will go through this common layer to verify various aspects of their data and state (payment overdue, is this new order valid? What are the new promotional prices? Can I add this service to this customer type? Has the shipment of the product gone out? Etc.) Some of the primary abstractions include:

   3.1. Business objects
   3.2. Business rules which should be applied to business objects (or non-business objects).
   3.3. Services part. Displaying the entire available services, registering/un-registering of a Service for a Service User and defining the Terms of Agreement are done in this section.

The generalization of the above, into a set of collaborating classes which are defined as interfaces and abstract classes, which users of this framework may customize for their unique business process are embodied in a Business Framework. The actual implementation we conducted happened to be in Java. That is why we call our implementation the Java Business Frameworks (JBF).

2.1 Example Use Cases from Two Perspectives: Contrasting Users and Developers

Example: E-business application Self-care Order-entry

The use cases of a typical business application from user's point of view

1. Browse through list of offered services
2. Select a service and view its proposed terms of service
3. Enter properties of a offered service and the term of agreement (for example enter the credit card number, the amount of money you will prepay, etc)
4. Register one or more services based on the on terms of agreement
5. Possibly withdraw (from a course) or discontinue a service(s) (membership) or return a purchased product or even cancel an order
6. Possibly alter properties of a service after its registration; for example, add a phone line to current long-distance package.

Development activities to implement the required architecture

1. Set up the service provider object and its properties
2. Create/Add services for the service provider (possibly from a database)
3. Remove services discontinued by the service provider (and possibly from a database)
4. Edit properties of services (possibly synchronize in a database) including setting up Service Offerings
5. Set up the Proposed or Agreed-Upon Terms of Agreement for a Service Offering
6. Provide the user interface for the user and let the user browse, register and maybe un-register a service
7. Whenever a new user registers a service, set up a new customer account
8. Provide a database persistency mechanism for customer accounts, services and proposed and agreed on terms.
9. Check user's subscription, purchase or enrollment against the business rules (e.g., credit verification) and the Terms of Agreement for the Service Offering (e.g., "18 month contract before eligible for a new cellular phone upgrade").
10. Define Terms of Agreement and Business Rules in the Rule Model (e.g., via a Rule Browser – see Rule Browser pattern in pattern language on scalable business rules [Arsanjani98]).

2.2 Typical Characteristics of Business Applications: Mining the Service Provider Pattern

During the course of the development (analysis, design and implementation) of business applications in various domains one recognizes the unmistakable feeling of *deja-vu* when one re-encounters a similar issue, problem, or begins to devise an analogous solution. Though the context may be slightly different, it is undoubtedly the *same* underlying combination of collaborative notions.

One may be able to take the time and gather support for scheduling sessions for mining out these patterns. In the case of Service Provider, there is a certain peculiarity in that the pattern is general enough to be used across domains, and yet, like a set of *n* equations with *n* unknowns, there may be many (if not infinite; this is engineering, not math!) solutions to it.

In a traditional business scenario, a person or corporation develops services or products (hereafter referred to as services) based on an initial demand or market assessment. They then compose the services into packages, deals, and service offerings, which are bound by some generic and specific Terms of Agreement. The generic ones are across product types, whereas specific ones arise and are resolved within the context of the actual business transaction (sale, purchase). Then the service provider uses a Service Connector or Marketer to present the potential set of planned Service Users or Subscribers with the Service Offerings that have been developed and composed. Prospective buyers review the Terms of Agreement for a given Packaged Service Offering. If there is mutual consent between them and the Service Provider, they either subscribe to or register for the services (as in telecommunications or higher education, respectively) or purchase the products (and possibly associated services; as in the case of a computer hardware purchase with technical support). In order to do so, they submit an Order that contains OrderLineItems related to the Service Offerings they have selected, and are in conformance with the Terms of Agreement.

Order Fulfillment happens when the Service Provider receives the request for subscription to services or purchase of products. Business Rules are checked and if everything is in order, items (products) are either manufactured, prepared or enabled (as in provisioning a telecommunications switch with the client's phone numbers for long-distance or cellular services). So products are picked and shipped; services may be scheduled and provided or installed and enabled.

The Business Transaction records the Terms of Agreement, the Parties and the Event (includes date and time, maybe circumstances in legal scenarios). Within the Terms of Agreement, can be found Payment Methods, Installments, Interest Rates, Down-payments, Credit, Rate Plans, and all other necessary abstractions that capture the prerequisites and constraints and agreements for a valid business transaction.

So, Parties [Fowler97] or Clients actually buy or subscribe to that service after accepting the terms of agreement under which they are bound by the business transaction. During this operation, money or Credit is exchanged based again, on some terms of agreement that are agreed upon by both the Service Provider and the Service User. In this sense, Service Provider is complementary to Ralph Johnson's *Transactions and Accounts* pattern language.

In an educational scenario, Service Provider focuses on the Provider, Services, and the actual ServiceOffering (such as the distinction between a Course and whether it is being this semester at a particular university campus for a particular degree program). It includes its Terms of Agreement (constraints, business rules) and the means whereby the ServiceUser or Subscriber or Buyer will interact with the actual incarnation of the framework that instantiates the pattern to actually subscribe.

Transactions and Accounts talk about the recording of the financial aspects, over time, for the Service Provider. Although in the Service Provider there is CustomerAccount which can be used to interface with an Account or participate in a Transaction by making that Transaction a Rule (using the Service Provider's Rule Model).

The Service Provider's intent can be summarized as follows:

Define a domain independent and generic architecture for enabling and facilitating an information system for a Service Provider that (develops, composes and presents) provides business Services, Offerings and Service Packages to Parties based on some Terms of Agreement to enact a Business Transaction.

In other words, the service provider pattern deals with all aspects of defining, packaging, presenting and accepting orders for services in a business context. It is based on the idea that the way business applications deal with services, products, membership, subscription, registration, are all very similar. This fact can be used to create a generalized solution, which in turn enables more extensibility of design and the achievement of a higher degree of reuse.

2.3 Definition of Key Terms and Participants

1. **Service Provider (SP)**: Typically, a composite hierarchy such as a corporation. A special case would be a small business or even a person, which provides some services or products. A telecommunications, utility, hardware vendor or software development house, bank, insurance company would be examples of larger SPs.
   - **1.1.** The Service Provider can assume a number of ***roles***. These roles are carried out by various departments within the organization or they may be outsourced to third-party organizations who will play the following roles. Alternatively, these roles may be carried out by the Service Provider's business partners.
     - **1.1.1. Service Developer (Supplier)**: Services that the Service Provider offers need to be developed (software), manufactured (Ics), prepared (food), cultivated (flowers).
     - **1.1.2. Service Composer:** The Service Provider often plays the role of Service Composer. SP will take one or more Services and "bundle" or package them according to various market trends and analyses, according to certain Conditions and Constraints (termed Terms Of Agreement) based on legal, engineering, marketing, strategic directives.

1.1.3. Service Connector (Presenter): This role is fulfilled by providing potential Subscribers, access to the Service Offerings. (Office of Admissions in a university, an Internet Service Provider, a marketing/customer service firm. They will then present Service Offerings to Subscribers.

2. **Service**: Services and products that a service provider provides are called Service. For example a course offered by a training center or a book offered by an on-line bookshop can be one of the available services a company provides.
   2.1. **Service Catalog**: Services typically need to be listed and accessed in several ways. This provides a selection point for whatever portal is physically displaying the Service Catalog. Thus, the Catalog is more of a business object (Model) that will get displayed (View) based on a presentation or portal manager component.
3. **Offered Service or Service Offering:** a service when is offered. The Subscribed Service or Service Offering will have its own attributes based on what the realization of the Service interface has to offer: for Course (As Service) we have Offering as Subscribed Service; and Offering will have an associated semester, units, teacher, room.
   3.1. **Service Offerings are** distributed, integrated and managed. They need to be deployed(distributed) so as to enable prospective customers (Prospective Customer is a Subscriber or Client State) to Subscribe to them.
4. **Term of agreement (TOA)**: Each service provider has some Terms of Agreement (TOA) for its services. Each service may have a term (or terms) and that term describes the conditions for that specific service. For example, how much money is paid, the discount amount or any other condition of the agreement being applied.
5. **Subscriber or Customer Account**: Someone or a corporation who subscribes to or registers for and uses sort of provided services. This is the person or account that is registering for a service.
   5.1. **Orders: Subscription or Registration are realized** by taking orders. Orders have order line items and are connected to the Generic Services (Products or Services) that will be undergoing the Business Transaction.
6. **Business Transaction:** Any order or subscription along with its Terms of Agreement between a Service Provider and a Subscriber will take place within the context of a Business Transaction.
   6.1. **Account:** The Transaction will be Accountable. This will be a portal to financial information.
7. **Service Location:** A Service Provider will provide service offerings at locations. These may be physical locations such as stores, or conceptual ones such as geographic regions (" our wireless coverage is limited to these states")
8. **Control Point:**
   8.1. Service Locations will need to go through a control point to be able to access a Service Provider's Service Offerings. For example, a telecom switch is such a control point, or you need to dial-in to an access point for an Internet Service Provider (ISP) to get online to their services.
   8.2. Control Points have security issues and these frequently keep Access Control Lists, Digital Signature or Public Keys for implementing controlled access.
9.

3. A Quick Look at Frameworks

Before getting into the details of the pattern itself, we will briefly touch upon key notions of frameworks in order to set the stage for subsequent discussion. Each definition below unfolds a slightly different perspective on frameworks.

A framework is a set of prefabricated software building blocks that programmers can use, extend, or customize for specific computing solutions. With frameworks, software developers don't have to start from scratch each time they write an application. Frameworks are built from a collection of objects, so both the design and code of a framework may be reused. It aims to capture the programming expertise necessary to solve a particular *class of problems*. Programmers purchase or reuse frameworks to obtain such problem-solving expertise without having to develop it independently.

A framework is a set of valid interface and abstract class interaction sequences which satisfy (solve, are a solution to the "equation represented by") the intent of a given problem domain.

It is a set of interfaces and abstract classes with default implementation, which capture the valid object interaction sequences (collaborations, interactions) of a domain architecture in the form of code.

Here is an interesting perspective that comes from the Patterns Generate Architectures (Kent Beck and Ralph Johnson) idea: a framework is the realization or instantiation of a domain pattern; a domain pattern is the blueprint on which a framework (for that domain) is built. The framework is the realization architecture of the domain pattern. Each domain pattern will contain a set of sub-domain and cross-domain patterns.

The cross-domain patterns or common business objects (Aka IBM San Francisco Project [IBMSF], see references) horizontally cut across domains. The sub-domain patterns are clusters containing classes or other clusters that are unique to the problem space of the specific domain.

3.1 Framework Domains

The problem domain that a framework addresses can encompass application functions, domain functions, or support functions:

- **Application frameworks** encapsulate expertise applicable to a wide variety of programs.

- **Domain frameworks** encapsulate expertise in a particular problem domain. These frameworks encompass a vertical slice of functionality for a particular client domain.
- **Support frameworks** provide system-level services, such as file access, distributed computing support, or device drivers. Application developers typically use support frameworks directly or use modifications produced by systems providers.

3.2 Key Advantages of Frameworks

The overall benefit of frameworks is that they enable a higher level of code and design reuse than what is practical with other design approaches.

- Provide infrastructure and architectural guidance
- The framework calls you, you don't call the framework; you get flow of control or collaborations “for free”
- Provide a mechanism for reliably extending functionality; the framework has been extensively tested and its reliability proven to a large extent.
- Reduce maintenance efforts and costs

4. Service Provider Domain Pattern

4.1 Intent

Define a domain independent and generic architecture for businesses (Service Providers) that provide Clients with business Services and Service Packages based on specific Terms of Agreement.

4.2 Motivation

The Mathematician and teacher, George Polya wrote a book *How to Solve It*. The first time I encountered this book was in 1979 when I was getting into algebra. The essence of the solution to any mathematical problem was explained as a series of heuristic techniques: “if you can’t solve this problem, ask yourslef, have you solved a similar problem? Can you use the solution of that problem here? ” Applied to programming, this would be *programming by meta-level analogy*. If you still can’t solve the problem, “Solve a concrete problem by solving a more general problem. The general problem has paradoxically a simpler solution.”

Here, writing an application is the “problem”. We solve this by writing a framework. Soon, we begin to see analogies between frameworks. We are bound to strive for greater generalization; being human. Thus, the problem is now: how do you write a generic framework that will span domains? The answer is to go one

level higher in abstraction (meta-level) and build a cross-domain pattern (not code; go beyond the framework into the concept) that describes the solution at a meta-level; for all businesses; not just for this particular business application.

We take a look at a decomposition paradigm based on “manners”. We call this “objects have manners”. Or “components have manners”. This signifies that in addition to behavior that is exposed through interfaces of a type, we need to have a same-level and /or meta-level specification of **how** *to use the interface*. The first and simplest scenario is a valid object interaction sequence: what are the valid orders in which I should invoke methods on this object (of this type). The second level is: what intent do I have for invoking any method on this object? Can I use a façade? Should I call a Mediating Façade? The answer to these questions have crystalized in the concept of a component or <<cluster>> (a stereotype of cluster in UML). See the rule model in the structure of the solution.

4.2.1 The Concept of a (meta-)domain pattern

A (meta)domain pattern is non-specific in the sense that is a reusable design structure able to span several business domains. This does not signify something like a windowing system or anything non-specific in that sense. We have been writing applications in several apparently different business domains. The commonalty between the application domains are captured in a (meta)*domain pattern*. This domain pattern will then guide the generation of a framework that can be customized for each domain. The following domains have been chosen as a *Running Example* simply because it is easier to see the commonalty between them.

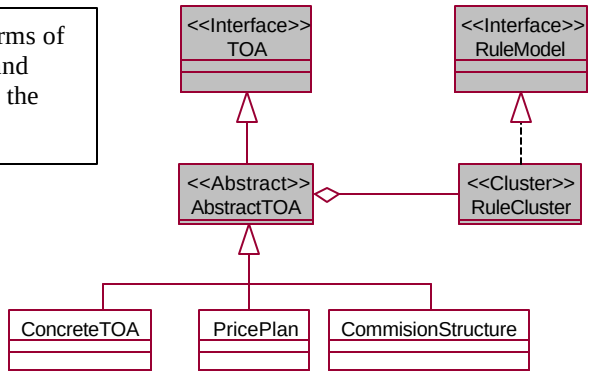
One of the themes and issues people encounter in writing business systems may be summarized as follows: how can you better manage the implementation of system architectures that support rapidly changing business services that have intricate dependencies with business constraints? Thus, we need to treat Rules as First-class Citizens [Arsanjani 97].

4.2.2 Example 1: University Registration or Training Center

Suppose you are writing a registration system for courses offered at a training center or a university. Students will want to log into the system, view or search course offerings and possibly register for one or more courses being offered this semester (the latter is an example of a Time Constrained Service Offering).

Registration will include payment of fees or tuition, based on a contract or a Term of Agreement (A Contract of sorts). This TOA may include course prerequisites, degree or certificate (university) or a special agreement (with a professional training provider) that the client will receive a group discount if they register five or more people by a certain date[2]. Also, the training center may have various other policies such as cancellation policies indicating whether you can cancel or change your registration up to a certain date and get a full refund or a percentage thereof.

Figure 1: Terms of Agreement and relation with the RuleCluster



[2] As another domain example, the purchase of airline tickets also follows the Time Constrained Service Offering pattern.

This is in fact a good opportunity to mention that in business systems, rules are "first –class citizens" of the object model. Thus, along with the identification of key abstractions within the domain (object identity), the next important aspect might be the object's "manners" or ***how*** *the object needs to behave* under various business scenarios and circumstances. These manners will then determine the actual finer grained behavior to be ascribed to the object in the form of methods (behavior) and how it affects and manipulates the object's state[3].

Going back to our example, in a university context, you would expect to have a course offered only within a given time frame; i.e., a semester or "block" [4]. This is an example of a Time Constrained Service Offering. This means you can enroll for a course only within a given time frame (e.g., semester or block). In a commercial training context, we would probably have an early withdrawal fee (as well as an early-bird registration discount). These constraints or business rules can get quite complex when they span geographic regions (such as wireless telecom services around the US), or if they involve the customer being on several "rate-plans" and having multiple locations, billing cycles and packaged deals.

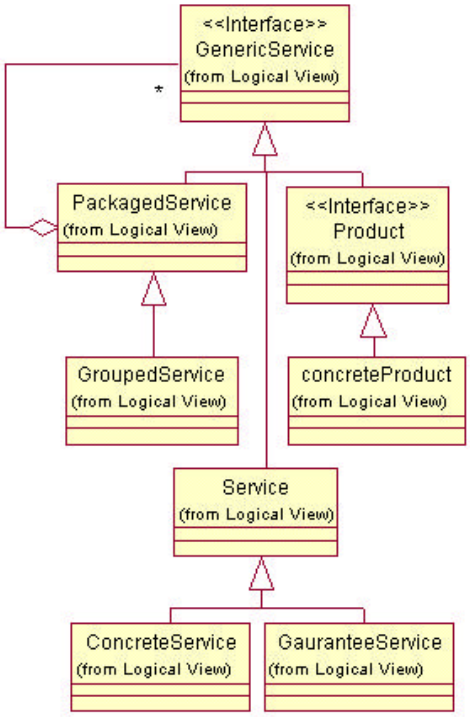


Figure 2: The Generic Service Cluster

Additionally, there is the notion of Location Dependent Services. These services can only be subscribed to relative when you happen to be within a certain geographic region or locale. For example, that if you are a student at the University of X at SomeCity, you can only apply for the software engineering bachelor's degree if you attend the on-campus full-time track (not distance education). Alternatively, you can enroll in the MBA with distance education, and be anywhere where you access to a 56K modem.

There can also be other forms of Location Dependent Services based on the geographical location of the Service Provider. For example our training center may have some branches overseas. A client calls the Maryland branch and wants to only enroll in classes held within the area. The Maryland training branch may not have the course available or the facilities to teach it at that location.

[3] The author is working on a pattern language for the scalable design and implementation of object manners and business rules.

[4] Some private universities have the "block" system where a student concentrates on one subject only during a one month period, rather than taking three or four courses simultaneously within a semester.

Service Provider Cluster, differentiates a Service Provider containing a hierarchy (either departments or local offices) and LocationDependentServiceProviders such as regional offices and stores or campuses.

Now lets take a look at another example in the academic domain (a course) and a product (a textbook):

The training center needs to handle additional Service Types. The company now has plans to publish books or Tech-Reports. So the application should be able to accommodate this addition (change) without impacting the entire system (Meyer's Open/Closed principle). Thus business systems should not be tied to a specific service such as a course, but should be able to add new services and products with their individual idiosyncrasies.

This pattern addresses the problem of the way the customer (a student here) and the service provider (the training center or a university) interact with each other and come to Terms of Agreement on enrolling for a set of service offerings[5].

In this scenario, one way to design the application is to have separate objects for each key abstraction; for example a TrainingCenter, Course1, Course2, CourseN, Student and TermsOfRegisteration. You may choose to include terms of agreement in the Student class if there were a one to one relationship between a student and a course, meaning that all students follow a single TOA strategy. But in our scenario (or more generically in most business systems) this is not the case. Often customers (or students in this case) are placed in different classes or categories, each class with its own unique Terms of Agreement. Students can be part-time or full-time; they may be taking a special track or emphasis with different requirements (TOA), stipends (interns, teaching assistants) or tutions.

The Service Provider (organization or company) should be able to look at the world from the point of view of the business (Service Provider, not you the Client), so there is an association between the Offered Services and the Customer or Subscriber.

A service (e.g., a Course at a University) can have several Offerings: Software Engineering may be offered during spring semester but not in the fall semester. So this is a "Time-Constrained Service Offering" . The general case would be a "Constrained Service Offering". This is embodied in the Terms of Agreement for a Subscribed Service.

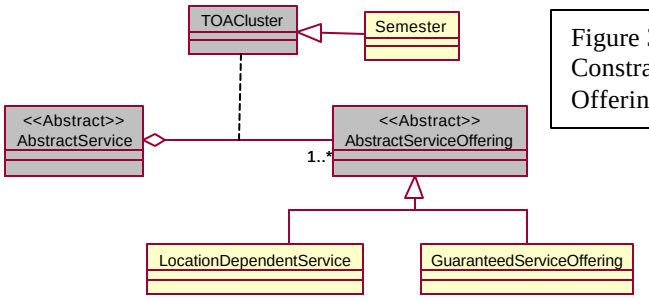


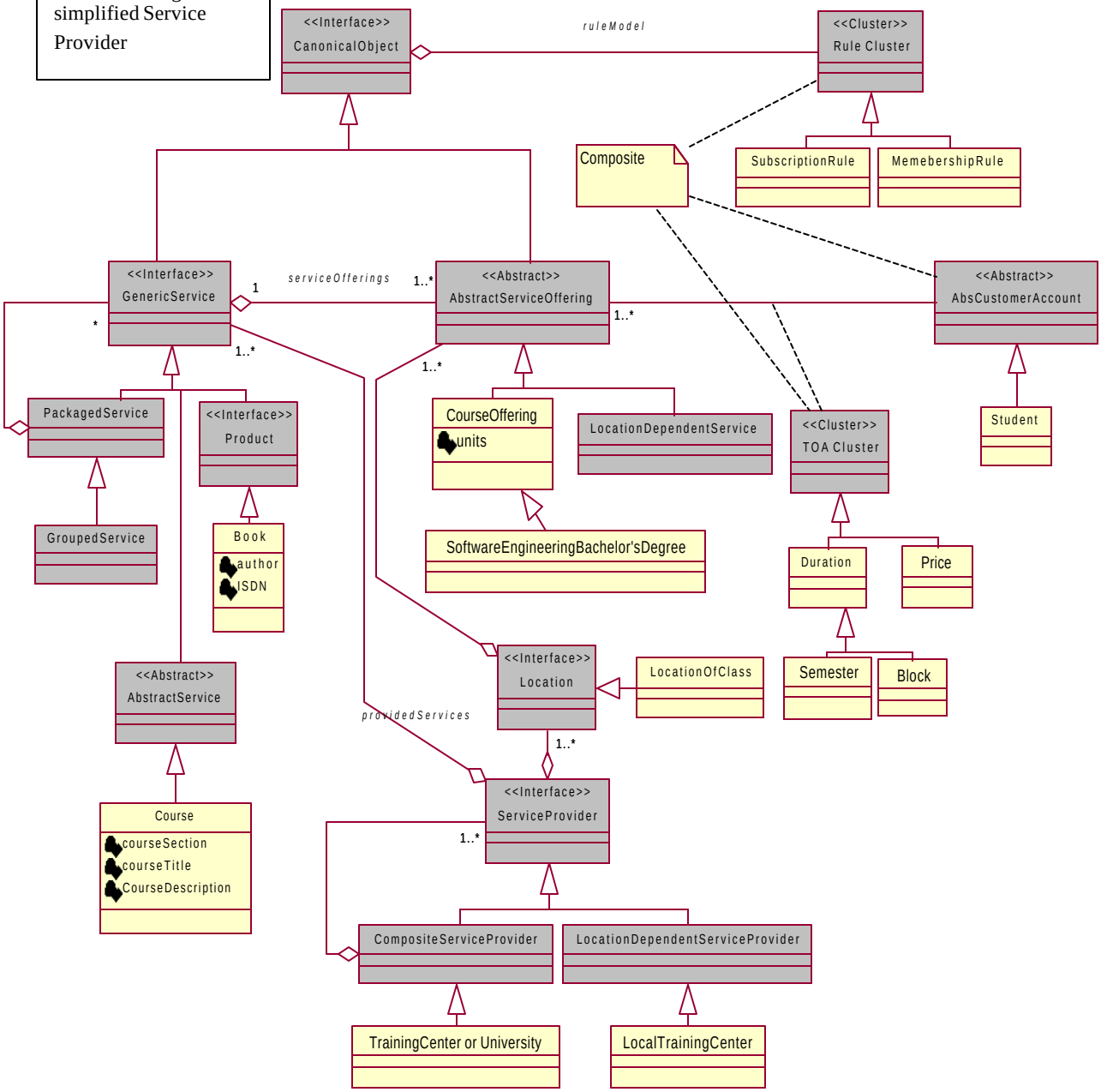
Figure 3: Time-Constrained Service Offering

When you, as a customer/client/party/account (whatever you want to call yourself in relation to the business; for the business will have a reference to you in their database, and the application will look at the world from the point of view of the business (Service Provider), not you the Client), subscribe to a service (or "sign-up" or "enroll" or "register" or "request" or "purchase" or "participate in a program"), the OfferedService you have subscribed to will have a Terms Of Agreement attached to it.

[5] Think of a typical order-entry system for customer service representatives for self-care using the Internet. The same general principles apply there as well.

This terms of Agreement will say, for example, that the course offering you enrolled for in the spring will be 4 units and has the following pre-requisites, will be held 3 times a week for 3 hours in room R025 and the instructor will be Ali Arsanjani. You will have to complete 3 labs, and hand in a semester project based on a selection of projects that will be given to you during the first two weeks of class. If you get a Grade Point Average of 3.5 or higher, you will retain your scholarship, if you had one. If you go below a 3.0, you will have to repeat the course. (Note: the last two rules were inherited from the generic Rules for the Generic Terms of Agreement that is attached to the Service (the Course), so each Service Offering will carry its parent's rules with him (due to Objects Have Manners pattern (see the author's pattern language on Scalable Business Rules Architecture)). Grey boxes represent the framework, while the white boxes are extension points; where developers customize the framework.

Figure 4: An Example of Customizing a simplified Service Provider



4.2.3 Example 2: Web-base e-commerce application

Suppose that you are writing an Internet-based e-commerce application. Users will want to log on to the system, view or search for books or compact disks, add items to their shopping carts, add or delete items from it and eventually purchase products using their credit cards.

Here, a purchase may be considered a form of enrollment; the Client is tacitly agreeing to the Service Provider's Terms of Agreement, for refunds, rebates, special deals (valid within a given time frame). This TOA may include the no warranty money refund period or a special agreement that a PHD student or a university faculty can buy books with 30% discount. The TOA will also include payment method information (coming from the user, such as credit card number, expiration date, shipping address, billing address, etc.) and the price of the items.

In this e-commerce system, items available for purchase are not considered Time Constrained Service Offerings. This means although a customer can refund the money in a period of time, but this does not mean the service (product in this case) is time constrained.

Here we also have the notion of Location Dependent Services. For example, overseas magazine subscriptions are often more expensive than in North America. Some software products may require an export license (part of the TOA).

5. Context

Companies and individuals conduct their business by providing services or products that can be bought (computers), leased (house), licensed (software), etc. The Party with which they do business may be an individual such as a consumer, or it may be a large organization (e.g., McDonald's franchise around the world, or a Bank).

The offering of services (for brevity, we will assume this to be inclusive of products) is usually based on a Terms of Agreement that is usually Time Dependent; (as in a special promotion for cellular services "until the end of the month"). Also, the Service Provider may typically have multiple sites or offices in different geographic regions (sometimes around the world). These sites may include sales representatives' offices, warehouses or stores.

This pattern can be used to generate frameworks and applications in which a composite business Service Provider conducts business based on the offering of Products and/or Services to Clients.

This pattern should be applied to customize applications for business domains, or to generate frameworks for domains. It allows the flexible allocation of time-constrained service offerings based on Terms of Agreement. Orders are taken and purchases are made in this context.

The relationship between the service provider, the offered or registered service and the customer should be generalized so that none of them are exclusively tied to each other. For example a Student should not be tied to a special course policy system. Nor a Subscriber to a particular set of telecommunications services for a specific company.

Developers should be able to customize a framework to a given domain covered by the framework's domain pattern. The developer uses the App model of the Service Provider for defining/removing service and service user objects in order to utilize the attribute management of Service Provider. The developer then uses the Rule model for implementing business rules: from attribute validation to complex, inter-dependent business rule inference mechanisms (if necessary). In order to define, subscribe to services, remove service offerings or service user objects, a presentation layer/mechanism is required to handle the user interface for entering/editing/displaying data. Here, the UI model is used. As business logic needs to connect to backend data stores or legacy systems, the DB model can be used connection to either a relational or object-oriented database.

The system should be easily able to accommodate new services and new packages (promotional service offering combinations). Businesses thrive on creating these packaged deals for a “limited time” and under special Terms of Agreement (such as subscribing for a minimum of two years, paying a monthly premium, etc.) Adding a new Service Type should not entail changing the Service Provider. For example our training center may choose to add consulting services in addition to public and customized training courses.

A business system should be able to offer services or products to customers (whether individuals or companies) regardless of type or number of services. It should be possible to compose or package services to define new service types. This is an activity often driven by marketing. For example a training center should be able to offer courses like “Java”, “OOA” and “OOD” and another grouped course called “OO

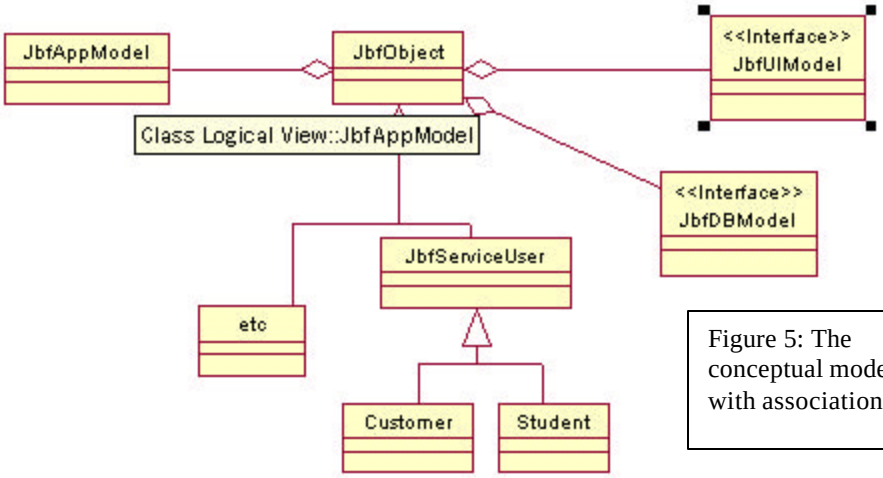


Figure 5: The conceptual model with associations

and Java” which consists of the three courses with a special terms of agreement (such as a price cut).

Customers should be able to easily register for/subscribe to/buy Service Offerings. A Service Offering can be a Generic Service like the I/T consulting offered by a systems integration company, or it can be a Product such as a book sold over the Internet. Maintaining/adding new Service Types should not require changing the Customer or ServiceSubscriber. Nor should registration options affect either the customer nor the offering. For example the client should be able to register for both a course and request follow-up consulting services and buy a software product from the same company (regardless of the type of service).

Each customer should be able to sign/accept a Term Of Agreement (TOA) when a service is being subscribed to/registered for. This acceptance may be required by law, as when changing a long distance carrier, a letter of authorization may need to be signed. The TOA is the link between the Subscribed Service and the user of that service (the Client/ServiceUser/ServiceSubcriber). The structure of terms of agreement should be generic and extensible to support different and always changing terms of the business world.

Registration in a back-office scenario as well a web-based system should be possible to be accommodated without making intrusive changes to the architecture or the domain objects that participate in the domain pattern.

In the domain of providing and requesting services, an entire supply chain may be involved: from raw material to the consumer (subscriber/requestor/client) themselves. There may be agreements (TOAs) that need to be enacted between many Parties (Service Providers, Composers, Presenters, Service Developers/Product Manufacturers). This is handles by the TOA being part of the Business Transaction that is non-repudiable, enforceable and documented. The TOA is really deriving default behavior from

Business Agreement which is a generic agreement between various Service Provider or Business Partner roles.

Services are requested or products ordered, and the Service Provider then processes this request. The request can come through a Portal and if the request requires any Location Specific Service Offering, it will go through the Gateway which acts as a Control Point.

6. Forces

There are many forces that need to be resolved in the conceptualization, design, development and deployment of business applications. Special considerations arise from building and utilizing business application frameworks.

- (+) One may create a business framework for each domain where the services are offered but this causes the development team to re-invent the solution for each domain.
- (-) Adding and packaging Services into Service Offerings can be hard-coded into the application. This does not allow the resilience a business system must provide the business with in order to react constructively to new market opportunities for closing business agreements.
- (-) Business Rules can be hard-coded and dispersed throughout the system, making changes difficult to manage. Centralizing the rules in each cluster facilitates making changes to the rules. Having Rules as First-class Citizens of the analysis method and final implementation technique facilitates mirroring the business into supporting coding structures.
- (+) Management needs to see the rules that are governing their business processes, explicitly. Centralizing rules in code does not allow this browsing and creation and update of business rules. So rules need to be centralized from the ease of programming and management changes and updates, but need to be belong to their respective object types during analysis and at run-time.
- (-) You can rewrite GUI patterns such as Search/List/Maintain (SLM) for every single project. You may also want to write it as a GUI or Presentation Layer sub-framework (cluster).
- (+) Businesses make profits by providing services for credit to a target society of potential subscribers.
- (-) Some Services are offered under certain conditions and constraints; e.g., for a limited time only, and only for new subscribers who buy “this packaged service”.
- (-) Some services are not being offered anymore, even though we have offered them in the past and clients have subscribed to them. We need to keep track of previous subscribers and their terms of agreement.
- (+) Subscribing to a service can be direct (self-care) or indirect (through a customer service representative). Each have their own forms of order entry and pre-requisite validation.
- (+) Rules must be applied to orders (subscription requests). These rules tend to change very rapidly with the market, with competition, with new trends, with new demands….
- (-) Rules must be trickled down from higher management and implemented in the code very rapidly.
- (+) Having a rule infrastructure facilitates productivity and time-to-market.
- (-) Rule infrastructures can be centralized (repository) or de-centralized (“object have manners” [Arsanjani 97])

7. Solution

Therefore, define a cluster[6] called ServiceProvider containing a set of Services. ServiceUsers will Subcribe to Services when they are Offered as ServiceOfferings according to Terms of Agreement. This will create a set of ServiceOfferings for that ServiceUser. The ServiceUser is a cluster that implements the composite Party analysis pattern [Fowler96]. ServiceProvider and ServiceUser (or ServiceSubcriber) are composite Parties: they can be an Individual or a hierarchy of Organzations.

[6] A “cluster” is a component with a level of reuse that is higher and coarser grained than a class. It is frequently realized as a Mediator encapsulating the collaborations of a group of classes that participate in valid role-object interaction sequences to fulfill a business intent.

Within the Service Provider, are several organizations which may be implemented as one Party taking on the role of all three: Service Composer, Service Connector and Service Developer (see section 2.2 for a description of participants) . The Developer manufactures and prepares Services. The Service Composer aggregates and composes them into Service Offerings. The Service Connector connects the Service Subscriber or User to the Service Composer. The Service Provider will typically have these three roles. Sometimes these roles may be spread out among different companies or subsidiaries in the marketplace/industry.

There are a number of underlying, common patterns that flow through each business domain; that can be captured and generalized into a domain pattern. A framework can then be generated from this domain pattern, to cover the domain. Applications written in that domain would make use of this framework and customize it for each particular sub-domain. They would, in essence add the necessary business vocabulary pertinent to that specific domain. Therefore, generate frameworks from domain patterns. Domain patterns generate generic architectures. These can be molded into one or more (related , sub-)frameworks.

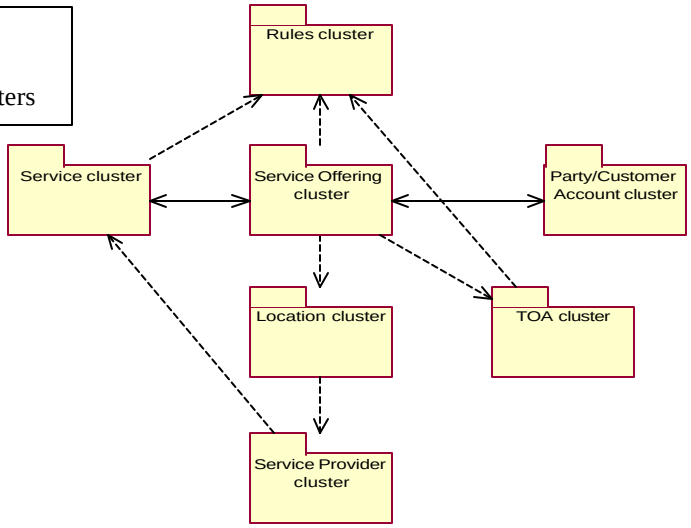
Writing individual Offerings without a generic context of Services that are constrained by Terms of Agreement (i.e., the rules that make up the package or deal “for a limited time only, we are having 20% discount if customers sign up by the end of this month in one of these packaged deals…”)

Businesses offer services according to some constraints (rules or Terms of Agreement wich the subscriber must agree to). These constraints tend to vary rapidly. Business need to quickly adapt by implementing these variations rapidly. Services may be dependent on location, time of offering, customer history, etc. These dependencies tend to change rapidly.
Information systems need to keep up with the pace of the business to retain a competitive advantage

8. Structure

This diagram shows all clusters involved in service provider pattern plus their dependencies to each other:

Figure 6: Dependencies between the clusters



The following figure shows a class diagram that demonstrates the interfaces and classes that participate in the Service Provider pattern:

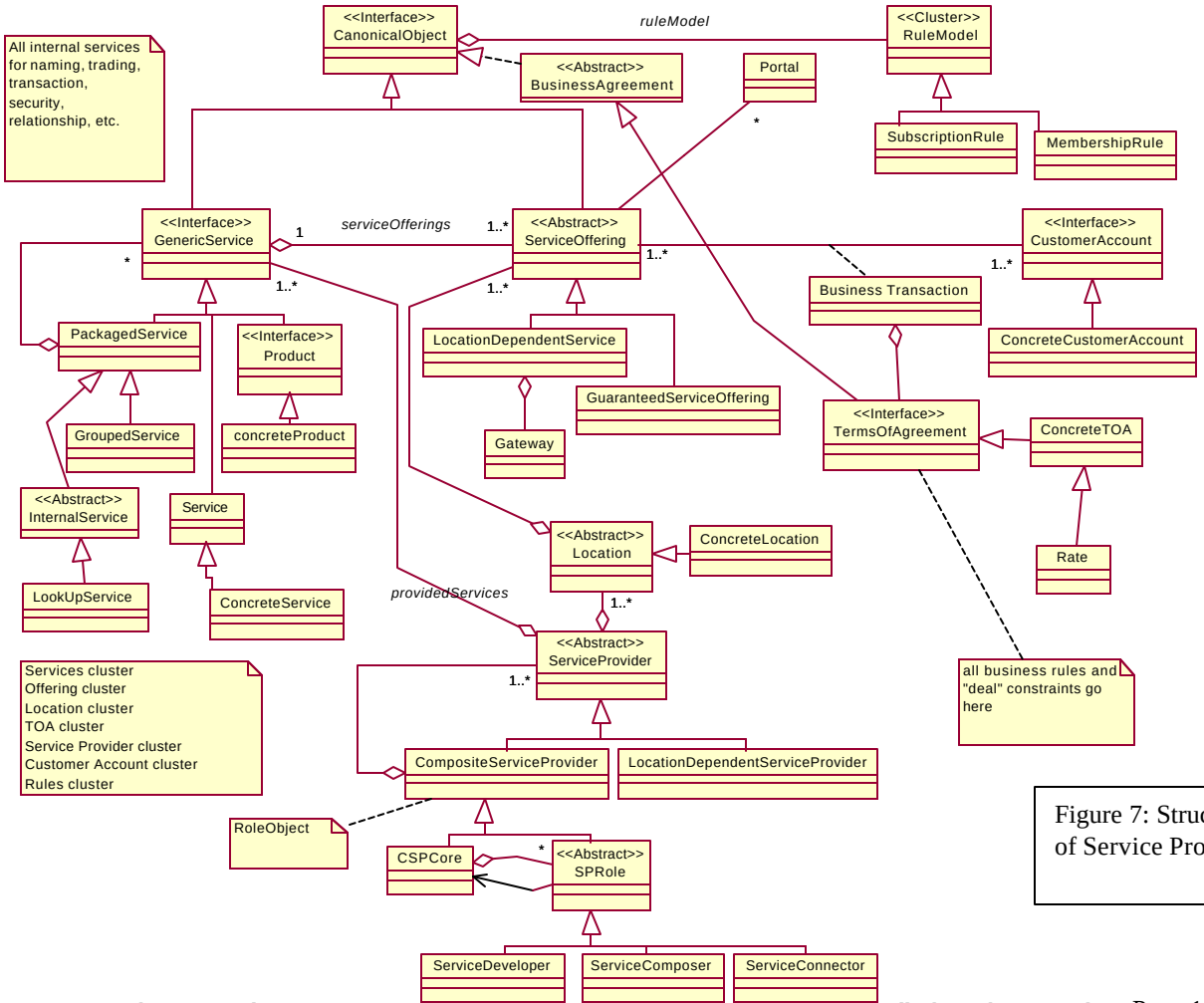


Figure 7: Structure of Service Provider

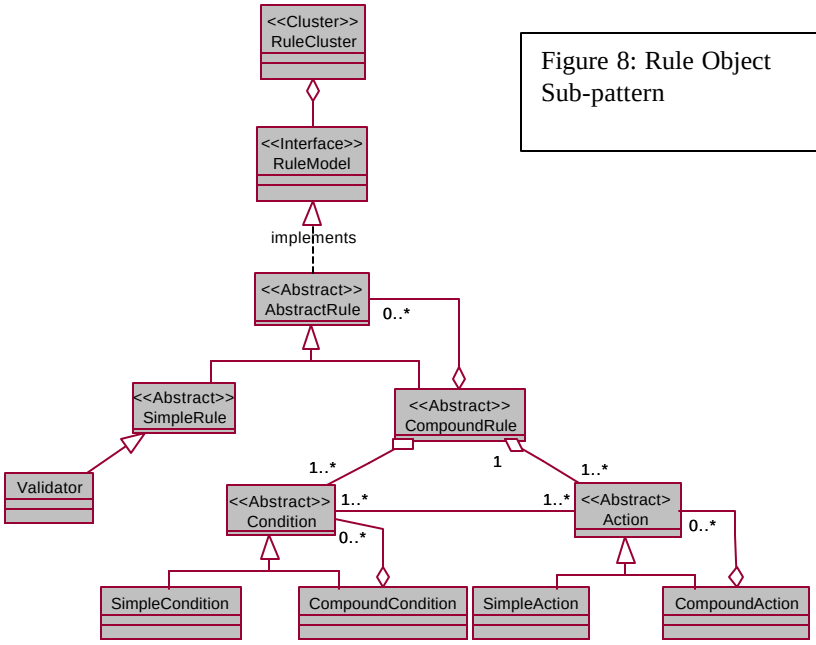


Figure 8: Rule Object Sub-pattern

Note: The application of a set of rules, may be optimized according to various Strategies. The CompoundRule has a Strategy for optimization of going through rules, assessing conditions according to the Assessor pattern and choosing the appropriate Action(s).

9. Participants (by cluster)

For participants by role, see section 2.3. There a more in-depth treatment of individual participants is given. Here, we take a look at the participants from the perspective of the cluster in which they reside.

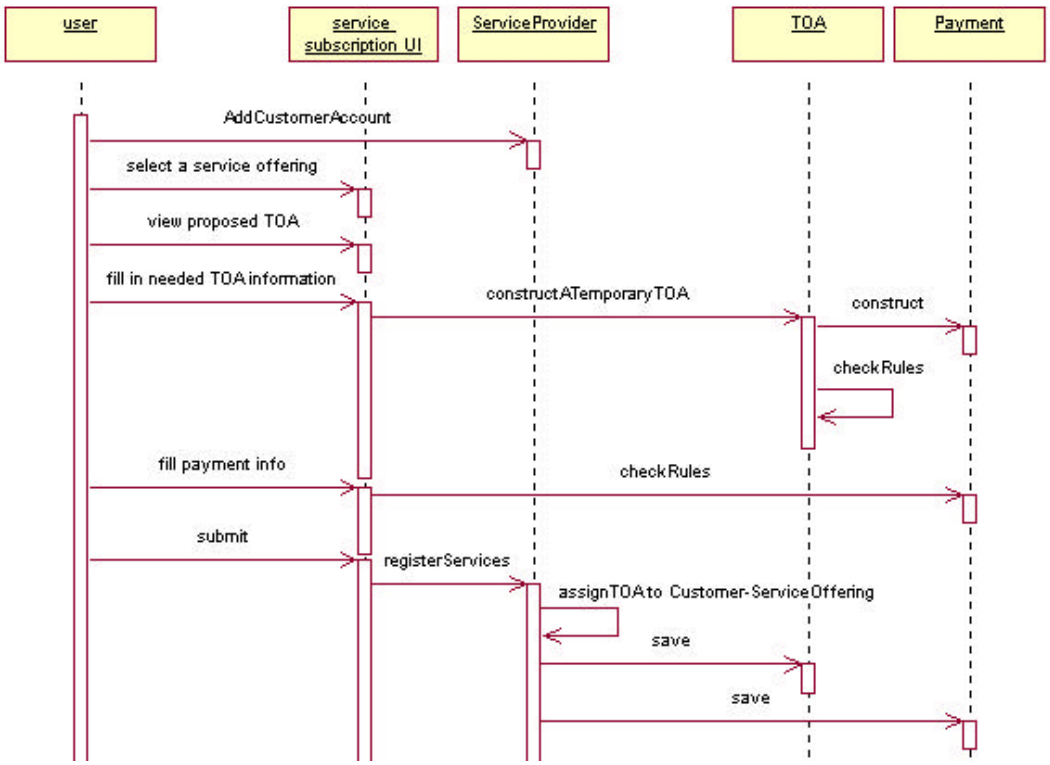
Looking at Figure 7, we recognize 5 main clusters within the Service Provider pattern:

- **ServiceProvider's cluster:** a service provider manages services and customer accounts. It has add and remove methods that add a GenericService to the list of available services and also methods for adding and removing a customer account.
  - **CompositeServiceProvider:** in addition to all characteristics of parent ServiceProvider class, a composite service provider can be composed of sub-services. For example a company, which has some branches and each branch, serves a special set of services, can be a composite service provider.
  - **LocationDependentServiceProvider:** a service provider which is offering location dependent services. It stores the service provider's location and can be made intelligent enough to offer those set of services to a user of in a specific location that can be registered.
  - **See section 2.2 on Service Developer, Service Composer and Service Connector (Presenter)**
- **Services cluster:**
  - **GenericService:** an abstraction for all different types of services in a generic and domain independent fashion. It typically has a name and a special rule object that is responsible for validity of the subscription process.
  - **Service:** the services of a corporation that are registered by clients. Many services are time constraint so a generic time constraint service is designed to handle them generically. A customer subscribes a service for typically a time interval and the service provider provides the customer with that specific service for typically a period of time. This class only contains attributes of a service in general regardless of the customer, it does not know anything about a customer. The 1-n association between Service and ServiceOffering means that a service (e.g., a Course at a University) can have several Offerings: Software Engineering may be offered during spring semester but not in the fall semester.

- **OfferedService or ServiceOffering cluster:** a service when is offered. The Subscribed Service or Service Offering will have its own attributes based on what the realization of the Service interface has to offer: for Course (As Service) we have Offering as Subscribed Service; and Offering will have an associated semester, units, teacher, room, prerequisites (the prerequisites will most likely be an override of its parent's generic prerequisites (I.e., each Course will have a set of pre-requisites; but each Offering may have additional pre-requisties including the ones from his parent. The OfferedService will also have additional rules based on its TOA.
  - **LocationDependentService:** some services are location dependent so a location dependent service object is designed to capture location attributes of a service. A rule model object that checks how a client can subscribe to a location dependent service typically manages the rules governing over a location dependent service.
- **Party or CustomerAccount:** An abstraction of customer account. It has a name and knows which services are registered for an account. It also has a MembershipRule object that validates the membership process. Typically a corporation may define several concrete customer accounts for different class or categories of clients.
- **TOA cluster:** an abstraction for the terms of agreements in a domain independent fashion. Many concrete TOA classes may exist for different types of valid agreements for different services. The CustomerAccount (the Client) will subscribe to the SubscribedService by accepting its attched ConcreteTOAs. We may need to model this as the SubscribedService having (aggregating) a set of TOAs (interface). Then, the actual ConcTOA will contain the actual rules and Terms of this Subscription. This is because, the egenral TOA hierarchy is a cluster of its own, so the intersection of the clusters is in the TOA interface.
- **Location cluster:** the location wherein a service is being offered. A location is an abstraction so that concrete classes such as Site or Address may exist to handle special location properties. A service provider typically needs to look up for locations and see which service offerings are offered at each or a special location. This is accomplished through the 1-n relationship between ServiceProvider and Location.
- **RuleModel cluster:** based on Rules as Objects pattern, defines the way rules are handled. See figure 9.
  - **What is not shown** is the rule optimization algorithm that is a strategy for determining what policy to use for rule order selection and firing: round-robin, heuristic searches, optimizations based on history, etc.
- **Transaction Cluster:** Consists of the ServiceOffering, the CustomerAccount information and the Business Transaction which encapsulates the Terms of Agreement to the transaction.
- **Service Location (Location)**
  - **Locations** offer their services via Gateways. An example would be a business –to-business server gateway or a small-business to internet gateway or a home network to a broadband external network.
- **Control Point**
  - Allow a central point of (remote) administration (or zero-admin for pervasive technologies) and security

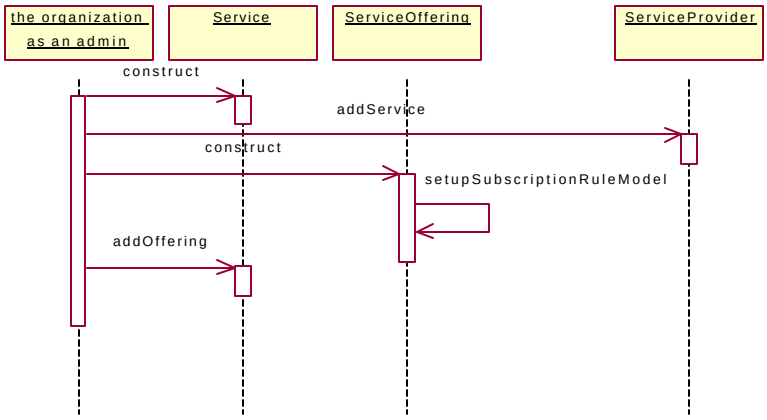
Collaborations

The following is a typical scenario of usage between the Service User and Service Provider. *Note* : This has been chosen as a representative collaboration related to a simple scenario from among many generic collaborations. Here are some representative collaborations for some of the main clusters.

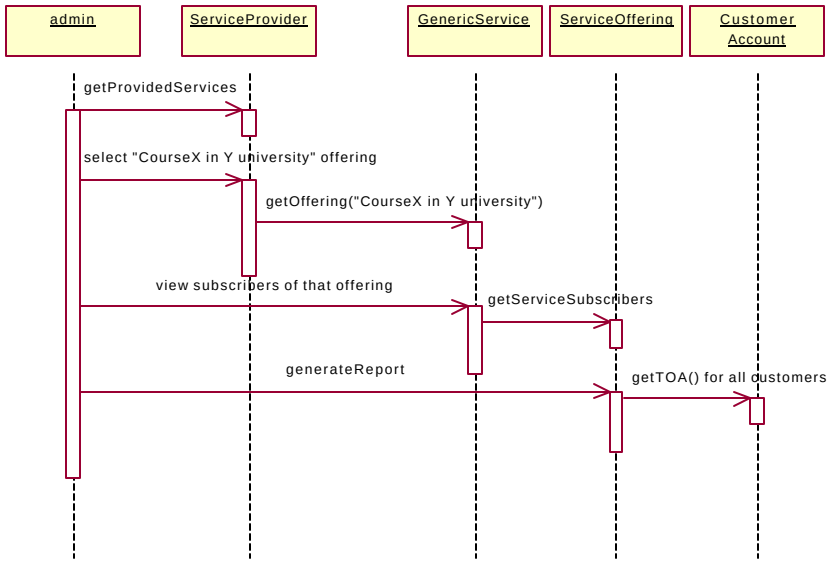


Here are some more typical collaborations based on clusters within Service Provider :

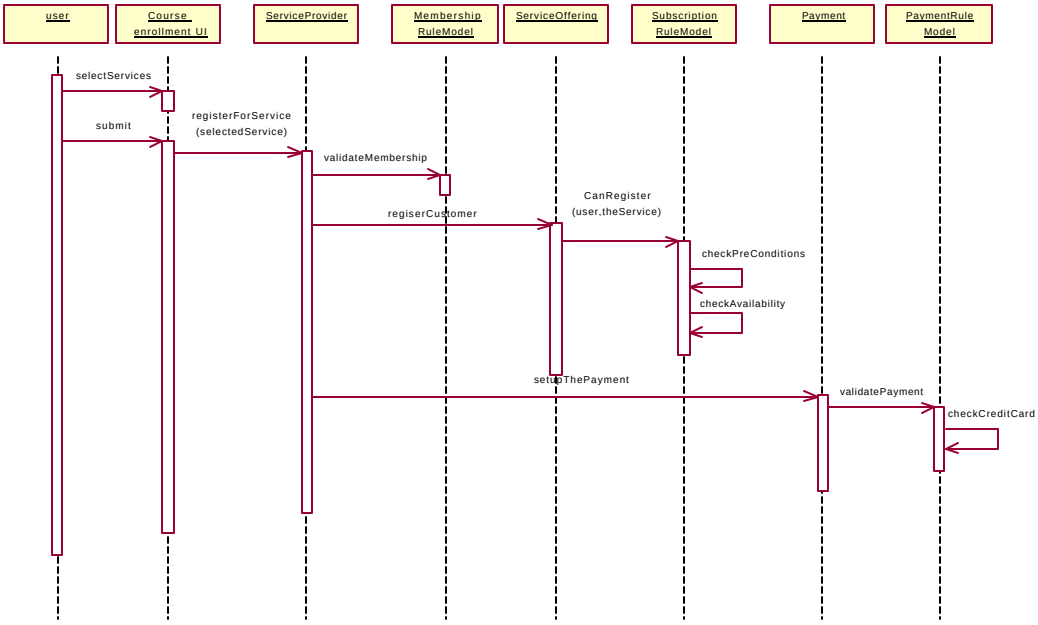
a)Service Offering Cluster Typical Scenario



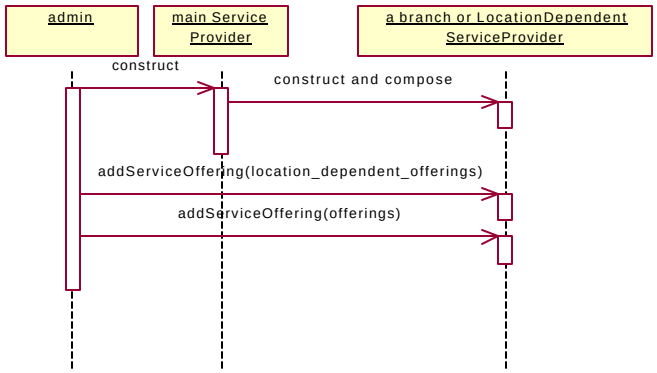
b) Reporting/ Request Typical Scenario



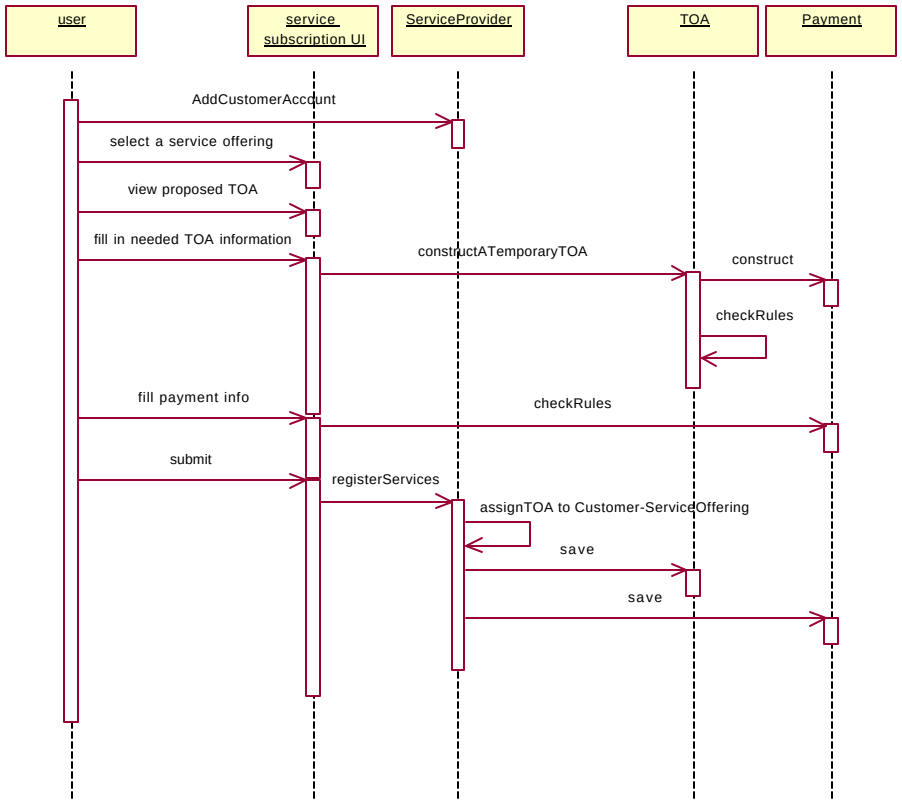
c) Rule Model Cluster Typical Scenario (instance is University Registration)



d) Service Provider Location Cluster Typical Scenario



e) Business Terms of Agreement Cluster Typical Scenario



Consequences

- A business application system is able to support business processes related to offering services to customers regardless of type or number of services. It is possible to compose or package services to define new service types.
- The relationship between the Service Provider, the offered or registered service and the customer is generalized so that none of them are exclusively tied to each other, leading to more maintainable code.
- New services can be easily added to system. A Service Provider is not impacted when we add a new Service Type. This is accomplished by abstraction of different services and that a service provider only talks to Service interface.
- Customers are able to register services easily. A customer object is not changed a lot to support registration of new service types. This is accomplished by abstraction of different services and that a service user only talks to Service interface via a ServiceProvider interface.
- Each customer or customer class is able to sign a term of agreement when a service is being registered. The term of agreement is the link between the service and the user of that service. All term objects adhere to a generic interface so working with terms is easier. For example a visitor class can visit all agreed terms of a customer to generate a total discount.
- A domain pattern contains other patterns at various levels and degrees: user-interface and business logic patterns. Other infrastructure patterns such as persistence mechanisms, ORBs, connectors to backend datastores, etc. are related but are really outside of the primary scope of domain patterns.
- Domain patterns reify use-cases; they embody the typical and valid object interaction sequences and scenarios of business processes.
- Seeing products and services; packaged services and “deals” as generic service types simplifies and unifies design decisions.
- Customization should take less time than re-writing code. Use this pattern in non-trivial, enterprise-wide systems.

Implementation

A ServiceProvider is typically a Singleton object. It may have Service listeners registered with it, that are notified of new services being added or removed, so a client application can be aware of most recent list of available services. ServiceProvider is typically an interface, but an AbstractServiceProvider class may exist to provide a default implementation and a user defined service provider class may derive from this default implementation.

GenericService is also an interface and Service and Product may be interfaces too, and may have default abstract implementations. The link relationship between Service, TOA and CustomerAccount can be implemented as a CustomerAccount having the list of its TOAs and the TOA knows which Services are registered using this TOA so CustomerAccount will know which services it has registered for.

Typically the OfferedService will have a list of TOAs or CustomerAccounts. A Registrar mediator class may exist to ease the process of service being registered for a customer and customer registering a service by a TOA. The ConcreteServiceProvider may itself be Registrar class or many Registrars may exist to balance the registration process between them.

Many concrete CustomerAccounts may exist to represent different classes or categories of customer account. A Corporation may decide to define a composition of customer accounts for this purpose.
Because typically many TOAs for different type of OfferedServices exist, a factory or builder cluster may be needed to handle this.

Finally the RuleModel cluster can be modeled differently. For example a CompositeRuleModel may exist that composes different RuleModel objects. An ActionConditionRuleModel may be needed if the sets of rules are complicated.

You may want to implement a portion or subset of the Service Provider. Or, your scenario requires less functionality and you do not want to have an over-kill in supplying functionality. Each sub-framework or sub-domain of the Service Provider can be used. Typically, the choice is based on the fact that the Service Provider is clustered (is composed of the Cluster stereotype). You may add detailed to the cluster (e.g., Location may or may not contain multiple locations, control points, transactions).

The Service Provider needs to have a number of internal services (such as CORBA Services) in order to provide naming, trading, security, transaction, etc., to its specific Services (e.g., Service Offerings) at given Locations. At each Location, there are Portals that allow access to the Service Offerings (Location-dependent Service Offering) through a Control Point which implements Authorization, Authentication and general Security and central point of administration (usually remote).

In addition, at each Location, the Service Provider will need a Gateway to make its Service Offerings available and accessible to the outside world and to Subscribers. It will act as an Adaptor (Pattern) to allow different protocols and interfaces to interpolate as seamlessly as possible.

Known Uses

CORBA Services, Jini and Java Embedded Server technology notions device and service registry and identification, various applications in many industrial sectors: order-entry, retail, billing, customer care, sales prospects, higher education, automotive industry, packaging, and in reality, most businesses; whether on the internet (“e-business”) or in traditional business contexts and scenarios.

Acknowledgements

The author would like to thank the JBF team at OOT, Ara Ebrahimi, Armond Avanes, Nooshin Hakimi, Behnaz Zolfaghari and Setareh Jalili for their efforts and insights in the development of JBF and for their suggestions and support.

References

[Arsanjani99;2] Ali Arsanjani, “E-business Web-development Best-practices”, Whitepaper.

[Arsanjani98], Ali Arsanjani, Object-oriented Technologies, Inc., “A Pattern Language for Business Rules Design and Implementation”, Whitepaper.

[Arsanjani95] Ali Arsanjani, Object-oriented Technologies, Inc. and Maharishi University of Management, “Rules as First-class Citizens of the Object Paradigm”, Whitepaper, 1995.

[Alexander77] Christopher Alexander et al., A Pattern Language, Oxford University Press, New York, 1977.

[Alexander79] Christopher Alexander, The Timeless Way of Building, Oxford University Press, New York, 1979.

[Beck96] Kent Beck, "Smalltalk Best Practice Patterns", Prentice-Hall, New Jersey 1996

[Buschmann+97] Pattern-Oriented Software Architecture, Prentice-Hall.

[Cunningham96] Ward Cunningham, "Episodes: A Pattern Language of Competitive Development", in [PLoP95]

[Foote96] Brian Foote and Joseph Yoder, "Attracting Reuse", PLoP'96 Proceedings.

[Fowler97] Martin Fowler, “Analysis Patterns: Reusable Object Models”, Addison-Wesley

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software." Addison-Wesley, 1995.

[Johnson95] Ralph Johnson, Dan Roberts, “Evolving Frameworks”

[Johnson96] Ralph Johnson, “Transactions and Accounts”, PloP2, Addison-Wesley.

[Johnson+88] Ralph Johnson and Brian Foote, “Designing Reusable Classes”, JOOP 1988.

[Johnson 92]Ralph Johnson , “Documenting Frameworks Using Patterns”, Proceedings of OOPLSA '92, Vancouver BC.

[Meyer87] Bertrand Meyer, “Object-Oriented Software Construction”, Addison-Wesley, 1987.

[PLoP94] Proceedings of PLoP-94 - "Pattern Languages of Program Design" published by Addison-Wesley in 1995.

[PLoP95] Proceedings of PLoP-95 - "Pattern Languages of Program Design" published by Addison-Wesley in 1996.

[IBMSF] IBM San Francisco Project, www.ibm.com/SanFrancisco

[Pryce] Nat Pryce, Abstract Session, Imperial College, London.

