International Journal of Engineering Research and Technology.
ISSN 0974-3154 Volume 11, Number 3 (2018), pp. 501-528

http://www.irphouse.com

Patterns for Effective Handling of Exceptions in Processes Using Different Modeling Languages and Notations

Preetesh Purohit[1]

Ph.D. Scholar, Institute of Engineering and Technology, DAVV, Indore, India.

Vrinda Tokekar[2]

Professor, Institute of Engineering and Technology, DAVV, Indore, India.

Abstract

Process modeling permits for scrutiny and refinement of processes that bring together multiple stakeholders working together to carry out a job. Process modeling generally concentrates on how the participation takes place when everything goes as expected. Sadly, processes of real-world rarely carry out that easily. A more complete and detail scrutiny of a process requires that the process model also include details about what to do when unhappy path or exceptional conditions occur. We have observed that, in many scenarios, there are abstract patterns that catch the relationship between the standard expected process and unhappy path. We have confidence that process patterns can simplify the documentation, development, and maintenance of process models. We report these patterns using Business Process Model and Notation (BPMN), Little-JIL, and Unified Modeling Language (UML). We represent both the rational structure of the pattern as well as specimens of the pattern in use.

Keywords: Activity Diagram, BPMN, Little-JIL, Patterns, Process Modeling, Process Modeling Languages and Notations, UML

I. INTRODUCTION

Sequence of activities set in motion to produce the desired result or a product is known as a process. The vital importance of being sure that processes in different

domains are free of defects and efficient has led to flourishing interest in how best to depict these processes with models. It is crucial for a process model to include details of the process behavior under exceptional conditions, to grasp and evaluate a process completely. Focus of our work is on routine approaches to exception handling in real-world processes and problems that occur when pointing out correctly these exceptions handling mechanism in dissimilar process modeling languages and notations.

Process models commonly depict how all entities are coordinated and related to support better comprehension of the processes being modeled and to support scrutiny that could lead to refinements to the processes. Process models have been used in many application domains, such as and e-government [1], [2], [3], [4], healthcare provision [5], [6], [7], business [8], [9], [10], [11], software engineering [12], [13], [14].

I.I Processes and Exceptions

We believe that models for processes that do not describe behaviors precisely and carefully are neither complete nor adequate. Thus, we are unsatisfied with the mechanism of managing with exceptions by permitting a process model to be changed dynamically in real time when an exception arises (for example, as suggested in [15]).

In the medical field, inexact or misplaced design of how a process should deal with exceptional circumstances can lead different people to handle the same circumstances in a different way, based on individual style, level of knowledge, level of experience, and the actions of other people [5]. So far, Henneman et al. [16] study that models of medical processes frequently capture only the normative process and leave out plan of how to handle exceptions.

Since we believe that it is necessary that process models incorporate sufficient specifications of exception handling, we have paid substantial consideration to exceptional scenarios in our effort with processes. Thus, the work we explain here has two aims: i) the recognition of usual exception management ways and ii) the lucid illustration of these ways as components of models of processes that incorporate exception handling.

I.II Patterns for Managing Exceptions in Processes

To tackle the first aim of this paper, we have extensively explored the use of patterns in the past work. The concept of patterns gained importance in the computer science society with the book of Design Patterns: Elements of Reusable Object-Oriented Software [17] in 1994. Stelting [18] presents how to use design patterns to handle exceptions. Haase [19] describes exception handling idioms in the context of Java programming. Longshaw and Woods [20], [21] explain patterns of exception handling for multitier information systems. The notion of process patterns has been explored by Coplien [22] and later by Ambler [23]. Russell, van der Aalst, and ter Hofstede have begun to examine the occurrence of patterns within workflow. They classify patterns

in four workflow definition semantic domains: control flow [24], [25], data flow [26], resources [27], and exception handling [28]. Osterweil's research [29] suggests that this is no less significant and no less possible in a process language and process model and than in programming languages and application software.

Our understanding in defining processes in a diversity of domains has indicated that certain behaviors return frequently and thus seem to contain specifiable patterns that are very much in the same spirit as design and programming language patterns. The recognition and the subsequent use of such patterns have facilitated writing and reasoning about processes that employ these patterns. Some of these patterns deal particularly with exceptions and their management. Therefore, we think that identification of exception handling patterns and use of regular idioms to code them can show the way to enhanced readability and understandability of process definitions.

I.III Establishing Patterns for Managing Exceptions in Process Modeling Languages

To tackle the second aim of this work, we support the use of process modeling language that includes precise services for robustly supporting the modeling of process exceptional conditions and their management. A suitably expressive language, for example, would be one that facilitates the preferred obvious division of exceptional conduct from expected conduct and can serve as a medium for keeping large and difficult process definitions in scholar control. It is generally thought that support for the clear, explicit specification and handling of exceptions in application programming languages such as Java makes programs written in these languages clearer and more agreeable to effective scholar control. Adding analogous mechanisms to a process modeling language gives process modeler comparable facilities when modeling processes. However, different process modeling languages include dissimilar constructs that get across with exception handling in appealing ways, which turns out to have a major impact on how obviously and particularly the different languages are able to depict the patterns.

I. IV Procedure

While describing processes, we have documented strong similarities among the traditions in which the field experts have described how they treat exceptional conditions. This led to vigilant attempts to outline and classify these dissimilar approaches to exception management. Primarily, we easily recognized three unlike rational ways:

- Placing other odd jobs prior to returning to the regular process.
- Terminating the in progress processing.
- Demonstrating possible choices to execute the identical job.

We considered, BPMN [30], Little-JIL [31] and UML activity diagrams [32] as medium for defining these patterns, and establish strengths and weaknesses, of notation used. We also analyzed the significant contributions of researchers made in connected areas [33] and our work is inspired as per the past contribution [17], [34]. We use all three notations to define exception handling patterns and demonstrate their possible use.

The article commences by describing the exception management patterns that we have recognized. Each one is first defined and described informally, and then we explain how the pattern would be depicted in two of the process notations whose modeling language characteristics exhibit compelling options in how the exception handling patterns can be expressed. We present instance of the pattern as well as general divergences of the pattern. The paper finally wraps up in section III.

II. PATTERNS FOR MANAGING EXCEPTIONS

Patterns are best known in the perspective of object-oriented design. Object-oriented design patterns [17] demonstrate fascinating ways to join classes and explain methods to tackle ordinary design issues, permitting software designers to reuse high-level answers to problems instead of recreating solutions for each new design issue. Likewise, we concisely launch the exception handling patterns that we have recognized, following the manner suggested in the classic Design Patterns book [17].

We arrange the patterns into a group of categories. We explain the character of each category and then introduce the exact patterns that it contains. Our cases are taken from diverse domains to propose the generality of the patterns.

II.I Embedding Manner of Conducting Action

Routinely noticed way to addressing a process specification issue is to embed supplementary actions that are required in order to fix issues that have been recognized during execution of some job. A chief feature that determines different repairing patterns is the timing of the fixing or repairing activity with respect to when the exception or error is found. In Instantaneous Repairing, the issues are handled before continuing with the job, whereas in Postponed Repairing the issue is noted, possibly worked around, and then addressed completely in future.

Another significant concern is the nature of the repairing activity. One option is for the repairing activity to be a completely new activity designed particularly for the intention of treating the particular exception. Another option is for the repairing activity is to integrate recurrence of earlier activities, resulting in a Reiterate or Error-Compelled Remake pattern.

In this segment, we first present the Instantaneous and Postponed Repairing patterns. Then, we show how Reiterate can be used to rerun a job at the time that it fails. Lastly, we present a more general Error-Compelled Remake pattern in which a job's failure is not detected instantaneously, requiring the job to be reperformed at a later

time.

II.I.I Name: **- Instantaneous Repairing** Pattern

Pattern's Intent: When a non regular circumstance is noted, a response is taken to resolve the issue that caused this situation before continuing with the rest of the process.

Pattern's Applicability: This template permits the infusion of additional action to handle expected, but non regular situations. It is helpful in circumstances where some likely troublesome issue may arise and a simple method exists to resolve the issue in such a way that the process can however pursue.

Pattern's Structure: Using BPMN we present the design of the Instantaneous Repairing Pattern in Fig. 1. Here, the non regular circumstance is depicted as an **Intermediate (catch) Event** connected to the end of a Job. When an **Event** with one of the particularized triggers is encountered, the outflow of the process is immediately diverted through the **Intermediate Event**, halting rest of the work within the job.

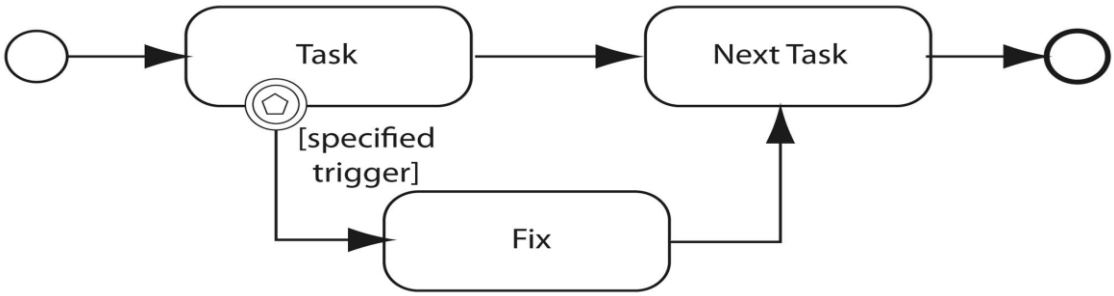

Fig. 1. Instantaneous Repairing pattern in BPMN

The new route of the process is called **Exception Flow**. In the structure, the **Exception Flow** points to a repairing activity before reconnecting the routine route (**Normal Flow**). The adoption of the control flow edges to place the exception handler in the process makes it so apparent where control flows upon culmination of the exception handler.

Control flow following exception handling is not depicted distinctly in either Little-JIL or in UML. Little-JIL additionally demands understanding of the semantics of the continuation icons being used, while in UML, it depends upon where the exception handler is associated. Fig. 2 shows the design of the Instantaneous Repairing pattern using UML. If any exception arises during the execution of the Job activity (details of the activity not depicted in the figure), the exception is raised to the **CallBehaviorAction** that initialized the activity call. The exception is then caught by the **Exception Handler**, which calls the *Repair/Fix* activity. After having repaired the issue, the process continues its execution by calling the *Next Job/Task* activity.

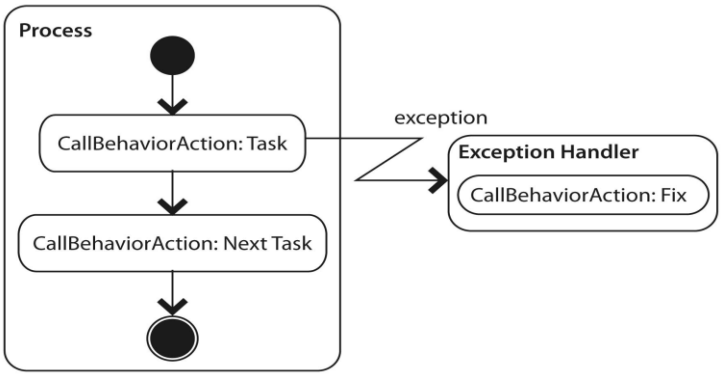

Fig. 2. Instantaneous Repairing pattern in UML

Pattern's Participants: Instantaneous Repairing pattern have two participants: the anomaly discoverer/detector and the repairer/fixer. The **anomaly discoverer** is the part of the process that identifies that an anomaly has occurred and notifies the process by throwing an exception. The **repairer** is the exception handler that repairs the issue and permits the process to continue.

Pattern's Instances: A sample of BPMN process in software development that illustrates the Instantaneous repairing pattern is shown in Fig. 3. Instantaneous repairing manages exceptions caused by compilation errors that may arise in the **Sub-Process** *Code the Modules* (that is executed multiple times). After repairing the error, the control flow for this instance of coding abolishes.

Pattern's Deviations: Moreover to embedding manner of conducting action, it is also practicable to use this pattern to omit some jobs in the process that are improper in the context of the exception. This is achieved by allocating the exception handler at the suitable level of the calling hierarchy.

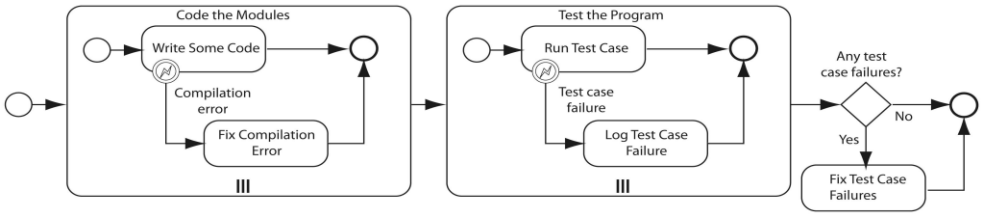

Fig. 3. Instantaneous and Postponed Repairing patterns in software development

II.I.II Name: - **Postponed Repairing** Pattern

Pattern's Intent: When a non regular circumstance is noted, action must be taken to note the situation and in some way address the situation either temporarily or partially because addressing the situation fully is neither necessary nor immediately possible. Afterward, a supplement action needs to be taken to complete the recovery from the

condition that caused the occurrence of the non regular circumstance.

Pattern's Applicability: This pattern is appropriate in preventing the process from coming to a stop even though the likely troublesome effects of an uncommon, yet foreseeable, circumstance cannot be addressed entirely. The pattern is helpful in situations where addressing the issue conclusively is attainable only when more time or facts becomes available, where the need for further work to finish the handling of the exception can be grabbed in the state of the process, and where interim measures can enable the process to proceed to the state where such additional time and knowledge have become available.

Pattern's Structure: Representation of the design of this pattern in Little-JIL is shown in Fig. 4. An exception is thrown at the time of the execution of *Substep 1*. The exception is managed by *Do interim/temporary repair/fix*, an exception handler that makes some worthwhile interim adjustment records the requirement for a more repair, and then, returns to regular processing, as marked by the **continue** handler. However, at some later point in the process, a supplement step (or group of steps), represented by *Some step* in the figure, must be executed to either complete the handling of the non regular circumstance or check that the non regular condition no longer exists. This test is made by an **edge predicate**, represented by the condition in parenthesis, before executing *Some step*, which examines the process state to find if the repair is needed. Note that the dotted line representation is not syntax, but is proposed just to perceive that an arbitrary amount of work may happen between when the temporary repair takes place and the repair is completed.

The structure of the Postponed Repairing pattern in BPMN is shown in Fig. 5. The **Exception Flow** includes an interim repairing activity that includes the creation of a problem report. It then flows back into the **Normal Flow**, which may include any number of activities (indicated informally using a dotted line). A **Gateway** is then used to test whether a problem report exists in which case a full repairing is carried out.

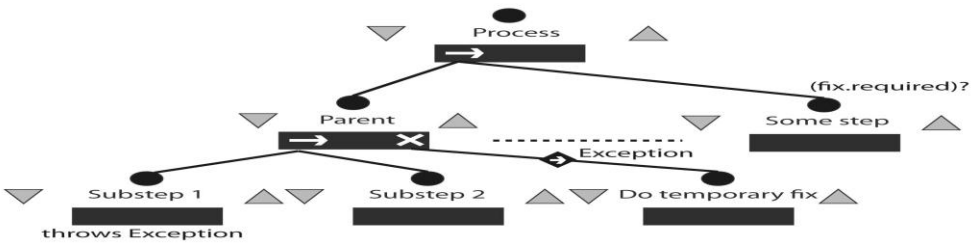

Fig. 4. Postponed Repairing pattern in Little-JIL

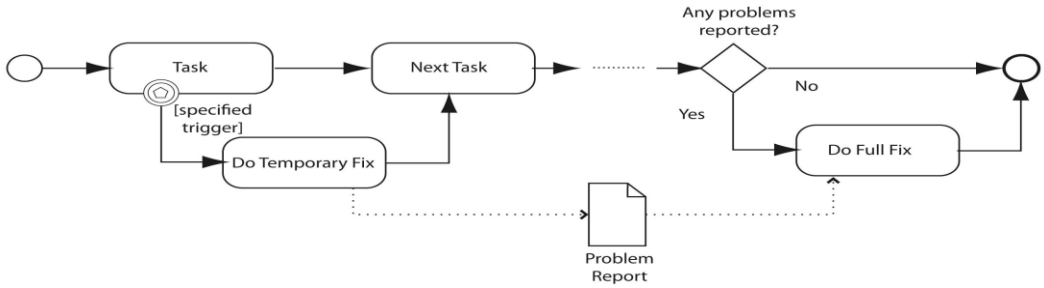

Fig. 5. Postponed Repairing pattern in BPMN

Pattern's Participants: There are three participants in the Postponed Repairing pattern: the anomaly discoverer/detector, the patcher/logger, and the repairer/fixer. The **anomaly discoverer** is the portion of the process that identifies that an issue has arisen and notifies the process by throwing an exception. The **patcher/logger** is accountable for recording the anomaly and possibly doing an interim repair. In the Postponed Repairing pattern, the patcher/logger is the exception handler. The **repairer** is the later step that examines the log and completes the handling of the non regular circumstance. Notice that the repairer does not use an exception handling mechanism, yet is a key participant in resolving the anomaly.

Pattern's Instances: Example of the Postponed Repairing pattern is included in Figure 3. Postponed Repairing handles exceptions caused by potential test case failures during program testing (represented in the **Sub-Process** *Test the Program*). Here, every failure is recorded in a test log before the control flow for this instance of testing terminates. Failures, if recorded, are fixed only after all instances of testing have been completed.

Another example of the Postponed Repairing pattern is shown in Fig. 6, in Little-JIL. Here, the commuter has successfully booked/reserved a flight, but the Website that is used to allow the user to choose a seat is unavailable. *Reserve flight* throws the *SeatSelectionWebsiteIsDown* exception. This is managed by making a note to choose seats later and then continuing with reserving the hotel and car. At some later stage in the process, verification is made to see if the seats have been chosen. If not, the *Select plane seats* step is executed.

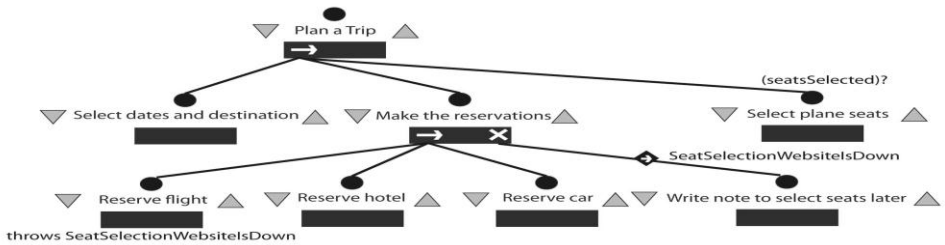

Fig. 6. Postponed Repairing pattern to complete seat selection at a later time

II.I.III Name: - **Reiterate** Pattern

Pattern's Intent: When an issue is identified immediately after the execution of the activity causing the issue, an action is taken to resolve the issue and then the activity that caused the issue is reiterated.

Pattern's Applicability: This pattern is useful when an activity fails but a change to the state or addition of a short delay seems likely to allow the activity to succeed if it is reiterated. This is a usual approach when input to an activity is wrong, such as a debit card number, or transient hardware failures take place, during working on Internet.

Pattern's Structure: The design of the Reiterate pattern in Little-JIL is depicted in Fig. 7. During the *Do the work* step, an exception is thrown. It is managed by the *Reiterate* step, which first performs a step to *Update Context* followed by recursively carrying out the *Task* step. The *Update Context* step may also be accountable for finding whether to continue with reiterating the *Task* or whether to leave it and propagate the exception to be managed somewhere else. This is crucial to refrain retrying the same task continually. On finishing of the *Reiterate* step, the *Task* that it is an exception handler for is complete. The structure of Reiterate pattern in BPMN is shown in Fig. 8. In design; the Exception Flow contains *Update Context*, and then, bends back to *Task*. An extra Exception Flow is defined for leaving out retry and making known the exception to be handled somewhere.

Pattern's Participants: This pattern contains the same two participants as in Instantaneous Repairing. The **anomaly discoverer/detector** is identical, but the **repairer/fixer** has a more polished design consisting of a step to revise the perspective prior to a **reiterable**/**retriable activity**, which is the activity that is iteratively or recursively called upon after the context update.

Pattern's Instances: We present the trip planning process using the Reiterate pattern in Fig. 9. If it is not likely to get a flight that fits the primary plan, the *Book/Reserve flight* step throws the *FlightNotAvailable* exception. This is managed by the *Revise plan* step, which updates the dates and then uses the *Make the reservations* step recursively. When the exception handler completes, the initial *Make the reservations* step is also complete due to the **complete** semantics associated with the exception handler.

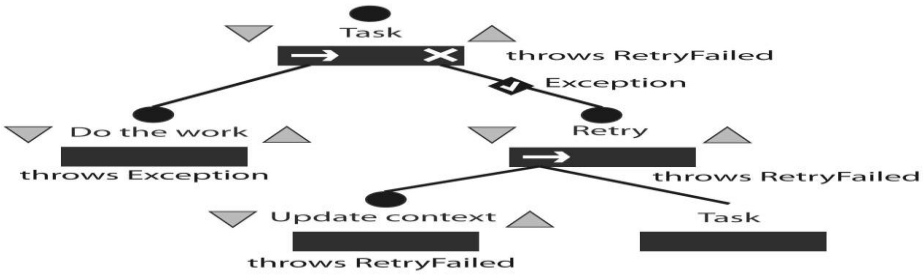

Fig. 7. Reiterate pattern in Little-JIL

Pattern's Variations: In few scenarios, it may be reasonable for the repairer to be missing or play a minimal role. For example, if the exception being handled is that a Website is not responding, the repairer might simply add a pause prior to retrying the Website or it might tally the number of retries and terminate if repeated efforts fail.

II.I.IV Name: - **Exception-Compelled Remake** Pattern

Pattern's Intent: Between the occurrence of an issue and its discovery an arbitrary amount of time can pass. During that duration, other activities whose executions depend on the activity in which the issue occurred can be executed. Once the issue is detected, the repairing of the issue includes the reexecution of the activity that introduced the issue originally.

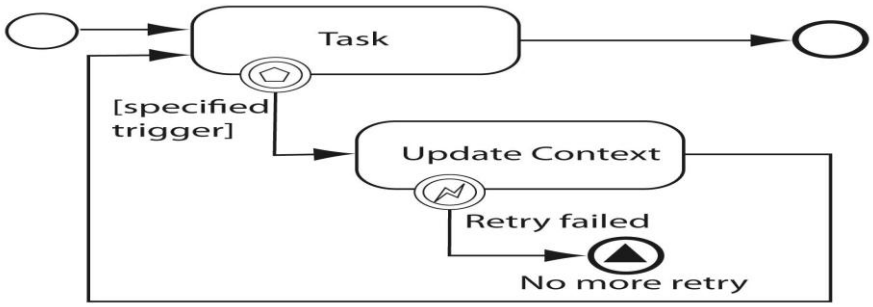

Fig. 8. Reiterate pattern in BPMN

Pattern's Applicability: Exception-Compelled Remake pattern is a generalization of Reiterate. It is suitable in almost the same scenarios as Reiterate, other than that it relaxes the requirement that no time expires between the occurrence and the detection of the issue. In the Reiterate pattern, a trouble with the original work is detected *immediately* after the occurrence of a trouble, and this discovery, in turn, causes the repeated work to also be done *immediately*. Exception-Compelled Remake permits for the detection of the trouble and the repeated work performed to repair the problem to take place at any time, maybe even after a significant amount of time has elapsed since the issue was created.

Pattern's Structure: Exception-Compelled Remake is a generalization of Rework, where the rework need not be triggered by an exception, but simply requires the reexecution of a step executed at some point in the past. Cass et al. [33] specify a much more complete definition and description of rework.

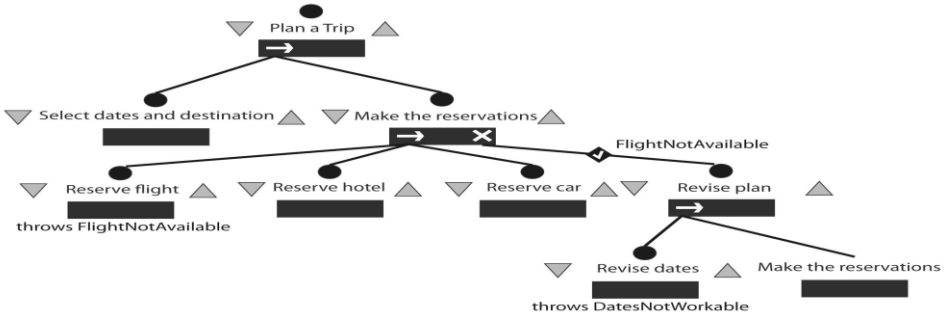

Fig. 9. Applying the Reiterate pattern to replan a trip

Pattern's Participants: The participants in this pattern are the **anomaly discoverer/detector** and the **repairer/fixer**. As in the Reiterate pattern, the repairer can be further disintegrated into a structure that contains a step to update the context before executing a **reiterable/retriable activity**.

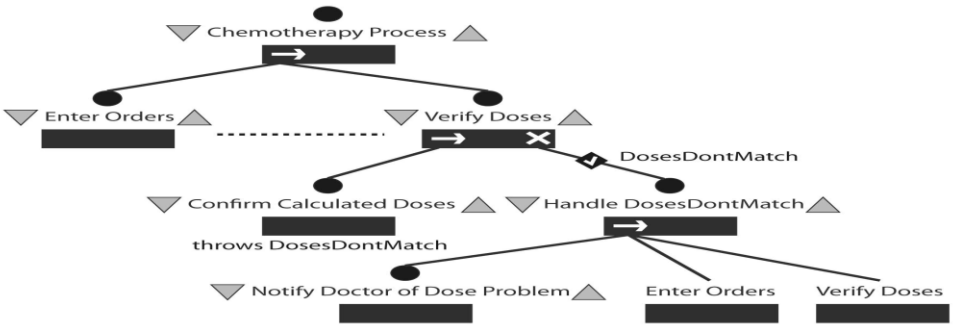

Fig. 10. Applying Exception-Compelled Remake in the medical domain

Pattern's Instances: An example of Exception-Compelled Remake can be found in the process of medical domain. A fragment from a process of chemotherapy preparation is shown in Fig. 10. A doctor has prescribed medication dosages as part of his/her orders for treatment of a patient at some stage earlier in the process. For protection, a nurse next uses patient height and weight data to manually recalculate the doses of these same medications and then attempts to verify that newly calculated doses match the doses ordered by the doctor. If the doses do not match, the nurse needs to notify the doctor of this issue, then the doctor needs to reenter the correct doses by remaking a previously executed medication entering activity, which now is done in a new perspective, namely, one in which the previous faulty performance is now a part of the history of the execution of the process. After the doctor has prescribed the new doses, the nurse needs to retry (also in a new perspective) the activity he/she failed to complete, namely, confirming that the manually calculated doses match the ones just prescribed by the doctor.

Pattern's Deviations: Remake/Rework is frequently followed by a *ripple effect*. Other already executed activities in a process may rely on the conclusions made in or the outputs generated by the problematic activity. In that situation, only reworking the problematic activity is not sufficient. To fully repair the trouble, the already executed activities that are relying on the problematic one should also be revisited.

II.II Nullifying Action

This type of exception handling patterns is one in which an action being pondered must not be permitted for some cause.

II.II.I Name: - **Discard** Pattern

Pattern's Intent: It occasionally becomes evident that an action being pondered should not be permitted. The driving force pondering the action must be informed and allowed to make adjustments or transforms and retry, if so required.

Pattern's Applicability: Discard pattern creates an admission obstruction to a division of a process.

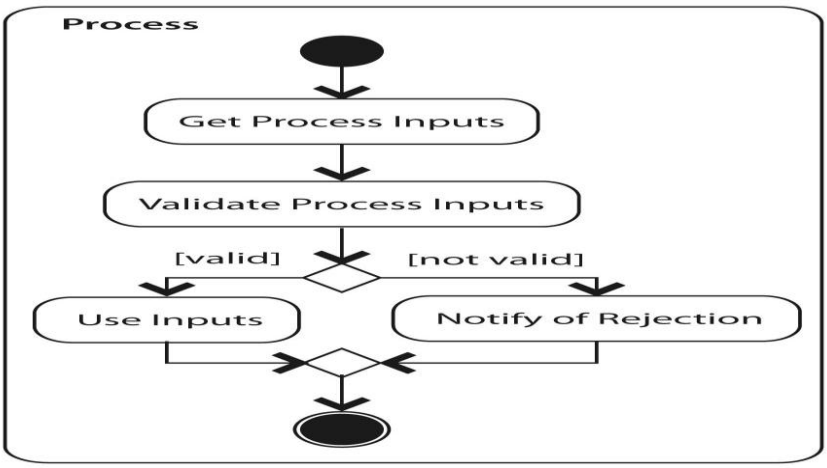

Fig. 11. Discard pattern in UML

Pattern's Structure: There are unusual ways to signify the organization of the Discard pattern using UML notation. One easy approach is given in Fig. 11 using a **Decision Node** symbolized in the stature by a diamond and **Guards** on its output edges. The guards verify the outcome of the *Validate Process Inputs* action, and consequently choose to continue to the next step if the inputs are suitable or to inform the representative of their dismissal if they are not. There is one final state which both the nominal and exceptional flows attain.

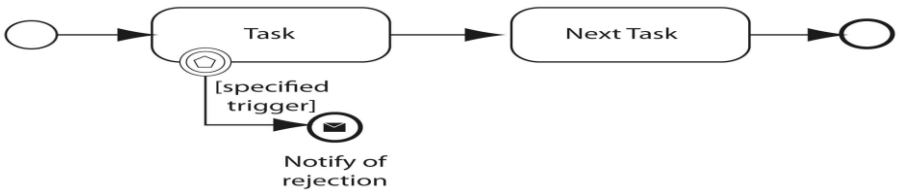

Fig. 12. Discard pattern in BPMN

Fig. 12 shows the Discard pattern in BPMN notation. The **Exception Flow** only includes an **End Event** that throws a **Message** of notification. Compare to the UML structure, we have two final states, one for the normal flow and an additional for the exceptional flow.

Pattern's Participants: The Discard pattern consists of a discarder/rejecter and a validator/confirmor. The **discarde**r is an activity that causes the portion of the process that manages the discarded/rejected input to be canceled. The **validator** decides if the input should be acknowledged or not.

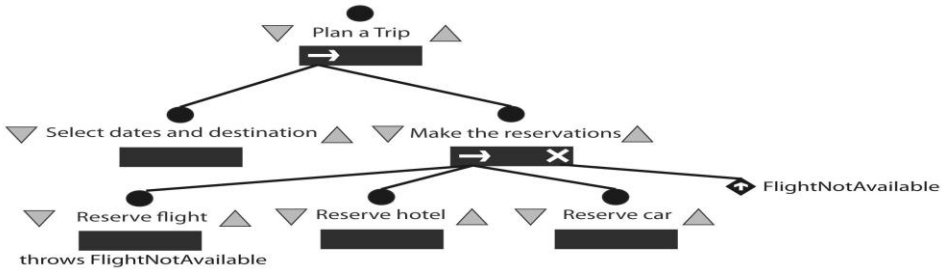

Fig. 13. Applying the Discard pattern to cancel a trip

Pattern's Instances: Lot of processes includes tests of different clauses that must be fulfilled in order for a part or the whole of the process is to continue. Hence, in Fig.13, we demonstrate a Little-JIL process that terminates a journey if there is no flight available for the journey. This takes place as the exception handling semantics in this example is to rethrow the exception, represented by the upward pointing arrow on the exception handler. Note that this process would give the impression to have the same objective as the process portrayed in Fig. 9, but the processes vary in the measures taken when there is no flight available. In the earlier example, we reworked plans and retried. In this case, we merely back out. Still a dissimilar way to dealing with this state of affair would be to permit the user to make the selection, leading to a process that makes use of both patterns. In this situation, discarding the input outcomes in the complete process being aborted.

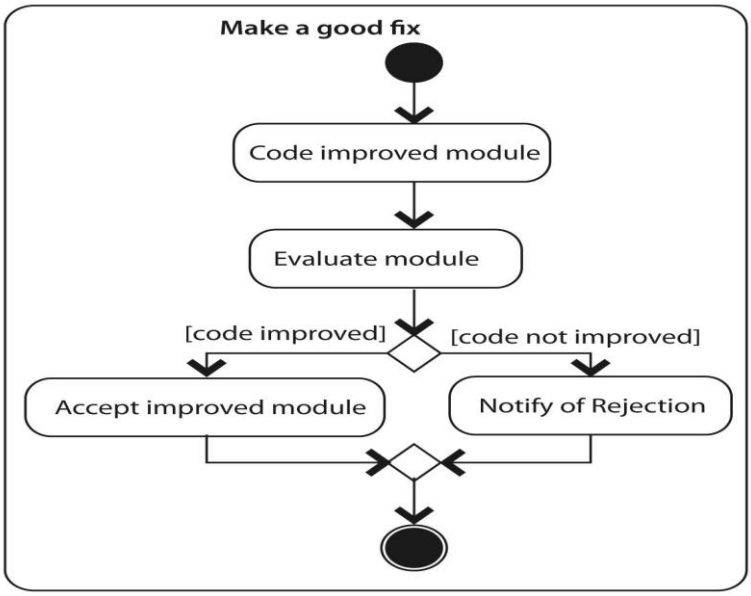

Fig. 14. Applying the Discard pattern to reject a code fix that does not work

Fig. 14 offers one other illustration of the use of this pattern, now drawn for software development process in UML. This instance demonstrates a process; *Make a good repair/fix*, for repairing/fixing a module. The process initiates by coding the superior module. Then, there is a test to observe if the module has actually been improved, by testing, formal and/or informal analysis. If we settle on that the alleged repair is not truly an upgrading, we discard the repair as a substitute of accommodating it in the subsequent step. At this point, an exception handler higher in the process (not revealed) would catch the propagated exception and permit software development process to carry on, but without the repair that was discarded.

Pattern's Deviations: The Discard pattern can be used either to end only part of the process or end the entire process. To end part of the process, an exception handler superior in the call hierarchy will require for managing the exception to let the process to continue.

II.II.II Name: - **Indemnify** Pattern

Pattern's Intent: When revoking an action, it is commonly essential to undo effort that has previously been finished. Indemnify pattern addresses the requirement to decide what effort must be undone and to then carry out the adjusting action(s) required in order to undo it.

Pattern's Applicability: Indemnify pattern is mainly valuable in circumstances in which it is not likely to identify at the beginning that a task will be successful, or the outcomes produced by the task will confirm ultimately to be suitable. In view of this, the process must include means for undoing the fraction(s) of the job that did

complete and/or replacing the outputs that proved to be improper. In a number of situations, the status of the process past compensation may emerge to be the matching as if the unsuccessful actions not at all happened. Frequently, nevertheless, there will be a testimony that the activity happened but the compensating activity cancels the impact of the original action, as when a debit/credit card credit repays for a debit/credit card fee for privileged customers of the bank.

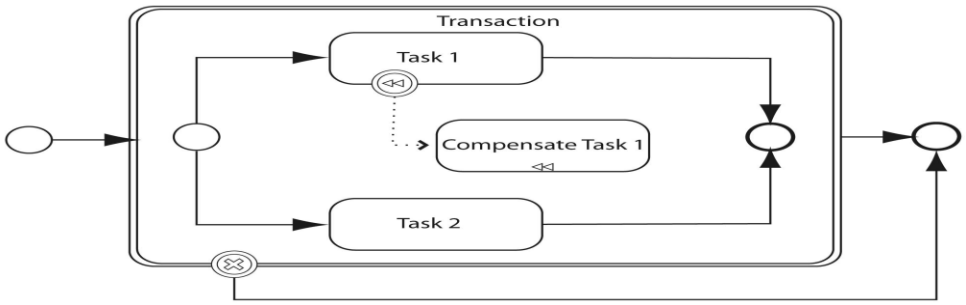

Fig. 15. Indemnify pattern in BPMN

Pattern's Structure: Fig. 15 shows the composition of the Indemnify pattern in BPMN, using BPMN's construct. **Indemnification/Compensation** rolls back some of the effects of a Transaction. A **Transaction** is based on a formal business association and undisputed agreement among two or more participants. It is represented as a **Sub-Process** with a double-line boundary. A **Cancellation/Termination Event** attached to this boundary will disrupt the **Transaction** and make the process continue along the **Exception Flow** specified. However, previous to opening the **Exception Flow**, any accomplished activities within the **Transaction** that have Indemnification/Compensation activities are undone by clearly defined rollback actions. This is designed by attaching a **Compensation/Indemnification Event** to the boundary of that activity, and linking it to a unique type of activity, a **Compensation/Indemnification** activity (symbolized using a rewind symbol). Figure shows, two normative jobs are defined in a parallel flow that permits their implementation in any order (parallel construct in Little-JIL). A **Compensation/Indemnification** activity is defined only for *Task/Job 1*. When a cancellation/termination of the **Transaction** happens after *Task/Job 1* has accomplished, the **Compensation/Indemnification** action is performed, and then, the **Exception Flow** defined for the **Transaction** is turned on. The BPMN illustration adequately shows how the Indemnify/Compensate blueprint is characteristically used within the larger perspective of a parent/creator process/procedure.

Neither UML nor Little-JIL has a Compensation/Indemnification design. This makes the Compensation/Indemnify model more complicated to articulate in the widespread case. For the reason that it at present becomes essential to define tests in the process to decide which steps are finish in order to make out which compensation/indemnification actions are essential when an exception raises. Fig. 16

illustrates how this would be signified in Little-JIL. Both *Step 1* and *2* are done side by side. If *Step 2* not succeeds but *Step 1* finishes, an exception handler is used to compensate/indemnify for the impacts of *Step 1*. Observe that the process wishes to verify clearly if *Step 1* is finish in its exception handler.

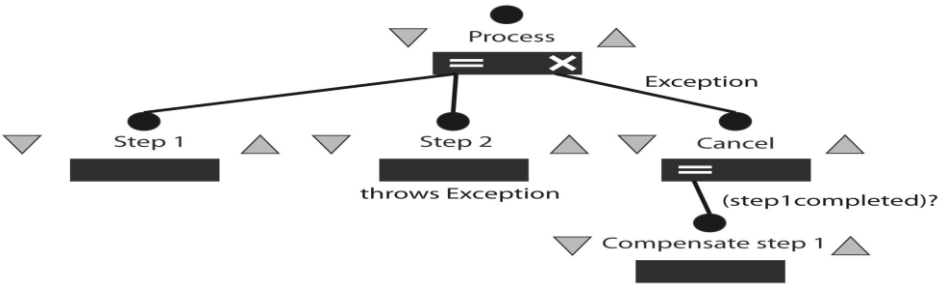

Fig. 16. Indemnify pattern in Little-JIL

Pattern's Participants: The members in this blueprint are the Performer, the Abolisher, and the Indemnifier. The **Performer** carries out few jobs that the **Abolisher** later wants to undo. The undo is performed by the **Indemnifier**, which comprehends the work that was accomplished and how to undo it.

Pattern's Instances: Fig. 17 illustrates one more deviation of the Little-JIL procedure of scheduling a journey. In this case, the bookings can be achieved in any sequence. If we not succeed to get an air travel, we terminate the journey. This will involve abolishing cab and inn bookings if those actions have previously completed. The vital distinction among this case and that in Fig. 13 is that, in the former case, the consumer got the airplane booking first and thus had nothing to revoke if there was no air travel available. In this case, the consumer can do the three bookings in any sequence, and conceivably simultaneously, therefore we have to find out what was done if we need to cancel the trip.

Fig. 18 demonstrates a BPMN case method of management for order of client. It is represented using a **Transaction** *Manage Client/Customer Order* that comprises actions of charging the client and dispatching a proof of payment, and selecting the planned product from the warehouse and delivering it to the client. Three of these actions are linked with **Indemnification Activities**. When abolition (for example, order abolition) happens, any of the three actions that has accomplished is recouped as stated in backward order of the regular course. Once the **Transaction** is completely rolled back, the control flow is diverted according to the **Exception Flow** described and defined on the parent stage.

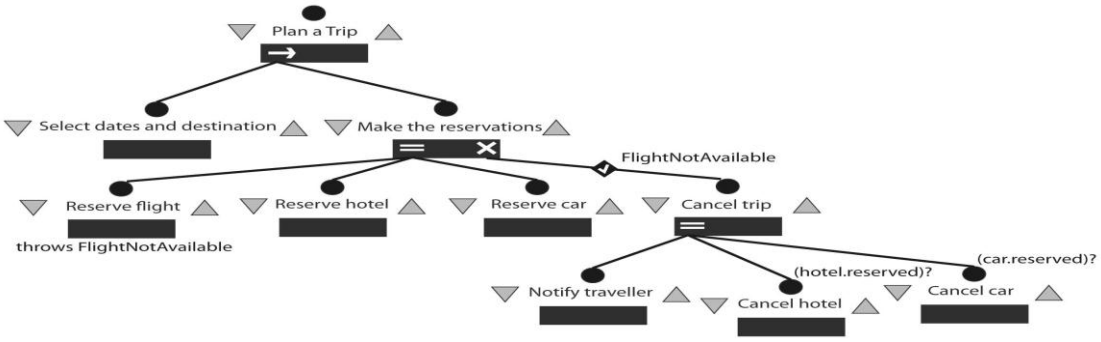

Fig. 17. Applying the Indemnify pattern to cancel a trip

Pattern's Deviations: Indemnification can be joined with supplementary patterns. In specific, any time that an activity not passes with an exception, it may be obligatory to unwrap several effort that has been accomplished. As a result, reparation could form component of the exception handling used in any of the previous patterns.

A different variation is that it is not every time essential to encompass in the process the checks to decide what work requires to be recouped, even in the absence of an indemnification build similar to BPMN has. This is the case if the place of the exception handler is enough to decide what job is accomplish, as would be the case if the indemnification was in the perspective of chronological jobs rather than parallel jobs.

II.III Seeking Other Options

One general class of exception management patterns defines how to deal with opinions about which of numerous alternative courses of action to follow. In few cases, such verdicts are based upon situations that can be set directly in the process, fundamentally with an if-statement to make the selection. In other situations, at prior, it may be hard to catch all situations for which each course of action is best suited. For specific cases, it is time and again most helpful to just present the process actor with options to try. If the option that is attempted fails, another option is to be attempted in its place, using exception management to move on to not attempted options. In this group, we have recognized two different exception management patterns: sequenced options pattern and disordered options pattern.

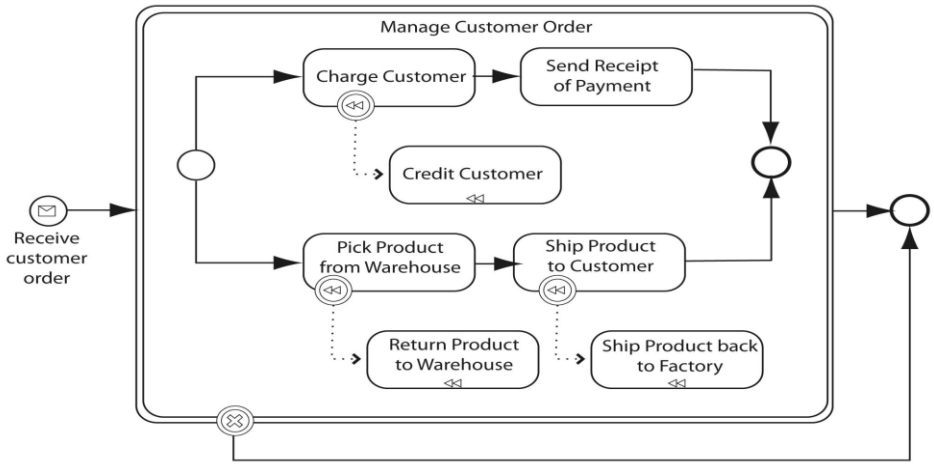

Fig. 18. Applying the Indemnify pattern to cancel an order

II.III.I Name: - **Sequenced Options** Pattern

Pattern's Intent: There are numerous ways to complete a job and there is a preset order in which the options should be attempted. Preparation must be made for the likelihood that no options will be successful.

Pattern's Applicability: Sequenced options pattern is appropriate when there is a chosen order amongst the options that should be attempted in order to carry out a job.

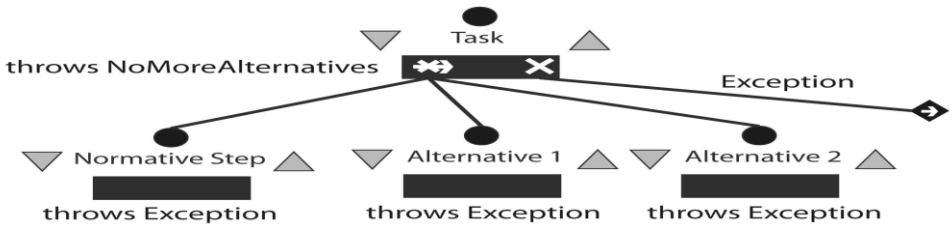

Fig. 19. Sequenced Options pattern in Little-JIL

Pattern's Structure: Fig. 19 shows the composition of the Sequenced Options pattern in the Little-JIL notations. Practices are symbolized in Little-JIL as ranked disintegrations into **steps**. At this point, we see the **step** named *Job/Task* through three substeps, each one defining one mean to accomplish the job. The symbol at the left side of the black step bar of *Job/Task* specifies that it is a **Try/Attempt** step. The interpretation of the semantics of the Little-JIL **Try/Attempt** step go with the meaning of this blueprint fairly closely, as the **Try/Attempt** step meanings states that the step's descendants correspond to options that are to be attempted in sequence from left to right. If an option accomplishes something, the ancestor step is concluded and

no additional options are proposed. If carrying out of an option raises an exception, the exception is managed by the handler affixed to the **Try/Attempt** step by the rightmost edge. The symbol linked with the exception handler signifies that the **Try/Attempt** step should carry on with the subsequent option. This goes on up to the time that one of the option substeps turns out successfully. If not any of the substeps accomplishes something, an unusual exception, called *NoFurtherOptions/NoMoreAlternatives*, is thrown. *NoFurtherOptions* exception must be managed by a predecessor of the Try step. Representing that all options have not succeeded is element of the blueprint, but the managing of that exception must take place in the perspective in which the blueprint is used rather than as component of the blueprint.

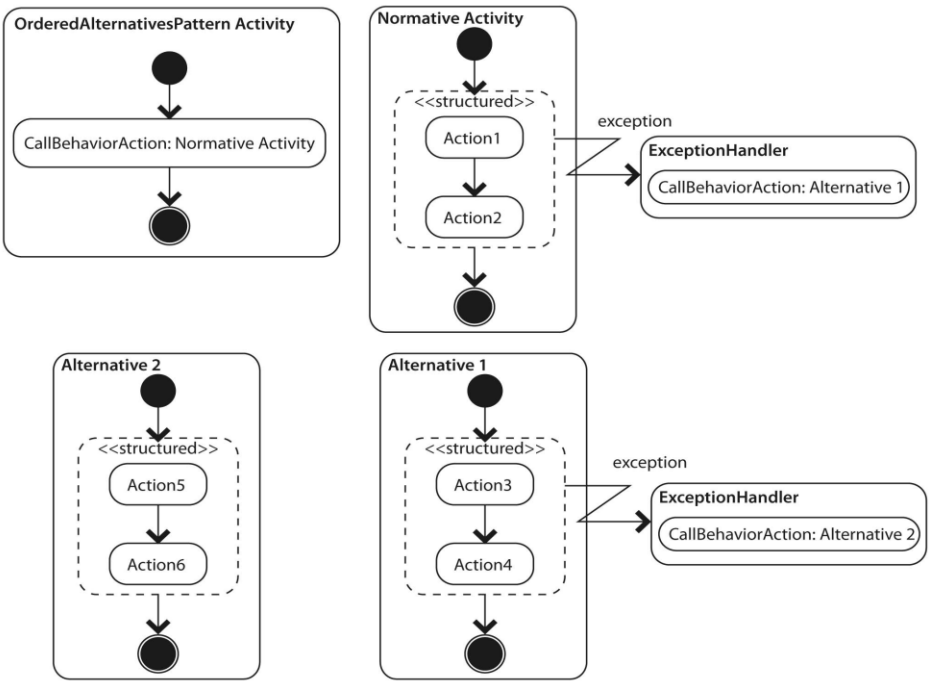

Fig. 20. Sequenced Options pattern in UML

Neither UML nor BPMN have a build resembling to the **Try/Attempt** step in Little-JIL. The outcome is that this model is conveyed by attaching jointly the options with exception handlers as revealed in the Activity Diagram drawn using UML in Fig. 20. A core action, in the build called *SequencedOptionsPattern/OrderedAlternativesPattern*, is used as a perspective to call the carrying out of the *Normative/Regular* action. This call is guaranteed by a **CallConductActivity**/**CallBehaviorAction**, which, in UML, serves as the ways to call an activity from inside a different one. *Regular* actions contains a set of operations to be executed by the representative and which are confined within a **DisciplinedPursuitNode**/**StructuredActivityNode**, a controlled segment of the

action that is not used in common with any supplementary segment and which can be sheltered by an **Exception Handler**. Any exception originated by the carrying out of any deed within the disciplined node and having a category equivalent to the exception types managed by the supervisor will be trapped by the exception handler. If an exception takes place, the control flow is ended within the disciplined node and the flow is transmitted to the **Exception Handler**. The exception handler will then call the first option, which is characterized in Fig. 20 by the *Option 1* action using another **CallConductActivity**/**CallBehaviorAction**. The actions in the *Option 1* activity may also be sheltered by an **Exception Handler**, which may call a second option if *Option 1* not succeeds as well. In case of no further options, *Option 2* will terminate, causing the *NoFurtherOptions* exception to disseminate to the call action that summoned *Option 2*, then to the ancestor activity possessing the call action, and so on.

Pattern's Participants: This blueprint has three types of members: the list, the options, and the pursuer. The **list** is the segment of the process that arranges the options into a sequence. The **options** are the diverse means in which the preferred job can be executed. While the illustrations confirm three options, there is no limit to the number of options that could be used in this model. Each option, excluding perhaps the very last, must have the capability to throw an exception that originates thoughtfulness of the subsequent option. The **pursuer** is the exception supervisor that specifies that the course of action should go on with the next option.

Pattern's Instances: Fig. 21 illustrates the utilization of the Sequenced Options pattern in a Little-JIL practice to arrange journey to be present at a symposium. This blueprint can be witnessed in the *Book inn/Reserve Hotel* step. At this juncture, the progression calls for first trying to get a booking at the symposium inn ahead of taking into consideration other inns. If the symposium inn is occupied, the *InnFull/HotelFull* exception is thrown. This is managed by causing the *Book other inn/hotel* step to be tried after that.

The implementation of the *Inn/Hotel Booking/Reservation* process in UML is shown in Fig. 22. The chief action calls the *Book/Reserve Symposium/Conference Inn/Hotel* action by means of a **CallConductActivity/CallBehaviorAction**. The *Book/Reserve Symposium/Conference Inn/Hotel* action holds an order of activities described within a **Disciplined Pursuit Node/Structured Activity Node** which is sheltered by an **Exception handler**. The structure of the Exception handler comprises of a call to the *Book/Reserve Other Inn/Hotel* activity and is prompted if an accomplishment within the sheltered node not succeeds.

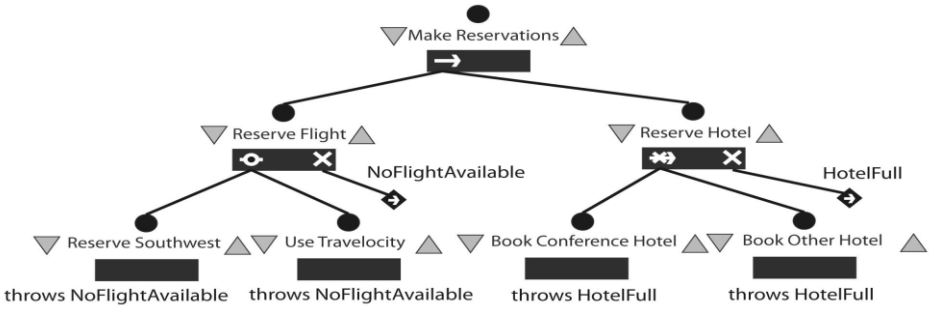

Fig. 21. Applying the Sequenced and Disordered Options patterns for a journey

Pattern's Deviations: One deviation of this model uses Boolean situations after an option is attempted, rather than anticipating the option to throw an exception. If the situation evaluates to correct, it conveys that the option has succeeded. If the situation evaluates to incorrect, it conveys that the option failed and the course of action should continue to the subsequent option. The compromise at this point is basically comparable as we observe in procedural programming approach when determining whether a function must return a status value to prove if it has accomplished something successfully or it should throw an exception, if failed.

If the states of affairs under which an option will do well are identified in advance, the options are better signified with a build similar to an if-else build in a conventional programming language. This permits the orders to be particularized at the same time as avoiding the call for exception handling. It is the Exclusive Choice pattern offered as a control flow pattern by van der Aalst et al. [24].

II.III.II Name: - **Disordered Options** Pattern

Pattern's Intent: There possibly will be several ways of finishing a task, however there are incidents when a predetermined order in which options are to be attempted is either not well-known or not required. If an exception takes place while attempting one approach, an option other than all of those that have been attempted previously is to be attempted in its place. This is to carry on until an option succeeds or until all options have been attempted and have failed. In this later case, the letdown of all options is indicated as an exception to be managed by the process perspective in which this pattern is implanted.

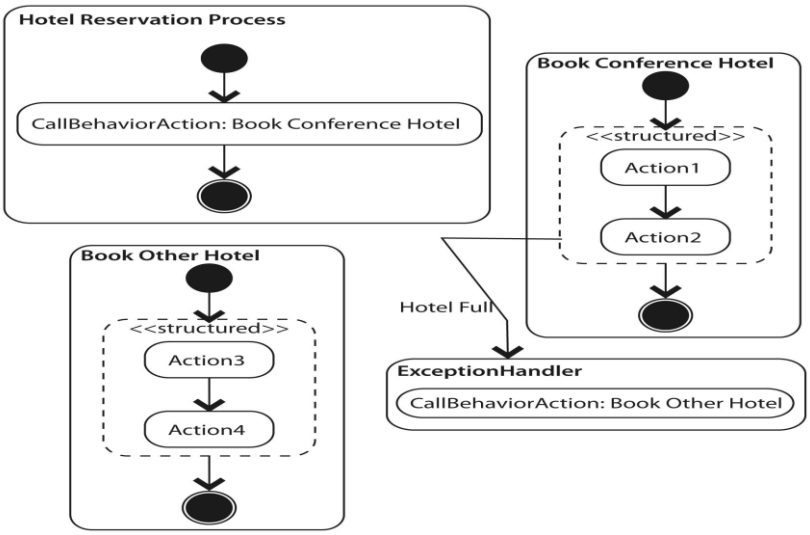

Fig. 22. Applying the Sequenced Options pattern to select a hotel

Pattern's Applicability: This model applies when there are numerous ways to complete a job and it is not recognized a priori which approach is the most suitable. Here, the choice of the order in which the options are attempted is postponed until runtime. If a tried option fails, there is a different effort to finish the work by selecting another option. There possibly will be numerous aspects that control the sequence in which the options are attempted. Such as, the state of the artifacts or the state of the process being controlled by the process may perhaps control the sequence.

In addition, a number of options can need dissimilar resources than others, so resource accessibility and availability may control the sequence in which the options are well thought-out. The information of the actors/performers contributing in the process may also control the sequence in which options are attempted. Specifically, a person actor might utilize knowledge concerning the results of trying earlier attempted options to determine which option is to be attempted subsequently. Note that the blueprint would have the same organization, autonomous of the aspects that control the eventual sequence that is selected, as in this case, the aspects controlling the sequence are dynamic while the blueprint captures only static information. In this fashion, the Sequenced Options pattern can signify either internal or external nondeterministic choice.

Pattern's Structure: The structure of the Disordered Options pattern is shown in Fig. 23 using Little-JIL notations. This pattern is identical like the preceding one apart from that a **Choice/Option step** is used rather than a **Try/Attempt step**. This is showed by the symbol at the left side of the black step bar for *Task/Job*. The semantics/meanings of the Little-JIL **Choice/Option** step go with the description of this blueprint moderately closely as the **Choice/Option** step semantics specify that the step's offspring stand for the options that are to be attempted, without representing any sequence. The semantics/meanings of the **Choice/Option** step indicate that only

one option is to be attempted. If the selected option is victorious, the job is accomplished. If the option is not victorious, then an exception is thrown, resulting in the options that have not yet been tried to be depicted to the manager. As with SequencedOptions, if all options are unsuccessful, the *NoFurtherOptions/NoMoreAlternatives* exception is thrown and should be managed in the perspective in which the blueprint is utilized. In the blueprint, there are three options to select from, however, in common; there can be random number of options.

Here also, neither BPMN nor UML have a control build comparable to the Little-JIL choice/option step. In these representations, Disordered Options are symbolized using a provisional build to decide which option is chosen by the client. If the chosen option not get pass, command loops back to permit the user to choose another time. Fig. 24 illustrates the depiction of the Disordered Options pattern in UML. Here, the *PonderClientOption/ConsiderUserChoice* action permits the client to make a choice. The **Conditional/Provisional Node/Point** contains a **check/test** and **structure/body** for every option. If the chosen option passes the check, the Disordered Options are accomplish. If fails the check, an exception is thrown. The exception handler upgrades the list that the client can select from. If further options stay, control flows back to the deed that shows the updated list to the client.

Pattern's Participants: Similar to Sequenced Options, this blueprint has three kinds of members: the list, the options, and the pursuer. The **list** shows the options, however, here, the sequence of the options has no semantics/meanings. The **options** are the numerous ways in which the job can be performed. Each option must throw an exception that can then be managed to permit the further options to be tried. The **pursuer** is the exception controller that causes the further options to be reviewed.

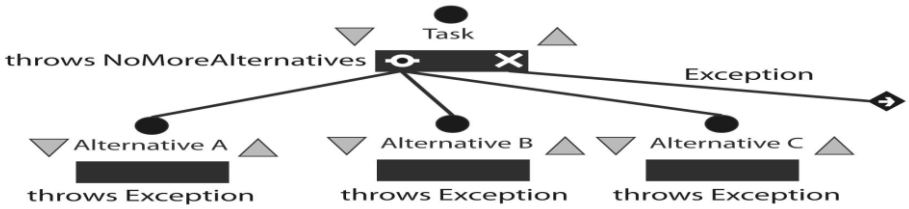

Fig. 23. Disordered Options pattern in Little-JIL

Pattern's Instances: Fig. 21 further illustrates the moves included in booking of air-travel for a journey. At this point, the client can select either to use Indigo/Southwest or to use Jet/Travelocity to book air travel on other airlines (since Jet/Travelocity does not list Indigo/Southwest schedule for flights). If there is no desired flight exists via the first amenity selected, an exception is thrown. The exception controller carries on with the *Book air-travel/Reserve flight* step by allowing the client attempt the further option.

To opt for a shipper disordered option pattern being used as depicted in Fig. 25. An exception will be thrown, if a shipper cannot satisfy the delivery conditions. The exception is managed by permitting the client to attempt the new shipper.

Pattern's Deviations: As the blueprint is represented here, whenever an option fails, only the options that have not yet been attempted are permitted. In a different distinction, all options are permitted every time. There are rewards and drawbacks in both variations. It could be that the options themselves have a bunch of substructure to them. Therefore, it may likely be that the same option could be executed numerous times, with dissimilar outcomes each time. In that scenario, it would be favored to permit all the options every time. By contrast, if each option at all times generates the identical outcome no matter how frequently it is attempted, it is significant to eliminate options from deliberation as they are tried to stay away from a never-ending loop.

III. CONCLUSIONS

We have discovered the blueprints for managing exceptions, which were depicted, to be helpful in raising the reflection level of process models. They give a method, for moving toward exceptional case, by giving a system of inquiries. Would we be able to settle the issue quickly? Is there another option the procedure should provide? Would it be advisable for us to dismiss this information completely?

As there are numerous utilizations of classes that do not play parts in standard object oriented designs, models and patterns, we expect that there are requirements for special exceptional case taking care of in forms that cannot be met by any of the examples and designs we characterize here in our research. Along these lines, in our research, we do not give thought to all lawful methods for joining approaches for exception handling. More willingly, we have concentrated on blends that we have experienced over and over in our work in characterizing forms and designs in such differing spaces as software design and development, commerce and business, transaction/negotiation and medical and health related aspects. While we trust that the decent variety of these areas affirms our claim that the examples of patterns proposed by us are broadly useful in nature, we unquestionably don't trust that this list of blueprints is finished and we expect that it will develop and grow further in near future.

While a few examples and blueprints are less demanding to express in a few modeling language and notation than others, we additionally trust that the examples and patterns are autonomous/independent of modeling language and notation used. Then again, the nearness and presence of specific builds/constructs using a modeling language and notation influences the manners by which one models processes. Indeed, our involvement with analyzing genuine procedures demonstrates that numerous procedures do exclude special exceptional case in their depictions and descriptions. This might be partially because of the noncritical idea of the procedures/processes, yet it might likewise be in any event somewhat because of the nonappearance of

builds/constructs that are helpful for expressing and dealing effectively with unhappy path in those modeling language and notation. We trust that this research on effective management of exception through patterns will urge process modelers, designers and developers to incorporate and deal with exceptional case more efficiently. We believe our contribution will motivate process modeling language designers, to give a (re)thought to builds/constructs, to ease and smooth the way for dealing with exception more cautiously.

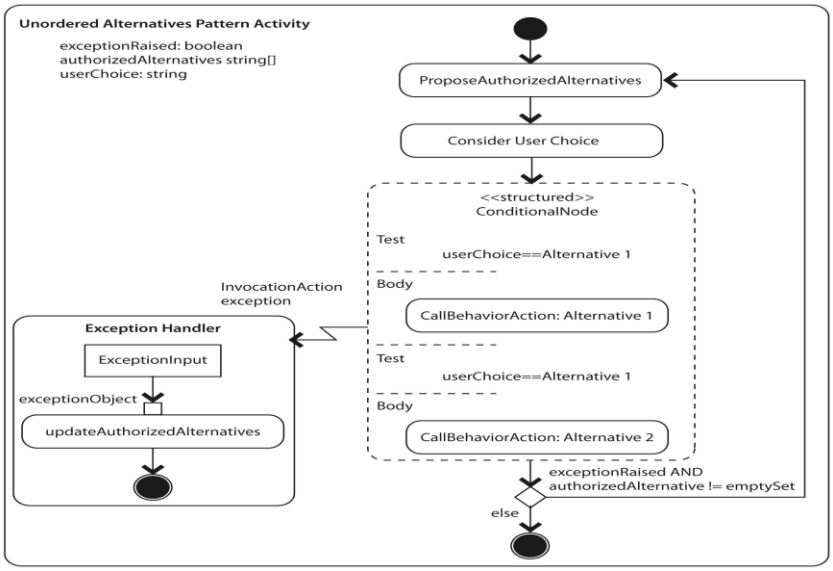

Fig. 24. Disordered Options pattern in UML

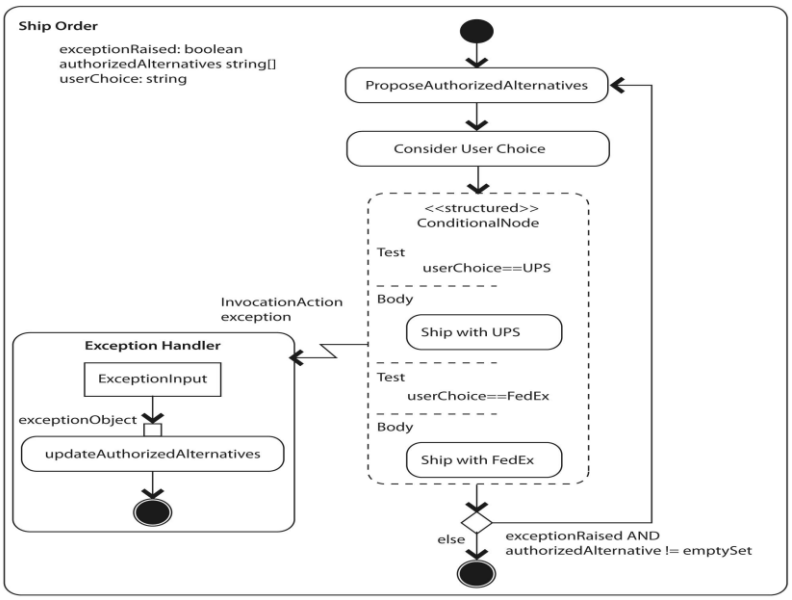

Fig. 25. Applying the Disordered Options pattern to select a shipper

REFERENCES

[1] M.S. Raunak, B. Chen, A. Elssamadisy, L.A. Clarke, and L.J.Osterweil, "Definition and Analysis of Election Processes," Proc. Software Process Workshop (SPW '06) and 2006 Process Simulation Workshop, pp. 178-185, 2006.

[2] L.J. Osterweil, C.M. Schweik, N.K. Sondheimer, and C.W.Thomas, "Analyzing Processes for E-Government Development: The Emergence of Process Modeling Languages," J. E-Govt., vol. 1, no. 4, pp. 63-89, 2004.

[3] B.I. Simidchieva, M.S. Marzilli, L.A. Clarke, and L.J. Osterweil, "Specifying and Verifying Requirements for Election Processes," Proc. 2008 Int'l Conf. Digital Govt. Research, pp. 63-72, 2008.

[4] L. Clarke, A. Gaitenby, D. Gyllstrom, E. Katsh, M. Marzilli, L.J.Osterweil, N.K. Sondheimer, L. Wing, A. Wise, and D. Rainey, "A Process-Driven Tool to Support Online Dispute Resolution," Proc. 2006 Int'l Conf. Digital Govt. Research, pp. 356-357, 2006.

[5] S.C. Christov, G.S. Avrunin, B. Chen, L.A. Clarke, L.J. Osterweil, D. Brown, L. Cassells, and W. Mertens, "Rigorously Defining and Analyzing Medical Processes: An Experience Report," Proc. Models in Software Eng.: Workshops and Symp. at MoDELS '07, 2007.

[6] A. ten Teije, M. Marcos, M. Balser, J. van Croonenborg, C. Duelli, F. van Harmelen, P. Lucas, S. Miksch, W. Reif, K. Rosenbrand, and A. Seyfang, "Improving Medical Protocols by Formal Methods," Artificial Intelligence in Medicine, vol. 36, no. 3, pp. 193-209, 2006.

[7] M.S. Raunak, L.J. Osterweil, A. Wise, L.A. Clarke, and P.L. Henneman, "Simulating Patient Flow through an Emergency Department Using Process-Driven Discrete Event Simulation," Proc. Workshop Software Eng. in Health Care, 2009.

[8] W.M.P. van der Aalst, "Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management," Lectures on Concurrency and Petri Nets, J. Desel, W. Reisig, and G. Rozenberg, eds., pp. 1-65, Springer-Verlag, 2004.

[9] A.-W. Scheer, ARIS—Business Process Modeling, third ed. Springer-Verlag, 2000.

[10] H.A. Reijers, S. Limam, and W.M.P. van der Aalst, "Product-Based Workflow Design," J. Management Information Systems, vol. 20, no. 1, pp. 229-262, 2003.

[11] D. Mu¨ ller, M. Reichert, and J. Herbst, "Data-Driven Modeling and Coordination of Large Process Structures," Proc. Move to Meaningful Internet Systems '07: Int'l Conf. Cooperative Information Systems, Int'l Conf. Distributed Objects and Applications, Conf. Ontologies, DataBases, and Applications of Semantics, Int'l Conf. Grid Computing, High Performance and Distributed Applications, and Int'l Symp. Information Security, 2007.

[12] A.G. Cass and L.J. Osterweil, "Process Support to Help Novices Design Software Faster Better," Proc. 20th IEEE/ACM Int'l Conf. Automated Software Eng., pp. 295-299, 2005.

[13] M. Li, B. Boehm, and L.J. Osterweil, Unifying the Software Process Spectrum.

Springer-Verlag, New York, 2006.
[14] Modelplex, IST European Project Contract IST-3408, http://www.modelplex-ist.org, 2010.
[15] M. Adams, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst, "Dynamic and Extensible Exception Handling for Workflows: A Service-Oriented Implementation," BPM Center Report BPM-07-03, BPMCenter.org, 2007.
[16] E.H. Henneman, R.L. Cobleigh, K. Frederick, E. Katz-Bassett, G.A. Avrunin, L.A. Clarke, L.J. Osterweil, C. Andrzejewski, K.Merrigan, and P.L. Henneman, "Increasing Patient Safety and Efficiency in Transfusion Therapy Using Formal Process Definitions," Transfusion Medicine Rev., vol. 21, no. 1, pp. 49-57, 2007.
[17] E. Gamma, R. Helm, R. Johnson, and J.M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented software. Addison-Wesley, 1994.
[18] S. Stelting, Robust Java: Exception Handling, Testing and Debugging. Prentice Hall, 2005.
[19] A. Haase, "Java Idioms: Exception Handling," Proc. Seventh European Conf. Pattern Languages of Programs, July 2002.
[20] A. Longshaw and E. Woods, "Patterns for Generation, Handling and Management of Errors," Proc. Ninth European Conf. Pattern Languages of Programs, July 2004.
[21] A. Longshaw and E. Woods, "More Patterns for the Generation, Handling and Management of Errors," Proc. 10th European Conf. Pattern Languages of Programs, July 2005.
[22] J.O. Coplien, "A Development Process Generative Pattern Language," Pattern Languages of Programs, 1994.
[23] S.W. Ambler, Process Patterns: Building Large-Scale Systems Using Object Technology. Cambridge Univ. Press, 1998.
[24] W. van der Aalst, A. ter Hofstede, B. Keipuszewski, and A.P.Barros, "Workflow Patterns," Distributed and Parallel Databases, vol. 14, no. 3, pp. 5-51, July 2003.
[25] W. van der Aalst, A. ter Hofstede, B. Keipuszewski, and A.P. Barros, "Advanced Workflow Patterns," Proc. Seventh Int'l Conf. Cooperative Information Systems, O. Etzion and P. Scheuermann, eds., pp. 18-29, 2000.
[26] N. Russell, A. ter Hofstede, D. Edmond, and W. van der Aalst, "Workflow Data Patterns: Identification, Representation and Tool Support," Proc. 24th Int'l Conf. Conceptual Modeling, L. Delcambre et al., eds., pp. 353-368, 2005.
[27] N. Russell, W. van der Aalst, A. ter Hofstede, and D. Edmond, "Workflow Resource Patterns: Identification, Representation and Tool Support," Proc. 17th Conf. Advanced Information Systems Eng., O. Pastor and J.F. e Cunha, eds., pp. 216-232, 2005.
[28] N. Russell, W. van der Aalst, and A. ter Hofstede, "Exception Handling Patterns in Process-Aware Information Systems," BPM Center Report BPM-06-04, BPMCenter.org,http://www.workflowpatterns.com/documentation/documents/BPM-06-04.pdf, 2006.

[29] L.J. Osterweil, "Software Processes Are Software, Too," Proc.Ninth Int'l Conf. Software Eng., pp. 2-13, 1987.
[30] OMG, Business Process Modeling Notation (BPMN) Version 1.2, http://www.omg.org/spec/BPMN/1.2, 2010.
[31] A. Wise, "Little-JIL 1.5 Language Report," technical report, Dept. of Computer Science, Univ. of Massachusetts, 2006.
[32] OMG, Unified Modeling Language, Superstructure Specification, Version 2.1.1, http://www.omg.org/spec/UML/2.1.1/Superstructure/PDF/, 2010.
[33] A.G. Cass, S.M. Sutton, and L.J. Osterweil, "Formalizing Rework in Software Processes," Proc. Ninth European Workshop Software Process Technology, pp. 16-31, 2003.
[34] B.S. Lerner, S. Christov, L.J. Osterweil, R. Bendraou, U. Kannengiesser, A. Wise, "Exception Handling Patterns for Process Modeling," IEEE Transactions On Software Engineering, pp. 162-183, 2010.