APPLICATION OF DECLARATIVE PARADIGM IN ENTERPRISE IS

David Klika[2,+], **Josef Smolka**[1,2,*]
[1]Faculty of Nuclear Sciences and Physical Engineering CTU in Prague, [2]SEFIRA spol. s r.o.
[+]klika@sefira.cz, [*]smolkjos@fjfi.cvut.cz

ABSTRACT:
This paper introduces one of possible applications of declarative paradigm in development of an enterprise information system. Advantages of the approach are shown on examples based on a complex system developed for an electricity company.

KEYWORDS:
declarative programming, enterprise system, metadata, Groovy, Java

INTRODUCTION

When designing an enterprise information system, developers should consider not only the costs of initial development itself, but also the costs of maintenance and expansion of the system during its lifetime. If whole business logic is hard-wired into the system source code, every future change must be made by a programmer and affected part must be redeployed to a runtime environment. This redeployment could be a real headache because of common policy in corporate environment, i.e. a supplier of the system cannot usually perform the redeployment himself, but has to ask an application administrator to perform the task. In case of a serious error, delay between report of the error and deployment of the fixed version can be unpleasant. This is what we call a closed system from the perspective of the maintenance and expansion. On the other hand, in open system, core business logic can be expanded and fixed at runtime not only by developers, but also by administrators if it is desirable.

In this paper we introduce a method we have employed during development of an enterprise information system for an electricity company. The system is used for planning and evaluation of electricity generation, so the core business logic consists mainly of calculation of many quantities for power plants and individual equipment. The main use case could be described as follows:

1. Power plants, equipment and commodities characteristics entry.
2. Power generation plan entry.
3. Actual power generation entry and evaluation.
4. Reports entry and evaluation.

As can be seen above, user works primarily in ‘data entry’ mode entering dozens of values for defined generation units. System then computes values of derived quantities and aggregations over a period and presents the results in the form of graphical reports. The system has been developed on J2EE platform with help of Spring framework and Groovy dynamic language.

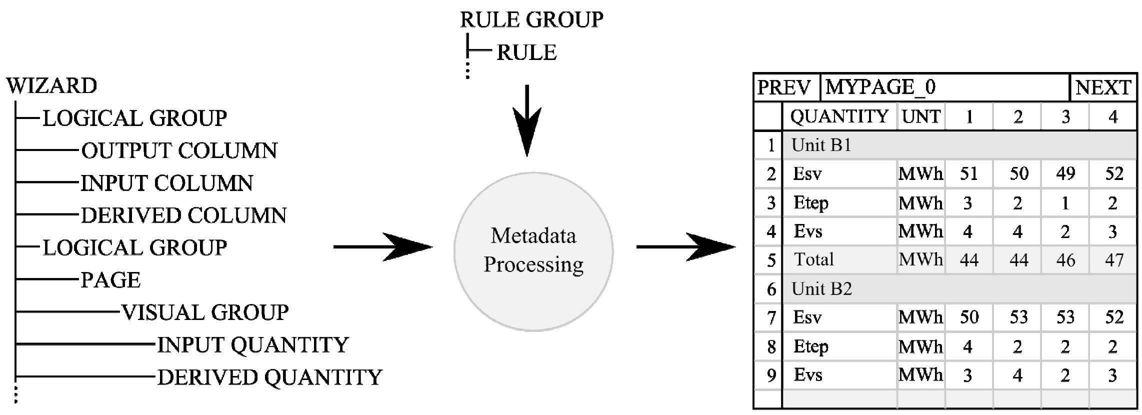
To describe these two mentioned common use cases (data entry in form of quantity values and reports generation), system of hierarchical metadata was created. That enables us to program new business logic in declarative manner without modifications of original Java code and to deploy changes to testing and production environment instantly. Utilization of this method naturally led to agile development and allowed the customer to take active role in the development process.

Declarative programming
In a declarative language programmer specifies what is to be computed in contrast to imperative language in which programmer has to specify how this is to be computed. This means that programmer can describe the logic of program without describing its control flow [1][2].

DESCRIBING COMMON USE CASES WITH HIERARCHICAL METADATA
As stated in the introduction, core of our solution is to describe common use cases in system with hierarchical metadata. In sum there are three classes:

- computational rules – define relations between quantities,
- input wizards and validation rules – define input forms,
- reports and exports – define output formats.

Computational rules are combined with input wizards and reports metadata to generate presentation tier.



	QUANTITY	UNT	1	2	3	4
1	Unit B1					
2	Esv	MWh	51	50	49	52
3	Etep	MWh	3	2	1	2
4	Evs	MWh	4	4	2	3
5	Total	MWh	44	44	46	47
6	Unit B2					
7	Esv	MWh	50	53	53	52
8	Etep	MWh	4	2	2	2
9	Evs	MWh	3	4	2	3

Figure 1: Input wizards and computational rules metadata trees and resultant spreadsheet.

Input wizards metadata
As can be seen in Figure 1 input wizard is described by a tree. Every node in the tree represents an element of a resultant spreadsheet through data is entered. All types of elements have some common properties, e.g.:

- Processing enabled – Logical expression stating whether the element should be processed during spreadsheet construction.
- Rendering enabled – Logical expression stating whether the element should be visible to the user. When element is not visible, but it is processed in a spreadsheet construction phase, it can still influence the data.
- Repeating expression – Expression that evaluates to a collection of objects for which is the sub tree repeated in result. Typical use case: nuclear power plant has several units, for every unit same quantities should be entered (see Figure 2).
- Repeating variable – Name of the variable to which current object from repeated collection is stored. This variable can then be used in other expressions.
- Definition of other variables that can be used in the respective sub tree.

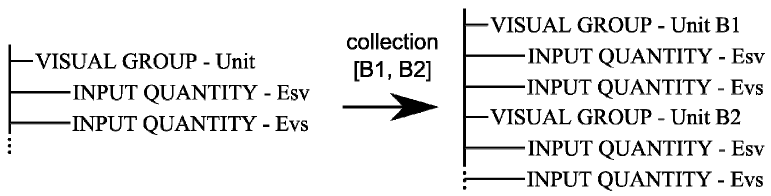


Figure 2: Repeating sub tree.

Substantial elements of the wizard tree are quantity elements: an input quantity element for quantities entered by user and a derived quantity element for quantities computed according to defined rules. Every quantity element is defined by its quantity number and other properties:

- Default value – Expression defining default value in case no other value of the quantity is available.
- Value storage – Expression stating whether the value should be persistent (stored in database) or not.
- Editable – Expression stating whether the value can be edited by a user.

Quantity elements can be grouped in visual and logical groups, the first one has graphical representation in the resultant spreadsheet. These groups can be organized into pages and pages into wizards.

Any sub tree can be shared among wizards or within one wizard. This means, that there can be only one definition of sub tree, but many usages. Such sub tree is in fact a reusable component. For example, all wizards use similar structure of columns (twelve columns for months, four columns for quarterly aggregations and one column for yearly aggregation), so there is no need to define them for every wizard separately.

Another important part is data validation. When user enters new data, go to the next page or save whole wizard, the system must ensure that entered and computed data is correct. This is achieved through a system of validation rules. Validation rule consists of:

- Severity – Specify whether validation failure is an error or just a warning.
- Left expression, an operator and right expression – Logic of the validation. When left expression is left blank, value of validated quantity is used.
- Application condition – Specify when this validation rule is applicable.
- Validation phase – Phase in which the rule should be applied.

This is quite simple but sufficient way of ensuring entered data validity. Other simple checks are defined by quantities itself: checks for zero, negative or positive value.

Reports and exports metadata

Reports in PDF format, CSV and database exports are described by the same set of metadata that is quite similar to previous class.

Computational rules metadata

Computational rules are key concept in our solution. Rules are used to describe relations between particular quantities, e.g. $q = Q / G$. When user enters values for Q and G, value of q is computed automatically. Every rule belongs to at least one rule group and is defined by the following set of properties:

- Quantity numbers – Collection of quantity numbers to which is the rule applicable. In many cases dozens of quantities can be computed in similar way.
- Priority – Rules are applied in order according to priority, i.e. system tries to apply rule with highest priority first, if it is not applicable second one is tried and so on.
- Expression – Definition of a computation rule.
- Facility/Device – Define whether the value is computed for whole facility or particular device.
- Commodity – Define whether the value is computed for particular commodity (fuel in most cases).

- Cumulation – Define whether the value is computed for individual months or it is aggregation for some period.

Example on using computational rules
Let there be a derived quantity input element in wizard definition for quantity X which should be computed for whole facility. During spreadsheet construction phase, system has to determine value of X in order to be able to present it to a user. Let suppose that wizard definition has associated rule group containing rule for quantity X and all dependencies as shown on left side in Figure 3.

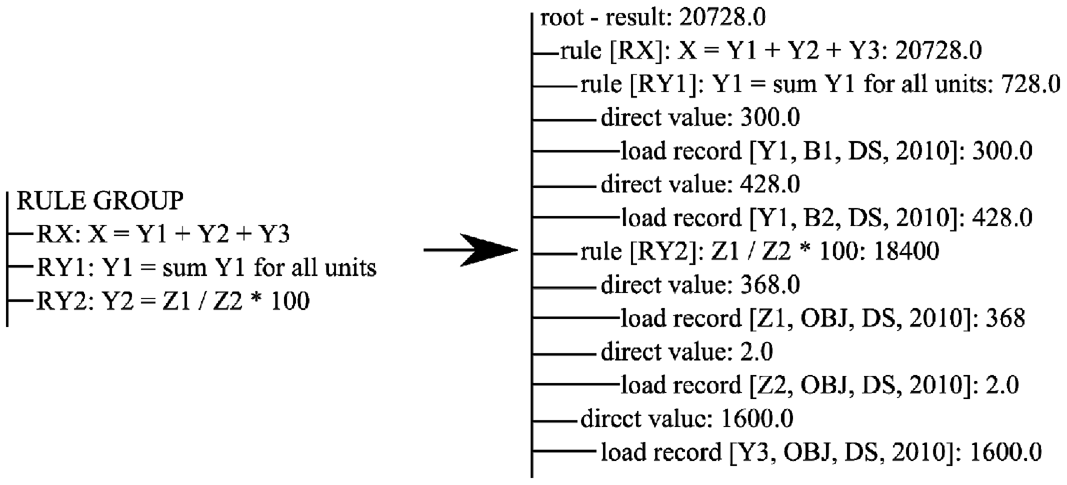

RULE GROUP
—RX: X = Y1 + Y2 + Y3
—RY1: Y1 = sum Y1 for all units
—RY2: Y2 = Z1 / Z2 * 100
root - result: 20728.0
—rule [RX]: X = Y1 + Y2 + Y3: 20728.0
——rule [RY1]: Y1 = sum Y1 for all units: 728.0
———direct value: 300.0
————load record [Y1, B1, DS, 2010]: 300.0
———direct value: 428.0
————load record [Y1, B2, DS, 2010]: 428.0
——rule [RY2]: Z1 / Z2 * 100: 18400
———direct value: 368.0
————load record [Z1, OBJ, DS, 2010]: 368
———direct value: 2.0
————load record [Z2, OBJ, DS, 2010]: 2.0
——direct value: 1600.0
———load record [Y3, OBJ, DS, 2010]: 1600.0

Figure 3: Quantity value computation according to computational rule.

Leave aside that there are some conditions of application and priorities and assume that the rule RX has been applied during value determination. Right side in Figure 3 shows what it means to apply rule RX:

1. RX is defined as summation of three other quantities. The first one is Y1, so its value must be computed first.
2. For Y1 there is a rule RY1 which states that Y1 for facility can be computed as sum of Y1 for individual facility units.
3. Y1 for units B1 and B2 can be loaded from storage because there is no applicable rule.
4. Next is Y2 which can be computed using rule RY2.
5. Quantities Z1 and Z2 can be again loaded from storage.
6. Finally, value of Y3 can be also loaded as there is no rule.

DISCUSSION

If there is such system of metadata as described above and appropriate tool for editing, preferably integrated into the enterprise system itself, programmer or system administrator can easily add new business functions into system or edit existing without knowledge of the system internals and moreover, with minimal knowledge of programming. Proposed approach also minimize error rate, because new logic to application is added through predefined wizards.

Groovy as expression language
Some element properties are stated to be expressions, so there must be some expression language employed. Requirements for the language are:

- good integration into Java project [5],
- dynamic language,

- support for lambda expressions.

Groovy meets all the requirements imposed and was used as expression language [3]. Despite the fact, that the main structure can be constructed in built-in editor with proposed hierarchical metadata without any programming, at least minimal programming skill is required to describe constraints and computational rules logic.

Great help are lambda expressions mainly in connection with collection methods *findAll* and *collect* [4]. For example when we want to get all used commodities from respective generation units to repeat some sub tree for them, we can write:

```
object.devices.findAll{dev -> dev.isType(20)}.collect{dev ->
dev.commodityUsages.collect{usg -> usg.commodity}}.flatten().unique()
```

At first glance it may seem difficult to understand this expression, but it is quite easy for person with at least minimal programming background.

Deploying metadata

Another advantage of the overall approach is an easy deployment of the business logic. All metadata can be exported to a XML file (natural choice with regard to hierarchical nature of data) and imported back to system with similar mechanism.

CONCLUSION

We introduced a way how we had utilized the declarative paradigm in development of enterprise information system for an electricity company and stated advantages of this approach: agile and rapid development, smaller error rate and easier deployment.

ACKNOWLEDGEMENT

Creation of this paper was partially supported by a grant SGS 11/167.

LITERATURE

[1] Jayaratchagan, N. Declarative Programming in Java. [cit. 2011-04-06]. Available from WWW: < http://oatv.com/pub/a/onjava/2004/04/21/declarative.html >.
[2] Lloyd, J. W. *Practical Advantages of Declarative Programming*. [cit. 2011-04-06]. Available from WWW: <ftp://clip.dia.fi.upm.es/pub/papers/PARFORCE/second_review/D.WP3.1.M2.3.ps.Z>.
[3] *Groovy – An agile dynamic language for the Java Platform*. Available from WWW: <http://groovy.codehaus.org/>
[4] *Groovy – Closures*. Available from WWW: <http://groovy.codehaus.org/Closures>.
[5] *Scripting for the Java Platform*. Available from WWW: < http://download.oracle.com /javase/6/docs/technotes/guides/scripting/index.html>