Conditional Execution: A Pattern for the Implementation of Fine-Grained Variabilities in Software Product Lines

JADSON SANTOS, Federal University of Rio Grande do Norte
GLEYDSON LIMA, Federal University of Rio Grande do Norte
UIRÁ KULESZA, Federal University of Rio Grande do Norte
DEMOSTENES SENA, Federal University of Rio Grande do Norte
FELIPE PINTO, Federal University of Rio Grande do Norte
JALERSON LIMA, Federal University of Rio Grande do Norte
ALEXANDRE VIANNA, Federal University of Rio Grande do Norte
DAVID PEREIRA, Federal University of Rio Grande do Norte
VICTOR FERNANDES, Federal University of Rio Grande do Norte

This paper presents the Conditional Execution design pattern that aims to help the implementation of fine-grained variabilities in the context of software product lines of information systems. The pattern has been used successfully in three product lines of web information systems developed by Informatics Superintendence (SINFO) at Federal University of Rio Grande do Norte.

Categories and Subject Descriptors: D.2.11 **[Software Architectures]** Patterns; D.2.2 **[Programming Languages]:** Language Constructs and Features—Patterns

General Terms: Design

Additional Key Words and Phrases: Design Pattern, Software Product Lines, Fine-Grained Variabilities

ACM Reference Format:

Santos, J., et al. Conditional Execution: A Pattern for the Implementation of Fine-Grained Variabilities in Software Product Lines. Proceedings of the IX Latin-America Conference on Pattern Languages of Programming (SugarLoafPLoP'2012), September 2012, 17 pages.

1. INTENTION

The Conditional Execution design pattern contributes to the implementation of fine-grained variabilities in the context of software product lines (SPL) [Clements and Northrop 2001] [Czarnecki and Eisenecker 2000] [Weiss 1999] of information systems. It provides support to the modularization of a certain variability located in one or more modules to values assigned to configurations parameters, allowing its activation (or deactivation) dynamically in the system.

2. EXAMPLE

In order to illustrate the need for the usage of the pattern, we present examples from a Library Management module of a software product line of an academic information system. We use the punishment functionality for users who return late borrowed books. This functionality defines a variability that determines the kinds of punishment, which can be: (i) a payment in cash or (ii) a suspension for a period of time without to make new loans.

During the execution of the system, which represents a product (instance) of the SPL, the decision for execution of such punishment variability can happen several times, for example, during the execution of a new loan, returning a book or printing a bill for payment. The variability code is also spread among the system layers, from the calculation of the amounts of punishment in the business layer to the user visualization of loan functionalities in GUI layer. In some cases, the system manager can provide a "period of forgiving" during which the users do not receive punishment, even if they return the material overdue. All variabilities related to

Author's address: Department of Informatics and Applied Mathematics (DIMAP) and Informatics Superintendence (SINFO), Federal University of Rio Grande do Norte (UFRN), Campus Universitário Lagoa Nova, Postal Code 59076-970, Natal, RN, Brazil; emails: {jadson}@info.ufrn.br, {uira}@dimap.ufrn.br, {demostenes.sena, felipe.pinto, jalerson.lima}@ifrn.edu.br, {gleydson.lima, drvpereira, strapacao, victorhcf }@gmail.com

the punishment feature must be deactivated from the system and then being re-activated immediately after the system manager request.

Listing 1 shows the code of the `ReturnLoanProcessor` class from business layer containing the functionality "return of a book loan" without the punishment variability. We want to create a variation point inside the core conditional statement that verifies if the return of the book is delayed. It is noticed that the code that creates a suspension is intertwined with business code that verifies the permission of the operator (library employee who performs the loan), updating the data to commit the loan and sending e-mail information to the user about the book return.

Listing 1. Example of fine-grained variability in business layer

```
public class ReturnLoanProcessor {

    public void execute(Loan loan){

        verifyOperatorPermition(getLoggedOperator());

        boolean delayed = limitDateExceeded(loan.getLimitDate(), new Date());

        if (delayed){
            createUserSuspension(loan);
        }

        updatesLoan(loan.getId()
        , new String[]{"returnDate", "userReturnLoad", "situation"}
        , new Object[]{loan.getReturnDate()
        , loan.getUserReturnLoad()
        , loan.getSituation()});

        sendEmailUserLoan(loan.getLibraryUser());

    }
}
```

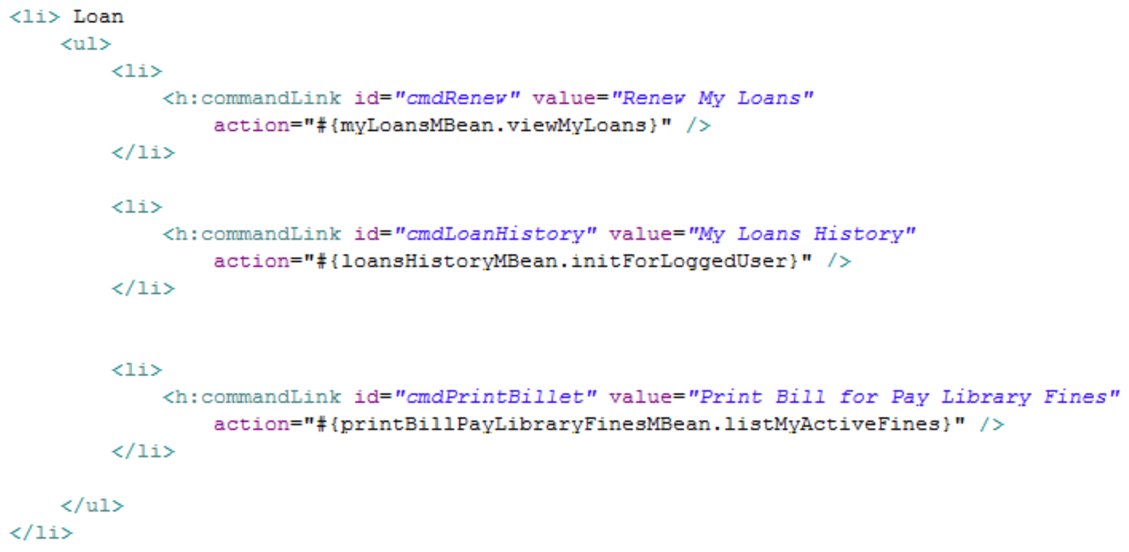
Software product lines of medium and large scale with a large number of features, tend to have fine-grained variabilities [Kästner, Apel and Kuhlemann 2008] that are intertwined with the implementations of other common and variable features.

It is necessary to implement the variability of execution the punishment feature in the example in Listing 1 in a way that, depending on the type of punishment that is currently active, the code that implements such variability is executed, creating the correct punishment for the user, instead always create a suspension.

The decision for the execution of the variability need be codified to allow its dynamic change, reflecting if the punishment variability is currently active or not. In addition, it is also need the introduction of the variability implementation inside of classes of the business layer. In Listing 1, for example, the punishment variability must be implemented using another conditional statement ("if") that determines its execution.

Finally, another requirement is that the proposed solution to implement the punishment variability can be used in other modules of the system, eventually implemented using other technologies. Listing 2 shows, for example, a Java Server Faces (JSF) web page from the same system, which implements a menu with the user options related to the functionalities of the book loan. The item "*Print Bill for Pay Library Fines*" should be visible to users only if the variability of fine is enabled.

Listing 2. Example of fine-grained variability in view layer



```
<li> Loan
    <ul>
        <li>
            <h:commandLink id="cmdRenew" value="Renew My Loans"
                action="#{myLoansMBean.viewMyLoans}" />
        </li>

        <li>
            <h:commandLink id="cmdLoanHistory" value="My Loans History"
                action="#{loansHistoryMBean.initForLoggedUser}" />
        </li>


        <li>
            <h:commandLink id="cmdPrintBillet" value="Print Bill for Pay Library Fines"
                action="#{printBillPayLibraryFinesMBean.listMyActiveFines}" />
        </li>

    </ul>
</li>
```

Object-oriented (OO) solutions such as design patterns [Gamma et al 1995] can contribute to the modularization of SPL variabilities in information systems by providing flexible structures to implement optional and alternatives features, which may be activated depending on the desired behavior in the product. For example, the State or Strategy [Gamma et al 1995] design patterns can be used to implement alternative variabilities, such as the example shown in Listing 1, which involves choosing one among a set of algorithms or possible states.

Although the adoption of traditional design patterns can help the modularization of SPL variabilities in information systems, the decision to instantiate or execute such variability may depend on many other parameters configured from the SPL domain. These parameters can be available during the process of derivation of the product (configuration parameters of the product) or only during its execution.

There are other modularization techniques that can be used in the implementation of fine-grained variability of information systems, such as aspect-oriented programming and conditional compilation. Each one of them faces difficulties when used in this context. Aspect-oriented programming (AOP) enables the modularization of such variabilities. However, the implementation of fine-grained variabilities is usually strongly coupled with the business code where it is located. Because of that, the use of aspects requires to expose all parameters used by the variability, which demands the codification of complex pointcuts [Kästner, Apel and Kuhlemann 2008]. Furthermore, in the context of the development of information systems in the industry, it is not common to find software engineers who have advanced knowledge for using such technique.

The Conditional Compilation is another technique that can be used for implementing fine-grained variabilities. This technique works by defining a set of pre-processor directives in the source code, which indicate the inclusion of a fragment of code on the product of a SPL. However, this technique provides a series of limitations compared to other existing ones: (i) it is possible to define only Boolean parameters when specifying the variability, and (ii) all the variabilities need to be addressed and solved during the compilation of the system, there is no support for the definition of variabilities, whose execution can be decided during the system execution.

3. CONTEXT

Nowadays there is a great demand in the industry for the customization of information software systems already developed for a given institution, in order to adapt those systems to be used in environments from other institutions. One of the motivations for such scenario is to increase productivity and quality of development of new information systems through the reuse of design and code of previously developed systems.

In this context, software engineering approaches to the development of software product lines (SPL) emerge as a promising proposal. A SPL can be seen as a family of systems that maintains: (i) a set of common features for all members of the family, known as similarities; and (ii) a set of variable features that changes based on specific needs of a SPL family member (product), called variabilities. A feature represents a functionality or property of SPLs that is used to capture their similarities and distinguish their variabilities.

The development process of SPLs involves not only the identification and modeling of their similarities and variabilities, but also the definition of a flexible architecture and the choice of adequate implementation techniques to modularize the SPL variabilities.

4. PROBLEM

The Conditional Execution pattern proposes to address the following question: How to implement fine-grained variabilities in information system product lines in order to allow dynamic product configuration, without causing major changes impacts on architecture designed, keeping the performance expected by the system and being flexible enough to be implemented by all its modules and/or layers?

The following forces emerge from this problem:

(1) Dynamic Configuration: The variability choice should be performed dynamically, thus giving more flexibility to the SPL customization.

(2) Low Impact: The implementation of the variability should not require substantial changes on the SPL structure.

(3) Fine Granularity and Crosscutting: The variability can appear in any layer or piece of code, being fine-grained and/or crosscutting.

(4) Performance: The solution should run with low overhead for the system.

5. SOLUTION

The pattern proposes to map each identified variability to a configuration parameter, which allows defining if the variation will be performed at runtime according to the parameter value.

The parameters are stored in a persistent repository, which allows easily reading and updating their values. A component is concerned in retrieving the parameter values in a transparent and efficient way, so it can instantiate the variability code to be executed based on these values.

The proposed solution is a combination of conditional statements "ifs" with configuration parameters of the system. Thus, it can decide which variability will be present in a particular product by adding or removing variabilities at runtime without keeping this decision hardcoded.

6. STRUCTURE

The Conditional Execution pattern consists of four main participants, which are illustrated in Figure 1:

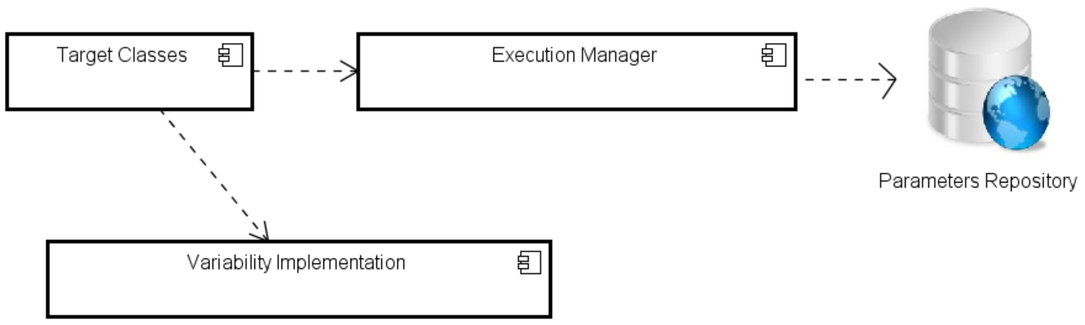


Fig. 1. Conditional Execution pattern structure

6.1 Target Classes

It represents the system components where variabilities happen. The execution of the variability code depends on the configuration of the system parameters values. In the context of web information systems, *Target Classes* may be in any layer of the application, including dynamic HTML content pages.

6.2 Variability Implementation

The code that modularizes the variability is located within the *Target Classes*. It usually consists on: (i) a direct implementation of variability code inside a *Target Class*; or (ii) a call to a component that implements the variability. The execution of this snippet is decided by the *Execution Manager*.

6.3 Execution Manager

The *Execution Manager* is the central element of the pattern. It determines if certain SPL variability will be executed or not based on configuration parameters from the system. The main tasks of this element are: (i) to get the parameters values from the repository as efficiently and transparently as possible; and (ii) to determine the execution of the variabilities associated with the retrieved parameters on *Target Classes*. *Execution Manager* contains the decision logic to determine if the variability is performed, removing this responsibility from the *Target Classes.* It provides access to configuration parameters, allowing the runtime adaptation of the variability. Finally, the *Execution Manager* must also provide mechanisms to optimize and minimize access to the repository, and it must maintain the parameters values synchronized with the values stored in the repository.

6.4 Parameters Repository

The *Parameters Repository* is responsible to store the parameters values used during the conditional execution. It allows storing and updating the parameters values during the application execution. It is fundamental that the retrieving of the parameter values is not costly for the system performance. Database systems or local files are examples of technologies that can be used as a *Parameter Repository*.

7. DYNAMIC

The following scenario defines the behavior of the Conditional Execution pattern during the runtime execution of variabilities

7.1 Scenario 1: Variability Execution

An instance of the *Target Class* calls the *Execution Manager* to determine if the variability will be performed. The *Execution Manager* retrieves the parameter value by calling the *Parameters Repository*. After that, the *Execution Manager* indicates to the *Target Class* if the *Variability Implementation* must be executed or not. Figure 2 shows a sequence diagram that illustrates this scenario.

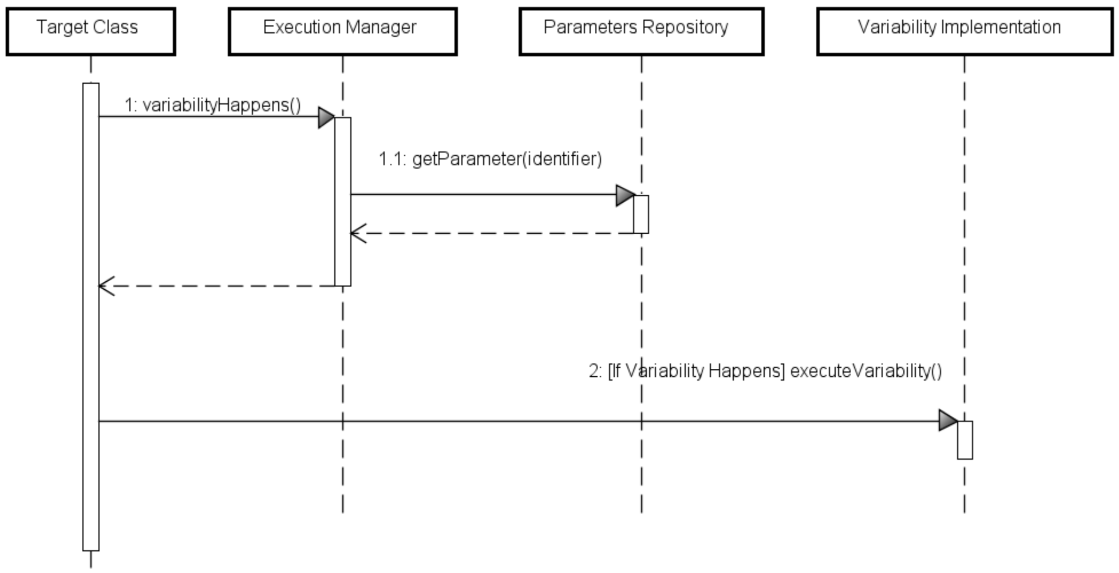

Target Class
Execution Manager
Parameters Repository
Variability Implementation
1: variabilityHappens()
1.1: getParameter(identifier)
2: [If Variability Happens] executeVariability()

Fig. 2. Dynamic of the Conditional Execution pattern

8. CONSEQUENCES

The Conditional Execution pattern brings the following benefits:

- Simplicity. It simplifies the implementation and customization of fine-grained variabilities by allowing the extraction or introduction of SPL variabilities in existing legacy information systems.
- Load-time or runtime customization. Variabilities can be configured at load or execution time. Changes in the system business rules, for example, that demands the addition or removal of variabilities may be performed by the software engineers without the need to recompile the system, such as using techniques like Conditional Compilation.
- Preventing compilations issues. All variabilities are present in the SPL products; the pattern eliminates the possibility of generating compilation errors during the product derivation process compared to the Conditional Compilation technique [Kästner and Apel 2009].
- Reducing code complexity. It avoids the implementation of aspects with complex pointcuts that need exposing several parameters. It allows introducing variability code based on parameter values that can be easily implementing in any mainstream programming languages.

On the other hand, Conditional Execution pattern has the following drawbacks:

- Poor variability modularization. When compared to other techniques such as aspect-oriented programming, the Conditional Execution pattern does not provide an adequate modularization, because it does not promote the separated and localized implementation of fine-grained variability. However, the usage and adaptation of virtual separation of concerns (VSoC) tools [Kästner, Apel and Kuhlemann 2008] can help to overcome this deficiency.
- Late detection of variability correct configuration. Because the Conditional Execution pattern is an implementation technique that enables the runtime decision for the execution of SPL variabilities, the effective test to verify the correct configuration of the system with their respective variabilities can be done just at load time and/or during the system execution.
- Exposure of full product line code. The code of all variabilities is always present in every SPL product, requiring only changes in parameters values for activating each of these variabilities. This may not be suitable for some business models, in which the product price depends on the amount and kind of variabilities enabled.

9. KNOWN USES

In recent years, the Conditional Execution design pattern has been used to implement different types of fine-grained variabilities in the information systems of the Informatics Superintendence (SINFO) at the Federal University of Rio Grande do Norte (UFRN) [SINFO / UFRN 2012] [GIS / SINFO 2012], such as:

- SIPAC - Sistema Integrado de Patrimônio, Gestão e Contratos: It provides fundamental operations for the management of the units responsible for finance, property and UFRN contracts, therefore, acting through activities "middle" of this institution. The SIPAC fully integrates administrative area since the request (material, service, supply of funds, daily tickets, lodging, informational material, maintenance of infrastructure) until internal distributed budget controls.

Among the several features of SIPAC using this pattern, we can cite:

- Lock Materials Purchase: It indicates whether the system will block the request of materials if the available balance is exceeded.
- Editing the Contract Occurrences: If the system possesses the operation to change/delete occurrences of contracts.
- Indication Request Period for Scholarships: The period in which the operation of request for financial aid for students is enabled.

- SIGRH - Sistema Integrado de Gestão e Recursos Humanos: It automates the human resources procedures, such as marking / changing vacation, retire calculations, functional assessment, sizing workforce, frequency control, trainings, online services and applications and functional registers, personal department reports, among others. Most operations have some level of interaction with the nationwide system SIAPE (Sistema Integrado de Administração de Recursos Humanos), while others are only local services.

Among the several features of SIGRH using this pattern we can cite:

- The Training Server: It sets minimum grades and work hours that an employee must reaches to pass in a training course.
- Chief/Vice-Chief Lock Vacations: It sets whether the chief and the vice-chief of a unit can enjoy the vacations in the same period.
- Validation of the Employee Vacations: It defines if the validation rules at the moment of creating a vacation will be applied or not when the manager of the Personal Department creates the vacations for an employee.

- SIGAA - Sistema de Gestão de Atividades Acadêmicas: It automates the procedures in academic area through the modules: undergraduate, postgraduate (stricto sensu and lato sensu), elementary school, high school and technical school; submission and projects control and scholarship research, submission and extension actions control, submission and control of projects education (monitoring and innovations), library management system, reporting of academic professors productions, teaching activities and virtual distance learning environment called "Virtual Class". It also provides specific portals for: rectory, teachers, students, tutors distance learning, coordination (stricto sensu, lato sensu and undergraduate), and evaluation committees (institutional and faculty).

Among the several features of SIGAA using this pattern we can cite:

- The Online Enroll: It sets whether students complete enrollment by the system or in person way only.
- Calculation of Academic Performance Indices: It calculates the indices that assess the academic performance of students in accordance with the strategies that are configured by the pattern.
- Method Evaluation of students: If students will be evaluated by grade, aptitude or concept.
- Punishment for the Delay in the Library: If users suffer fine or suspension for delaying loans of books in the library.

The mechanism of user authentication is another feature that uses the pattern, it is present in all three systems mentioned above, which can use a database or network authentication with LDAP.

Since one of the characteristics of the pattern is its simplicity of implementation, it can be applied easily in information systems. An example of these systems can be the ERPs systems (software integrated enterprise resource planning). These softwares have features for presenting a large amount of variability because they are designed to be deployed in as many customers as possible and each of these customers have a business processes whose variability must be answered.

10. IMPLEMENTATION

This section presents an example of how the Conditional Execution pattern can be implemented. This example is the same shown in Section 2 about the punishment variability for the overdue materials (books, journals, proceedings, among others) from Library Management module. During the discussion of the example is highlighted how the forces involved with the pattern were implemented.

10.1 Step 1: Identifying the Target Classes

Inside the SPL SIGAA, there are two variabilities related to the user punishment functionality: (a) cash fines for each day of delay in return of the material, and (b) suspend the user for a certain amount of days, during which the user cannot do new loans.

The using of the punishments functionality is related mainly with two scenarios of implementation in Library Management module: (i) the execution of a new loan when the system must not allow punished users get a new loans and (ii) at the moment of material return when the system must verify if it has been returned late and then it generates a punishment for the user. These two actions are performed in SPL SIGAA respectively by business classes `LoanProcessor` and `ReturnLoanProcessor`, which are characterized as *Target Classes* of this pattern. Both classes are part of the business layer of the SPL SIGAA which is organized following the tier architectural pattern [Buschmann et al 1996].

Inside the functionality user punishment, there is another action that involves print bill for pay library fines. This action should be only enabled if the variability "fine" is properly configured. Inside the SPL SIGAA, the places where this variability happens, which are characterized as Target Classes are: (i) the Managed Bean `PrintBillPayLibraryFinesMBean`, and (ii) the following JSP pages: `student_menu.jsp`, `professor_menu.jsp` and `server_menu.jsp`.

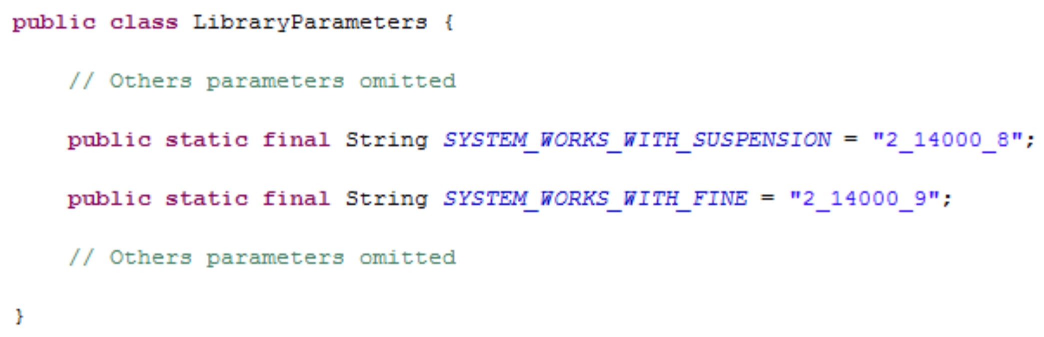
10.2 Step 2: Mapping each identified variability to configurations parameters

After identification of the *Target Classes*, you need to create the parameters corresponding to punishment variability. For this case, there are two punishment variabilities. Thus, two parameters were defined:

- SYSTEM_WORKS_WITH_SUSPENSION
- SYSTEM_WORKS_WITH_FINE

The parameters are declared in a Java interface called `LibraryParameters`, which is shown in Listing 3. This interface is used by *Target Classes* and *Execution Manager* to reference the parameters via Java source code. Each parameter has an identifier. These identifiers are created sequentially, so they are unique for the entire SPL.

Listing 3. Interface with parameters declarations



```
public class LibraryParameters {

    // Others parameters omitted

    public static final String SYSTEM_WORKS_WITH_SUSPENSION = "2_14000_8";

    public static final String SYSTEM_WORKS_WITH_FINE = "2_14000_9";

    // Others parameters omitted

}
```

After that, we need to persist the parameters in the *Parameters Repository.* In the specific case of SPL SIGAA is used the technology of the database to persist parameters. It facilitates create, access and change parameter values. Thus, the creation of parameters is inserting them into the database. Listing 4 shows the creation of the SYSTEM_WORKS_WITH_FINE parameter in the database. Besides the identifier and the parameter value, other s informations such as a brief description and identification of the module of the SPL SIGAA that the parameter belongs are persisted, to facilitate its use through the system.

Listing 4. Parameters creation in repository

```
INSERT INTO comum.parameter (name, description, value, id_subsystem,
id_system, identifier)

VALUES ('SYSTEM_WORKS_WITH_FINE',

'Define if the system will use fine like punishment', 'true', 14000, 2,
'2_14000_8');
```

The extra information above referred is not essential to the operation of the pattern. It is not the purpose of the pattern to define of persistence techniques that have to be used, other techniques can be tried to implement the *Parameters Repository*.

The product derivation can be done by changing the parameter value in the database, as shown in Listing 5. An interesting strategy is creating a use case for manage the SPL parameters, in order to pass the responsibility to choose which variability will be presented in a particular product for some system manager.

Listing 5. Making product derivation using Conditional Execution pattern

```
UPDATE comum.parameter set value = 'false' where identifier = '2_14000_8';
```

Thus, simply changing a value in the database determines the pattern characteristic of **dynamic configuration**.

10.3 Step 3: Implementing the Execution Manager

The *Execution Manager* is the main component in the implementation of this pattern. It contains a logical decision whether a particular variability will be performed or not.

Listing 6 shows a possible implementation of the *Execution Manager*. In this implementation a method checks if a given parameter variability happens retrieving the parameter from repository and returning its Boolean value. If the parameter value is true, the variability is present in the product line and must be executed, if the value is false, the variability must be not executed.

Listing 6. The Execution Manager

```java
public class ExecutionManager {

    public boolean variabilityHappens(String identifier){
        Parameter p = getParameter(identifier);
        return p.getBooleanValue();
    }

    private Parameter getParameter(String identifier) {
        /* ... */
        if (cache.get(identifier) != null) {
            /* Checks if the parameter time in the cache has expired */
            Long time = cacheTime.get(identifier);
            Integer maxTime = cacheMaxTime.get(identifier);
            if(System.currentTimeMillis()-time > maxTime ) {
                cache.put(identifier, null); /* Force the loading */
            }
        }

        if (cache.get(identifier) == null) {
            /* ... */
            ParametroDao dao = new ParametroDao();
            /* Updates the value in the repository */
            Map<String, Object> map = dao.getParameterAsMap(identifier);
            /* ... */
            cache.put(identifier, new Parameter());
            cacheTime.put(identifier, System.currentTimeMillis());
            cacheMaxTime.put(identifier, maxTime);
            /* ... */
        }else {
            return cache.get(identifier);
        } return null;
    }
}
```

Normally, due to the use of this pattern, there is a considerable amount of access to the configuration parameters that can affect the system performance. The large number of data accesses to the *Parameters Repository* is because the system, during its execution, has to check in all places where a determined variability may happen (variations points), if this variability is present in that particular product or not. Even if changes in a specific variability present in a product are rare, this checking is always necessary.

For this reason, it is recommended to adopt a strategy to minimize the number of accesses to the *Parameters Repository*. This strategy can be implemented in various ways. It is not the objective of the pattern to define a specific implementation strategy to achieve this aim.

How in this example of implementation we used a database system, to minimize the impact on system performance, it was implemented a cache strategy. This implementation maintains information about the last time that the value of each parameter was synchronized with the value of the repository. When timeout is reached, the *Execution Manager* reloads the parameter value. This strategy is implemented by the method `getParameter` in Listing 6.

If you followed this strategy of cache, the *Execution Manager* should also keep an operation that forces a reload of a parameter or set of parameters whenever requested. This is useful when the value of a parameter is changed and this change should be immediately reflected in the behavior of the system.

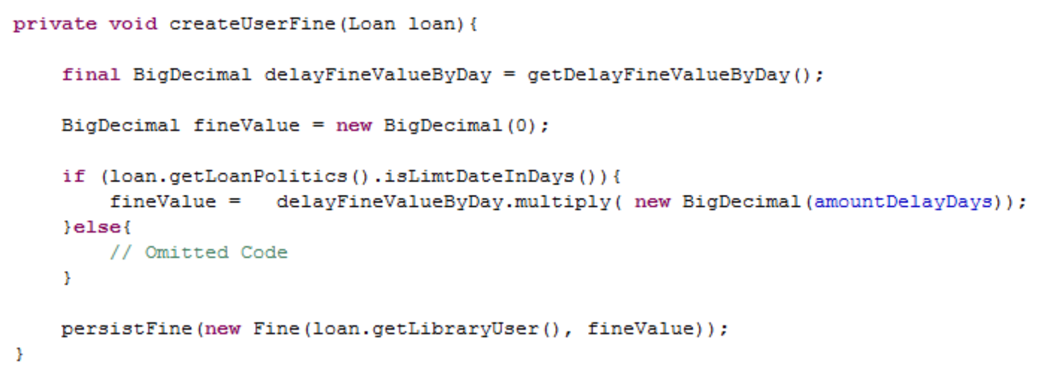
The caching strategy used by *Execution Manager* ensures low impact on system **performance**, reducing queries to the repository, one of the forces that the proposed pattern undertakes to deliver.

This third step of the pattern implementation needs to be done once. This implementation can usually stay in the SPL architecture and it be used by all variability.

10.4 Step 4: Defining the Variability Implementation

The next step is to implement the specific code for each system variability. This code will be called by the *Execution Manager* if the variability happens in the product. Listing 7 shows a simplified code snippet of fine variability implementation in SIGAA SPL.

Listing 7. Fine variability implementation

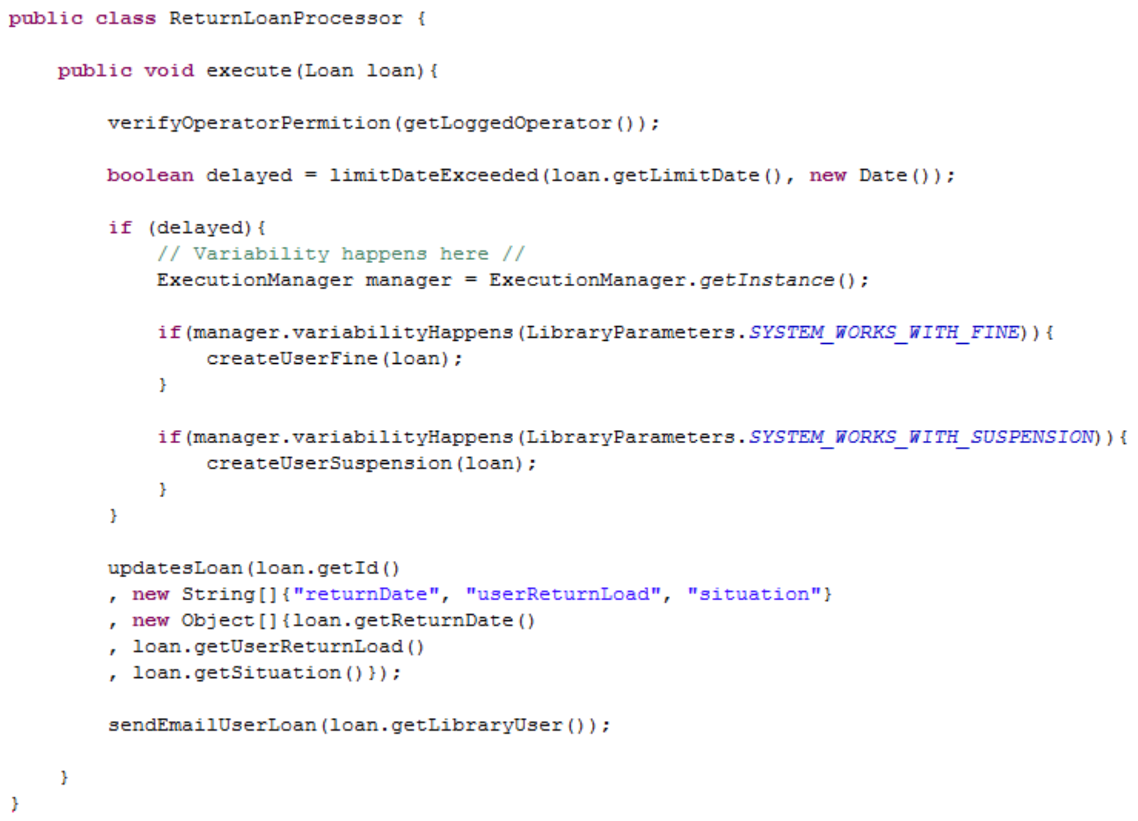


```
private void createUserFine(Loan loan){

    final BigDecimal delayFineValueByDay = getDelayFineValueByDay();

    BigDecimal fineValue = new BigDecimal(0);

    if (loan.getLoanPolitics().isLimtDateInDays()){
        fineValue =   delayFineValueByDay.multiply( new BigDecimal(amountDelayDays));
    }else{
        // Omitted Code
    }

    persistFine(new Fine(loan.getLibraryUser(), fineValue));
}
```

10.5 Step 5: Calling Conditional Execution from Target Classes

A conditional execution is implemented to generate the punishment for the material delay return depending on which of variability is currently enabled. Listing 8 shows the business class code `ReturnLoanProcessor` that is responsible to implement such functionality with the introduction of variability. To have a clear perception of this pattern, we can compare the code in Listing 8 with the code shown in Listing 1, which is presented in the same business class without using the Conditional Execution pattern.

Listing 8. Example of Conditional Execution in business layer

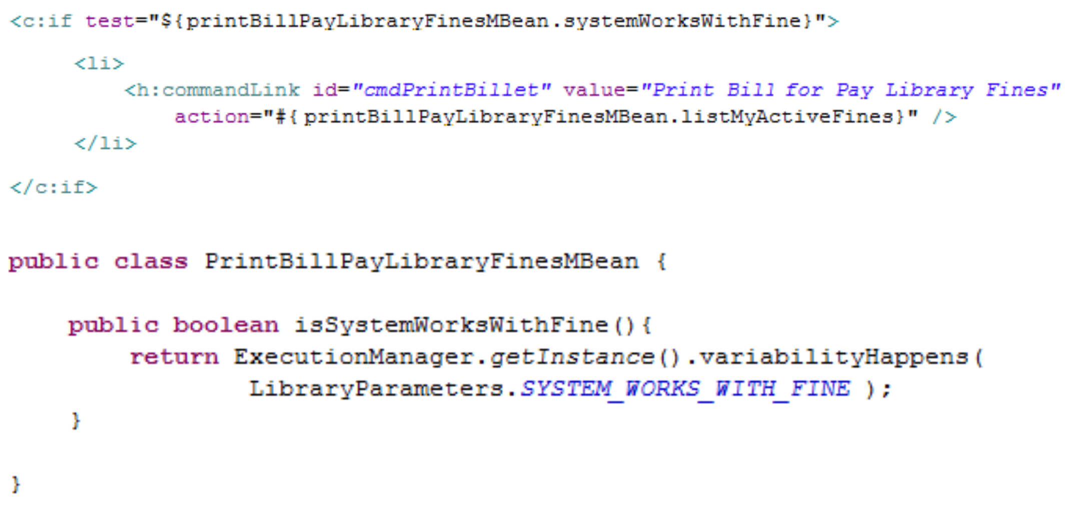


```
public class ReturnLoanProcessor {

    public void execute(Loan loan){

        verifyOperatorPermition(getLoggedOperator());

        boolean delayed = limitDateExceeded(loan.getLimitDate(), new Date());

        if (delayed){
            // Variability happens here //
            ExecutionManager manager = ExecutionManager.getInstance();

            if(manager.variabilityHappens(LibraryParameters.SYSTEM_WORKS_WITH_FINE)){
                createUserFine(loan);
            }

            if(manager.variabilityHappens(LibraryParameters.SYSTEM_WORKS_WITH_SUSPENSION)){
                createUserSuspension(loan);
            }
        }

        updatesLoan(loan.getId()
        , new String[]{"returnDate", "userReturnLoad", "situation"}
        , new Object[]{loan.getReturnDate()
        , loan.getUserReturnLoad()
        , loan.getSituation()});

        sendEmailUserLoan(loan.getLibraryUser());

    }
}
```

We can see in the code in Listing 8 that the implementation was carried out with a **low impact**, since no change was needed in the core of SPL. Besides, the call to *Execution Manager* is a simple implementation of the method invocation and it can be perfectly included at the point where the dynamic derivation should occur.

Another variability related to the delay of the loan is SIGAA SPL variability *"Print Bill for Pay Library Fines"*. This functionality should be performed only if the variability "fine" is active in the system. In this case, a link is shown to the user allowing the printing of payment bill. This is also a typical example of conditional execution of fine-grained variability that can be implemented using the pattern. Listing 9 shows a snippet of a webpage of LPS SIGAA, which includes a link to print the bill only if the variability "fine" is active. Moreover, it is also presented the code snippet of class `PrintBillPayLibraryFinesMBean` that is responsible for accessing the *Execution Manager* to verify the occurrence of variability in the product.

Listing 9. Example of Conditional Execution in view layer



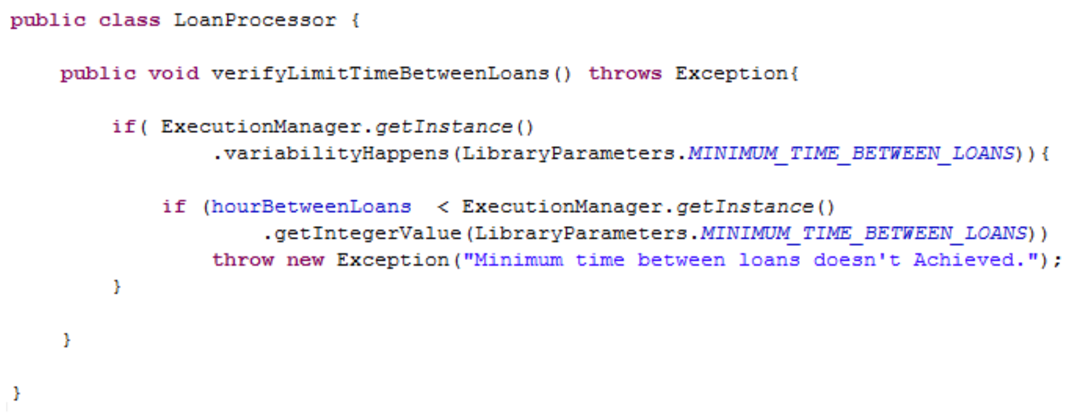
```
<c:if test="${printBillPayLibraryFinesMBean.systemWorksWithFine}">

     <li>
         <h:commandLink id="cmdPrintBillet" value="Print Bill for Pay Library Fines"
             action="#{ printBillPayLibraryFinesMBean.listMyActiveFines}" />
     </li>

</c:if>


public class PrintBillPayLibraryFinesMBean {

    public boolean isSystemWorksWithFine(){
        return ExecutionManager.getInstance().variabilityHappens(
                LibraryParameters.SYSTEM_WORKS_WITH_FINE );
    }

}
```

11. VARIANTS

The Conditional Execution pattern can be used in a way that it performs more complex decisions than just Boolean decisions that verify if a variability will be present or not in a product. This section describes two variant implementations of the pattern that are being using by SINFO/UFRN.

To illustrate the first approach, consider the functionality that checks the timeout between loans in the Library System modulo of SPL SIGAA. For this functionality there is an integer value that defines the minimum time, in hours, that a user can make a new loan of the same material. Besides, it makes possible to determine a minimum time between different loans for each product, this verification can be disabled in some products. Listing 10 shows the code of the *Target Class* where occurs this verification.

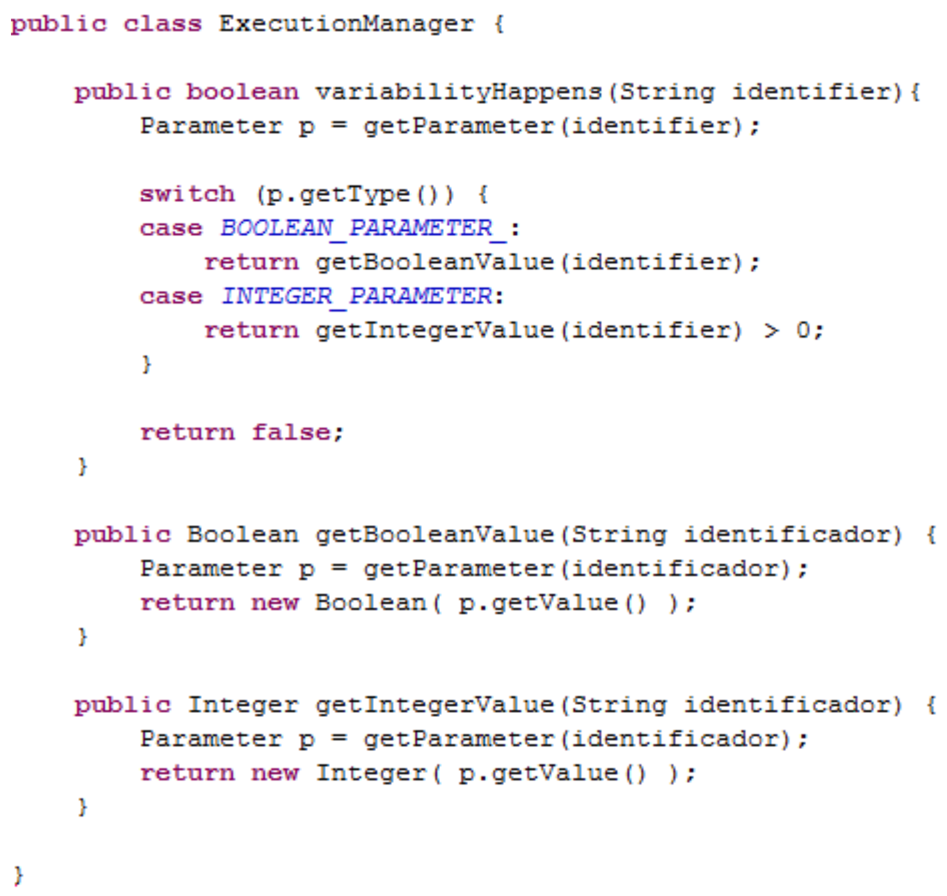
Listing 10. Conditional Execution of Integer Variability



```
public class LoanProcessor {

    public void verifyLimitTimeBetweenLoans() throws Exception{

        if( ExecutionManager.getInstance()
                .variabilityHappens(LibraryParameters.MINIMUM_TIME_BETWEEN_LOANS)){

            if (hourBetweenLoans  < ExecutionManager.getInstance()
                    .getIntegerValue(LibraryParameters.MINIMUM_TIME_BETWEEN_LOANS))
                throw new Exception("Minimum time between loans doesn't Achieved.");
        }

    }

}
```

Instead of using a Boolean parameter to determine if the variability should occur, it was created only one parameter of type integer. If this parameter value is greater than zero, the *Execution Manager* enables the execution of verification. At the same time, the parameter value is also used to calculate if the minimum time between loans was obeyed, throwing an exception if the business rule is not followed. Listing 11 shows an alternative implementation of the *Execution Manager* that verifies if a variation of type integer occurs, besides the method that returns this integer value to be used by the variability.

Listing 11. Execution Manager for Integer Variabilities



```
public class ExecutionManager {

    public boolean variabilityHappens(String identifier){
        Parameter p = getParameter(identifier);

        switch (p.getType()) {
        case BOOLEAN_PARAMETER_:
            return getBooleanValue(identifier);
        case INTEGER_PARAMETER:
            return getIntegerValue(identifier) > 0;
        }

        return false;
    }

    public Boolean getBooleanValue(String identificador) {
        Parameter p = getParameter(identificador);
        return new Boolean( p.getValue() );
    }

    public Integer getIntegerValue(String identificador) {
        Parameter p = getParameter(identificador);
        return new Integer( p.getValue() );
    }

}
```

To this alternative implementation was added the information “type” to the parameter, beyond the information previously shown in Listing 4. This information allows determine how the variability should be treated. We call the variabilities that use an integer type parameter of “Integer Variabilities”. We emphasize that this was a solution adopted and it is not the goal of the pattern determines how these implementation details must be performed.

The second variant is based on removing of the dependency that the *Target Classes* have with the *Variability Implementation*. In this approach, the *Execution Manager* doesn’t just determine whether a *Variability Implementation* will be performance in a *Target Class*, but it instantiates and returns to the *Target Class* the *Variability Implementation* to be executed.

The solution adopted for this second approach makes use of the Strategy design pattern [Gamma et al 1995]. Some strategies are defined. The names of the classes that implement these strategies are mapping for parameters. If the variability is presented in the product, the *Execution Manager* instantiates the concrete strategies contained in those parameters.

For this variant was created a new type of parameter called “Strategy Parameter”. This kind of parameter doesn’t contain a Boolean or Integer value, but a class name that implements the strategy. The variabilities that use this kind of parameter we call of “Strategy Variabilities”. Figure 3 outlines the structure for this second variant in the implementation of the pattern. The functionality used is the same of Listings 1 and 8, the punishment of users for Library Management module of SPL SIGAA.

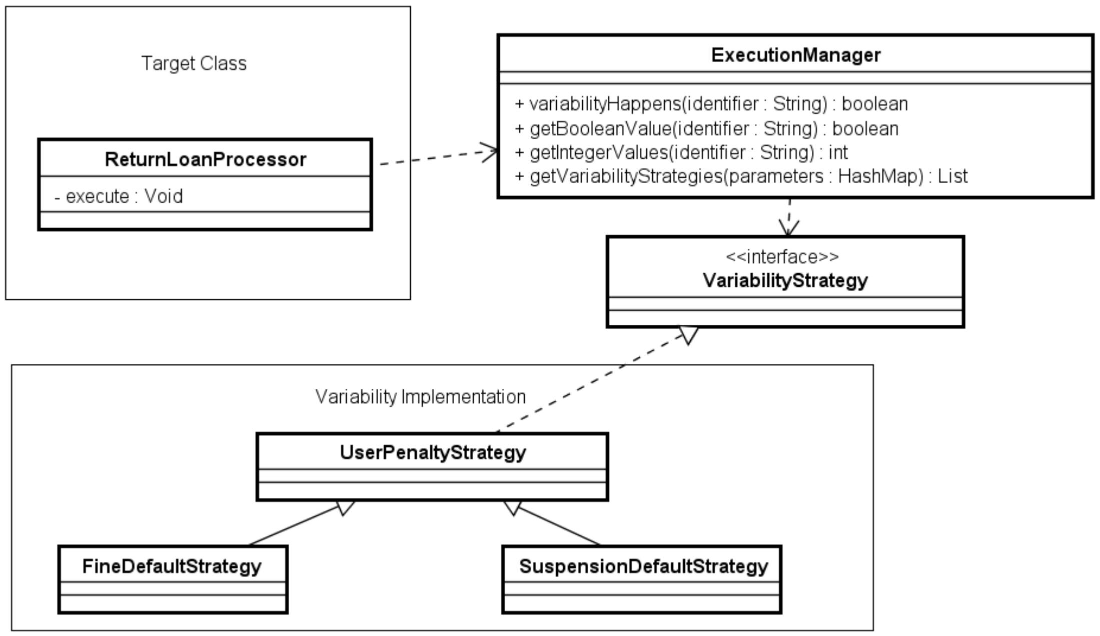

Target Class
ReturnLoanProcessor
- execute : Void
ExecutionManager
+ variabilityHappens(identifier : String) : boolean
+ getBooleanValue(identifier : String) : boolean
+ getIntegerValues(identifier : String) : int
+ getVariabilityStrategies(parameters : HashMap) : List
<<interface>>
VariabilityStrategy
Variability Implementation
UserPenaltyStrategy
FineDefaultStrategy
SuspensionDefaultStrategy

Fig. 3. Example of Strategy Variability implementation using Conditional Execution pattern

In this variant, the method `getVariabilityStrategies` of *Execution Manager* receives a map of parameters. The map keys are Boolean parameters which determine whether the variabilities happen and the map values are strategy parameters containing concrete strategies to be instantiated. Listing 12 contains the code for a possible implementation of the method `getVariabilityStrategies`.

Listing 12. Example of Execution Manager for Strategy Variability

```
public class ExecutionManager {

    public List<VariabilityStrategy> getVariabilityStrategies(Map<String, String> map){

        List<VariabilityStrategy> list = new ArrayList<VariabilityStrategy>();

        for (String parameterId : map.keySet()) {
            if(variabilityHappens(parameterId)){
                String strategyId = map.get(parameterId);
                String strategyClass = getParameter(strategyId).getValue();
                // Instantiates the class of concrete strategy using Reflection
                list.add( ReflectionUtils.newInstance( strategyClass ) );
            }
        }

        return list;
    }

}
```

If the variability contained in the map key occurs, the *Execution Manager* instantiates the strategy contained in the map value, returning to the *Target Class* a list of strategies that should be executed in the product and performing the product derivation at runtime.

Listing 13 shows the implementation of the same variability shown in Listing 8, but now using the approach of strategy variability modularizing to treat the functionalities "fine" and "suspension".

Listing 13. Target Class using Conditional Execution for Strategy Variability

```
public class ReturnLoanProcessor {

    public void execute(Loan loan){

        verifyOperatorPermition(getLoggedOperator());

        boolean delayed = limitDateExceeded(loan.getLimitDate(), new Date());

        if (delayed){

            ExecutionManager manager = ExecutionManager.getInstance();

            List<VariabilityStrategy> strategys = manager.getVariabilityStrategies(map);

            for (VariabilityStrategy strategy: strategys) {
                UserPenaltyStrategy ups = (UserPenaltyStrategy) strategy;
                ups.createUserPenalty(loan);
            }

        }

        updatesLoan(loan.getId()
        , new String[]{"returnDate", "userReturnLoad", "situation"}
        , new Object[]{loan.getReturnDate()
        , loan.getUserReturnLoad()
        , loan.getSituation()});

        sendEmailUserLoan(loan.getLibraryUser());

    }

}
```

In this context, to add a new variability to SPL is necessary to create a new strategy punishment that "extends" the class `UserPenaltyStrategy`. It must insert the new parameters to the variability in *Parameters Repository* and add them to the map passed as a parameter to the method `getVariabilityStrategies` of the *Execution Manager*. Thus, a new strategy is instantiated by the *Execution Manager* and returned to be performed without need to add a new conditional statement in the *Target Class*, always that a new variation is added to the product line.

Listing 14 shows an example of information about three kinds of parameters that can be used to implement this pattern.

Listing 14. Example of three kinds of parameters used by the suggested implementation of the pattern

```
-- boolean parameter --
INSERT INTO comum.parameter (identifier, name, value, id_system, id_subsystem,
type )
VALUES ('2_14000_8', ' SYSTEM_WORKS_WITH_FINE', 'true', 2, 14000, 1);

-- integer parameter --
INSERT INTO comum.parameter (identifier, name, value, id_system, id_subsystem,
type)
VALUES ('2_14000_10', 'MINIMUM_TIME_BETWEEN_LOANS', '24', 2, 14000, 2);

-- strategy parameter --
INSERT INTO comum.parameter (identifier, name, value, id_system, id_subsystem,
type)
VALUES ('2_14000_20', ' CLASS_NAME_SUSPENTION_STRATEGY',
'br.ufrn.sigaa.circulacao.negocio.SuspensionDefaultStrategy', 2, 14000, 3);
```

Other variants to the pattern implementation can be created, it depends if will appear other variabilities which behave differently from those ones, described in this paper.

ACKNOWLEDGEMENTS

We are grateful to Professor Dr. Eduardo Guerra for his valuable suggestions during the shepherding process that contributed to a broader understanding of the proposed pattern. We also thank all the suggestions given by the participants of the Writers' Workshop of SugarLoafPLop'2012.

REFERENCES

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996): Pattern-Oriented Software Architecture, A System of Patterns, vol. 1. Wiley.
Clements, P., Northrop, L. (2001): “Software Product Lines: Practices and Patterns”, Addison-Wesley Professional, 2001.
Czarnecki, K., Eisenecker, U. (2000): “Generative Programming: Methods, Tools, and Applications”, Addison-Wesley, 2000.
Gamma, E., Helm, Richard., Johnson, R., Vlissides, J. (1995). “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
Kästner, C., Apel, S., Kuhlemann, M. (2008). “Granularity in software product lines”. In Proceedings of the 30th international conference on Software engineering (ICSE '08). ACM, New York, NY, USA, 311-320.
Kästner, C., Apel, S. Virtual Separation of Concerns - A Second Chance for Preprocessors. Journal of Object Technology. Vol. 8. No 6. 2009
SIG/SINFO – Management Integrated Institutional System of Informatics Superintendence at UFRN: Available at: http://www.info.ufrn.br/wikisistemas, Oct 2012.
SINFO/UFRN – Informatics Superintendence at UFRN (2012). Available at: http://www.info.ufrn.br, Oct 2012.
Weiss, D., Lai, C.. (1999): “Software Product-Line Engineering: A Family-Based Software Development Process”, Addison-Wesley Professional, 1999.

Received May 2012; revised June 2012; accepted August 2012