A. S. BORANBAYEV

PRINCIPLES FOR DEVELOPING ARCHITECTURE AND DESIGN OF WEB APPLICATIONS AND INTERFACE IN INFORMATION SYSTEM

When architecting and designing an application, you are in the first phase of the software life cycle. There are usually many existing blueprints and design patterns that can be used as a starting point to build the application.

1. Overview

When designing an application, the advantage of using an existing blue print or design pattern is that they solve many issues common to application development. There are also many factors to take into account when planning the application. This includes performance, scalability, packaging and deployment restrictions.

One of the biggest promises of object-oriented development has been that of promoting code reuse. That promise is built on the idea that if you were to build generic objects, those could be used and reused. Some prime examples of such code reuse would be frameworks like SDF, etc.

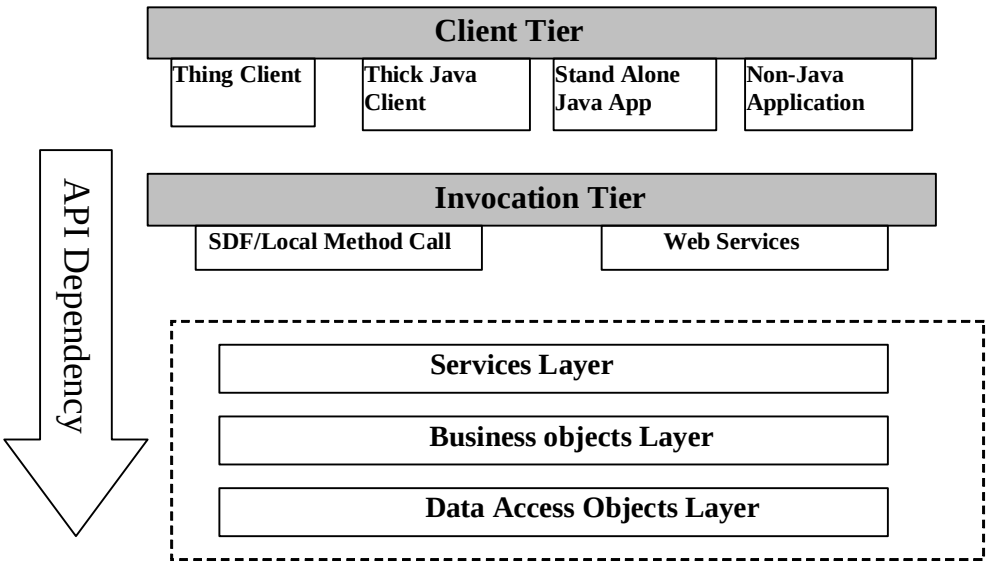
By Software Development Framework (SDF) I mean a programming language, standardized libraries, and any tools that are part of the normal development process. Software Development is about getting specified tasks done by recording appropriate directions which will be performed later by a mechanism.

The question that comes to mind is how to take that one step further and be able to reuse business functionality. Till so far, architects and designers have been able to contain business logic in a set of business objects that comprise the Business Layer. How do we package this business functionality such that different types of clients could reuse it? This article aims at addressing such concerns.

Layered Approach 2.

The first principle of creating business services is that of separation of layers [1]. A layer consists of entities that share similar responsibilities. For instance, the data access object layer contains objects that are responsible for providing access to various data stores. The diagram above depicts the various layers in a typical application. In order to eliminate circular dependencies, a lower layer never depends on a higher layer. For instance, the Services Layer should not accept HttpServletRequest as an input parameter. Doing so makes it dependent on a type of Invocation Layer. A higher layer would always depend on the functionality provided by the lower layer. Based on this principle the Services layer should not contain any of the following:

- HTTP/ Servlet / Swing API.
- Java remoting or UI technology specific constructs.
- JDBC constructs.



3. Interface Driven Design and Development

The second principle is that of Interface driven design and development. It states that software designers should define clear interface specifications for components [1]. All collaborations amongst components within a layer and across layers should be driven by these interface specifications or contracts. The big advantage of this principle is that it forces designers to assign specific responsibilities to components, and it helps to identify the various collaborations amongst components. An additional advantage is that implementations of these interfaces can evolve over time, without disrupting the other dependent components.

The Factory design pattern is a good way to cleanly separate the instantiation of an implementation of an interface, from the use of the interface itself.

4. Business Services

A service consists of a collection of components that work in concert to deliver the business function that the service represents. It is an abstraction layer on top of the functionality exposed by business objects, which in turn are fine-grained and are solely focused on implementing the individual steps of a business process workflow. Services orchestrate various operations across multiple business objects, thus promoting the reusability of business objects. Any extensive orchestration of service operations by the calling clients should essentially be exposed as a coarse-grained operation on the service itself. At times, services may appear as a pass-through layer but they provide the flexibility of being able to create orchestrations of the business objects, in future.

Services are process-centric, in that they map to the business functions that are identified during business process analysis. They encapsulate business functionality and for all practical purposes can be treated as black boxes.

- All validations are performed within the operations
- All entity relations are handled within the operations

Invocation of services is stateless, as service requests don't depend on each other. Services may be fine- or coarse-grained depending upon the business processes. However fine-grained calls to a service, over a network can increase network chatter and clog the network.

5. Operations on Business Services

The operations on a service can be exposed as:

1. Local calls within an application server.
2. Local calls in stand-alone / desktop based program.
3. Local calls in batch program invoked by a shell.
4. XML based Apache soap, axis etc for Java based providers and consumers.

The types of input and output parameters of the operations on a service are restricted. This is mainly for the benefit of the various channels accessing the service. The following JAX-RPC restrictions apply to the input and output parameters.

6. JAX-RPC Restrictions

JAX-RPC (Java API for XML-based Remote Procedure Calls) allows invoking from a Java application a Java based Web Service with a known description while still being consistent with its WSDL description [2] [3].

JAX-RPC uses SOAP and HTTP to do RPCs over the network [4]. RPC stands for remote procedure calls, and is used for making procedure or function calls and receiving the responses. The SOAP specification defines the necessary structure, a convention for doing RPCs, and its corresponding responses. The RPCs and responses are transmitted over the network using HTTP as the primary transport mechanism.

From an application developer's point of view, an RPC-based system has two aspects: the server side (the Web service) and the client side. The Web service exposes the procedures that can be executed, and the client does the actual RPC over the network.

The Web service environment is based on open standards such as SOAP, HTTP, and WSDL. It is therefore possible that a Web service or a client wasn't developed using the Java platform. However, JAXR-RPC provides the mechanism that enables a non-Java client to connect to a Web service developed using Java platform, and vice versa [5].

The runtime protocol works as follows:

1. A Java program invokes a method on a stub.
2. The stub invokes routines in the JAX-RPC runtime system.
3. The runtime system converts the remote method invocation into a SOAP message.
4. The runtime system transmits the message as an HTTP request.

The consumer becomes aware of the provider services through the machine readable WSDL file. Construction of the consumer relies on the WSDL from the provider.

JAX-RPC is WS-I compliant and is widely accepted among the industry. WS-I stands for "The Web Services Interoperability Organization". WS-I is an industry consortium created to promote inte-roperability among the stack of web services specifications [6].

The following are the JAX-RPC supported Java types:

- One of the Java primitive types (boolean, byte, short, int, long, float, double)
- A subset of the standard Java classes (as specified in the J2SE APIs)

1. java.lang.String
2. java.util.Date
3. java.util.Calendar
4. java.math.BigInteger
5. java.math.BigDecimal
6. javax.xml.namespace.QName
7. java.net.URI

- A Java array with members of a supported JAX-RPC Java type.
- A service specific exception; declared in a remote method signature must be a checked exception. It must extend java.lang.Exception either directly or indirectly but must not be a RuntimeException.
- A JAX-RPC value type is a Java class, whose value can be moved between a service client and service endpoint. A Java class must follow these rules to be a JAX-RPC conformant value type:

1. Java class must have a public default constructor.
2. Java class must not implement (directly or indirectly) the java.rmi.Remote interface.
3. Java class may implement any Java interface (except the java.rmi.Remote interface) or extend another Java class.
4. Java class may contain public, private, protected, package-level fields. The Java type of a public field must be a supported JAX-RPC type.
5. Java class may contain methods. There are no specified restrictions on the nature of these methods. Refer to the later rule about the JavaBeans properties.
6. Java class may contain static or transient fields.
7. Java class for a JAX-RPC value type may be designed as a JavaBeans class. In this case, the bean properties (as defined by the JavaBeans intro-spection) are required to follow the JavaBeans design pattern of setter and getter methods. The Java type of a bean property must be a supported JAX-RPC type.

7. Configuration and Deployment of Business Services

Services are independent of the way they are deployed. Specifically, in an SDF deployment setting, Services should not rely on the Proxy Servlet to initialize other framework-related services like the DBConnectionProvider, etc. The services and the related business objects should take that fact into account and use alternate API to initialize and access such services.

8. Exception Handling in Business Services

An exceptional condition prevents the continuation of an operation that has been called on a service. At that point control is relegated from the service back to the client. The service then provides adequate information on the exceptional condition. The following describes how exceptions should be handled in the services layer:

1. Each service operation will throw one and only one exception. This exception relates directly to the service domain. For instance, the Enterprise Alerts service throws an 'AlertsException'.
2. The exception object will support error codes, their corresponding messages, and nested exceptions.
3. Each layer underneath the service layer is responsible for handling its own exceptions. A higher layer will not directly deal with exceptions that originate in a lower layer. Exceptions will be handled in the layer in which they originate.
4. Each exceptional condition will have a corresponding error code and a detailed message. Upon catching an exception, assign it the appropriate error code and message, and then encapsulate it in a layer specific exception object. This object is the one that will be thrown back to the calling layer.

For instance an object in the Data Access layer, upon encountering either a CompanyDataException or a CompanyConnectionException will assign it the appropriate error codes, detailed messages, and will then encapsulate it in a 'MyCustomDataLayerException' object before throwing it back to the business object layer. The Business Object layer in turn will wrap the exception in a 'MyCustomBusinessObjectLayerException' object before throwing it back to the services layer. Finally the Services Layer will wrap this exception in a service domain specific exception (described earlier), and throw it back to the client [7].

9. Ensuring Backwards Compatibility for Java Clients

The service interface is a contract between the service provider and the consumers of that service. Once the interface has been published, any changes to it will make its existing clients incompatible with the service. However not all changes have such an affect. The following guidelines should help protect existing clients from any evolution in the service:

- The best way to ensure backwards compatibility is to always extend an interface. That cleanly separates the contracts between the existing and the new clients.
- It is a good idea not to modify or remove operations from a published interface. The same applies to objects that are published as input/output parameters, and exceptions, on the operations of a service.

10. SDF and Code Reuse

It is asserted that the primary goal of development frameworks is to manage complexity. Closely related are the requirements that code be maintainable, extensible, and implementable by separate programmers.

It is important that code be verifiably performing as specified. It is also desirable that code be easily created quickly. It is then argued that fundamental to any attempt to manage complexity is a strong and extensible type system. It is in no way asserted that a strong and extensible type system by itself guarantees an excellent SDF.

- Code Reuse => Parameterized Functions => Interface Definitions => Type Validation => Strong Typing.
- Data Structures => Per Structure-Instance Methods => Classes.
- Predictable Behavior => Object References.

Code reuse is the bedrock of reducing complexity. If the same thing is done twice, then it should be done in one place, so that improvements in the process need only be done once. Most modern languages encourage code reuse.

A parameterized function allows code to perform the same actions on different data. The concept of parameterization has been extended in some languages to allow parameters to determine what the function does.

An ideal SDF automatically detects code reuse, and allows the reduction of repeated code to a single parameterized function without incurring any hidden costs.

Every non-trivial parameterized function can only successfully perform its work on a subset of the possible data-objects that could be used as parameters. The definition of what work is performed, what subset of objects may be appropriately used as parameters, and what qualities any returned objects can be expected to have is called the Interface.

11. Conclusion

In this article, I have talked about business functionality reuse, code reuse, and about guidelines for creating Business Services, when developing Web Applications. I have showed how to package the business functionality such that different types of clients could reuse it. I have addressed important architecture topics and development steps that one should consider in a J2EE project. The information is taken from real-world experiences, and is intended to help developers build J2EE systems.

REFERENCES

1. *Boranbayev A.S.* Reference Architecture for Web Applications // Доклады национальной академии наук Республики Казахстан. 2007. №5.

2. WebService, from Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Web_service

3. Web Services Activity - http://www.w3.org/2002/ws/

4. *Boranbayev A.S.* Optimal Methods for Java Web Services // Известия национальной академии наук Республики Казахстан, серия физико-математическая. 2007. №5.

5. Chapter 11: Working with JAX-RPC from the book «Java APIs for XML Kick Start» by Aoyon Chowdhury and Parag Choudhary, published by Sams Publishing.

6. Web Services Interoperability, from Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/WS-I

7. *Boranbayev A.S.* Development of E-business websites using a multi-tiered architecture and J2EE // Материалы VI Казахстанско-Российской международной научно-практической конференции "Математическое моделирование научно-технологических и экологических проблем в нефтегазодобывающей промышленности". Астана, 2007. С. 87-91.

Резюме

Мақала Веб косымшаларды жасауға және жобалауға арналған. Бұл бағдарламалық жүйелердің өмірлік циклінің бірінші фазасы.

Резюме

Статья посвящена разработке архитектуры и проектированию Веб приложений. Это является первой фазой цикла жизни программного обеспечения.

UDC 681.3

L. N. Gumilyov Eurasian National University

Поступила 13.12.07г.