Documenting Patterns with Business Process Models

PEDRO MONTEIRO, Universidade Nova de Lisboa
MIGUEL PESSOA MONTEIRO, Universidade Nova de Lisboa

The last years have seen a rise in the number of tools and methods developed to help developers and domain users communicate. One such method is Business Process Modeling. The Business Process Modeling Notation provides a graphical visualization and modeling tool for end-to-end business process documentation and thus excels in directness and ease of communication between non-expert users and technical people. The goal of this paper is to present the basis for a new process-oriented perspective into pattern design representation. We claim that this modeling technique can be effectively used as a representation format for most patterns, bridging across domains and styles.

Categories and Subject Descriptors: **D.2.9 [SOFTWARE ENGINEERING]**: management—*Software process models*; **K.6.3 [MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS]** Software Management—*Software process*; **D.3.3 [PROGRAMMING LANGUAGES]:** Language Constructs and Features—Patterns

General Terms: Business Process Models

Additional Key Words and Phrases: BPMN, UML, Design patterns

ACM Reference Format:

Monteiro, P. and Monteiro, M. P. 2012. Documenting Patterns with Business Process Models. In Proceedings of the 19th ACM Conference on Pattern Languages of Programs (PLoP 2012) (SPLASH, Tucson, Arizona, USA, October 19-21 2012).

1. INTRODUCTION

Over the past 30 years, the field of software development evolved at an accelerated rate. New and progressively more advanced software development techniques arise on a daily basis to the extent that no single person can hope to grasp the sheer volume of available knowledge. Nowadays, the number of available software development, design and maintenance techniques is so vast that programmers are ever more specialized in narrow, specific areas of the software domain. Simply deciding on the best methodology and development strategy with which to implement software is at best difficult. Programmers often opt to use the solution they know best, even if it may turn out not to be optimal. Thus, in order to reuse good software development practices and techniques, it is essential that expert programmers identify and document what are considered to be best practices in their specific domain. In this context, Software Patterns represent well-documented template solutions that can be reused whenever a certain specific problem arises in a well-known and clearly defined context [Buschmann et al. 2007].

Patterns capture formal solutions to specific problems, while maintaining a high level of abstraction. Thus, they help support a high-level form of reuse, which is mostly independent from methodology, language, paradigm and architecture [Buschmann et al. 1996]. Using patterns, both expert and non-expert programmers can employ and build upon proven concepts and solutions to some of the most common problems in a specific domain, avoiding recurrent pitfalls and benefiting from carefully thought design strategies. Patterns must be concrete enough so as to represent valid solutions, yet their context should be flexible enough to allow their application to a variety of problems.

As the pattern community continues to grow, software developers are once again confronted with an overwhelming amount of knowledge. Patterns are swiftly becoming what they were created to prevent in the first place [Henninger and Corrêa 2007]. We believe one of the problems is the visual representation of a pattern, or lack thereof.

Patterns are usually delivered in textual format, with a few illustrative figures or UML models. In this case, the aim of a UML model (more frequently than not a class diagram) is to provide the means for a quick implementation of the pattern in a generic OO programming language. This is a generally well

Author's address: Pedro Monteiro, CITI, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal; email: pmfcm@campus.fct.unl.pt; Miguel P. Monteiro, CITI, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal; email: mtpm@fct.unl.pt

accepted practice, however it also provides the non-domain fluent developer with a hard-set solution which is hard to pull away from.

The goal of this paper is to present the basis for a new perspective into pattern design representation.

The last years have seen a rise in the number of tools and methods developed to help developers and domain users communicate. One such method is *Business Process Modeling*. It is our belief that Business Process Modeling, specifically the *Business Process Modeling Notation* [Lohmann et al. 2009], can be used as a representation format for most patterns, bridging across domains and styles.

BPM is an ideal technique to represent patterns because:

- BPM was created specifically to provide a way for Business domain experts to communicate with software engineers in a common language, ergo notation.
- Most patterns are actually algorithms, in the sense that an algorithm is the representation of the process of accomplishing a task. Even design patterns can be considered algorithms insofar as the implementation of a particular design forces a specific usage process. In BPM, Engineers see algorithms while Domain experts see business processes and activities; both of which can be seen as structures of concepts.
- The simplicity and potential of use of BPM is directly linked to its directness. The symbolic language used in BPM is easy enough to be recognized by someone who has had little contact with BPM, while at the same time is expressive enough to represent the most complex processes.
- As BPM is used to model real world processes, it is well adapted to represent even the most complex situations, such as parallelism, concurrency, synchrony and asynchrony, message passing, etc.
- BPM's usage is widespread enough for it to be considered a valid complement to UML; one which brings added value.
- BPM's directedness makes it easy to learn and understand; therefore it would not be a burden for pattern writers to use.
- UML cannot easily support a process-centric way of thinking as BPM does.
- BPM is about agility; therefore it is a good representation option for patterns that are easily adapted to its surrounding context.
- BPM is not without its shortcomings and is not intended to replace current diagraming practices; however we trust that it can help bridge the conceptual gap between the textual description of a pattern and its mental representation, while providing the means to connect patterns from different domains and levels of abstraction.

Other representations such as UML are mainly technical in nature and don't favor learning/usage by non-technical people, i.e. business domain experts. Through this diagrammatic representation business domain experts, which are more familiar with BPM than with other more technical representations (such as class or sequence diagrams) will have the opportunity to grasp the pattern concepts, thus fomenting its widespread use as business artifacts and enhancing pattern understandability.

The remainder of this paper is organized as follows: Section 2 describes traditional pattern representation formats while Section 3 presents Business Process Modeling as an optional notation for patterns; Section 4 presents examples on documenting patterns with BPMN; Section 5 describes related work; Section 6 and 7 concludes the paper and describe future directions for this work. An appendix containing some BPMN definitions is also presented.

2. PATTERN REPRESENTATION

The textual representation of patterns has been the subject of great debate. There is no consensus on the formal structure of pattern description and many authors coin their own format. The main templates of

pattern description relate to the Alexandrian form [Alexander et al. 1977], Gang of Four form [Gamma et al. 1995] and Coplien form [Coplien and Woolf 1997]. In Alexandrian patterns, the general form and style includes pattern name, context, main (problem statement, forces, solution instruction, solution sketch, solution structure and behavior), and consequences. The Gang of Four design patterns present a structure composed of Pattern Name and Classification, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, and Related Patterns. Coplien Patterns are based on Alexandrian form but present patterns in a reduced format, which includes Pattern Name, Problem, Context, Forces, Solution, Rationale, Why does this pattern work? and Resulting Context.

This freedom of letting the pattern author decide on the representation is also used for the visual representation of patterns. Yet there is no clearly defined basic style for the pattern writer to follow. More often than not, the usual chosen representation style is UML class diagrams, mainly due to the popularity of the Gang Of Four book [Gamma, Helm, Johnson and Vlissides 1995].

Freedom of notation and description are some of the characteristics that make patterns accessible to a large audience – this is in no way a bad thing. However, there are times when the differences between authors can inhibit the use of patterns from different catalogues.

3. BUSINESS PROCESS MODELING

There are few Modeling Notations that were successfully absorbed by the business world [Nysetvold and Krogstie 2006]. The most successful are EEML (Extended Enterprise Modeling Language), UML activity-diagrams and BPMN (Business Process Modeling Notation). All these are similar in the features they support and choosing one over the other is not a trivial matter. EEML is probably the most “featuristic” of the three and is able to describe a wider set of business concepts. UML Activity Diagrams have the added advantage of being part of the general UML Meta-Model and thus contributes to the model’s expandability and composability with other diagrammatic views of a given system. BPMN outmatches the others in terms of “user friendliness”, being both easier to understand and use. On the overall, BPMN is reportedly more adapted to the business world, be it by satisfying inter-department communication requirements or procedural user needs. Through the Semiotic Quality Framework, Wahl and Sindre [Sindre 2006] explored the features of these three notations using a set of common criteria. In general BPMN achieved the highest score in all criteria, except domain appropriateness. The authors also show that BPMN excels in comprehensibility appropriateness since it possesses the highest number of modeling artifacts of the three notations. BPMN was compared to UML activity diagrams by White [White 2004] by comparing the support both languages offered in modeling Workflow Patterns [Van Der Aalst et al. 2003]. The author assessed that all patterns could be successfully mapped or modeled through BPMN artifacts in a more intuitive manner than UML activity diagrams. Rosemann et al. [Recker et al. 2006] analyzed the representation capabilities of BPMN on different approaches, concluding that BPMN can be regarded as a mature modeling notation. Another advantage of BPMN is that it has a direct (if incomplete) mapping to the process execution language BPEL[White 2005], which makes it a valuable technical resource.

Other workflow models, such as Petri nets [Peterson 1981] have also been successfully used to model processes, providing strong semantics which are especially tailored for process analysis of concurrent models. The body of work behind Petri nets is indeed massive and, being a very specific notation with only three types of artifacts (Places, Transitions and arcs) it is used to validate other models including BPMN [Lohmann, Verbeek and Dijkman 2009] and other Business process notations. However, Petri nets’ formalization and notation is very distant from the common reader’s conceptual idea of a process and this translates into weak readability and learnability. This complexity makes Petri nets a less popular notation.

The Business Process Modeling Notation was designed with the intent of providing graphical visualization and modeling tool targeting end-to-end business process documentation [Aagesen and Krogstie 2010; White 2004]. It is defined by a flowchart-inspired Business Process Diagram (BPD), where activities are grouped into networks with control-flows indicating relationships. BPMN provides a simple and intuitive way for business experts to communicate with technical people while at the same time maintaining a level of formality that helps connect the design and implementation details of a system, by specifying its processes through visual representations of algorithms.

BPMN is considered a mature notation mainly because of the success of its predecessors; its design has strong influences from Petri nets, UML Activity Diagram, UML EDOC Business Process, IDEF, ebXML BPSS, Activity-Decision Flow Diagrams, RosettaNet, LOVeM, and Event-driven Process Chains.

BPMN also has some shortcomings. The most important missing feature is probably the lack of a formally defined way to represent the state of activities. There is no way to decide if an event is triggered by a condition or set of conditions. Also, within a model, there is no clear separation of concerns and the system being modeled might be seen as another activity within a much larger model. This introduces a measure of confusion when the model grows too large and/or there are separate models for certain processes within a larger process.

For the above reasons, and mainly due to BPMN's ease of use, we have come to believe that it would be the most likely candidate for a representational notation for Patterns and Pattern Languages, so we describe it greater detail in the next sections. A short introduction to BPMN artifacts is included in the appendix.

BPMN has been used in a variety of domains, as is the case of SOA with the BPEL mapping for web services[White 2005]or the Outsystems agile development framework for web applications , for instance. More recently, some work has been developed to show that BPMN can be used to model the runtime of complex algorithms. Figure 1 presents a proof of concept example showing the Transactional Locking II algorithm [Dice et al. 2006] in BPMN.

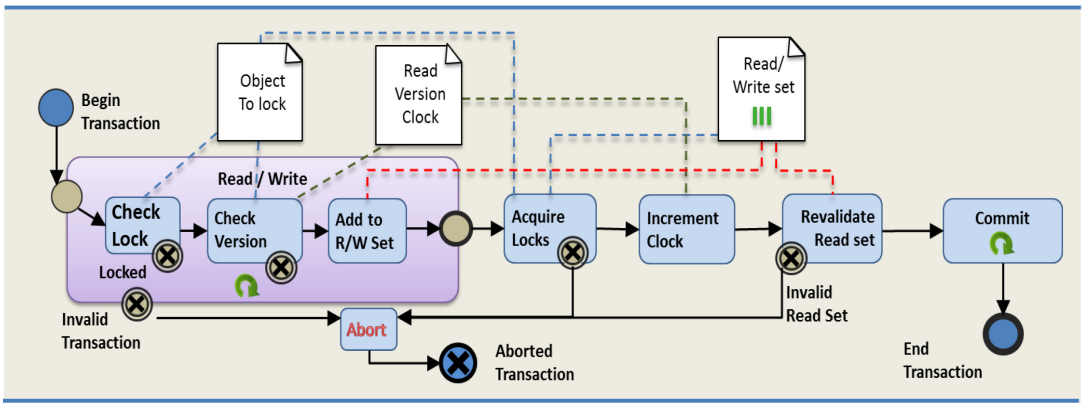

Fig. 1. Example implementation of the Transactional Locking II algorithm in BPMN. TLII is a complex algorithm used to implement transactional memory semantics in non-transactional systems. It's these same expressive capabilities that allow BPMN to be an optimal candidate representation format for patterns

4. MODELING PATTERNS

4.1 Design Patterns

The true impact of patterns for software development first became apparent when, in 1995, Gamma, Helm, Johnson, and Vlissides published *Design Patterns: Elements of Reusable Object-Oriented* [Gamma, Helm, Johnson and Vlissides 1995] containing 23 software design patterns. The book was so widely accepted that the Gang of Four quickly became synonym with software design patterns. Design patterns represent design problems and their respective solutions and entail cooperation between classes and objects.

The GoF book [Gamma, Helm, Johnson and Vlissides 1995] uses the OMT notation to represent patterns. OMT is a close ancestor to UML class diagrams and thereof most books about patterns forego OMT in favor of UML. This type of modeling technique suits well with design patterns because of the inherently static nature of their implementation: Given any type of object oriented programming languages with classes, it is a rather trivial task to map the class diagram onto code. However in our view, the way the patterns exist in the book introduces two problems dealing with pattern representation.

First and foremost, the behavior that the patterns aim to represent is sometimes hard to memorize. If we put aside the best known patterns like Singleton, Iterator, Observer, Proxy, Interpreter we are left with the hard cases. For instance: What is the essence of the Builder pattern and what is the difference to Abstract Factory? What are the steps required to use a Builder? How many different classes do you need to use to implement a Builder? And more importantly, what is a Director?

A textual description with the help of UML can provide evidence on what the pattern represents and how it should be implemented. However, textual description more often than not cannot fully describe the

process of using/implementing a pattern without getting the reader confused. If patterns are to truly be widespread, we should strive to achieve a clear representation of the process that defines the pattern.

In a way, this is already partly achieved through the use of UML sequence diagrams. The main difference between BPMN and UML sequence diagrams is the target audience. BPMN is favored by business people while sequence diagrams are mainly used by software engineers. UML sequence diagrams are used to represent object instantiation and reference calls, representing the execution process with much more detail than that which is achieved through BPMN. However, BPMN exists on a higher abstraction layer but provides a set of modeling artifacts that can be easily understood by the reader.

In the case of patterns, not all patterns have the same level of detail and most are conceptual in their description of the problem and solution. In order for it to be applicable to a wider range of contexts, a more abstract solution is always preferable. Also, there is a plethora of domains outside of software that could possibly benefit from this BPMN representation.

Let us consider the Abstract Factory pattern[Gamma, Helm, Johnson and Vlissides 1995], for instance: It provides an interface to creating a product family by specifying an Abstract Factory instantiates a concrete factory according to the product that is intended. The same factory will build different products based on who is the client using the factory. One could use the Abstract factory to build a way to load files of different types into the same database (see Figure 2).

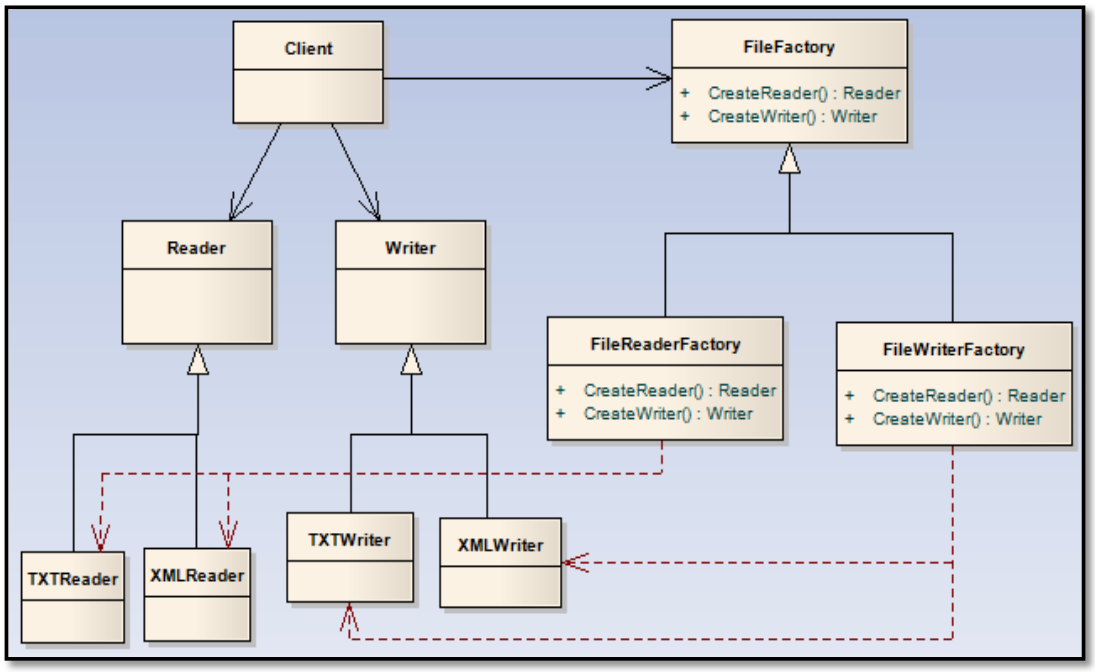

Fig. 2. UML class diagram of a File Reader/Writer Abstract Factory.

The class diagram does help to show the pattern's intent, it however does not show the process of accomplishing said intent. As discussed earlier, to help complement this shortcoming we usually see sequence diagrams such as the one in Figure 3 which provides a runtime view of how the Abstract Factory pattern behaves. This is a standard view in software and is widely recognized, thus it is a valuable asset in any documentation. The fact that it is a software artifact and that it represents runtime characteristics, such as function calls and class instantiations, makes it largely unknown to the non-technical people.

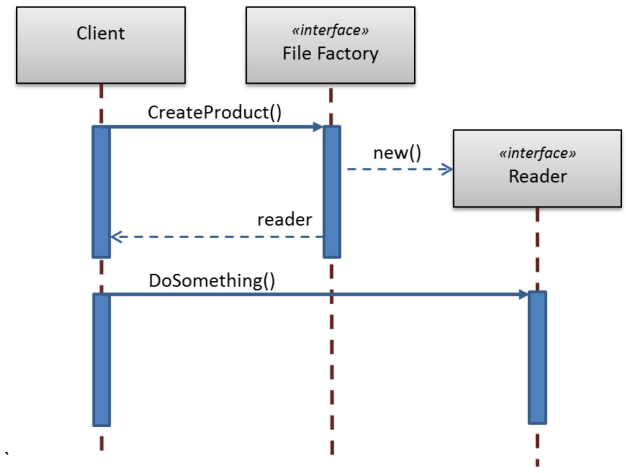

Fig. 3. UML sequence diagram of the Abstract Factory pattern

For the reasons described above, we claim that BPMN is a more than valid alternative to sequence diagrams (see Figure 4). BPMN can provide a similar understanding of the runtime flow of a pattern while preventing alienation from non-technical people. Furthermore, it can represent the same information and, since BPMN has more artifacts than sequence diagrams these specialized artifacts add to the pattern's understandability.

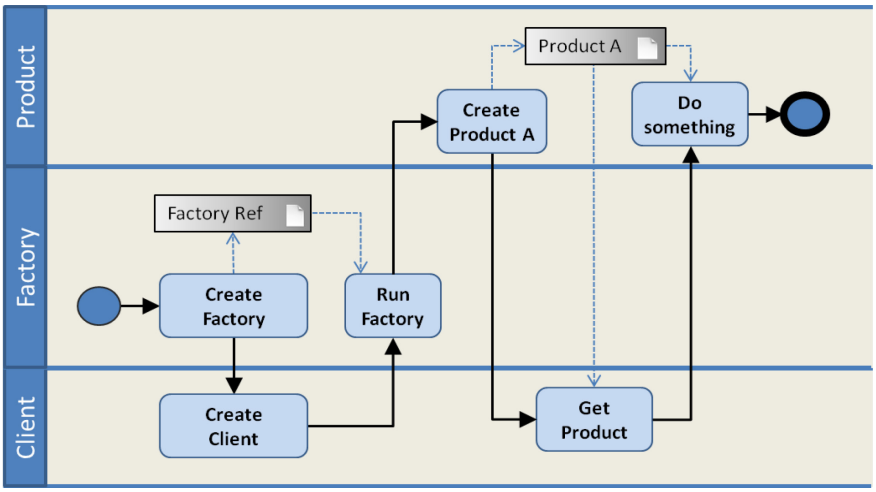

Fig. 4. BPMN version of the *Abstract Factory* pattern.

The same can be achieved for Builder [Gamma, Helm, Johnson and Vlissides 1995]: the Builder represents a similar concept to that of the Abstract Factory; however it focuses on step by step construction of a complex object rather than a single step construction. Builder allows for smaller variations in the way that an object is constructed.

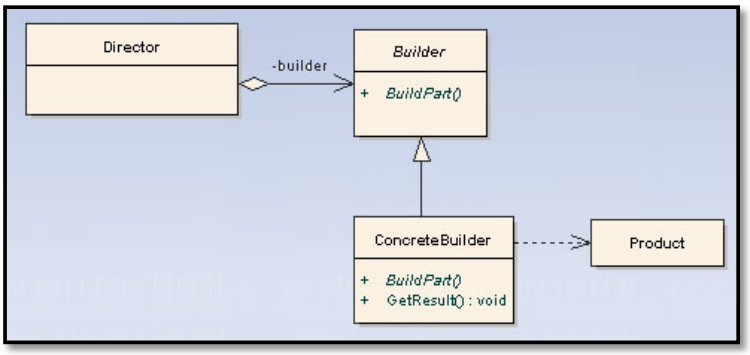

Fig. 5. UML Class Diagram of the *Builder* pattern.

Comparing the traditional UML Class Diagram pattern representation (Figure 5) with the representation of the pattern in BPMN (Figure 6) we can see that the process of instantiating and executing the pattern and obtaining the complex object becomes clearer.

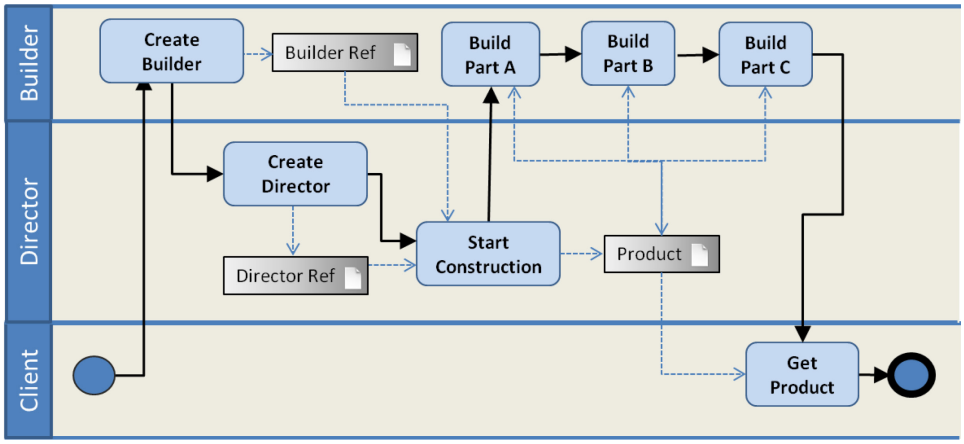

Figure 1 – BPMN version of the *Builder* pattern

Fig. 6. BPMN version of the Builder pattern.

4.2 Security Patterns

In the world of patterns, not all are Design patterns. Most patterns are conceptual by nature [Fowler 1997; Grone 2006; Molina et al. 2002; Monteiro et al. 2011] and as such do not use the traditional UML class diagram representation. These patterns would gain the most with the process representation using BPMN; the pattern concept and the provided solution could be effectively mapped to a model without the pattern losing the abstraction level that makes it conceptual in nature.

Let us consider two examples taken from the domain of security patterns: *Front Door* and *Trusted Proxy*:

A security pattern is a well-understood solution to a recurring problem within the information security domain [Adams et al. 1996; Schumacher 2006]. These patterns are important artifacts because they crosscut many different domains and, as patterns are developed as the means to spread best practices of a given domain, developers who are not security engineers can use and adapt these patterns to achieve security properties in their specific domain.

The *Front Door pattern* [Schumacher 2006] provides a solution for single sign-in in web applications. This is achieved through the use of a proxy between the client and server that checks for user credentials

and grants or denies access. This pattern is easy to envision from the overall description. Nonetheless, there are advantages to formalizing the process described in the pattern in a model.

The BPMN representation of *Front Door* is shown in *Figure 7*. Here since the pattern has communication handled through asynchronous message passing between servers, the model is slightly more complex. However, this is still a very simple and readable model which fully conveys both pattern intent and the process by which it is achieved.

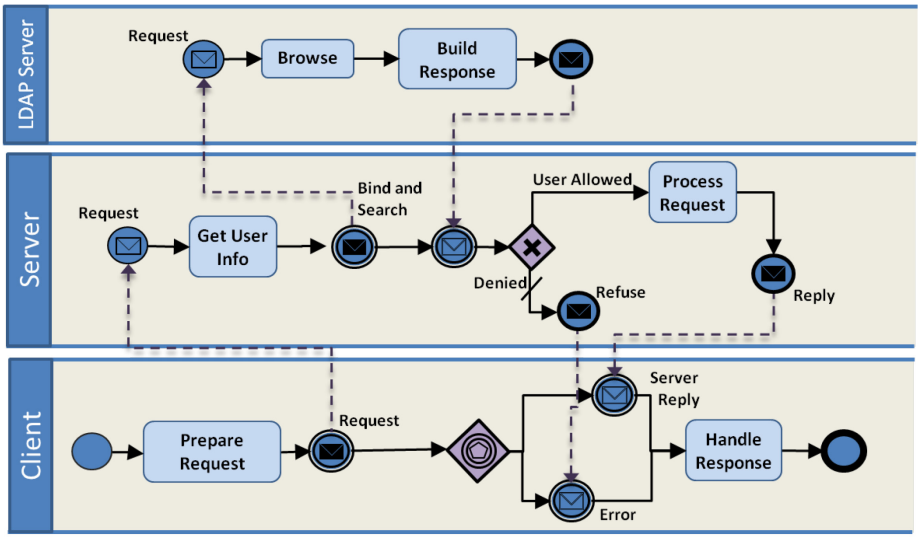

Fig. 6. BPMN version of the Front Door pattern.

The *Trusted Proxy* pattern[Kienzle et al. 2002] allows communication from client to server to be filtered by a proxy. The proxy provides all security mechanisms to ensure that only valid requests reach the server. This way, the server is oblivious to requests from untrusted clients and can use all resources to continue to serve requests from trusted clients. A BPMN representation of *Trusted Proxy* is shown in Figure 8.

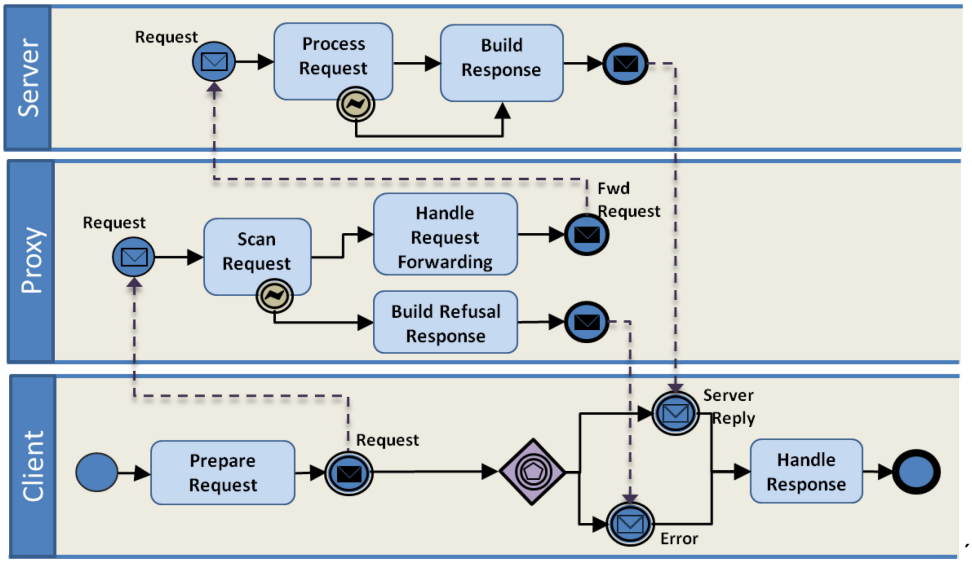

Fig. 8. BPMN version of the Trusted Proxy pattern.

4.3 Pattern Languages and Pattern Composition

When patterns are considered in isolation, as single entities, it is difficult for readers to be fully aware of how the pattern was originally intended to be composed with other patterns. Using stand-alone patterns for real-world systems frequently results in an increase in design complexity, since single patterns often cannot consider the multifaceted context of large scale software requirements [Buschmann, Henney and Schmidt 2007]. *Pattern catalogues* are groups of patterns that share the same domain of application and are written with similar goals in mind. When a set of patterns or a catalogue takes into consideration inter- and intra-pattern dependences and provides a way to guide readers in the steps needed to implement the set of patterns a *Pattern Language* is achieved. The pattern language forms a complete set of patterns, such that choosing to use one pattern will eventually direct the software developer to another related pattern [Meszaros and Doble 1998]. Following the sequence of pattern dependences will lead to complete solutions for a complex contexts.

Yet, there is no formally defined rule for describing pattern dependences in pattern languages. Dependences are usually introduced by a dependence graph or, more traditionally, through a Related Patterns section in the body of the pattern.

Graphical descriptions of the dependences between patterns help pattern readers have an overview of how patterns interact. In software patterns for instance, this means that without extensive knowledge of the pattern language, the designer can have an idea of what patterns will be used to develop a software system. However, this type of graphical description often fails to provide the needed level of description and only presents short non-descriptive comments. In contrast, a Related Patterns section is much more verbose but can often lead to slightly different interpretations of pattern interactions. Therefore, most modern pattern languages use a composition of both forms since they are complementary.

As patterns from pattern catalogues and languages are intended to be used together or at least taken in consideration under similar contexts, more often than not patterns end up being composed together and the borders of each individual pattern get diluted and it becomes difficult to identify the composition as being a set of patterns weaved together instead of a single pattern. This is a problem mainly for evolutionary design where maintenance is a pressing concern. Through BPMN this behavior can be ameliorated. Business process modeling was designed to allow the seamless integration of different processes into a common workflow, while preserving the process form. The remainder of this section will be used to provide evidences of this through the previously described GoF and Security patterns (see sections 4 and 4.2).

The GoF Design Patterns[Gamma, Helm, Johnson and Vlissides 1995] are a pattern set and not a pattern language per se, however, there are some cases when composing design patterns is required. Consider as an example a food court available in any Shopping Center. There are many restaurants available and the menu of any of those restaurants has entrées, main courses and desserts. Each of those products is composed of a variety of elements and it's not uncommon for different restaurants to serve the same dish but cooked using a different recipe. In This case, the Restaurant is a factory of food and each food item is built from a set of other items (see Figure 9).

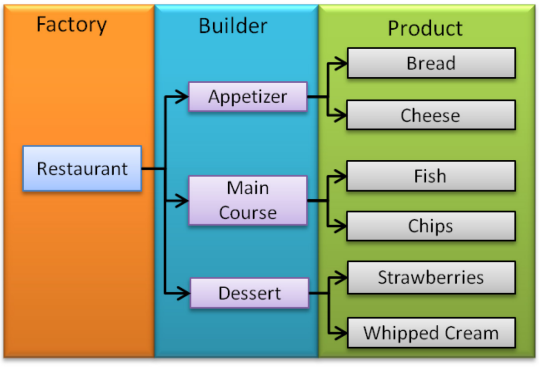

Fig. 9. Abstract Factory and Builder composition example.

Modeling this composition of design patterns in UML class diagrams would do little to clarify the example in question. However, using BPMN, it is a rather simple process of collecting the models of Figure 4 and Figure 6 and creating a common process that describes the act of fabricating products that build other products, as is the case of this example. The BPMN representation of the composition of *Abstract Factory and Builder* can be seen in Figure 10.

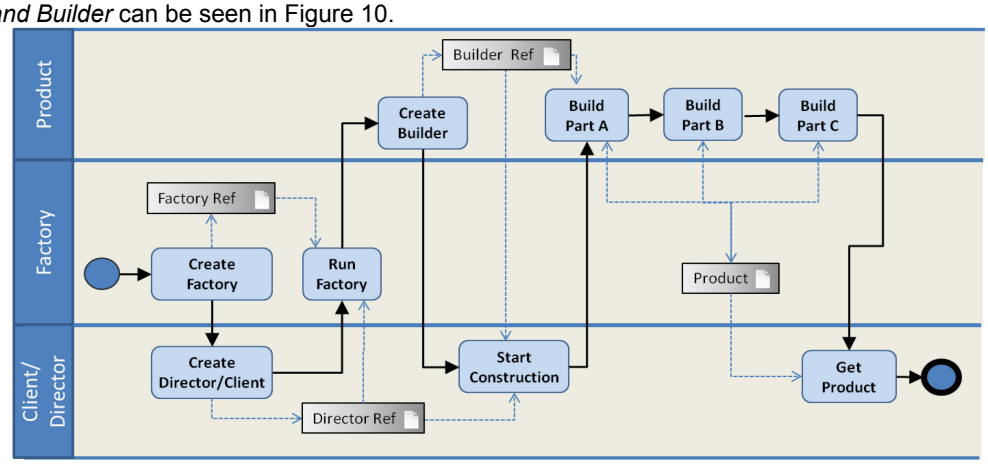

Fig. 10. Abstract Factory and Builder composition in BPMN.

In pattern languages the process is similar. For instance, in *Growing a Pattern Language (for Security)*[Hafiz et al. 2011], the authors succeeded in gathering all published security patterns into a single pattern language. Although the effort is applaudable, the end pattern language, containing a grand total of 96 patterns, is largely unreadable. The way that BPMN can help such cases is to weave the patterns together as a giant process, in fact creating a system of patterns with end-to-end functionality. Picking the previous examples of the *Front Door* and *Trusted Proxy* pattern, both included in the previously cited pattern language, we can envision a case scenario where it would be required to allow single sign-in capabilities only to trusted clients (being a trusted client is not the same as being allowed to login). The BPMN representation of the composition of *Front Door* and *Trusted Proxy* can be seen in Figure 11.

These examples serve to show the potential of BPMN as a pattern composition tool. There is a direct representation of both the process of the independent patterns as of the composition. Furthermore, as was the case with *Front Door* and *Trusted Proxy,* BPMN can help compose patterns from different pattern languages, as long as there is a common context.

4.4 Applicability to other pattern domains

Pattern catalogues and languages for software design represent a widely prolific area of development. Mainly due to the success of the GoF book [Gamma, Helm, Johnson and Vlissides 1995], patterns became popular in the field of reusable design, branching different application areas such as object-oriented programming [Noble and Sydney], aspect-oriented programming [Zdun 2004], framework design [Roberts and Johnson 1998; Wolf and Liu 1995], software architecture [Avgeriou and Zdun 2005; Buschmann, Meunier, Rohnert, Sommerlad and Stal 1996], and even patterns about patterns [Coplien and Woolf 1997; Meszaros and Doble 1996].

HCI patterns for instance, are usually visual artifacts or concepts (like Shopping Cart or Wizard [Van Welie]) whose behavior can be modeled using BPMN but that might not actually merit the effort. Many times these patterns are self-explanatory. Consider the BPMN version of the Shopping Cart and Wizard pattern (Figure 12 and Figure 13). These diagrams are pretty straight forward; however there is little that can be learned from the pattern's BPMN representation that cannot be anticipated by the pattern's name. In the case of the Wizard pattern the diagram might even serve to make the meaning of the pattern more obscure.

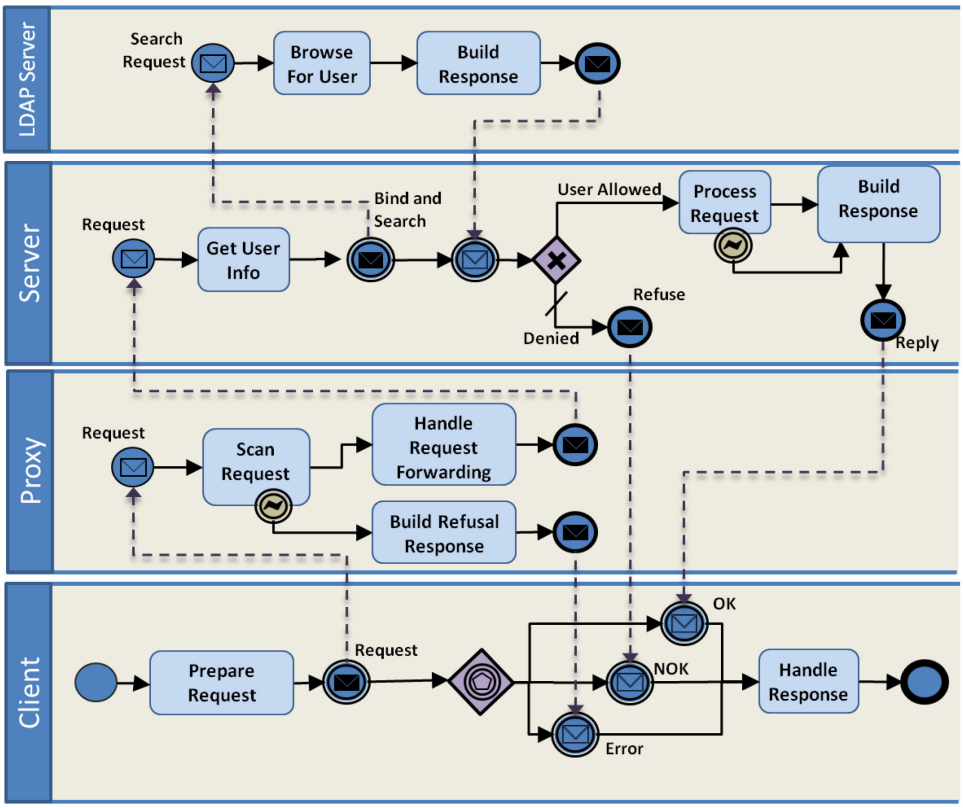

Fig. 11. Front Door and Trusted Proxy composition in BPMN.

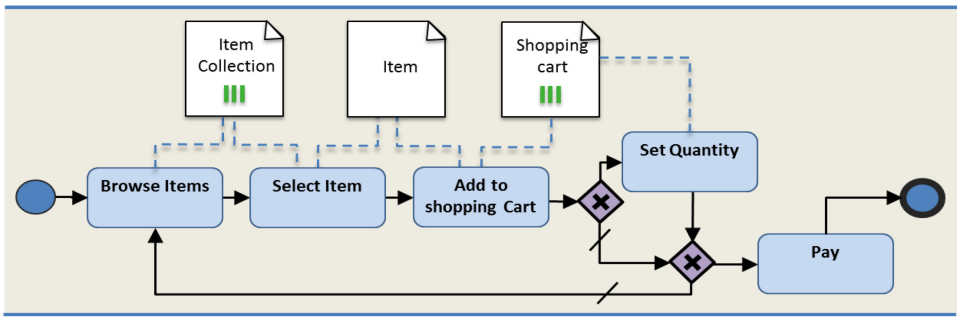

Fig. 12. BPMN Shopping Cart Pattern.

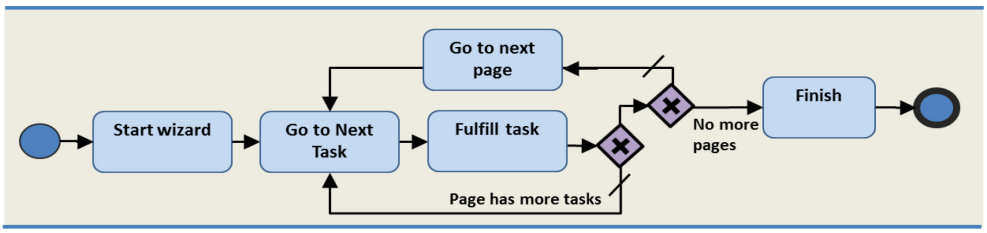

Fig. 13. BPMN Wizard Pattern.

So, while this BPMN representation seems to be expressive enough to express patterns from different domains and with different levels of granularity, in some cases it is not justified to represent it so. This is true for HCI patterns but the same thing holds for learning patterns or architecture patterns. The closer a pattern is to its conceptual model the less need there is of representing its processes.

Nonetheless, if the pattern is to be used in a business context, then a BPMN representation would stimulate the pattern's usage and adoption by the business domain experts, as well as facilitate its inclusion in existing and future business process models, thus increasing adoption and recognition.

5. RELATED WORK

In regards to patterns, very few considerations were ever made about how patterns should be visually represented: patterns are, by design, textual descriptions of best practices, dutifully collected by domain experts. Research relating modeling techniques to patterns mostly considers pattern composition and interdependence, for instance in a pattern language or a set of patterns with a common context. Many pattern languages and pattern catalogues have been successfully created; however finding a way to create relationships between patterns of different authors is at best a difficult endeavor. The importance concept of providing a clear way in which readers can apprehend the patterns intent has also been discussed in the Intent Catalog pattern from Meszaros and Doble [Meszaros and Doble 1998]. One such catalogues for the GoF design patterns was presented by Zimmer [Zimmer 1995] with the intent of showing pattern relationships. However, since the intent of individual patterns is lost by the need to use reduced captions, the intent of the bird's-eye view of the GoF patterns is also lost. Knowing how to navigate a pattern language or set is very different from knowing how to navigate between the intent of the patterns focusing on other patterns that have similar intents.

In *Growing a Pattern Language (for Security)*[Hafiz, Adamczyk and Johnson 2011], the authors make such an attempt and develop a pattern language that encases the full set of patterns ever written in the context of the security domain. Due to its size, the final pattern language representation is so immense that the reader has no sense of where to start and which patterns are useful for his specific context. This is a problem shared with many other large pattern languages [Alexander, Ishikawa and Silverstein 1977; Buschmann, Meunier, Rohnert, Sommerlad and Stal 1996; Van Der Aalst, Ter Hofstede, Kiepuszewski and Barros 2003]. Design pattern rationale graphs[Baniassad et al. 2003] help reduce this problem by providing a direct model representation of implementation code and the pattern that justifies that code. This allows pattern relationships within pattern languages to be reinforced through the composition of the patterns related code.

Recently Long et al [Long et al. 2011] took a somewhat stronger approach to this problem by suggesting a high-level uniform representation of patterns (HiLPR). They suggest that each pattern's implementation follows three stages (Software Design, Hardware Characterization and Optimizations) and each stage has different features that suggest implementation choices. The features can be special considerations, implementation pathways or other patterns; thus HiLPR provides a way to compose patterns of different authors. This notion of a pattern being a feature of another pattern's solution is also present in Three Layer Cake [Robison and Johnson 2010]. Nonetheless, it is our view that HiLPR fails by presenting a difficult to read process flow tree, i.e. it does not provide a simple way in which a programmer can know the purpose of the pattern by following the process associated with that pattern. In this our approaches differ: while HiLPR considers implementation choices as the main motivation for this representation, our approach considers that providing a model of the base process flow for the pattern's solution will allow the reader to better understand the pattern's intention and to learn from it, facilitating future optimizations via implementation alternatives and variants. However, we agree that it is important to represent special considerations that readers need to be aware of when implementing the pattern and sustain that BPMN could also be used towards that end. A different approach was used in our previous work on parallel programming patterns [Monteiro, Monteiro and Pingali 2011]: pattern relationships within the pattern language were represented through Feature Diagrams [Kang et al. 1990] which allowed us to represent mandatory and optional patterns within the pattern language.

6. CONCLUSION

In this paper we have presented preliminary work on offering a Business Process Modeling representation of patterns as a complement to traditional UML Class Diagrams. UML was developed to aid developers

and is not, in its essence, suitable for communicating with business domain experts. In contrast, the Business Process Modeling Notation was designed with the intent of providing graphical visualization and modeling tool for end-to-end business process documentation. Therefore it excels in directness and ease of communication between non-expert users and technical people. It is our belief that this method of representing patterns will prove to be a valuable contribution to bridging the conceptual gap between pattern writers and readers.

There is still much work to be done in this field; however, we have argued that, for a considerably varied set of patterns and pattern languages, using BPMN as a complement to traditional pattern representations is not only feasible but also desirable, as composition of patterns intra- or inter-pattern languages is easily achieved.

7. FUTURE WORK

This paper is part of a larger body of work intent on studying BPMN's suitability to model parallel algorithms. Our Hyper Algorithmic Recipe for Productive Parallelism Intensive Endeavors (HARPPIE) [Monteiro 2012] aims to dilute parallel programming difficulties by providing a BPMN-based model driven approach to programming parallelism. HARPPIE stands on four cornerstones: 1) Make parallel programming accessible; 2) On-demand prototyping; 3) Little information overload; 4) Guide design with fast feedback.

8. ACKNOWLEDGEMENTS

We would like to thank Rosana Braga, whose comments have been incredibly helpful and insightful. Her shepherding skills helped us significantly improve the quality of this paper. We would also like to thank our program committee member Takashi Iba.

APPENDIX: BPMN Notation

This section intends to provide a small overview of some of the different elements that compose the Business Process Modeling Notation. We will focus on the elements used in this paper and refer the reader to the BPMN 2.0 specification [Model 2009]for further reading.

Activities. A business process is generally defined as a set of activities occurring in coordination within an environment. These activities are specific to that environment and together contribute to a common business goal. However, activities and processes within an environment can interact with other environments. Activities represent the design-time definition of a concept and can be either atomic, i.e. representing a single task, or composed of sub-activities, allowing the process to be defined with different levels of granularity. Therefore a compound activity is in itself a process composed of other BPMN elements.

An activity can also occur multiple times, according to its attributes. If a multi-instance activity is sequential it describes a loop iteration over that activity; if it is parallel it describes multiple activities occurring at the same moment.

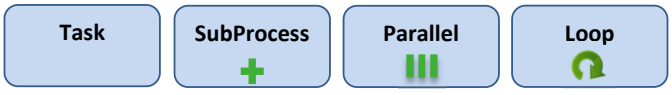

Fig. 14. BPMN Activities

Swimlanes. These are used to distinguish among different environments related to the process. Each participant or environment has its own individual swimlane, aka a Pool. Communication between pools is

equivalent to communication between processes. Pools can be further subdivided into lanes, which are little more than grouping elements.
Artifacts. Are elements that add information to the model, not necessarily modifying its behavior. The most interesting in this case are the data artifacts, namely the Data object and the Collection. These represent data that can be used as input or output of the activities and are linked through associations.

Fig. 15. BPMN Data Artifacts.

Flows. A flow can be either a sequence flow, a message flow or association. The Sequence flow represents the transition from one activity to another. Message flows are used to connect processes from different pools are the only type of flow that can cross pool boundaries. Associations are used link information and data with activities and process elements.

Fig. 16. BPMN Flows

Gateways. A Gateway controls logical flows within the processes. There are different types of gateways, each with a different marker identifying its behavior. There are many types of gateways: Exclusive OR (XOR) and Parallel (AND) Gateways, Inclusive (OR), Exclusive Event-Based and Complex gateways. For this paper we will only focus on the gateways used in the examples, namely the exclusive and the exclusive event-based gateways.

The exclusive gateway is a graphical representation of an IF clause, meaning that only one of the outgoing flows can be used at a time. The Exclusive Event-Based is used to control external messages or events and when one such event is triggered, all out flows are activated. This means that this gateway is reactive and oblivious to data. It is up to the outgoing flows to decide if they should be executed or not.

Fig. 17. BPMN Gateways

Events. Events represent actions that are triggered or caught by the process. Events can be used to Start or End a flow or alternatively to trigger intermediate actions, such as exception or timed events. Additionally, there are untriggered events to represent the start and end of a business process. Intermediate events belonging to an activity can be placed at the border of said activity to indicate that they represent exit conditions.

Message events are responsible for either sending or receiving messages to/from other Pools or processes. They represent the way different processes communicate and share information and requests. Error events, as the name indicates, refer to exceptions that occur within a process, while cancel events refer to voluntary terminations of a flow based on some condition. Additional types of events are described in the BPMN 2.0 specification [Model 2009].

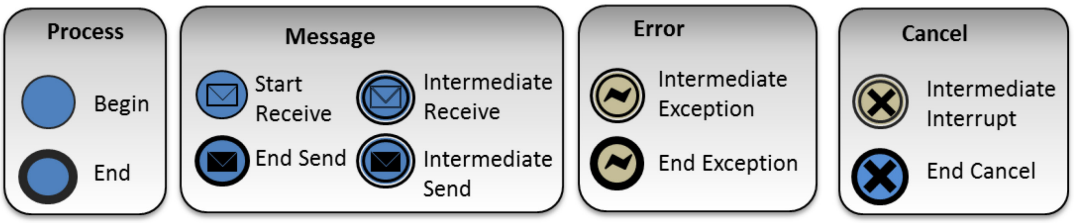

Fig. 18. BPMN Events

REFERENCES

Aagesen, G. AND Krogstie, J. 2010. Analysis and design of business processes using BPMN. *Handbook on Business Process Management 1*, 213-235.

Adams, M., Coplien, J., Gamoke, R., Hanmer, R., Keeve, F. AND Nicodemus, K. 1996. Fault-tolerant telecommunication system patterns Addison-Wesley Longman Publishing Co., Inc., 549-562.

Alexander, C., Ishikawa, S. AND Silverstein, M. 1977. *A pattern language: towns, buildings, construction*. Oxford University Press, USA.

Avgeriou, P. AND Zdun, U. 2005. Architectural patterns revisited–a pattern language Citeseer, 1-39.

Baniassad, E.L.A., Murphy, G.C. AND Schwanninger, C. 2003. Design pattern rationale graphs: Linking design to source. In *25th International Conference on Software Engineering* IEEE Computer Society, Portland, Oregon, 352-362.

Buschmann, F., Henney, K. AND Schmidt, D. 2007. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. John Wiley and Sons.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. AND Stal, M. 1996. A system of patterns: Pattern-oriented software architecture Wiley New York.

Coplien, J. AND Woolf, B. 1997. A pattern language for writers workshops. *C Plus Plus Report 9*, 51-60.

Dice, D., Shalev, O. AND Shavit, N. 2006. Transactional locking II. In *20th International Symposium on Distributed Computing (DISC'06)*, 194-208.

Fowler, M. 1997. *Analysis Patterns: reusable object models*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. AND Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented*. Addison-Wesley Longman.

Grone, B. 2006. Conceptual Patterns.

Hafiz, M., Adamczyk, P. AND Johnson, R. 2011. Growing a Pattern Language (for Security). In *Proceedings of the Pattern Languages of Programs (PLoP 2011)*2011.

Henninger, S. AND Corrêa, V. 2007. Software pattern communities: Current practices and challenges Citeseer.

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E. AND Peterson, A.S. 1990. Feature-oriented domain analysis (FODA) feasibility study DTIC Document.

Kienzle, D.M., Elder, M.C., Tyree, D. AND Edwards-Hewitt, J. Security patterns repository version 1.0. In *DARPA, Washington DC*, (retrieved July,2012) http://www.scrypt.net/~celer/securitypatterns/repository.pdf,

Lohmann, N., Verbeek, E. AND Dijkman, R. 2009. Petri net transformations for business processes–a survey. *Transactions on Petri Nets and Other Models of Concurrency II*, 46-63.

Long, D.K., Gibbs, C. AND Coady, Y. 2011. The High-Level Pattern Representation with Pretty Pictures. In *Proceedings of the Pattern Languages of Programs (PLoP 2011)*, Portland, Oregon2011.

Meszaros, G. AND Doble, J. 1996. Metapatterns: A pattern language for pattern writing Citeseer.

Meszaros, G. AND Doble, J. 1998. A pattern language for pattern writing. *Pattern Languages of Program Design 3*, 529-574.

Model, B.P. 2009. Notation (BPMN) Version 2.0 OMG Standard, Object Management Group/Business Process Management Initiative.

Molina, P., Meliá, S. AND Pastor, O. 2002. User interface conceptual patterns. *Interactive Systems: Design, Specification, and Verification*, 159-172.

Monteiro, P. 2012. HARPPIE: Hyper algorithmic recipe for productive parallelism intensive endeavors. In *34th International Conference on Software Engineering, ICSE 2012* IEEE, Zurich, Switzerland, 1559-1562.

Monteiro, P., Monteiro, M. AND Pingali, K. 2011. Parallelizing Irregular Algorithms: A Pattern Language. In *Proceedings of the Pattern Languages of Programs (PLoP 2011)*, Portland, Oregon2011.

Noble, J. AND Sydney, A. Towards a pattern language for object oriented design. *Proc. of Technology of Object-Oriented Languages and Systems (TOOLS Pacific) 28*, 2-13.

Nysetvold, A.G. AND Krogstie, J. 2006. Assessing business process modeling languages using a generic quality framework. *Advanced topics in database research 5*, 79-93.

Outsystems, (retrieved August, 2012) http://www.outsystems.com

Peterson, J.L. 1981. Petri Net Theory and the Modeling of Systems. *PRENTICE-HALL, INC., ENGLEWOOD CLIFFS, NJ 07632, 1981, 290*.

Recker, J.C., Indulska, M., Rosemann, M. AND Green, P. 2006. How good is BPMN really? Insights from theory and practice.

Roberts, D. AND Johnson, R. 1998. Evolving frameworks: A pattern language for developing object-oriented frameworks. *Pattern Languages of Program Design 3*, 471–486.

Robison, A.D. AND Johnson, R.E. 2010. Three layer cake for shared-memory programming. In *Workshop on Parallel Programming Patterns* ACM, Carefree, Arizona, USA, 5.

Schumacher, M. 2006. *Security Patterns Integrating Security & Systems Engineering*. Wiley-India.
Sindre, G. 2006. An analytical evaluation of BPMN using a semiotic quality framework. *Advanced topics in database research 5*, 94.
Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B. AND Barros, A.P. 2003. Workflow patterns. *Distributed and parallel databases 14*, 5-51.
Van Welie, M. Interaction design patterns, (retrieved July, 2012) http://www.welie.com/patterns/index.php
White, S. 2005. Using BPMN to model a BPEL process. *BPTrends 3*, 1-18.
White, S.A. 2004. Introduction to BPMN. *IBM Cooperation*, 2008-2029.
White, S.A. 2004. Process modeling notations and workflow patterns. *Workflow Handbook*, 265-294.
Wolf, K. AND Liu, C. 1995. New clients with old servers: A pattern language for client/server frameworks. *Pattern Languages of Program Design*, 51–64.
Zdun, U. 2004. Pattern language for the design of aspect languages and aspect composition frameworks. *IEE Proceedings-Software 151*, 67-84.
Zimmer, W. 1995. Relationships between design patterns. *Pattern Languages of Program Design 1*, 345-364.