Sovereign Value Object Design Pattern

Zdravko Roško
University of Zagreb
Faculty of Organization and Informatics, Varaždin, Croatia
zrosko@yahoo.com

Abstract. *Distributed systems architecture (CORBA, DCOM, EJB, etc.) and currently recommended design patterns, do not successfully help to solve the problem of data transport from client to server, data usage, data integrity, data transformation, internal programming techniques, program response time, methods signature flexibility, etc. From an integrated architectural point of view, proposed Sovereign Value Object (SVO) design pattern is a reusable component which helps to solve many of the complexities which currently exist.*

Keywords. Design Pattern, Value Object, Java, Service Oriented Architecture (SOA), component, method signature, property container, façade, Model View Controller (MVC), Core J2EE Design Patterns, XML, SOAP, Result Set, Copy Helper, name-value pairs.

1. Introduction

Sovereign Value Object is a design pattern response to the industry need to simplify application programming, improve performance, encapsulate data on a reusable way, and make the programs less error prone. Throughout the history of distributed systems and application programming, there were no recommendation specification or design pattern available to address the overall data encapsulation issue. IBM *Copy Helper* solution for transfering data in the Component Broker distributed envirnoment, was the first significant attempt to address the data transfer between distributed objects. A known issue with CORBA and EJB distributed systems in the past was the need to invoke network transport for each attribute *setter* and *getter* call from client to the server. Service method signature, reusability of once encapsulated Value Object [1], and many other issues are solved by using *Sovereign Value Object* design pattern. Recently published specification (November 2006) of Service Data Objects (SDO) by BEA Sysems, Inc., is the most complete specification in regards to data programing architecture, but it still does not specify the potential usage, and effective design of SDO. The author has been using Java SVO for 8 years and has improved its functionality over time.

2. Value Object

The idea of the Value Object (Sun Microsystems 2001) design pattern was to solve a performance issue with clients need to exchange data with enterprise beans. The client invokes business object's *getter* and *setter* methods for each field, which are potentially remote and cause network overhead. Use of Data Transfer Object (variation of Value Object) was the solution to the issue. A single method call is used to send and retrieve the Data Transfer Object. When the client requests the enterprise bean for the business data, the enterprise bean can construct the Data Transfer Object, populate it with its attribute values, and pass it by value to the client [2]. The major limitation of Value Object is the need to use many other patterns in connection to the Value Object.

Example 1. Implementing the Data Transfer Object Pattern - Transfer Object Class

```
public class ProjectDTO implements
java.io.Serializable {
   public String projectId;
   public String projectName;
   public String managerId;
   public String customerId;
   public Date startDate;
   public boolean started;
   public boolean accepted;
   public Date acceptedDate;
   public String projectDescription;
}
```

The issue with Value Object was defined too narrowly. This usage model of Value Object could be characterized as *transient,* since the Value Object is used for secondary reasons. *Sovereign* usage model, in contrast, uses a Value Object as a primarly data container throughout the complete life cycle of data, from creation to persistence or removal.

Sovereign Value Object is a pattern around which a number of transient functions will revolve, such as validation, mapping, caching, logging, etc.

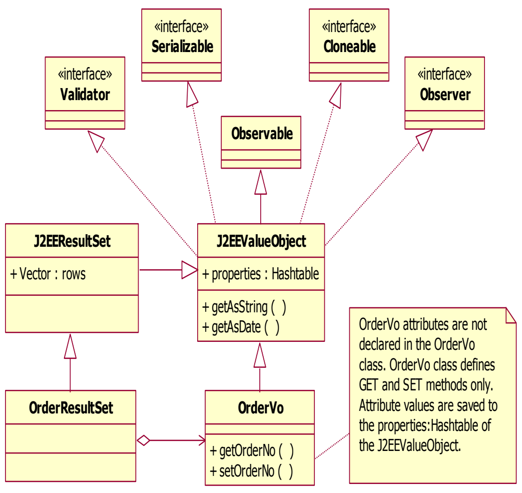
3. Sovereign Value Object

SVO appears as single row (contains maximum one instance of any type of attribute) and inherits from J2EEValueObject. SVO can contain a result set (multiple instances of any type of attribute) by inheriting from J2EEResultSet class.

3.1. Design

While designing the interface to its business methods, application development needs to define a SVO for each significant business entity and use it in all tiers within the application. When multiple entities are needed (SQL query), SVO result set needs to be created and used independently from the type of the persistence.

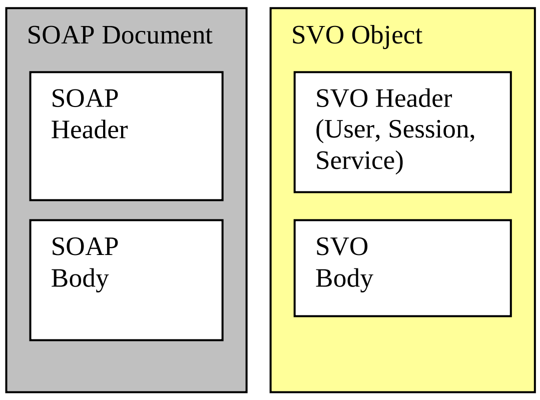
Figure 1. Sovereign Value Object design



3.2. Structure

SVO inherits property container and implements *getters* and *setters* for each of its attributes. Like a SOAP document, SVO object has a header part to keep the user, session and information about class and method (service) to invoke in the case the SVO is used as *command* [3]. SVO inherits *toXML* and *fromXML* methods to enable transformation from XML to Java object and vice versa. Transformation from SOAP document to SVO object and vice versa is supported to enable Service Oriented Architecture (SOA). SVO contains name-value pairs of preferably *String* data fields, but it also supports name-object pairs. Nested SVOs are supported to enable complex structured data, by having a reference to another SVO. SVO inherits standard conversion methods such as *getAsDate*(), *getAsStringOrEmpty*(), which ensures no *NullPointer* exceptions while using SVO. Implementation of SVO converts between *String* and any data types. SVO is *Cloneable* which makes it possible to perform *deep* copy of its data when needed. As *Serializable* object SVO can be transported through the network or to the file.

Figure 2. Sovereign Value Object and SOAP



4. Sovereign Value Object applicability

SVO is a complete data encapsulation and is used on the client, transport, business logic, data access, and services layers of a typical application. Having SVO as a part of the infrastructure core classes, the application SVOs need to inherit the *J2EEValueObject* or *J2EEResultSet* and implement the *get* and *set* methods only.

4.1. Model View Controller (MVC)

SVO can be used as a model (*Observable*) component of the MVC architectural pattern. The same model can be transferred to the next layer (ProxyFacade) or could be used for other applications processing needs (validation, local cache, etc.).

4.2. Interface to methods

Using SVO as the only input or output method parameter, ensures the method signature stability from any future changes. Having stable interface is a must for today's applications. SVO has unlimited capacity for accepting any kind and any number of attributes. Changes to the number and types of the input or output data parameters to a business object method, does not mean change of the method interface if using SVO.

4.3. Validation

Data integrity, field edits, cross edits and other types of validation while using SVO as data holder is made simpler. Sending SVO objects to a Rule Engine, Workflow Engine or Validation service is supported, since SVO is designed as a bean with *setter* and *getter* methods for its data attributes manipulation.

4.4. Command design pattern

SVO is an implementation of the Command [3,4] design pattern. Having an *execute* method, SVO is capable to start the service as described in its header. SVO is for client to use it locally, but it actually executes within the remote server, transparent to the client. Command pattern is used to encapsulate Façade business method calls, such as *placeOrder*, *transferFunds*, etc. The client interaction with a command SVO is simple. Once the client instantiate the SVO, it simply sets attributes onto the SVO, until the command contains all the data required to execute a specific business method.

4.5. Data caching

By implementing *Observable* and extending *Observer*, it is possible to store SVOs on the client or server cache for later retrieval. SVO result sets can be cached at the persistence tier, and monitored for changes which are reflecting the data stored within SVOs at the cache.

4.6. Services

SVO passed to a different type of service infrastructure components serves as a source of data to be used for logging, security check, workflow process, session, etc.

4.7. Persistence mapping

By using the same standard *data dictionary* (metadata or origin names) on the client and on the server layer, SVO enables persistence of its attributes without programming intervention. Data values available at the Data Access Object are mapped to the Persistence using metadata of the persistence layer. Also, it is possible to use SVO for validating the fields length against the persistence fields definition.

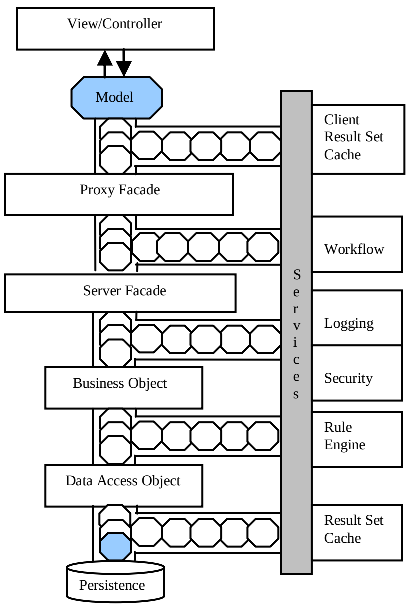
4.8. Result Set

J2EEResultSet is a practical way to marshal tabular data from a JDBC ResultSet or other data source, down to all application layers, without the overhead of converting the data to other kind of objects. J2EEResultSet is used to encapsulate the data from many kinds of data sources such as RDMS, Mainframe Application Server, ERP, Messaging Server, File System, etc. Data is transferred to the calling program and connection to the data source is not maintained. This way, any strategic change of the data source does not affect the business logic, or any other part of the application programming code. Replacing RDMS with ERP should not cause a single line of a code change if SVO is used in connection to Data Access Object design pattern which is an implementation of Strategy [3] design pattern.

4.1. JSP Bean

SVO can be used as a JSP *bean* or *bean list* to act as a model in the JSP architecture context.

Figure 3. Sovereign Value Object usage context



5. Sovereign Value Object implementation

Example 2. Implementing the base infrastructure Sovereign Value Object Class

```
public class J2EEValueObject extends Observable
implements Serializable, Cloneable, Obeserver,
Validator, XMLConvertable {
   public Object clone();
   public void deleteProperty(String name);
   public J2EEValueObject fromXML(String xml);
   public Date getAsDate(String name);
   public void set(String name, String value);
  public J2EEValueObject execute();
 public void update(Observable o, Object arg);
}
```

Example 3. Implementing the application Sovereign Value Object Class

```
public class ProjectSVO extends J2EEValueObject {
   public String getProjectId();
   public String getProjectName();
   public Date getStartDate();
   public void setProjectId(String value);
   public void setProjectName(String value);
   public void setStartDat(Date value);
}
```

5. Related Patterns

SVO can be used by Proxy, Front Controller, Business Delegate to reduce the coupling between the presentation-tier clients and business services.

Command design pattern hides the destination service which handles the data inside the SVO.

Façade needs to have a simple and flexible interface by using SVO as its only input or output parameters.

Business Object (pass through type) and Data Access Object keep its data inside SVO while accepting, manipulating, and passing it to the other destinations.

6. References

[1] Core J2EE Pattern Catalog, VO

[2] Core J2EE Pattern Catalog, DTO

[3] Eric Gamma. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company., 1995.

[4] Mark Grand. *Patterns in Java.* John Wiley & Sons Inc., 1998.