An approach to data validation based on lifecycle-bounded metadata

Vadim Surpin

The Institute for information transmission problems of the Russian Academy of Sciences (Kharkevich Institute),

Moscow, Russia

vadim@iitp.ru

Abstract – **Data validation is known to be the task performed in almost every business application and occurs through almost all levels of modern multi-tier application. Being a crosscutting concern validation requires some extra effort to make sure that it works right and consistent in all tiers thus increasing time to test the application and possible number of application bugs. The article describes an approach to describe complex lifecycle-bounded validation in a declarative manner making it reusable through the application.**

Data validation is one of the most common tasks in business application. Validation is a process of checking that data conforms to constraints applied to it and producing a list of validation messages that clearly describe failed checks. Problem of data validation attracts more attention since the time when Model-View-Controller (MVC) [1] become a standard for building modern application architecture. Such layered code separation allows to build more secure and scalable software but leads to undesirable code duplication between layers. The validation code is that one that being duplicated. It is being used in all application layers from model to presentation making a problem to coordinate validation rules on different levels.

Last few years there are several attempts to make data validation some more formal description and establish a standard that describes data validation approaches. There are several standards one of which is JSR303 [2] if we look in the area of Java technology [3]. The standard has a working implementation, the one that was a prototype for the standard. It is open-source project Hibernate Validator [4] from the Red Hat company. High page rank of the project official website in Google shows interest of software developers to the data validation problem. Here are some data validation approaches referenced in the JSR303 standard and implemented in Hibernate Validator project.

The standard based on idea of applying constraints to object's fields and offers to look at that constraints as metadata bound to thy fields. The most common constraints such as "not empty", "not large than N symbols", etc. are supplied as out-of-the-box implementation. Software developer may add some more complex field constraints that conforms to the standard. Every constraint may be associated with one or more groups that allows to validate object against several validation sets.

The approach described is suitable to accomplish simple validation tasks when validation constraint set isn't vary very much. Such as in the case of validating domain model objects before they're being send to persistence layer and database. This peculiarity is due to tight integration between Hibernate Validator that was the prototype for the standard and Hibernate [5] object-relation mapping solution. Being good at that field the standard doesn't address validation issues that exist in more complex workflow-based scenarios where typical tasks are:

- Constraints on fields that depend on each other
- Constraints on associated objects or object graph
- Constraints that depend on the lifecycle stage of the business object
- Constraints that depend on context parameters

To deal with first and second problems it is enough to allow object level validation and give a developer possibility to implement validation logic as a program code. This will give also an opportunity to validate complex dependencies between object fields and deep relationship between associated objects.

The lifecycle dependent validation is a common case in application where two or more users work on the same data. In such a case data travel from one user to another in accordance with application workflow, the data contained in the same business objects grows along it's way in workflow so it's consistency depends on the phase of the lifecycle. This is the most common case for every quite complex business application. To address this issue clear principles of business object lifecycle management should be described and implemented in an application architecture.

The MVC architecture states that at least three general classes of objects exist:

- Data access objects or domain-model objects (Model)
- Business logic objects (Controller)
- User interfaces objects (View)

From the perspective of high level system architecture interaction between these classes may be presented as on the UML[6] diagram Fig. 1

The diagram shows that all data changes only when it passes throw the methods of controller object that implements business operations of the system. This means that all object lifecycle-management occurs when data goes across the border between View and Controller layers where the View layer initiates object state change based on the user request and controller performs the requested business operation. That's why passing the boundaries between View and Controller layers is a good place to perform object validation. It solves at least three problems from the validation field:

1. Assure that controller receives correct data that won't corrupt data storage integrity if malicious data will be send by the user.
2. View layer is able to show validation messages informing the user about mistakes in just entered data in context of the requested operation thus giving a developer to supply more specific and clear validation message compared to ones that can be produced without this operation-context dependency.
3. Forces consistency between data check on View and controller layers by using the Don't Repeat Yourself (DRY) [7] principle eliminating code duplication.

The approach suits well to the modern application architecture where system modules have to be terminated by well defined interfaces and the module implementation is a "black box" for the cooperating party. An amount of modern programming languages have a notation of interface in their syntax, e.g. Java language interface syntax may look as follows:

```
@Remote public interface
BusinessOperationsRemote {

        void doSomething(T param);

}
```

The example of a simple business component interface that uses Enterprise JavaBeans 3 (EJB3)[8] technology is shown. Here are it's

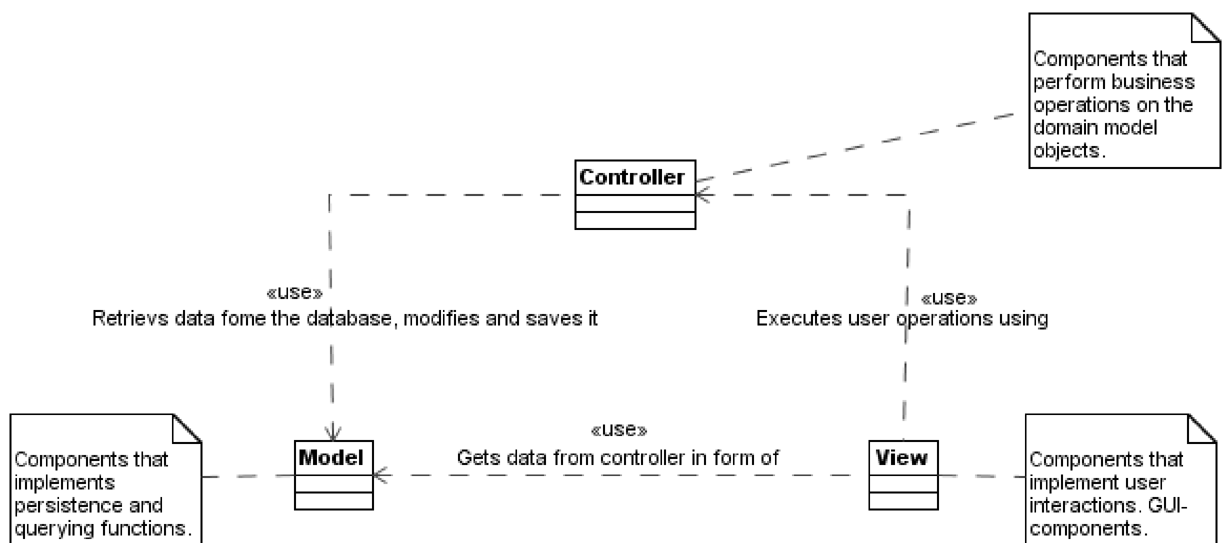

Fig. 1 Interaction between MVC objects

meaningful parts:

1. @Remote – so called “annotation”, an implementation of metadata facility from Java technology. The annotation means that the annotated interface belongs to a business object which lifecycle is managed by an EJB container and the interface methods are accessible remotely by network calls.
2. void doSomething(T param) – business method signature that states the method returns no result and accepts a parameter of type T.

When the component implementation is accessible only to the container which manages that component, it’s interface is visible to both component implementation and client from the view layer which calls business its methods. Keeping that in mind it becomes clear that the interfaces are a proper place to put method parameters validation metadata. The metadata take a form of @Validator annotation on the doSomething() method as follows:

```
@Validator(implementation=DoSomethingValidator.class)

void doSomething(T param);
```

Referenced by the “implementation” attribute class DoSomethingValidator implements the doSomething() method parameters validation logic:

```
public class DoSomethingValidator implements InputValidator {

	@Override

	public List<ValidationMessage> validate(Object… params) {

	…

	}

}
```

This class bytecode should be available both on controller and view layers and can be shipped with business interface description in the same deployment unit. Having access to the BusinessOperationsRemote business interface view layer can simply invoke parameter validation just before the method call:

```
List<ValidationMessage> messages = validator.validate(BusinessOperationsRemote.class, “doSomething”, param);

if(messages == null || messages.isEmpty()) {

	businessOperationsRemote.doSomething(param);

} else {

	//Show validation messages to the user

}
```

The solution described easy integrates with a JSR303 standard-compliant validation using it’s feature to set up validation groups for each object property. These groups may be just fully qualified names of BusinessOperationsRemote interface methods. Such choice of group naming has an advantage of hiding internal object lifecycle from the interfaces client describing transitions between lifecycle phases only in terms of business operation invocations. This gives view layer object only the required knowledge about object lifecycle and removes the need to specify validation groups at view layer. Spreading such information between layers causes numerous errors due to module miscoordintaion during system development process. Miscoordintaion is impossible when using interface level metadata because it can be detected on the compilation stage by compiler error messages.

So the approach described solves the problem of complex object validation in the process of object lifecycle transition process in the way clear to the developer. It coordinates check being performed at differed layers of a multi-tier application and refactoring-friendly since it describes all it’s metadata using language syntax available to the compiler. These simplifies software development that involves data validation facilities (a great part of modern software) and decreases number of hardly testable logical errors in the application design that occur when validation code at different layers gets miscoordinated.

REFERENCES

1. Design Patterns: Model-View-Controller, Java Blueprints, http://java.sun.com/blueprints/patterns/MVC.html
2. Java Specification Request, JSR303, http://jcp.org/en/jsr/detail?id=303
3. Java Technology, http://java.sun.com
4. Hibernate Validator Project, https://www.hibernate.org/412.html

5. Hibernate ORM, https://www.hibernate.org/344.html
6. M. Fowler "UML Distilled: A brief guide to the standard object modeling language", Addison-Wesley Professional, 2003.
7. William Crawford, Jonathan Kaplan "J2EE Design Patterns", O'Reilly Media, 2003
8. Enterprise JavaBeans Technology, http://java.sun.com/products/ejb/