Domain-Specific Language data-validation and manipulation in a case handling system

by

Erik Timliden

LIU-IDA/LITH-EX-G--14/076--SE

2014-08-27
Examensarbete

Domain-Specific Language data- validation and manipulation in a case handling system

av

Erik Timliden

LIU-IDA/LITH-EX-G--14/076--SE

2014-08-27

Handledare: Christian Svedin
Examinator: Petru Eles
Abbreviations

DSL – Domain-specific language
EDSL – Embedded Domain-specific language
GPL – General purpose language
GUI - Graphical user interface
UML – Unified modeling language
AST – Abstract Syntax Tree
IDE – Integrated development environment

Keywords: Domain-specific language, DSL, Groovy

Title: Domain-Specific Language data- validation and manipulation in a case handling system
Author: Erik Timliden

Abstract
This thesis creates a Domain-Specific Languages (DSL) implementation and integration inside the iipax case handling software. The implementation investigated advantages with Domain-Specific Languages. Following agile software development methods the paper results in a functioning Domain-Specific Language emulating written English.

Sammanfattning

# Table of contents

Abbreviations ......................................................................................................................... 1  
Sammanfattning ......................................................................................................................... 1  
List of figures ............................................................................................................................. 3  
1  Introduction ........................................................................................................................... 4  
   1.1  Motivation ......................................................................................................................... 4  
   1.2  Purpose ............................................................................................................................. 5  
   1.3  Aim and objectives ........................................................................................................... 5  
2  Materials .................................................................................................................................. 5  
   2.1  Limitations and delimitations ......................................................................................... 6  
3  Background .............................................................................................................................. 7  
4  Theoretical background .......................................................................................................... 8  
   4.1  Domain-Specific Language ............................................................................................. 8  
      4.1.1  Principles of a Domain-Specific Language ............................................................... 8  
      4.1.2  Internal- versus External Domain-Specific Languages ............................................ 9  
      4.1.3  Structure of a Domain-Specific language ............................................................... 10  
   4.2  Groovy ............................................................................................................................ 10  
   4.3  Domain-Specific Language syntax ................................................................................ 10  
6  Method ................................................................................................................................... 11  
   6.1  Software development ..................................................................................................... 11  
8  Preliminary investigation ....................................................................................................... 12  
   8.1  Domain-Specific Language demands ............................................................................ 12  
   8.3  Software design ............................................................................................................... 12  
   8.4  Integration with iipax ....................................................................................................... 13  
   8.5  Script syntax design ......................................................................................................... 14  
9  Implementation ....................................................................................................................... 15  
   9.1  Graphical User Interface ................................................................................................. 15  
   9.2  Code ................................................................................................................................ 15  
10 Evaluation ............................................................................................................................. 17  
   10.1  Syntax .............................................................................................................................. 17  
11 Testing .................................................................................................................................... 18  
12 Results ................................................................................................................................... 19  
13 Evaluation ............................................................................................................................. 20
List of figures

Figure 1 Standard process for a case in iipax .................................................. 4
Figure 2 Graphical user interface for script composing .................................... 7
Figure 3 Anatomy of a Domain-Specific Language ......................................... 9
Figure 4 Dynamic systems development method ............................................ 11
Figure 5 Parameters in the Mapdrawer ......................................................... 13
Figure 6 Mapdrawer as a whole ................................................................. 14
1 Introduction

1.1 Motivation

Domain-Specific Languages (DSL) are a special-purpose languages, designed to make a domain more effective [1, 4]. A DSL is customized for the purpose it will serve. Increasing the productiveness [4].

A case handling system or an issue tracking system is used in different situations such as in government work and insurance companies. A case handling system can be described as an assembly line where the case reaches different stages. To transfer stage certain demands must be achieved. The DSL suggested in this thesis will control that the demands are achieved and automatically process the case to the next stage. A case handling system is more dynamic than an assembly line. In the case handling system the case can choose different paths. Figure 1 is an arbitrary structure of a possible case path.

![Figure 1, Standard process for a case in iipax](image)

Iipax is a case handling system developed by the company Ida Infront AB. For a more dynamic system different assembly lines can be created and modified in iipax.
1.2 Purpose
The purpose of this thesis is to create and design a DSL that is integrated inside the iipax system. The DSL goal is to automatize the stage transfers with an easily understood DSL. Focus will be on a simple syntax, which is still powerful enough to utilize.

As iipax is written in Java, the DSL engine is written in Groovy. Groovy is an objective oriented language compiling on the Java virtual machine (JVM) [11]. Groovy was chosen to improve the integration process of a DSL in iipax.

1.3 Aim and objectives
The intention is to develop and implement a DSL that is integrated in a case handling system. Described in the introduction section, a DSL should serve a purpose. This purpose is defined with the following goals:

- Create a Domain-Specific Language integrated with the iipax software
- Create a crude syntax control for the Domain-Specific Language
- Create the possibility to change and validate already existing data in the case
- Create automatic transitions in iipax with the Domain-Specific Language
- Create a Graphical User Interface in iipax for the Domain-Specific Language

To achieve these aims:

- Data will be collected from literature in the Domain-Specific Language field
- Data and results will be compared to existing Domain-Specific Languages

1.4 Materials
To fulfil this project the following materials are used:

Literature:

- Groovy for Domain-Specific Languages by F. Dearle
- Enterprise Integration Patterns: Designing, Building and Developing Messaging Solutions by G. Hohpe and B. Woolf
- Groovy in action by D. König
- DSL for the Uninitiated by D. Ghosh
- DSLs in Action by D. Ghosh
- Domain-Specific Languages by M. Fowler
- Software Engineering for Students by D. Bell
- Design Patterns: Elements of Reusable Object-Oriented Software by E. Gamma, R. Helm and R. Johnson
- Incremental concrete syntax for embedded languages with support for separate compilation by T. Dinkelaker, M. Eichberg and M. Mezini
- Domain-Specific Languages and Code Synthesis Using Haskell by A. Gill

The Integrated development environment (IDE) was Eclipse Kepler for Java-EE with a Groovy plugin. To keep consistency with the iipax source code, an integrated Maven for Eclipse was used.
1.5 Limitations and delimitations

The ambition is to develop a DSL within the time limit. The thesis is limited to Groovy language. Comparisons will be made towards Java so that it appeals to a wider audience. This is my first time encountering DSL development, Groovy and involvement in iipax.
2 Background

This thesis idea was generated by an employee at Ida Infront, who became the product owner. Together with the product owner we specified a system requirement.

The system requirement involved information whether a DSL is a possible plugin to iipax. What operations and functions the DSL should support. Emphasis is on user simplicity, in both the GUI and the script syntax.

Together with the handler at Ida Infront we specified a rough blueprint. We design how the DSL would interact in iipax and where the script would be composed. This is visualised in figure 2.

![Graphical user interface for script composing](image)

Figure 2, Graphical user interface for script composing

The idea is that a user will compose, name and save the script as in figure 2. In figure 5 the user will decide what script will be executed at that specified node in the assembly line by entering the script name.
3 Theoretical background

Regarding Groovy and DSLs the following books and information have been used: *Groovy for Domain-Specific Languages* by F. Dearle [1], *Domain-Specific Languages* by M. Fowler [7], *DSL in Action* by D. Ghosh [6]. In addition a summarization of G. Laforge: lectures and presentations [10] has been of great help. The articles DSL for the Uninitiated by D. Ghosh and Domain-Specific Languages Code Synthesis Using Haskell by A. Gill [12], have both helped clear out concepts about Domain-Specific Languages.

3.1 Domain-Specific Language

Domain-Specific Languages are a common element in programming. Most developers have encountered more than one DSL. Some well known DSLs are MATLAB and SQL. These languages do not produce a feature missing in a General-Purpose Language (GPL) such as Java. However they are known to be efficient in their specific purpose and often regarded as more simple [5].

A GPL can be compared to a Swiss army knife, it can solve most problems. A DSL can be compared to a hammer or a screwdriver, that solves a more specific problem and often more efficiently [10].

3.1.1 Principles of a Domain-Specific Language

There are some principles when developing a Domain-Specific Language. One of the foundations of a DSL is an economical efficiency benefit or cognitive intentions before developing [6]. A DSL demands a lot of planning and a keyword is “implement the DSL to the end users” [6]. Since the end users often know the purpose domain better than the developer, the communication between them is essential.

The communication gap between a software developer and an end user is created by their vocabulary [5]. Developing a DSL after the end users terminology could facilitate the learning curve when introduced to the DSL.

Since a DSL is not a multipurpose language it should focus on the essentials [6]. This focus often results in the DSL being less cluttered. Almost like *lean software development* [8], where the principles are: *eliminate waste, amplify learning and see the whole*.

A DSL has many advantages, such as the communication described above. With the high-level of abstraction characteristic to a well-designed DSL, the end user does not need knowledge about memory allocation or type definitions [5]. Figure 3 shows the anatomy of a DSL. Separating the application programming interface (API) from the core implementation, this separation delimits the end user from corrupting data [5].
Disadvantages are the difficulty of designing a DSL to the end user [5]. To succeed, the developer should understand the target domain and users.

### 3.1.2 Internal- versus External Domain-Specific Languages

There are two kinds of Domain-Specific Languages, internal and external. An internal domain-specific language is often called embedded domain-specific language (EDSL); an external domain-specific language is called standalone domain-specific language [12].

The design language in this thesis is Groovy. Other common languages EDSLs are developed in are Scala and Ruby; they all have a strong metaprogramming and allow code generation at runtime. The biggest difference between internal- and external DSLs is the decoupling from the design language (sometimes host language). Embedded DSLs are tighter coupled to the design language and standalone DSLs are less coupled. An embedded DSL is often less extensive in implementation [12].

EDSLs are more like a “language inside a language”, while standalone DSLs have their own ecosystem [12].

An external DSL (EDSL) can be a shallow EDSLs or a deeply EDSLs. A deeply EDSL creates its own structure, where a shallow EDSL operates directly on values [12]. A standalone DSL is an own language composed strictly to a purpose. An example of a standalone DSL is SQL [12].
3.1.3 Structure of a Domain-Specific language

The structure of a DSL is similar to an embedded- and a standalone DSL. They both have the same purpose principles described in *Principles of a Domain-Specific Language*. As shown in figure 3, there is a syntax abstraction overhead, the semantic model. The syntax abstraction is often called the façade or symbol table [5, 7]. The semantic model is known as the core or base abstraction. The Façade provides the expression based on the core implementation.

There are many different patterns when implementing a DSL. The book *Domain-Specific Languages* by M. Fowler [7], is focused around this area. Different patterns conform to different situations.

One important decision is whether to choose an Abstract Syntax Tree (AST)* or Binding to integrate and translate the user script. They both have different strengths and weaknesses. Roughly it can be summarized that binding is simple and unable to handle complex syntax, while an AST is more extensive to implement but can handle complex and nested syntaxes. An AST conforms to deeply EDSLs while binding conforms more to shallow EDSLs.

3.2 Groovy

*Groovy* is a very dynamic language that supports both static and optional typing. It is compiled by the Java Virtual Machine (JVM). Making one of the *Groovy* main features the ability to create non verbose code. *Groovy* is weakly typed with optional definition of variables, semicolons and optional return statements.

*Groovy* is influenced heavily by other languages, mainly *Java*. Other influences in *Groovy* are *Ruby* and *Python*[10].

3.3 Domain-Specific Language syntax

Creating a DSL implies that a new syntax is developed towards the end-users. This thesis focuses on a easily understood syntax. That will look similar to this: “*when equal year, 2014 then transit to process*”. The function vocabulary in this line of code is: *equal, when* and *transit*. The bubble words to get a more readable syntax are: *then* and *to*. The parameters are: *year, 2014* and *process*.

*An AST parses the input into a tree-structure, enabling the opportunity to customize the DSL syntax further. Binding binds certain keywords to functions that can be explicitly declared.*
4 Method

This chapter describes how a Domain-Specific Language was created inside iipax.

4.1 Software development

Creating a DSL is software development. To develop software in an effective manner, the Dynamic systems development method (DSDM) [8] was used. DSDM is one of the agile software development methodologies, constructed to improve software development.

DSDM fits well for this project due to MoSCoW method (Must, Should, Could, Won’t) [8]. These principles can be translated both to the chapter Aim and Objectives and the lean software development principles brought up in the Theoretical Background chapter. DSDM begins with a planning stage, followed by the foundation then rotating and switching between exploration, incremental deployment and engineering. The planning stage is extra important in DSLs [1, 5]. The following stages are testing, and iteratively developing the system [8]. A graphical representation of DSDM is illustrated in figure 4.

![Diagram of DSDM methodology]

Figure 4, Dynamic systems development method

For the tight timeline in the development process, a Middle-out implementation was chosen. Middle-out focuses on starting with a central component in the software. This central point is tested and the system comes forth from this central component [8].
5 Preliminary investigation

As described in the Background chapter, this thesis was supervised and ordered by the company Ida Infront. The process began by gathering common knowledge about DSLs. Together with the client, a software requirements specification was formulated, so that the end result would satisfy both parts.

In order to get a feeling for the Groovy language and environment, a smaller DSL was created that converted different currencies. After investigating the different solutions available, the design phase began.

5.1 Domain-Specific Language demands

The requirements specified in the software requirements specification included the following:

- Ability to validate and change values of predefined parameters in the case
- Ability to automatize case process
- Ability to log data
- Integrate with iipax

These demands set the bounds to the thesis and will shape the DSL.

The following functions are required to achieve the demands above:

- Comparisons
  - Equal, less, greater
- Assign values
  - Set
- Set next process for case

For the comparisons to work a conditional branching must be implemented.

5.2 Software design

An important decision is the software design and how to implement the DSL. To gather information about the software design I reviewed the patterns discussed and presented in M. Fowlers book Domain-Specific Languages [7].

The pattern that corresponded best to the Domain-Specific Language demands above was Symbol Table. Symbol Table is described as “a location to store all identifiable objects during a parse to resolve references”[7] and common in smaller DSLs. It is a straight forward approach and collaborates well with binding. The two defined flaws are the quotation marks for strings and that the compiler does not include type checking.
The **Symbol Table** pattern is defined as two parts. One part called **Symbol Table**, the other part called **Semantic Model**. Dividing the program into two separate files: the **Symbol Table** fetches the script, prepares the data and evaluates the script with an engine bound to the **Semantic Model**. The **Semantic Model** preforms the practical execution and integrates the functions to iipax.

The **Symbol Table** pattern together with binding would be a working solution towards the **Domain-Specific Language demands**. **Binding** the functions as references and develop a small shallow EDSL.

For code readability the **CamelCase** system was used to define functions and variables.

### 5.3 Integration with iipax

iipax is, as described, an existing product that the DSL is going to be integrated with. iipax has a **Register** section, where new registers can be created. The **Register** is graphically shown in figure 2. A new sub-function will be created in the **Register**. This sub-function will create a new register, which has two text input fields. One name field and one script field that compiles to the DSL.

This new sub-function will be defined upon existing structure in iipax, with the slight modifications of the two text field inputs. The already existing structure handles register saving in a satisfactory manner.

Earlier the report explained when the DSL would execute the script. Figure 6 is the graphical representation of the **Kartritaren (Map drawer)**, in iipax. In the middle of the left side of the figure, there is a **plugin parameter**. This plugin parameter will search for a file to execute. This file is the symbol table (the façade in figure 3). The symbol table must know what script to evaluate. This is decided by a parameter in figure 5 called **std_reg_regel**. The parameter and the plugin parameter are individual for each node represented in figure 5. The tree structure shown in figure 1 is one of the standard examples in iipax of a possible case path.
5.4 Script syntax design

The script that is evaluated by the DSL needs to be designed and formulated after the target audience. Instead of interviewing the end users, discussions with the assigned company handler took place. This solution was chosen since there is no specific end user for iipax. Usually it is a case worker in government or insurance companies. The company handler has a close communication as a developer with one of the major consumers of iipax, understanding the end users proficiently.

The common denominator in the end users is the general lack of knowledge in programming. Therefore, the goal is a simple syntax that emulates fluent writing.
6 Implementation

This chapter will explain the implementation process.

6.1 Graphical User Interface

The first implementation was the graphical user interface (GUI). As analyzed in the Preliminary Investigation chapter, the GUI would be built upon the existing structure of iipax. The company handler introduced iipax and foundation of the structure, explaining that the GUI is represented in Extensible Markup Language (XML).

The GUI was placed in the existing GUI XML code that constructed the Register (See Integration in iipax for further information). The code resembled existing sub-functions except for the name and input fields. The code is visible in the Appendices chapter.

6.2 Code

As described in the Software Development chapter, a Middle-out implementation was the planned approach. Much of how the code should be structured is explained in the Software Design chapter.

A prototype DSL was created separated from iipax. This prototype did not fulfil any purpose except testing script code in order to see that the DSL responded correctly. The entire process of prototyping began with creating the definitions of the required functions from Domain-Specific Language demands chapter. The following development was iterative: taking one step at a time and remaking that function until it worked correctly.

All the functions where defined in one file called functionTable. It can be found in the appendices for complete implementations. To facilitate the original functions where supplemeted to their opposites, for example is Equal got the contrary Differ. functionTable is the implementation of Semantic Model described in the Software design chapter. The following functions where implemented:

**When**

Enables conditional statements and works as an IF-statement. It is dependent upon the Boolean Wrapper that does the Boolean execution for single statements together with “and” and “or” conditions.

**Unless**

Opposite of when, created as a substitute of else and can perform the same actions as when.

**Equal**

Comparison between two undefined variables. Can compare strings, integers or any other class.

**Differ**

Calls on equal and returns the opposite. This function was defined to avoid the vague “!” syntax.
Set
Maps a string to a value to define variables.

Greater
Returns a Boolean value if the first parameter is larger than the other. JVM does not support comparisons between strings and integers, so a typecast is made in the comparison, causing the possibility of defective return values between strings and integers (Returns false).

Less
Calls greater and returns the opposite.

Transfer
Sets the path deciding what the next node will be; controls that the path exists and is valid to take.

As a safety precaution invokeMethod, methodMissing and propertyMissing was defined in the functionTable. These methods all work as an error handling system. They write error messages indicating what went wrong when calling for a function or an undefined variable. They are all defined by two functions, one with a name parameter and the other one with a name and value/arguments parameters to extend the catching functionality, depending on the call.

Together these functions can accomplish the aims in the Domain-Specific Language demands chapter. The functions are all located in the functionTable file and implement the Semantic Model.

The other file symbolTable, defines the overhead of the DSL. Before the DSL was integrated into iipax, symbolTable defined the binding. It bound certain keywords to the functions above. This binding was then used as an initialize parameter to GroovyShell. GroovyShell is a built-in script engine in Groovy, that evaluates given text. When the GroovyShell was set up with binding, it was tested to parse and run variables defined in the Symbol Table file. The variables consisted of predefined strings testing via DSL script the functions in functionTable.

When symbolTable worked sufficiently well, it was time to integrate it into iipax. The difference when inside iipax was that the script would be fetched via a query. The name of the script was defined at the local node (current process in the case).

The new symbolTable file is renamed to scriptPlugin. At first it logs to the iipax logging that the file has started to execute. It fetches the current case and all pre-existing data on the case. The parameter in the node labelled std_reg_regel contains the name of the script created in Register. scriptPlugin then queries iipax for the script. If no script is found it logs “unable to open script”, else the same process as in symbolTable file is executed. When scriptPlugin has been executed the system logs the upcoming node. scriptPlugin then returns the value of the upcoming node and iipax will try to execute the plugin on that node. scriptPlugin is executed if the node has scriptPlugin as the designated plugin.
7 Evaluation

The method chosen to reach the project goal could be followed without any major problems. Using an agile method DSDM provided the opportunity to make changes in late stages and was probably the best decision possible in this scenario. Changes always occur and in this case the opportunity to add and modify features in the end phases was useful.

7.1 Syntax

Normally, conditional branching can include an else statement. After some discussion with the company handler we agreed that to a person without prior knowledge to programming, the most appropriate way is to have

\[
\text{When equal \textit{income}, 20000 then transaction \textit{handle} end}
\]
\[
\text{Unless equal \textit{income}, 20000 then transaction \textit{assign} end}
\]

Instead of

\[
\text{When equal \textit{income}, 20000 then transaction \textit{handle} end}
\]
\[
\text{Else then transaction \textit{assign} end}
\]

In logical context they are the same, except that the first alternative might be more self-explaining.

One feature that was not intended is that the parentheses after the function is necessary in the syntax if conditions are used. This results in that

\[
\text{When equal(\textit{income}, 20000) then transaction(\textit{handle}) end}
\]

is needed to execute the script. This only occurs when a condition is used. So the following statement

\[
\text{Set \textit{income} 25000 transaction \textit{handle}}
\]

will compile. This is caused by the Command-Chain in Groovy. Command-Chain maps the functions and methods with arguments, removing the necessity of parentheses and dots. However this did not function with the When/Unless functions.
8 Testing

The testing phases were divided into smaller stages. The most intensive and comprehensive testing was after the merge.

Testing was based on the *Aim and objectives* and *Domain-Specific Language demands* chapters. The most extensive test was malicious commands and error handling, a combination of *white-box testing* and *black-box testing*, a so-called *gray-box testing*. 
9 Results

The results in this thesis are hard to measure in facts and figures. The results are, instead, measured compared to the Aim and objectives and Domain-Specific Language demands chapters. The syntax can be measured in the absence of punctuation marks and the perceived Human-computer interaction.

Measured towards the Aim and objectives, the product reached the proposed goals. The syntax is readable and resembles a normal language and can execute the appropriate functions.

The GUI is easy to employ and provides a quick way to change and create new scripts. The quick modification is provided by the fact that the GUI is completely integrated with iipax.

A crude syntax control is implemented. It is not implemented in the intended way. At first the syntax control would occur in the GUI. The script in the GUI would not be executed until the actual process transaction when running iipax. The system instead logs the error in the universal iipax log file. If a syntax error was encountered the log will look like this “ERROR: <TIME/DATE> file: scriptPlugin 111 script.run() “eqaul 3,3”. This error was in this case thrown by a spelling error resulting in the DSL not identifying eqaul.

One important result is that DSL adapts well into a case handling system like iipax.
10 Evaluation

Testing could have been more thorough and include stress test and other more extensive tests. The testing indicated that there were flaws in the error handling, the syntax control could have been more comprehensive, making the product better and more user-friendly. This could have been solved with more time spent on testing and error handling in the final stages, if more time had been available.

The tests done were an acceptance test and a smoke test, they both went well. All the testing was manual and did not include any automated tools.

One of the main objectives was to create a syntax that was appealing to human cognition. To achieve this, unnecessary parentheses and dots where avoided to differentiate the DSL from a low-level machine language.

The achieved syntax looks like the following:

When greater ("income", 20000) and less ("age", 20) then
  When equal ("name", "john") then
    transit "handle" end
  When equal ("name", "doe") then
    transit "assign" end end

And is representative for the following code in C++:

```cpp
If(income >= 20000 && age < 20) {
  If(name == "john") {
    transit(handle);
  } else if(name == "doe") {
    transit(handle);
  }
}
```

This piece of code takes full advantage of the syntax supplied in the DSL. It uses multiple conditions with parameters that are based on every specific case. The parameters income, age and name are fetched from every specific case. These parameters are generated when the case is created and stored as variables inside the case.
11 Future Work

What has been product part of this project is a rather small scale DSL that could be extended in functionality. The current model is open for extensions.

The syntax did not reach the level of simplicity I first imagined. Without the parentheses the syntax would have more correctly matched my expectations.

There are some functions that could be extended. Mainly a better implementation on the If-else cases is possible. The syntax could be extended with more bubble words, removal of parentheses in all situations and removal of quotation marks around strings.

An interesting idea would be to test the DSL on the normal iipax user. Unfortunately, that was not an option because of time restraints.
12 Discussion

12.1 Results
As mentioned in the Results chapter, this thesis is rather hard to measure in facts and figures. The results did not correspond entirely to the desired results, a DSL with easy fluent syntax readability, with good cognitive cognition.

Whether the quotation marks are interfering in the syntax can be argued. They make the parameters more notable, but reduce the fluency in reading the syntax. Parentheses do interfere with the readability.

The aim and objectives of the project were reached but not exceeded. My opinion is that the DSL developed in this report is dynamic enough to function inside a case handling system and powerful enough to utilize. There could be reinforcements in the utilization by increased functionalities or a greater variety of the different functions. Unfortunately, there was no chance to interview or discuss with the end users to gather a common vocabulary for the DSL. The syntax discussion was done with the company supervisor and we tried to stick to common English vocabulary with obvious commands. It could be researched further whether the syntax is clear enough for the end users.

12.2 Method
A lot of information had to be gathered before developing a DSL. This is a valuable lesson and, if I were to do the same project again, I would generally take the same steps and the same order but change the time spent on all the different parts in the process.

In the information gathering process, I would take more notes to avoid stepping in circles. Take the time to read the literature more thoroughly, and try to discuss the different approaches with a knowledgeable source. Both for ideas and avoid stepping in pitfalls.

The development phase went well. One mistake was not asking more questions and asking for help when necessary. The method could be replicated by anyone with knowledge in programming and development.

I recommend the literature used in this project, including the online material used. The sources have been inspected and deemed trustworthy in this scenario.
13 Conclusion

This project has been a new area to me. My sincere hope is to encourage innovation and the courage to try a new field. It has been a very instructive project. The knowledge acquired about Domain-Specific Languages and software development is valuable.

Hopefully, the results in this report can guide and help decision making when implementing a DSL or give inspiration to develop one. This report is focused on data validation and manipulation in case handling systems. It proves that an integrated DSL inside software can appeal to a wider audience as it is a universal solution that could streamline coding.


Appendices
class Function {
    public Function(ScriptPlugin sP) {
        this.sp = sP
    }

    def propertyMissing(String name) {
        outgoingMessage("Called propertyMissing1 with ${name} and value ${value}")
    }

    def variableMap = [:]

    def propertyMissing(String name, value) {
        incomingMessage("Called propertyMissing2 with ${name} and value ${value}")
    }

    def invokeMethod(String name) {
        outgoingMessage("Called invokeMethod $name")
    }

    def invokeMethod(String name, args) {
        outgoingMessage("Called invokeMethod $name $args")
    }

    def methodMissing(String name) {
        methodMissing(name, null)
    }

    def methodMissing(String name, args) {
        def method = metaClass.getMetaMethod(name, args)
        if (method != null) {
            method.invoke(this, args)
        } else {
            throw new MissingMethodException(name, this.getClass(), args)
        }
    }

    def set(String variableName, def value) {
        variableMap[variableName] = value
    }

    def change(String variableName, def value) {
        variableMap[variableName] = value
    }

    // Method Reference & Get a reference to a method, can be useful for creating closures from methods
    def when(BooleanWrapper bool, closure) {
        if (bool.value) {
            closure.delegate = delegate
closure(closure)
        }
    }

    def unless(BooleanWrapper bool, closure) {
        if (!bool.value) {
            closure()
        }
    }

    def equal(def variableA, def variableB) {
        boolean isEqual
        if (variableMap.containsKey(variableA)) {
            if (variableMap.containsKey(variableB)) {
                isEqual = variableMap[variableA].equals(variableMap[variableB])
            } else {
                isEqual = variableMap[variableA].equals(variableB)
            }
        } else if (variableMap.containsKey(variableB)) {
            isEqual = variableMap[variableB].equals(variableA)
        } else {
            isEqual = variableA.equals(variableB)
        }
        return new BooleanWrapper(isEqual)
    }
}
```java
def differ(def variableA, def variableB) {
    return new BooleanWrapper(!equal(variableA, variableB).value)
}

def greater(def variableA, def variableB) {
    boolean isGreater
    if(variableMap.containsKey(variableA)) {
        if(variableMap.containsKey(variableB)) {
            isGreater = variableMap[variableA] <= variableMap[variableB]
        } else {
            isGreater = variableMap[variableA] <= variableB
        }
    } else if(variableMap.containsKey(variableB)) {
        isGreater = variableA <= variableMap[variableB]
    } else {
        isGreater = String.valueOf(variableA) <= String.valueOf(variableB)
    }
    return new BooleanWrapper(isGreater)
}

def less(def variableA, def variableB) {
    return new BooleanWrapper(!greater(variableA, variableB).value)
}

def transit(String transition) {
    sp.setNextNode(transition)
}

def transfer(String transition) {
    transit(transition)
}
}

public class BooleanWrapper {
    def value
    BooleanWrapper(def value) {
        this.value = value
    }

    def or(def valueWrapper) {
        return new BooleanWrapper(this.value || valueWrapper.value)
    }

    def and(def valueWrapper) {
        return new BooleanWrapper(this.value && valueWrapper.value)
    }
}
```
13.2 symbolTable

```java
package idainfront.std.server.plugin.workflow.dsl
import java.util.logging.Level
import java.util.logging.Logger
import org.codehaus.groovy.control.CompilerConfiguration
import groovy.lang.Binding
import iipax.generic.objectbase.client.api.iface.CallerObject
import iipax.generic.plugin.PluginException
import iipax.generic.plugin.StringMap
import iipax.generic.plugin.process.CaseProcessHelper
import iipax.service.casekernel.client.api.CallerObjectFactory
import iipax.service.casekernel.client.api.iface.CaseObject
import iipax.service.casekernel.client.api.iface.CaseRegister
import iipax.service.casekernel.client.api.iface.CaseSupportObject
import iipax.service.casekernel.client.api.iface.RegisterQuery
import idainfront.std.common.util.metadata.StdAttributtyper
import idainfront.std.server.plugin.workflow.dsl
import idainfront.std.server.plugin.workflow.BasePlugin

public class ScriptPlugin extends BasePlugin {
    private static final String CLASS_NAME = ScriptPlugin.class.getName()
    private static final Logger LOG = Logger.getLogger(CLASS_NAME)
    public String nextNode
    private String mRule

    // Reachable from FunctionTable, the return from execute = next node
    public void setNextNode(String s) {
        nextNode = s
    }

    @Override
    protected String executePluginSpecificTasks() throws PluginException {

        final String methodName = "executeBindingEvaluate"
        LOG.entering(CLASS_NAME, methodName)

        def function = new Function(this)
        Binding binding = new Binding([
            function: function,
            when: function.&when,
            unless: function.&unless,
            otherwise: function.&otherwise,
            publish: function.&publish,
            set: function.&set,
            change: function.&change,
            equal: function.&equal,
            differ: function.&differ,
            greater: function.&greater,
            bigger: function.&bigger,
            less: function.&less,
            transit: function.&transit,
            transfer: function.&transfer,
            invokeMethod: function.&invokeMethod,
            methodMissing: function.&invokeMethod,
            propertyMissing: function.&propertyMissing
        ])
    }
}
```
```java
function.variableMap += getCase().getAttributes().entrySet()

String ruleScript = getRule()
CaseObject co = getCase()
if (ruleScript != null && !ruleScript.equals(""))
    {
        try
        {
            GroovyShell shell = new GroovyShell(binding)
            Script script = shell.parse(ruleScript)
            script.run()
        }
        catch (Exception e)
        {
            throw new PluginException(e)
        }
    }
else
    {
        String msg = "No rule was supplied during initialization!"
        LOG.logp(Level.WARNING, CLASS_NAME, methodName, msg)
        throw new PluginException(msg)
    }

    LOG.exiting(CLASS_NAME, methodName)
    return nextNode
}

void setRule(String rule) {
    mRule = rule
}

// Kludge
String ruleRewrite(String rule) {
    rule = rule.replace(" then", ", {")
    rule = rule.replace(" end", "}")
    return rule
}

/**
* Look up the given rule
* Fetch script by ruleName
* @return Script
*/
private String getRule() {
    CaseCallerFactory factory = CaseCallerFactory.getFactory()
    RegisterQuery query = factory.makeRegisterQuery()
    query.addCondition(CallerObject.A_DISPLAY_NAME, RegisterQuery.OPERATOR_EQUALS, mRule)
    query.addAllowedObjectType("std_reg_regel")

    CaseRegister register = factory.getCaseRegister()
    CaseSupportObject obj = register.search(query).iterator().next()
    String mRuleName = obj.getAttribute("std_reg_regel")//regelattribut, regelnamn
    mRuleName = ruleRewrite(mRuleName)
    return mRuleName
}
```
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Erik Timliden]