Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

Complexity Management Patterns and Generative Software Development

Mohammad Reza Besharati[1], PhD Candidate, Sharif University of Technology, Tehran, Iran
Mahdi Mostafazadeh, PhD Candidate, Sharif University of Technology, Tehran, Iran
Mohammad Izadi, Sharif University of Technology, Tehran, Iran

Abstract

According to many researchers, Abstraction is the basis of mathematics, computing, counting devices, and computer science and engineering. What is more, all of the above deal with complexity management in some way, and abstraction is the most basic mechanism of complexity management.

Generative software development - whether in the sense of empowering humans by machine to create software or in the sense of reusing products - has been and is one of the serious concerns and goals of software engineering. The interesting thing is that in both views of generativity, the main issue is still, in a way, complexity management: whether this complexity management is to achieve diversity and reuse management (Czarnecki's approach) or to Structuring from existing structures (the approach of Alexander and his followers in an object-oriented society).

In this article, we will first look at complexity and its various definitions. The definitions that show, despite the different perspectives on complexity in different disciplines and domains, all point in one direction. We will conclude that complexity is rooted in multiplicity. In the following, we will formally define complexity. In the following discussion, we will look at the generative patterns of software development, and then we will look at the complexity management patterns at seven levels.

In this article, the author has tried to maintain a comprehensive approach to complexity and to consider the approaches of different domains to complexity.

Keywords: Abstraction, Complexity Management, Patterns, Generative Patterns

[1] Corresponding Author: besharati@ce.sharif.edu

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

Introduction

Definitions of Complexity

Complexity is a feature of objects and systems that is usually considered the quality of being hard to separate, analyze, or solve (as a problem) [5]. In different domains, various features have been proposed for complexity. Apparently, most of the definitions share the following features [5], [15]:

1. Size, count, and number can cause complexity.
2. Relationships and dependency of components can cause complexity.
3. Complexity prevents perception, identification, and conclusion (before it is managed).

Some references have mentioned other features of complexity which seem to be variants of the above features (*e.g.* resource limitations [4] and time limitations, which are actually variants of the second abovementioned feature, or uncertainty and inability to predict the system behavior [24], which would be considered variants of the count of system states).

Complexity is sometimes considered a factor that increases workloads or concerns of controllers in systems [15]. Within the project management domain, project complexity is considered the nature, quality, and magnitude (= quantity) of organizational subtasks and their relationships which have been added to an organization for the sake of a project [20].

Some references have distinguished between two classes of complexity definitions, *i.e.* physical manifestation complexity resulting from components and their relationships and mental complexity from the perspective of human users [5]. There is also another definition encompassing the two previously mentioned ones. It defines complexity as a feature that prevents a comprehensive and thorough evaluation of a system even if all information regarding system components and their relationships is available [5]. This definition is also characterized by mental complexity and physical manifestation complexity.

Basically, psychologists consider complexity a feature that prevents cognition; therefore, it induces anxiety. In other words, complexity means the perception of the fact that an object of system of interest is something but what has been supposed to be or should be [9].

A study has found interesting results by discovering people's perception of complexity in an organization. Asked what complicated or had complicated the module which they were creating, 80 individuals involved in creating an engineering product (from the senior project manager to engineers and technical managers) gave different responses. Figure 1 demonstrates their responses [21].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

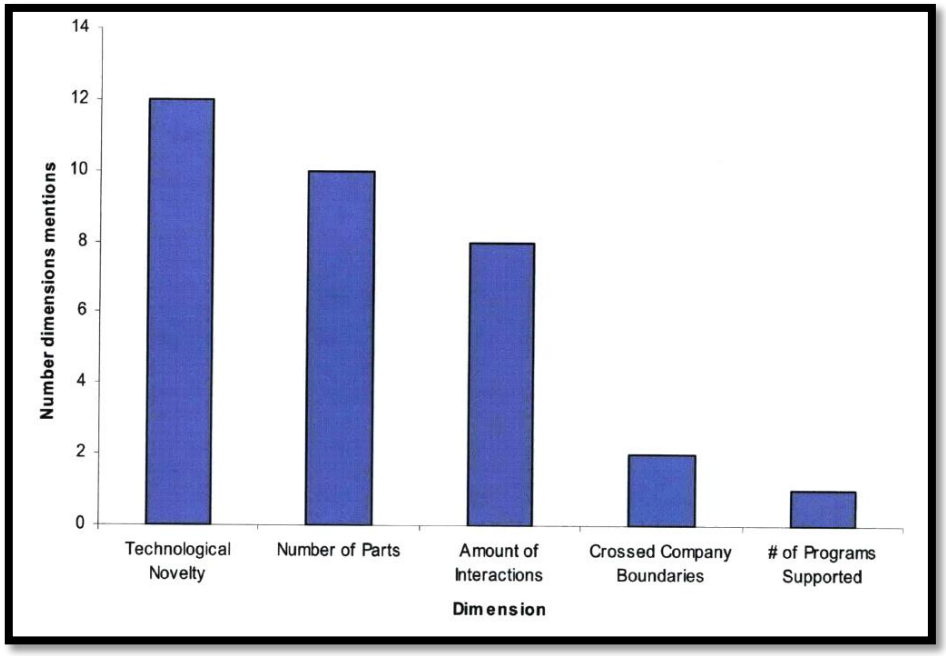

Figure 1- Responses given 80 individuals involved in creating an engineering product in an organization to a question regarding the source complexity

Within the scientific scope of systems engineering, complexity was considered to result from the multiplicity of components and their relationships in the 1970s. A more complex approach to complexity has recently been popular. This approach separates the three areas of complexity known as structural complexity, behavioral complexity, and interface complexity [21], [24].

To the author, it is interesting to realize that structural complexity has been more prominent than behavioral complexity in software engineering texts where there is a more complex approach to complexity. This is also observed in software architecture and design patterns, which are mostly structural in aspect. A question arises here. Does behavioral complexity really have less importance and a less prominent role in software systems and software applications than structural complexity? Moreover, are programmers following the right paradigm? Probably, programming and its convention, which is based on programs with static structures, have led programmers to keep the behavioral aspect diminished and rely on the structural aspects of software systems. Can relatively newly-emerged approaches such as search-based software engineering, generative software development, software development through animation of formal specifications be considered the sparks that might change the dominant approach to the world of software someday?

Kolmogorov (1963) defined complexity within the scope of structural complexity [17], [24]. Accordingly, the complexity of a program (or an algorithm) equals the shortest description length [24]. Evidently, this definition can scientifically be used if an invariant basic language is presented to describe programs and algorithms — similar to a Turing machine. An important advantage of this definition is that it has inherently valued the simplest and shortest description, which will be considered the most abstract

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

description in this field.[2] Therefore, a component of this system which can be abstracted will have no effect on complexity. In other words, if a complexity is to be reduced, it should be made more abstract and simpler. It will then be shown that software engineers agree with Kolmogorov and that abstraction is the most basic mechanism for complexity management.

Formal Definitions of Complexity and Complexity Management Based on the Set Theory

As discussed earlier, there are various definitions of complexity, each of which has been formulated for a specific scientific and engineering area within the contexts of its relevant concepts. Despite differences and due to the similarity of basic and original concepts, it is possible to achieve a high-level, formal definition of complexity. A formal count-based definition of complexity is given first. Accordingly, a formal definition of complexity management is then presented. Similar to the concept of cardinality in the set theory, this definition results from the abstraction of various definitions which have already been presented.

Formal Definition of Complexity

Before complexity is defined, it is necessary to define what complexity should be calculated for. Since the set theory has already been accepted as a basis for mathematics and abstraction,[3] the concept of set is selected as the most basic construct, structure, and concept.

The concept of set is defined as a collection (set) of objects. If "abstraction" and "object" are considered predefined basic concepts, the concept of set can then be defined accordingly. Inversely, if the set is considered "abstraction" and "object" can then be defined accordingly. The first approach, *i.e.* using "abstraction" and "object" as basics, seems to be more natural to the author.

Definition 1: The complexity of a set equals the number of its members.

Definition 2: An N-set is simple if its complexity is lower than or equal to N.

Accordingly, this definition resembles the definition of cardinality in the set theory. If the concept of "abstraction" is considered a presumption and base, then the concept of "number of members" — used in the above definition — will at least be defined for countable and finite sets. This concept can also be made compatible and well-defined for more complex structures such as uncountable and infinite sets, something which is performed in the set theory.

---

[2] "The Kolmogorov measure hints at a key idea, which is the use of abstractions to reduce the length of the system structure description and its Kolmogorov complexity." [24]

[3] The fundamental objects in mathematics are sets and their constituent elements [14].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

In the approach adopted by the author for complexity and complexity management in this paper, countable and finite sets are only dealt with. Therefore, there is no need to define complexity for uncountable and infinite sets.[4]

Relationships between Complexity, Abstraction, and Set

It can be concluded that complexity is definable for every set. Inversely and according to the above definition, if there is complexity somewhere, there should also be a corresponding basic set. Hence, this rule of thumb can be achieved to state that "the presence of a set denotes the presence of complexity, and the presence of complexity means the presence of a set". Therefore, "presence of complexity" and "set" are somehow two equivalent concepts, which are considered equal in this study.

Every different abstraction of an object, a system, or an objective reality leads to a different and even single-member set.[5] Every set can also be considered to be corresponding to an abstraction; therefore, "abstraction" and "set" are equivalent concepts, which are considered equal in this study.

As a result, a general but very important rule is achieved: "presence of complexity", "abstraction", and "set" are equivalent concepts which are considered equal in this study.

Complexity of Complexity Calculation

It should be noted that every abstraction — differing from an object or a system which can consist of complex components — will lead to a different set. Hence, different complexities can be calculated for an object based on different abstractions. The output of an abstraction can also be used as the input of another abstraction. With every abstraction, a set (that might even have only one member) is defined; therefore, although the proposed definition of complexity seems slightly simple, "calculation of complexity" depends on how the analyzed system is viewed. Based on the abovementioned reasons, the calculation of complexity is a complicated process of many counts. Every different type of view leads to various abstractions of the system [12] and the results of different calculations for complexity.

It should also be noted that the proposed approach focuses rarely on the quantitative calculation of complexity in this paper. It will be shown that detection of complexity is more important to authors than its calculation.

[4] However, a definition of complexity can be achieved for uncountable and infinite sets by applying abstraction many times — or through the abstraction of a set of abstractions. In the set theory, the cardinality of uncountable and infinite sets is also calculated in this way.

[5] A set is an abstraction of the informal concept of a collection of objects [14].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

Formal Definition of Complexity Management

The existing definitions of complexity management have already been reviewed. In fact, "interface" is a basic concept mentioned in all of those definitions. Generally, complexity management means relating two complexities with each other. According to a previously introduced formal definition, complexity corresponds to the count of a set.

Consider set A with M members and set B with N members. The relationships of M members of set A should first be defined in order to define the relationship between these two sets. However, a relationship is an abstract concept that seems to be simple; therefore, it appears that the most basic level of complexity management is to establish a relationship between a complex set and a simple set. According to Definition 2, simplicity is a relative concept for which a bound should be considered with respect to every application and context. Therefore, a broader definition is presented as below:

Definition 3: Establishing a relationship between the M-member set A and the N-member set B is an act of complexity management on them.

Properties of Complexity Management:

First: Complexity management is a Holonic concept. In other words, when sets A and B are managed in terms of complexity, both their internal structures (*i.e.* members) and external structures (defined as the externally observed relationship between these two sets) are considered important. Therefore, a set is considered both a whole (including members) and a component (*i.e.* one side of a relationship) [23].

Second: This concept can also be used as a fractal, for the mechanism in which two sets are related can be applied to the internal members of every set to establish internal relationships in that set. In fact, there is no presumption regarding the abstraction levels of both sides in a relationship, and it can be reapplied at any level.[6]

Third: Every complexity management process can be viewed as a structural operator — in structural mathematics — which takes two operands. The parties on the two sides of a relationship (*i.e.* two sets and two complexities) are the operands. This operator yields a novel complexity space including the two previous sets and a third set that results from a relationship between those two sets. Hence, this definition of complexity management is totally consistent with the type theory and structural mathematics, which make this definition well-defined.

[6] The main characteristic of fractals is self-similarity that implies recursion and pattern-inside-of-pattern. Mandelbrot's sets display self-similarity because they not only produce details on finer scales but also generate details with certain constant proportions or ratios, although they are not identical [2].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

Fourth: Without abstraction, there will be no sets, without which it is impossible to define complexity management. At the same time, the concept of "relating" is a genuinely a kind of abstraction; hence, it can be stated that managing any complexity requires the creation of a new complexity in the form of a few abstractions.

Software Engineering Patterns and Their Roles in Complexity Management

Although Pattern Driven Paradigm started with Cunningham and Beck's inspiration from a book by Christopher Alexander, who was an architect living outside the realm of software engineering, this school of thought evolved and reached dynamism quickly. Not only patterns (patterns) were started to be employed to convey and record experiences and widely-used proven solutions in 1994, but it was also the time when the idea of having a common language for design and reuse of products came into realization also for the automation of software development processes [7].

The roles of patterns in complexity management should be analyzed in two aspects, one of which resembles their roles in other areas of software engineering. In other words, they record and convey methods, experiences, and widely-used proven solutions. They also become a common language to allow for reuse. This role of patterns can always be considered.

However, the most important role that patterns can play in complexity management is something else. Patterns can be used as a base for abstraction in software engineering.

As discussed earlier, the most important mechanism and concept of complexity management is abstraction. However, the problem arises when the space of possible choices for abstraction and the space of possible abstractions are complicated. Although abstraction simplifies the current problem space and decreases its complexity, the complexity of the entire space of existing solutions will increase. This can cause the current crises such as the maintenance phase crisis, the software analysis crisis, the crisis of no reuse of software products, and high risk in the development process. All of these problems are rooted to some extent in the complex space of existing solutions in software engineering and redevelopment of every solution.

However, what benefits can patterns provide? Patterns can define a kind of common infrastructure of abstractions between different solutions to prevent the complexity and multiplicity of a solution space in software engineering. In fact, similar patterns of abstractions result in harmony and compatibility of solutions in different software projects, something which mitigates the existing crises in software engineering.

As discussed earlier, complexity management is a key principle in software engineering, whereas abstraction is the main element and mechanism for complexity engineering.

It was interesting for the author to find out that the harmony and compatibility of structures were among the goals pursued by Christopher Alexander, who was invited to OOPSLA in 1996 to lecture the object-

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

oriented community.[7] He stated, "What is now my evaluation of what you (the object-oriented are doing with patterns in computer science? When I look at the object-oriented projects on patterns, I can see the format of one pattern (context, problem, solution, *etc.*). Therefore, object-oriented patterns constitute a good interface for relationships (between software designers). The existing patterns are also useful tools for software design. However, this is not everything that patterns and pattern languages are supposed to achieve. The pattern languages, which I was seeking in the 1970s, had other inherent features. Moreover, these languages aimed to achieve compatibility and coherence, which are the outcomes of what these patterns build. In addition, they should be generative and allow people to generate compatible and sensible objects by encouraging and enabling them to follow this process (= generativity)" [3].

Therefore, not only are the special patterns of complexity management useful and necessary for this purpose, but also the general patterns of software design and analysis can be useful for complexity management as long as they help develop an integrated and common context infrastructure for abstraction in different software projects.

Generative Patterns

Between 1994 and 1997, generative patterns were the ones which would not only describe the final desirable state but also present the method of reaching the final state and developing a solution.[8] By contrast, non-generative patterns are the ones which only describe the state structurally on the way to the final solution and offer no special suggestion to achieve the final state and apply or develop the solution.

In fact, according to those definitions, the quality of being generative depended more on the description and expression of a pattern than the pattern itself. In a paper (1994), Kent Beck and Ralph Johnson reviewed a few GoF design patterns (*e.g.* observer, composite, *etc.*) and stated, "The version of patterns proposed in this paper differs from the version of patters in the catalog (GoF) in two aspects. Another difference lies in the fact that the proposed version is more generative than the version in the catalog. In other words, this paper emphasizes and states under what conditions the pattern is applied and what transformations will be caused in the design by the emergence of every pattern after it is applied. The catalog emphasizes the typology of solutions; however, this paper focuses on how to use the solutions. Only a few patterns of the catalog were rewritten to make programming more generative, and there is no reason that it is impossible to create a similar version for other patterns of the catalog" [16].

---

[7] To the author, the quoted lines of Christopher Alexander's lecture are very thought-provoking, deep, and meticulous.

[8] Rather than describing the solution to a problem (non-generative patterns), generative patterns describe how to come to the solution. Generative patterns exist for a number of different phases and situations related to software development. There are generative design patterns, motifs (*i.e.* documentation patterns) and maintenance patterns [22].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

However, software generative patterns are not limited to the generative patterns of design. For instance, James Coplin (PLoP, 1994) proposed a set of generative patterns for an organizational process [6]. Those patterns were generative because they could be used directly to develop an organizational process. In fact, they did not just describe "what was supposed to be" but included the method of reaching the final construct. This is exactly the same concept that was attributed to generative patterns in 1994.

Since 2000, the concept of generativity has been defined through reuse.[9] The generative patterns were then known as the patterns which could be used as reusable structures even at the code level [25]. For instance, the CO_2P_2S (correct object-oriented pattern-based programming system) framework was proposed by researchers at the University of Alberta in 2002 to solve the problem of generativity in software patterns. In other words, the framework was meant to transform reusable structures even at the code level [25]. The researchers adopted the pattern parametrization approach to achieve generativity, which they considered to be the ability to produce code generativity for every specific situation in which the pattern was supposed to be used in. It means that users can set the values of parameters to inform the generative pattern of the special situation of a problem. Afterwards, the generative pattern can generate its code automatically through a code generator. Image 1 demonstrates their proposed approach:

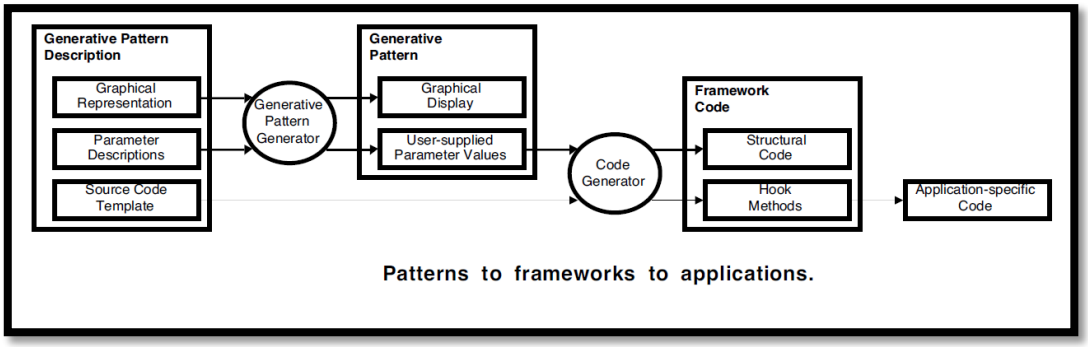

Image 1. The CO_2P_2S approach to achieving generative patterns in software design [25].

If the CO_2P_2S approach to generativity is to be stated in Czarnecki's literature, it should be stated that this framework is based on configuration at a high level but based on transformation at a low level for the realization of configuration.[10]

[9] The developed of this trend was Christopher Czarnecki who proposed his thesis entitled *Generative Programming [8]* in 1999. He was still the most prominent researcher of genericity — through reuse of products and codes — in the past years. However, the term "generative pattern" existed relatively extensively before Czarnecki. Interestingly, there is no sign of this term in his thesis. Apparently, Czarnecki's success resulted in the association of genericity with reuse, something which did not use to be necessary true. As discussed previously, generative patterns from 1994 to 1997 included the patterns that would include how to build them.

[10] Czarnecki categorized the feasible approaches to genericity as two classes or meta-patterns: genericity through configuration and genericity through transformation. In other words, it is necessary to use the configuration of a regular structure or create

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

Another approach to achieving generative patters is to use pattern description languages, especially XMLs such as PLML (pattern language markup language) and UsiXML [26], to generate codes automatically. This approach has been relatively successful in certain domains, *e.g.* deigning user interfaces. Similar to the above approach, user interface description languages are now used in most of commercial and open-source frameworks of software development to generate user interfaces.

Simaro *et al.* drew a comparison between descriptive patterns and generative patterns. The more generative the pattern, which enables developers to create a solution and shows how to reach the solution, the less general the pattern. However, the pattern expressivity increases, and the pattern becomes more accurate and detailed. The following figures shows an overview of the pattern:

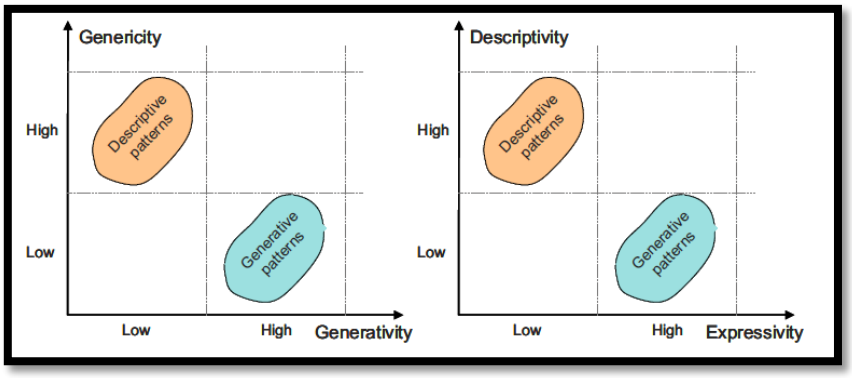

Figure 2- Generativity, Descriptivity, Genericity and Expressivity [26].

There is another distinct and interesting approach to generative patterns. Discussed and analyzed by Hoverd in 2008 [13], this approach appears to be a resurrection of Alexander's approach to generative patterns. As discussed earlier, since the generative programming discourse was introduced by Czarnecki, generativity has been summarized as reuse. However, Alexander's approach to generativity in the object-oriented community (during the 1990s) was much more extensive. In this approach, generativity is defined beyond statics and dynamics. The evolutionary path of systems towards becoming complex but more natural is as follows: static systems, dynamic systems, and then generative systems. A static system does not change during execution, whereas a dynamic system changes during execution. However, a generative system is generated during the execution time. In other words, it emerges during execution. Accordingly, the generative patterns of software are the patterns which are software structures that result in the emergence of other software structures. In this regard, generative patterns are considered an order which can lead to new orders that will then repeat this procedure. Moreover, Alexander's approach is not general and inaccurate. The following figure shows a meta-model of Alexander's approach to generative patterns by Howard in the format of a UML chart.

---

that structure by applying the transformations to the structures of the problem realm to map the problem space onto the solution space.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

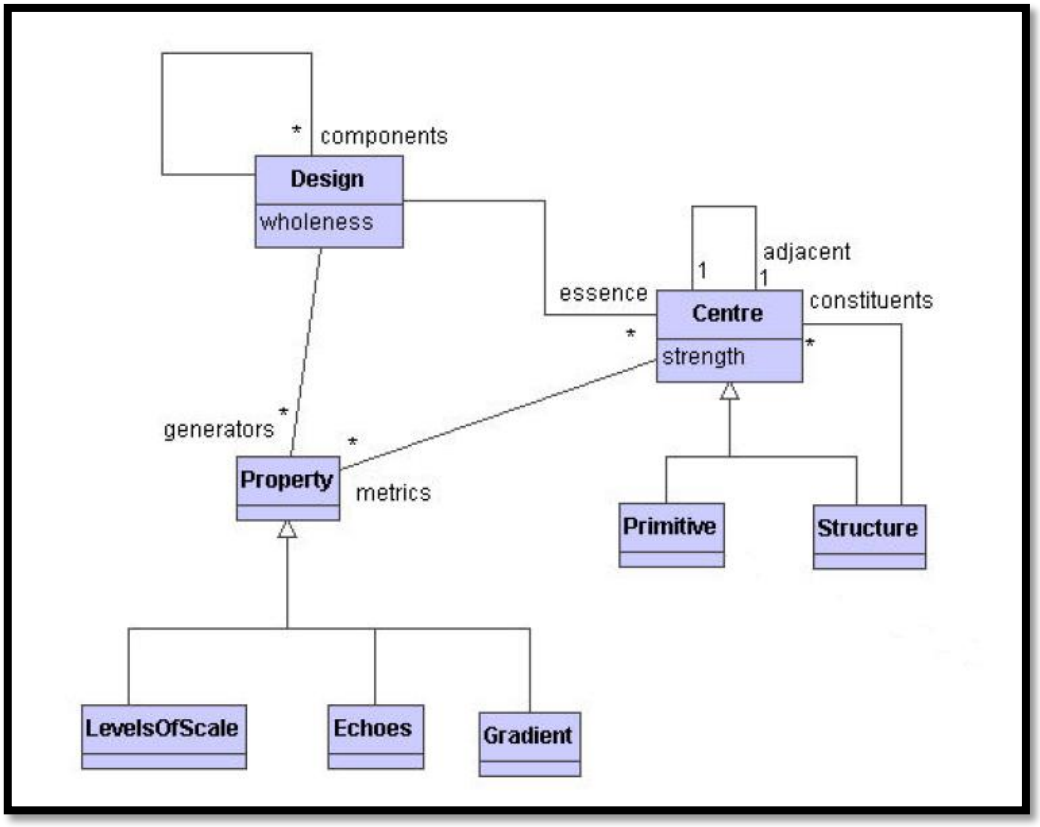

Figure 3- Meta-model of generative patterns [13]

Accordingly, this recent approach to generativity and generative patterns is very close to the complexity problem and complex systems because every emergence is actually the realization of an abstraction. As mentioned earlier, abstraction is the main mechanism of complexity management. This recent approach to generativity is related to i*, factorism, and adaptive systems through the concept of "complexity emergence". To the author, the concept of "emergence" equals the concept of "abstraction" and complexity management in terms of complexity.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

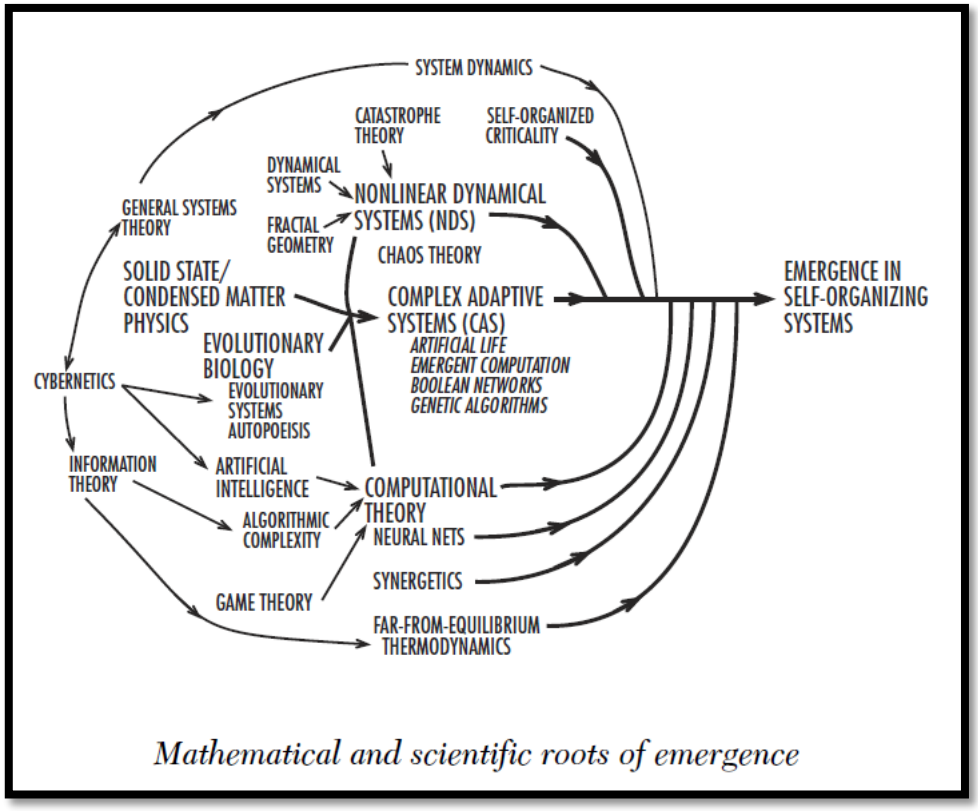

Image 2. Linking the concept of "emergence" with other prominent concepts of engineering and sciences, especially such concepts as complexity, fractals, self-adaptive systems, and self-organizing systems [11].

Complexity Management Patterns

This subsection introduces a few complexity management patterns. In fact, complexity management patterns exist at different levels of complexity. Some of them, *e.g.* abstraction, are very high-level and simple. Such patterns are more descriptive than generative. In other words, although they describe a solution, they give little information regarding how they act. However, other patterns, *e.g.* the basic components of BPMN, are totally tangible and generative (according to the above definition of generosity).

At the same time, patterns of higher levels are used in some other complexity management patterns. For instance, abstraction patterns are employed in all complexity management patterns; therefore, these patterns can be sorted relatively in terms of being primary and basic.

Therefore, the author decided to introduce patterns at several levels. The most basic, highest-level, and descriptive pattern is the abstraction pattern that exists at the first level. As the level number increases, the pattern loses the quality of being a high-level pattern and becomes more tangible. In addition, the pattern will not be a basic pattern anymore and will itself become a complex or compound pattern. As the level number increases, the pattern goes from being descriptive to being generative with respect to the meaning

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

presented above. Hence, as the patterns are traversed from top to bottom, they lose generality but gain expressivity.[11]

First-Level Patterns

Abstraction

Abstraction is the most basic mechanism for complexity management. With abstraction, a numerous set can be mapped onto a set of lower count, which is usually a single-member set. Therefore, according to the proposed definition of complexity, the complexity of the final set is lower than that of the first set. Many researchers consider abstraction the base for count devices, calculations, logical devices, and computer engineering and science.

The mechanism for the realization of abstraction depends on the patterns from the next levels of complexity management. The instances include abstraction by similarity, abstraction by generalization, abstraction by specialization, abstraction by selection, and abstraction by definition. These cases are analyzed in the patterns of next levels.

According to cognitive psychologists, humans have a specific numerical sense, which some consider it to be 4, whereas others consider it to be 7. They also think that the human mind does not consider the number equal to or smaller than the human numerical sense to be complex and abstracts them unconsciously to their numerical symbol without counting.

In a high-level view, any structuring framework, like what is also called “from mod to structure”, is considered a kind of abstraction. For instance, if a structure is considered for the concept of time (*e.g.* an order or sequence), an abstraction emerges.

The separation of concerns can also be categorized as abstraction because its main mechanism is to abstract the concerns in separate structures.

Second-Level Patterns

In fact, descriptive languages are certain mechanisms for complexity management, for they should organize the complexity of described objects. At the second level, the basic patterns of descriptive languages are presented. Furthermore, these two patterns are combined with abstraction to develop different types of networks, especially semantic networks (in which descriptive languages can also be considered a subclass in a certain view).

[11] A similar case lies in descriptive languages. A language is more general if it is more abstract and belongs to higher levels; however, it will have lower expressivity. Another similar case is observed in the comparison between descriptive and generative patterns. In fact, generative patterns have higher levels of expressivity but lower generality.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

Equality Pattern

When two objects are placed in an equivalence class and considered the same, an abstract view of their reference set is presented. The existing count of that set is decreased in that view. The equality pattern can be refined into "abstraction by specialization" pattern.

Sometimes, this pattern is applied after another abstraction is applied. In other words, an abstract image of an object is first created. It is then equalized to the image of another object. The complexity management of the Liskov substitution principle is an instance of this case, in which an abstraction of "abstraction by generalization" emerges first, and the equality pattern is then applied by analyzing the accuracy of is-a.

Membership (Assignment) Pattern

If an object is a member of another object (= set), the complexity management pattern is then employed to express this relationship. It should be noted that complexity decreases after an object is assigned to a set in an abstract view where the members of that set have been abstracted. As discussed earlier, every set is actually equals to an abstraction. Assigning an object to a set means assigning that object to an abstraction and introducing a method for abstracting that object. The membership pattern can be refined into an "abstraction by generalization" pattern.

Third-Level Patterns

Although some references have classified this pattern as the "divide and conquer" approach and considered it to be at the same level as the abstraction pattern, it can apparently be considered a refinement of the second-order pattern of "membership", which itself is a refinement of the first-level pattern of "abstraction".

However, the specific feature of the divide-and-conquer pattern should be taken into account. In most other patterns of complexity management, which are all based on abstraction, the identity of set members under complexity management will not be maintained after abstraction. However, the identity of components and members will be maintained after abstraction in the divide-and-conquer pattern.

In other words, the abstraction of the divide-and-conquer pattern, like any other abstractions, divides the space into an internal view and an external view. In most other patterns, the internal view and the external new are not considered at the same level of abstraction. In fact, either an internal or an external view is used at any level of abstraction. However, in the divide-and-conquer pattern, both the internal and external views are simultaneously taken into account. This relatively unique feature of the divide-and-conquer pattern has made some researchers consider it equivalent to the abstraction pattern and name "divide-and-conquer" and "abstraction" patterns the basic patterns of complexity management.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

Order

Order is among the most important patterns of complexity management and is usually applied to the results of the divide-and-conquer pattern. Other patterns such as enumeration and induction are the variants of this pattern.

In one aspect, programming is complexity management. A basic structure of procedural programming is the very order pattern. Another structure of the procedural programming is a loop. In fact, a loop is a compound pattern that divides the state space into variant and invariant sections. Thus, it as a divide-and-conquer pattern inside. It also includes the steps that are taken in a particular order; thus, it has the order pattern. At the same time, it exists in an abstract construct of a higher level — than the other ordinal structures existing in the program, *i.e.* commands — called the loop. Therefore, the loop can be considered a compound pattern of divide-and-conquer, order, and abstraction. The loop pattern is classified at next levels under the general patterns of process (basic components of BPMN).

Furthermore, there might be order but no divide-and-conquer, and vice versa. A specific state of divide-and-conquer which is not governed by an order is another case of compound patterns called the parallel pattern. It is classified at next levels under the general patterns of process (basic components of BPMN).

Selection

Also known as "abstraction by election" in the literature [19], selection is another pattern of complexity management. It is also among the major structures of the procedural programming. In this pattern, abstraction emerges by selecting a member and introducing that member as the representative of all members. Therefore, the identity of one member is maintained, whereas the other members lose identity during the abstraction mechanism and an abstraction view at a complex set.

Statistical sampling is a case of this complexity management pattern.

However, if a number of members selected when other members are not selected, the "abstraction by simplification" [19] or "abstraction by removing detail" [18] complexity management pattern emerges. It is another variant of the very selection pattern, for the remaining identities are not artificial in the abstraction image but have the original identities (= they have only been selected).

Aggregation

In this pattern, the individual identity of the complex set members will vanish through abstraction; however, all characteristics and features of members will remain in the novel abstract structure. For instance, the white light has all the physical characteristics of the seven lights in a rainbow; however, the individual identity of every light (red, yellow, *etc.*) is lost. This problem has decreased from the seventh-order complexity to the first-order complexity through the aggregation pattern. This pattern emerges in

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

the form of encapsulation principle in object-orientation. It is also seen as the form of synergy in the general theory of systems.

Generalization (Abstraction by Generalization)

Generalization is among the most widely used patterns of complexity management in software engineering [19]. Some researchers have only used authorized this type of abstraction in the object-oriented analysis and design of systems (Liskov's substitution principle) because it maintains the is-a principle.

In this pattern, the individual identify of set members will not be maintained in an abstract view of the set, and one abstract identity can only be seen. This abstract identity has the common features shared by the members of a complex set. Their uncommon features will not be seen in this abstract identity.

The concept of class and type represent the practical manifestation of this pattern in the contemporary object-oriented literature. It should be noted that abstraction by generalization differs from abstraction by elimination of details [18]. In the latter, the identity of the complex set members will be maintained in the abstract image. Some of them will be selected, whereas some others will not (this type of abstraction was classified under the selection pattern). In abstraction by generalization, although the common features and nature of the complex set members are seen in the abstract image, it is impossible to see any of the complex set members. In fact, the individual identity of the complex set members will not be maintained in the abstract image created by generalization.

Specialization (Abstraction by Allocation)

Abstraction by specialization resembles abstraction by selection. In fact, the latter is a refinement of abstraction by specialization. However, there is a prominent difference between these two patterns. In the selection pattern, the individual identity of the complex set members should be maintained through the selection process. In other words, the result of abstraction in the selection pattern is a member of the complex set. However, the result of abstraction does not need to be one of the members in abstraction by specialization. In fact, the final abstract structure might have a completely abstract identity. The result of abstraction includes a subset of the features shared by the complex set members. The example of this complexity management pattern is the famous *Elephant in the Dark* case. However, this complexity management pattern is not always considered inefficient, for it has efficiency in some cases.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

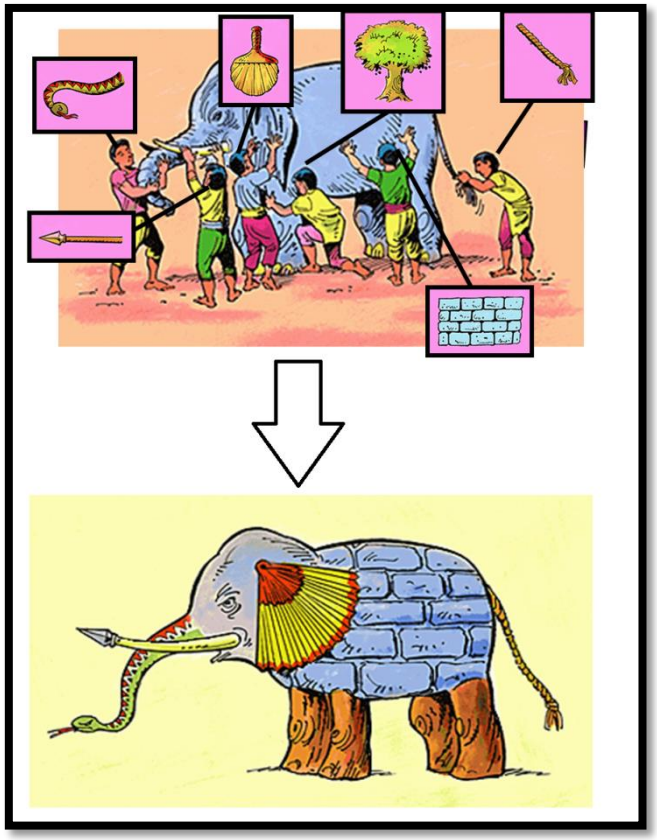

Figure 4- An example of complexity management through abstraction by specialization: when there is no selection, the result might have a completely abstract identity [1].

Fourth-Level Patterns

A feature of this level of patterns is that they can be applied to a context in which the concept of time has already been abstracted. In other words, time shown as an order or a sequence.

General Patterns of Process (BPMN Basic Components)

Although these components are the basic patterns of the commercial process, they are actually the general patterns of workflow and process. A task needs no process as long as it is not complex. In fact, a process comes into view when there are complex, proliferating, or multiple tasks at hand. Therefore, work process patterns are actually considered complexity management patterns. It should be noted that cases such as parallelism, integration, repetition, intersection, and occurrence emerge in a context of divide-and-conquer and abstraction patterns. In other words, work components (*i.e.* stages) are first determined by applying the divide-and-conquer pattern. After that, other cases of abstraction and complexity management are applied to them through BPMN components in the same way as defining an order on them, defining parallelism on them, defining repetition on them, *etc.*

It should be noted that time is also abstracted as an abstract structure in the first stage, something which helps define patterns such as emergence and intersection.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

CRUD Operations

Create, *Read*, *Update*, and *Delete* operations are the patterns that are signified after time is abstracted and applied to another abstracted structure such as data or information. The *Create* and *Delete* patterns can be considered the ordinal definition over the time in which data or information is monitored. The *Read* pattern can also be considered an abstraction. Moreover, the *Update* pattern can be considered an ordinal definition over the time in which two abstract structures (previous value and current value) are monitored.

GRASP Patterns

These patterns, which are used for delegating responsibilities to objects, can be considered in this class because they address complexity management and also use the patterns of both previous and current levels. For instance, CRUD patterns were used in GRASP, especially the *Creator* pattern.

Hierarchy

This very widely used pattern results from the application of the divide-and-conquer pattern, the order pattern, and then the divide-and-conquer pattern again. This procedure can be continued by applying the repetition pattern (which is among the patterns of general process, a type of divide-and-conquer, and then abstraction). In other words, there are a few levels at the highest view (= divide-and-conquer pattern), and an order is defined between levels (= order pattern). Every level can then be divided and given an order again.

Fifth-Level Patterns

Algorithms, Mathematical Theorems, Macros and Programming Basic Functions, Domain Objects, Process Segments, Circuits, Rules, *etc.*

These patterns are also the abstractions built by the major components of complexity management (patterns of the previous levels). However, they should also have the other attributes of patterns (*i.e.* correct and proven solution to an unrepeatable problem in their own area).

Software Architecture, Analysis, and Design Patterns, Refactoring Patterns

These patterns are also put in this class because they have been structured by the patterns of the previous levels (*e.g.* GRASP patterns). However, these patterns have different levels of abstraction. For instance, the layering pattern is a pattern of general application.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 March 2021 doi:10.20944/preprints202103.0651.v1

Sixth-Level Patterns

Programs, Processes, Development Methodologies, Frameworks, Architectures, Logics, Mathematical Theories, *etc.*

However, they should possess other attributes of patterns (*i.e.* accurate and proven solution to an unrepeatable problem in their area).

Seventh-Level Patterns

Complexity Management through Complexity (or Systems)

Operating in a context of complexity, systems are known as complex structures. Sometimes, a complex system (in terms of having numerous capabilities) is forced to perform complexity management instead of directly encountering complexity management. However, the complexity of a system should be equal to the order of complexity which it is going to manage. Some agile methodologies such as ASD can be classified as this class of complexity management patterns.

Programming Languages, Grammars, Logics, Mathematical Theories, *etc.*

The Complexity-Constructors are a sort of Complexity-Constructions, per se.

References

1. "Software Architecture Course Materials", *Sharif University of Technology*, 2012.
2. Tharumarajah, A., Wells, A. J., And Nemes, L., "Comparison of Emerging Manufacturing Concepts", *IEEE International Conference on Systems, Man, and Cybernetics,* 1998.
3. Alexander, C, "The Origins of Pattern Theory: The Future of the Theory, and the Generation of a Living World", *IEEE Software,* vol. 16, N. 5(September/October), 1999, pp. 71-82.
4. Becz, S.,"Design System for Managing Complexity in Aerospace Systems", *ATIO/ISSMO Conference, 2010.*
5. Bhagat, R., "Documenting & Using Cognitive Complexity Mitigation Strategies (CCMS) to Improve the Efficiency of Cross-Context User Transfers*",* Waterloo, Ontario, Canada: the University of Waterloo, 2011.
6. Coplien, J. O., "A Development Process Generative Pattern Language", *Pattern Languages of Programming Conference (PLoP'94), 1994*.
7. Lopes, C. V., Lieberherr, K., "Generative Patterns", in *Proceedings of the N^{th} European Conference on Object-Oriented Programming,* 1994.
8. Czarnecki, K., "Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models", Ilmenau, Germany: Technical University of Ilmenau, 1999.

9. Peterson, J. B., Flanders, J. L., "Complexity Management Theory: Motivation For Ideological Rigidity and Social Conflict", Department of Psychology, University of Toronto, 2002.

10. Vanderdonckt, J., Simarro, F. M., "Generative Pattern-Based Design of User Interfaces", in *Proceedings of the ١st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems,* pp. 12-19, 2010.

11. Goldstein, J., "Emergence as a Construct: History and Issues", *Emergence, A Journal of Complexity Issues in Organizations and Management,* vol. 1, no. 1, ١٩٩٩.

12. Grady Booch, R. A .(٢٠٠٧) .*Object-Oriented Analysis and Design with Application*.Third Edition.

13. Hoverd, T .(٢٠٠٨) .*Generative Patterns of Softwar: Qualifying Dissertation* .University of York.

14. Hwang, A. D .(٢٠٠٨) .*Calculus for Mathematicians,Computer Scientists,and Physicists: An Introduction to Abstract Mathematics*.

15. Jonathan M. Histon, R. J .(٢٠٠٨) .*Mitigating Complexity in Air Traffic Control: the role of structure-based abstractions* .MIT International Center for Air Transportation (ICAT)

16. Kent Beck, R. J .(١٩٩٤) .Patterns Generate Architectures٨ .*th European Conference for Object-Oriented. Programming* .(١٣٩–١٤٩ .ص) ,Bologna, Italy.

17. Kolmogorov, A .(١٩٦٣) .On tables of random numbers .*The Indian Journal of Statistics*.٣٧٥–٣٦٩ ,

18. Kramer, J .(٢٠٠٧) .Is abstraction the key to computing ?*Communications of the ACM*.٤٢-٣٦ ,

19. M. Wimmer, G. K .(٢٠١٠) .Surviving the Heterogeneity Jungle with Composite Mapping Operators .*Third international conference on Theory and practice of model transformations (ICMT 2010).*

20. M.V. Tatikonda, S. R .(٢٠٠٠) .Technology novelty, project complexity, and product development project execution success: a deeper look at task uncertainty in product innovation .*IEEE Transactions on Engineering Management*.٨٧-٧٤ ,(٤٧)

21. Makumbe, P. O .(٢٠٠٨) .*Globally distributed product development : role of complexity in the what, where and how* .Massachusetts Institute of Technology.

22. Mattsson, M .(١٩٩٦) .*Object-Oriented Frameworks: A survey of methodological issues* .Department of Computer Science, Lund University.

23. Mella, P .(٢٠٠٩) .*The Holonic Revolution Holons, Holarchies and Holonic Networks, The Ghost in the Production Machine* .Pavia: Pavia University Press.

24. Moses, J .(٢٠٠٤) .Foundational Issues in Engineering Systems: a framing paper .*MIT Engineering Systems Symposium* .Cambridge.

25. S. MacDonald, D .(٢٠٠٢)Generative Design Patterns ١٧ .*th IEEE International Conference on Automated Software Engineering (ASE 2002).*

26. *Francisco Montero Simarro, J. V. (2010). Generative Pattern-Based Design of User Interfaces. 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems . Berlin, Germany: ACM.*