Abstract

Writing patterns is a very important task for leveraging knowledge within an organization or in the software engineering community as a whole. Patterns are more than text, diagrams or source code. Patterns are knowledge that comes from experience. Sharing patterns is sharing knowledge. Currently, an abstract and standardized way for specifying software patterns doesn’t exist. There isn’t any single meta-definition model that governs patterns description. This leads to the problem of finding a way to represent patterns in a self-contained that leverages the literary information and the implementation information independent of the programming languages and execution platform and a way for sharing those patterns. The creation of a language for pattern meta-specification and a catalog of patterns from different pattern languages described using this language is a clear step towards the solution of this problem. The creation of a web-based visualization tool for the catalog makes this knowledge available to the world, allowing searching, linking and using the patterns in the catalog. In this paper, we will present the advances and results of our research work on meta-specification and cataloging of software patterns.

Keywords: patterns, Meta, meta-specification, classification, patterns browser, pattern oriented software engineering, domain specific languages, adaptive object model, knowledge management.

1. Introduction and Problem Statement

In the last years, the patterns community grew at a very fast pace. A lot of pattern languages and pattern systems emerged. Some of them are very popular, some of them are not. To show the tip of the iceberg, we propose a very simple, yet mind provoking, exercise: let’s sum the number of patterns in 3 reference and mainstream books, “Design Patterns” [GoF95], “Pattern Oriented Software Architecture, Volume 1” [POSA96] and “Patterns of Enterprise Application Architecture” [Fowler03]. The final number is 100… quite big to remember all! To worsen things, each pattern language is described using different formal elements (templates) and the samples are written with different programming languages. This produces a considerable impedance mismatch among all of them. There is not a single model to guide their definition. Such a guidance model may not be restrictive: the existence of a set of rules doesn’t imply fewer possibilities or more difficulties of representation. In fact, we can take as an example the C programming language: the same program can be written in almost infinite ways and in all cases will be a set of instructions understandable by someone or something, in this case, a computer.

We found some “calls to action” in several reference books in the patterns literature ([POSA96], [GoF95], [Berczuk94]) that made evident the need for a standardized way to describe, classify and store patterns and a simple way to use these descriptions. This makes patterns description and cataloging a first class problem that must be solved and hasn’t been successfully addressed yet.

The clearest use case and “call to action” for the development of a meta-specification language and a cataloging tool can be found in the last chapter of POSA Volume 1 [POSA96], where the authors refer to the future of patterns:

...In conclusion, using patterns successfully still requires the intellectual skills of the software developer. We believe that a well-designed pattern browser or World Wide Web tool can be much more efficient in helping a developer to find and use patterns that a fully integrated “pattern-supporting” software development environment ever could be. [POSA96].

Copyright © 2006 León Welicki. Permission is granted to copy for the EuroPLoP 2006 conference. All other rights reserved.
A more complete list of calls to action in some of the most important books of the patterns literature is presented at [Welicki05].

Sharing patterns is sharing knowledge. The lack of a single description, classification and sharing approach makes difficult the transmission of the knowledge that the pattern intents to share. As each catalog focuses on one particular point, its users only see the pattern in the catalog’s point of view. We have found that in the .NET world there is a very popular web site [DoFactory06] that has the intent, participants, structure diagram and implementations for the 23 GoF patterns. In lot of development forums when people refer to the pattern they are just referring to the implementations, but not to the pattern itself. They are confusing one concrete implementation with the real concept, with the knowledge that can allow a person to get to that implementation.

Some patterns description, cataloging and browsing approaches exist, but all of them fail in some aspects. In general terms, patterns are not fully described, are described much focused on implementation issues or are targeted only to a specific programming language and execution platform. In all cases is not possible to describe supporting entities such as categories, refactorings, object oriented concepts, sources of information, etc. with same syntactic and semantic constructs that are used for patterns description. But the description is only a part of the problem. All this knowledge is sparse, distributed through lots of sources of information. It would be a good thing to group this knowledge, creating a single point of access for it. Regarding the catalog and visualization, almost all public catalog browsers lack of a clear and easy to use navigational model that allows easily finding a pattern or focusing on a concrete aspect of a pattern. Additionally, each catalog visualization tool is focused on a single aspect (as we shown with the DoFactory example that is focused on the implementation). The catalog should not impose a single view to the user. Users need to decide in which view he wants to focus according to his needs on a particular moment.

All the exposed makes evident the need of a single way for describing, grouping and sharing patterns (and associated concepts) that helps to really understand the patterns, focusing on different concrete aspects of the pattern (literary information, diagrams, artifacts generation, etc.) according to the users need.

In this paper, we will explore the patterns description and cataloging problem and present our solution to it: the creation of a meta-specification language, a catalog of patterns and concepts described within that language and a viewer for the catalog. In the next chapter, we will go through the state of the art regarding our problem. In the next chapter, we will establish the objectives, audience and requirements for our research. Following we will present our solution, that consists in a meta-specification language called EML, a catalog of patterns and a viewer for the catalog (Patterns Browser). The proposed solution combines domain specific languages and adaptive object models to create an extensible an expressive model. In the next chapter, we will present some EML samples. Next, we will present some experiences with the system. Finally, we will share some conclusions and future work.

2. State of the Art

A lot of efforts for patterns description and cataloging have been made. This problem seems to be very important, since these efforts have been done equally by the academia and commercial software vendors.

Today most modern modeling tools [Borland04], [Rational05], [Sparx05] have pattern facilities that allow navigating a catalog of patterns and generating software artifacts out of them. In the same way, a lot of research approaches exists [BFYV96], [PLML03], [Hasso05], [WOP05], [Booch05]. In all cases we can find some drawbacks and limitations regarding our original problem.

In the last years, some web-based patterns catalogs emerged. All of them have the same problems: navigability, lack of homogeneity, incompleteness, passiveness (is not possible to generate software artifacts from them), rigidity, and lack of extensiveness among other problems.

In this section we will briefly analyze some modeling tools, some research approaches and some Web-based pattern catalogs showing how all of them lack an abstract and standardized way for specifying and navigating the patterns.
2.1 Patterns in commercial and research tools

Although for some people the idea may sound new, lots of efforts on this area had been done in the last years by modeling tools vendors, by the academic community, and by some research organizations.

In all cases, there is a clear lack of an abstract meta-model approach to guide patterns definition. In the following table, some commercial and research tools are briefly analyzed.

<table>
<thead>
<tr>
<th>Tool</th>
<th>How Patterns are represented…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rational XDE</td>
<td>Patterns are specified using a combination of XML, XSD and HTML files. For each programming language-specific pattern implementation a set of files must be created. The most important file is an XML document (generally its name coincides with the pattern name) and has all the necessary information to use the pattern within XDE.</td>
</tr>
<tr>
<td>Together</td>
<td>Patterns are specified using a set of XML files with a proprietary format. Description level is incomplete and for each programming language-specific implementation a set of files must be created.</td>
</tr>
<tr>
<td>Enterprise Architect</td>
<td>Patterns are specified in an XML file, using XMI. The pattern definition does not include full information of the description level nor algorithm implementations (implementation level). It only contains structural information.</td>
</tr>
<tr>
<td>IBM Research Prototype [BFVY96]</td>
<td>The description level of the patterns is represented as a set of HTML files, where each one represents a section of the pattern template (using the GoF template). Therefore, 13 HTML files are necessary to represent the literary level of a pattern. The relationship between them is only syntactic. The source code (implementation level) is described using COGENT (specially created for this project). It is not language independent (is necessary a template for each language) and is not related with the literary information (description level).</td>
</tr>
</tbody>
</table>

Table 1 – Analysis of some commercial and research tools with patterns support.

There is a common factor in all the tools: none of them has a unified and abstract way for representing patterns in the way proposed at the beginning of this work. In all cases, separate information exists for each target programming language and environment. Additionally, in almost all cases literary information (that helps to really understand the pattern) is not present or is not related to the implementation level. A detail analysis of each of these tools can be found in [Welicki05] and [Welicki04].

2.2 Some existing catalogs

It is unfair to say that no web-based patterns catalog exists today. In fact, there are many of them and some are very good. As an example, we can name PatternShare [Microsoft04], a Microsoft initiative led by Ward Cunningham.

In the last months Grady Booch is working towards the creation of a patterns catalog. At the moment of this writing he has catalogued more than 400 patterns [Booch05].

Linda Rising’s Patterns Almanac 2000 [Rising2000], a very comprehensive directory of all the patterns published as of year 2000 (unfortunately not actualized with current information). It is a very comprehensive catalog that shows the state of the art at the moment of its publication. The almanac is a directory that contains minimum information on each pattern (classification information and the intent). It intends to organize patterns to help users to find them. A web version of the almanac exists.

Other available catalogs are the Portland Pattern Repository, Sun's J2EE Blueprints, The Server Side Patterns, Enterprise Integration Patterns Catalog, UI Patterns and techniques, Implementing Finite State Machines, etc.

In all the tools mentioned above we have found occurrences of some of this problems: literary information (description level) is not present or is incomplete, patterns have some dependence on a particular platform, the
implementation level is not described in an abstract format, the navigability is not always very clear, search facilities are not always present or usable, and most important of all, there is no a single meta-definition model that rules the patterns description.

Another very important and subtle problem that is ubiquitous in all the catalogs is that it is not possible to have different views on the pattern according what we want to do with the pattern. A single view on the patterns is enforced (according to the interest of the tool’s authors).

### 2.3 Other approaches for patterns description and cataloging

There are several approaches that intend to solve the meta-specification problem, the cataloging problem or both of them. In this section we will enumerate some of them.

At CHI 2003, 15 participants gathered at a workshop to discuss perspectives on HCI patterns. A significant outcome of the CHI2003 workshop is the Pattern Language Markup Language (PLML) specification [PLML03]. The goal in deriving PLML was to bring order to the many (inconsistent) forms pattern authors have used [PLML03]. PLML helps to bring order to the pattern description, but has some drawbacks and limitations, such as a fixed set of elements, lack of description of the implementation level. There may be a gap between PLML document type definition (DTD [PLML03]) and the template used to describe certain patterns. For example, there are some of the sections of the GoF pattern template that are not included in the DTD. The same happened for POSA patterns or the ones included in PEAA [Fowler03]. Describing this patterns using PLML implies a considerable translation effort. Additionally, it does not include constructs to describe the implementation level in an abstract way. Finally, PLML is intended to describe patterns. It can’t express supporting concepts such as object oriented principles, pattern types, pattern languages, etc.

The Web of Patterns Project is aimed to represent software design using semantic web technology, in particular RDF and OWL [WOP05]. It intends to represent patterns, anti-patterns and refactorings using the concepts developed in the ontology. Its objectives include the creation of the ontology and a set of tools and sharing them with the software engineering community.

It consists of three parts [WOP05]:

- The Object-Oriented Software Design Ontology ODOL
- Pattern descriptions and catalogues based on ODOL.
- The WOP Pattern Scanner, a tool that finds design patterns in Java programs.

This approach does not include the description level (literary information). It is more focused on implementation issues. Additionally, the creation of the ontology is a complex process that in some cases requires a bottom-up approach. At the moment of this writing, there is an Eclipse plugin that allows navigating the 14 supported. [WOP05]

Hasso and Carlson, from the Illinois Institute of Technology, proposed at PLoP 2005 a model for pattern classification based on linguistics theory and complex algebraic structures [HC05]. The model is very powerful, although is complex to put in practice. It takes a lot of analysis work to classify a group of patterns. Another problem is the difficulty of the evolution of the stated classification.

### 3. Objectives, target audience and proposal

#### 3.1 Objectives

Writing patterns is a very important task for leveraging knowledge within an organization or in the software engineering community as a whole. Patterns are more than source code and diagrams. We want to create a set of technologies that allow people to know, understand and use the patterns. A set of tools and technologies like the ones presented in this paper could simplify the pattern mining and writing tasks, enhancing the efficiency in knowledge sharing.
These technologies must support the description of patterns and related concepts (such as pattern languages, categories, concepts, authors, etc.), the cataloging, relations between concepts, visualization and searching, leveraging the process of using, hunting navigating and sharing software patterns.

The objectives of our research are:

- Represent patterns (and related concepts) in a format independent of the programming languages and execution platform. This representation must be self-contained, leveraging the description and implementation levels.
- The creation of a catalog of patterns (and supporting concepts, for example, pattern languages, categories, authors, etc.) represented with that format.
- The creation of a browsing tool to navigate, manipulate and use that catalog. The tool must be easily accessible for a broad audience.

Achieving these objectives will allow us to:

- Share knowledge about patterns and the software engineering concepts on which they are founded.
- Support empirical knowledge management within the software engineering community through a set of technologies and tools.
- Gather and formalize sparse knowledge.

### 3.2 Intended Audience

The technologies and tools presented in this paper are targeted to a very broad audience. They are targeted mainly for everyone that has interest in any area of software development. Since there are patterns for very diverse disciplines regarding software construction (this includes engineering, architecture, design, etc.), the audience for each kind of patterns may be different. As an example, we can quote that the audience for interaction design patterns may not be the same that the audience for design patterns or the audience for anti-patterns (although of course, it could be the same).

In the following paragraphs, we will present some of the groups with their expected interest and usage of the tools.

- **Software Developers:** the technologies and tools presented in this paper are of strong interest to everyone involved in software development. This group is not limited to programmers and designers: there are patterns that can assist managers and graphic designers, to quote some groups of interest. In this group of users, we found two main sub-groups:
  
  - **Novice Software Developers:** this group can leverage the knowledge of the experts. Due that our approach is to cover all the information necessary to really understand the patterns, we can give to this group a valuable source of information that can be used in their daily activities.
  
  - **Experienced Software Developers:** this group has two functions in the whole model. The first one is to share their experience with the novices. Even if the members of this group don’t publish patterns, they can help others through the forums or helping in the collaborative cataloging effort. Additionally, being an expert does not mean having all the patterns in you memory, but knowing which patterns exists, what are they for and where to find them. The tools presented here can be of great help for these experts, being the repository where they can find the patterns, giving them a tool that allows finding easily the patterns.
- **Patterns Authors**: authors can share their patterns within the technologies proposed in this paper. They can describe their patterns using a meta-specification language and register them in the catalog (the meta-specification language should be flexible enough to allow to use any one of the existing templates for describing patterns). This gives them all the advantages inherent to the proposed model that are going to be presented later in this paper. Another possible use for pattern authors is to assist them in the process of finding the structure of a pattern language [Schuemmer03]. This could be achieved modifying the classification metadata and looking at the results, taking an iterative and incremental approach to accomplish this task. The idea of including this group came from the feedback obtained by the authors in PLoP 2005.

- **Teachers and Students**: teachers could use this technology to teach patterns to students. Since the tools presented here allow focusing in a very specific aspect of a pattern, they are very good for abstracting different aspects of the patterns. Students can examine the pattern from different point of view, according their concrete interest.

### 3.3 Our Proposed Plan

To solve the meta-specification, classification and sharing problem, we propose the following steps:

- Create a meta-specification language for describing the patterns. This meta-specification language must be able to represent the description and implementation levels. The definition of the implementation level must be independent of any programming language and implementation platform.

- Create a catalog of patterns (and supporting concepts) described with the language created in the previous step.

- Create a tool to browse a catalog of patterns described with the language mentioned above. Ideally, this tool must be based on Web technologies, to make it accessible to a broad audience through a web browser (without the need of any additional software). Additionally, it may expose some of its functionalities as Web services which may allow using the catalog in different ways (for example, it could be invoked from development IDEs like Eclipse, Visual Studio .NET, etc.). It must allow users to input their feedback and discuss the concepts from the catalog, in order to make the catalog “a living space”

#### 3.3.1 Patterns Definition Used

It is very difficult to establish a definition of the term “pattern”. There are several slightly different definitions in the reference books that exist in this field. In order to create our meta-specification language based on solid foundations, having a clear definition of this term is mandatory.

For the purpose of this research work, we will define a pattern as a “piece of knowledge that includes information about a problem and its solution in a specific context, with the trade-offs and all the literary information needed to have a good understanding of the issues related with it. Eventually, it may contain implementation specific information, which will allow generating the source code for the proposed solution to the problem”.

### 3.3.2 Requirements for the components of the proposed plan

In this section, we will briefly establish the requirements for each part of our solution. A broader discussion on the requirements can be found at [Welicki05] and [Welicki04]

- **Requirements for the meta-specification language**: The main requirement for the solution is to establish a uniform and standardized way to represent patterns and supporting concepts. This pattern definition language must provide appropriate syntactic and semantic constructs to represent the description and implementation levels in a homogeneous way. The description level must support all the existing pattern templates [C2PatternForm]. Regarding the implementation level, there shouldn’t be special clauses to describe platform or language specific semantics. The behavior of the participants of the patterns must be described as abstract as possible.
• **Requirements for the patterns catalog:** The patterns catalog gathers the meta-specifications of the patterns created using the language presented in the previous section. It must support easily adding, removing, modifying and linking patterns. It must support bottom-up editing approach (it is not necessary to have full information of an entity to introduce it in the catalog, it can be completed later). Additionally, it must contain specifications of supporting concepts that may help to get a better understanding of the pattern.

• **Requirements for the Catalog Browser:** The catalog browser is the component that shows the contents of the catalog and allows the user browsing and using the patterns. It must be easily accessible and easy to use. It must provide an easy to use façade on the catalog, allowing the users to find, navigate and learn the patterns and concepts stored in it. It must include feedback mechanisms to allow users to discuss and enhance the contents in the catalog, in order to make it a “living information space”.

### 3.4 What we are NOT proposing

The set of tools and technologies that we propose are not automatic: they require the intellect of people. They support patterns usage or a pattern-oriented process, helping users to find and use patterns and authors to describe their patterns and concepts but they are not automatic tools. They neither do automatic pattern mining nor automatic pattern application. In all cases, the intervention of a human user is mandatory.

It is neither oriented to assist non-software people for creating software. It does not fulfill Alexander objective that a user can design his house [Alexander79]

Finally, we are not proposing a theoretical model. All what is presented in this paper has been implemented in a research prototype. Every single idea is proved empirically as soon as possible, in order to have a full implementation of our complete proposal.

### 4. Solution to the Stated Problem

Through our research, we found a solution for the meta-specification and cataloging problem. We built a research prototype composed of the definition of a patterns meta-specification language, a catalog of patterns and concepts described using that language and a web-based patterns browser.

The elements that shape this solution (based on the approach proposed previously in 3.5) are the ones as follows:

• **EML (Entity Meta-specification Language):** EML is a pattern meta-specification language based on XML. It allows defining a pattern (and supporting concepts) in single self-contained file. It includes language constructs that allow the abstract representation of the description and implementation levels. It also allows describing the relationships between entities. The implementation level is described with total independence of the target language and platform.

• **Patterns Catalog:** is a catalog of representative patterns, written using EML. It also contains supporting concepts described used EML (pattern languages, sources of information, types of patterns, object orientation concepts, etc.)

• **Catalog Browser:** is a visualization and navigation tool for the catalog. It supports multiple views on a pattern. It also exposes the catalog contents through a SOA interface (based on SOAP Web services).

In the remaining part of this section we will explore each one of the building blocks that conforms the solution. We will mainly put our focus on the EML language. We will also show the Adaptive Object Model that acts as its execution platform. As we will see, Domain Specific Models and Adaptive Object Models are central concepts in our solution architecture.
4.1 EML – The Meta-Specification Language

EML is the acronym for Entity Meta-specification Language. EML is a modularly composable domain specific language [Czarnecki00] created with the purpose of describing all kind of software patterns and supporting concepts. The entities described with EML must have literary information (description level) and may also have behavior information (implementation level). EML is based on XML.

EML includes the entire necessary infrastructure to describe the description and implementation levels and the relationship among patterns and supporting concepts. In its first version [Welicki05] it was only enabled to describe patterns using rigid previously defined templates. In the current version it can describe any kind of entity using dynamic templates. It can also describe the relationships among entities of the same or different types (for instance, it may express that “a pattern is contained in a pattern language”, “a pattern conform an object oriented principle”, etc.)

For the description of the implementation level, a Domain Specific Language (DSL) containing the representation of abstract constructions of object oriented programming languages has been created. The full definition of this subset of the language can be found at [Welicki04].

4.1.1 EML is a Domain Specific Languages

A domain specific language is a specialized, problem-oriented language [Czarnecki00]. The basic idea of a DSL is a computer language that's targeted to a particular kind of problem, rather than a general purpose language that's aimed at any kind of software problem [Fowler04]. A DSL may be textual or graphical. They may also have different levels of specialization. In general you will need several DSLs to specify a complete application. [Czarnecki00]

EML is a high-level modularly composable [Czarnecki00] DSL that includes 4 narrower domain specific languages:

- **Relationship definition language**: its main purpose is the definition of the relationship among entities.
- **Properties definition language**: its purpose is defining the literary information of an entity. It can be dynamically defined. It is interpreted by an adaptive object model.
- **Structure definition language**: describes the structure of the entity (participants and their relationships)
- **Implementation definition language**: describes a very high level and abstract definition of the behavior of each participant. Contains abstractions of constructs of programming languages.

Each one of these languages has a very well defined purpose and very responsibility definition. The result of the interaction among them is a framework for describing patterns and supporting concepts.

4.1.2 Specifying entities using EML

Patterns do not live in isolation. In the same way that an implementation is not enough for understanding a pattern, a pattern itself is not enough without any supporting concept.

In order to cope with this situation we defined a low level ontology to formalize the domain of the patterns definition. What is ontology for the purpose of our work? We are using the definition given in [Gruber], that says that “an ontology is a specification of a conceptualization” and that “an ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents”.

It is very important to say that the objective of the ontology is enabling knowledge share and reuse [Gruber], which are also the main objectives of our model.
In the proposed structure (shown in the next figure) the pattern is the central concept, and related to it there are several other concepts that are related among them as well.

As we previously mentioned, EML contains a DSL for establishing these relationships. Why do we choose describing this domain and its relationships with EML and not used OWL instead? Because EML based relationships can be created, established and modified in runtime in an emergent way, without the need of any high level coordinator or high level design.

The question, then, is what this model is for. The answer is very simple: we used it as a starting point for creating the catalog. As we are expecting the catalog to be a “live” entity, it surely may vary as new patterns and concepts start to appear. We want to create a classification system founded on the community knowledge.

Additionally, we want to establish an “emergent feedback system”, that allow all the users to “tune” in a cooperative way this relationships. We are currently working on this issue, with the goal of having an implementation of this technique soon.

### 4.1.3 Sections of an EML Entity

EML allows representing an entity in a single file. This representation could be used to view the literary information, view CRC cards from the pattern or generate code in any language (in the current version, C#, Java and VB.NET code generators are provided).

An EML pattern representation is self-contained: this means that it contains all the literary information needed to understand the pattern, all the implementation information to use the pattern (to allow code generation from it or using it in a case tool) and metadata to assist the search engine and the catalog browser.

EML is not limited to pattern description. EML can describe any entity.

An EML definition consists of the following sections:

- **Tags**: includes tags for annotating the entities to enhance searching the entity being described.
- **Relationships**: relationship between the pattern and other entities ("Abstract Factory is contained in the GoF pattern language", "Abstract factory is a Design Pattern", "Abstract Factory conforms the Dependency Inversion Principle", etc.)

- **Summary**: a summary about the entity being described.

- **Template**: literary information of the pattern.

- **Structure**: structure of the pattern (participants with their relationships and responsibilities).

- **Implementation**: implementation information of the pattern (implementation level).

As we can see on figure 3, the description level consists of the Tags, Relationships, Summary, Template and Structure sections. The implementation level consists of the Structure and Implementation sections. As we’ve seen in the figure 3 above, Structure section is shared by the knowledge and the implementation levels, because it contains a formal enumeration of the participants and their responsibilities (implementation level) and textual information about their responsibilities (description level).

It is important to notice that the Structure and Implementation sections may be optional. Some patterns may not need them. For example, the patterns at “A UML Pattern Language” [Evitts99] may not have an implementation section. The Anti Patterns [BMMM98] may have both structure and implementation as well. The design patterns from GoF [GoF95] have both knowledge and implementation sections. As a conclusion, we can say that the description level is always mandatory and the implementation and structure are optional.

### 4.1.4 Describing the Description Level

Describing the description level is a very complex task. A big number of pattern templates exist and creating a new template that summarizes all of them is not a trivial task. In order to create a template for the description level, we
analyzed the most popular ones (including the GoF Template, Analysis Patterns Template, POSA, Patterns of Enterprise Application Architecture, Alexander Form, Canonical Form, etc.).

In the current version of our research prototype we created a template definition system that allows combinations of elements of any pattern form [C2PatternForm]. This flexible template approach is based on the dynamic composition of simple atomic units of information (called “properties”).

4.1.5 Describing the Implementation Level

EML contains a set of constructs that allows describing the behavior of a set of software artifacts without using any particular programming language and without any platform compromise.

- Artifact description: describing code modules (e.g. classes) and its elements
- Algorithm description: describing the behavior of a code module function (e.g. methods)

EML can be translated to any high-level programming language. In the first version of the solution prototype, 3 translators are provided: C#, VB.NET and Java.

A full definition of the implementation level constructs can be found in [Welicki04].

4.1.6 An Adaptive Object Model as the Execution Platform of EML

An Adaptive Object-Model (AOM) is a system that represents classes, attributes, relationships, and behavior as metadata. The system is a model based on instances rather than classes. Users change the metadata (object model) to reflect changes in the domain. These changes modify the system’s behavior. In other word, the system stores its Object-Model in a database and interprets it. Consequently, the object model is adaptable; when the descriptive information is modified, the system immediately reflects those changes [YJ01].

EML provides the entire necessary infrastructure to describe patterns, supporting concepts and relationships among them. EML can be consumed by any client. In our model, the execution platform for EML descriptions is an AOM. It is there where EML descriptions go live. The platform is based on a highly dynamic and adaptive object model. It can bee configured, adjusted and changed at runtime, based on EML definitions.

The next figure shows the structural view of the AOM for the properties and relationships of the entities. It also contains some objects devoted to UI, specified at the visualization level.

![Figure 3 – Adaptive Object Model class diagram](image-url)
Our adaptive object model is mainly based on the “Type Object Pattern” and is an adaptation of the “Type Square” to our problem domain. It contains different types of property object aimed to represent the different elements that may be part of a template.

To manage the relationship between entities we used the Accountability pattern [Fowler97]. To parse and build the run-time instances we use a combination of Builder, Interpreter [GoF95] and Registry on a plug-in architecture.

In the adaptive object model presented in the figure 3, three levels are specified:

1. **Knowledge**: records the general rules that govern the structure [Fowler97]
2. **Operational**: records the structure of the domain (in our case the structure is composed of patterns, properties of the template, etc.) [Fowler97]
3. **Visualization**: contains instructions on how to show the elements in the model. This instructions can be attached and composed at runtime (from configuration information)

### 4.1.7 Semantic Gradient

Authors may not have all the information at the moment of describing patterns or entities and relationships among them with EML. Our approach does not force them to have all the information in advance in order to create the specification. Instead, we created what we call the “semantic gradient”.

Any concept has a life cycle that may go through the semantic gradient. This gradient has three steps:

1. **Word**: A concept is a single word in the patterns description. There is no semantics associated to the word. It can be used in a full text search.
2. **Tag**: a concept may be a tag. It does not have a specification, but can be used for tag based searches.
3. **Entity**: a concept is specified as a full EML entity. It has full meaning and can be addressed as any other EML entity.

EML does not enforce to specify entities up-front. Entities can be discovered through the description process. Let’s make a quick sample to explain the concept. While specifying the Abstract Factory and Factory Method patterns [GoF95], the factory word is present in both. Making a full text search using the keyword “factory” would give as a result both patterns (among others that contain this keyword in its definition). We may find that the factory concept is more than a word, but we do not have a definition for it. In that case, we can create a “factory” tag and attach it to both patterns. When we make a tag based search, we will get only the patterns that are annotated with that tag. Finally, after some time we may found more information on the factory and we may decide to promote it to a full EML entity. In this case we will create the “Factory” EML entity and relate it with both patterns.

### 4.2 The Patterns Catalog

The patterns catalog has been totally built using EML. In its first version, it includes the patterns from the GoF book, “Evolving Frameworks” [RJ98] and some from GRASP [Larman99].

The catalog is a set of files, where each one contains the definition of a pattern in EML (each EML file represents a pattern in a totally self-contained fashion). There are other EML files that have information on the categories, pattern languages, authors, sources of information, principles, etc.

Additionally, there is a registry where all the patterns are “registered” in the catalog. This means that is not sufficient to write a EML pattern file on disk. It must be also “registered” in the patterns registry.
The registry is a combination of a relational database (where data about the registered entities and relationships live) and a file-system based vault where the actual EML definitions live.

The catalog is maybe the most complex and important part of our prototype.

### 4.3 The Patterns Browser

The patterns browser is a web-based tool that allows browsing through the contents of the catalog. The objective of the pattern browser prototype is to provide a viewing application for the EML patterns catalog. The pattern browser allows navigating through the pattern languages registered within the catalog.

![Patterns Browser](image)

**Figure 4 – Complete view for the Abstract Factory pattern in the Patterns Browser.**

A high level architectural view and a further discussion on the browser can be found at [Welicki05].

#### 4.3.1 Views on a pattern

A common problem in almost every public patterns catalog viewing application (including all the catalogs mentioned previously on section 2.2) is that the user interface forces a single view on the information to the end-user. Often this view is directly related with the interests and objectives of the publisher of the information. We think that this is a very undesirable situation and that users must freely decide on which aspect of the pattern they want to focus. To solve this situation, we propose applying the classic separation between information and representation. The information is an EML meta-specification. The representation is encoded in a view that can be applied to a meta-specification. A view is a component that makes a transformation, taking an EML meta-specification as input and rendering UI code for a specific aspect of the entity as output.

For each pattern, several views may be available. Therefore, taking a pattern as the model (in a MVC like architecture), multiple views can be applied to it (as shown in figure 4). Moreover, the views can be dynamically attached to a pattern, allowing changing the visualization style for different patterns based on arbitrary criteria (using a plug-in architecture [Fowler03]. In the current version, there several views are provided for a pattern:

- **Complete View**: shows all information on a pattern (taken from the template)
- **Summary View**: a summary of the description level of the pattern
- **CRC View**: shows CRC cards for the participants of the pattern
- **Source Code View**: shows the source code for the pattern. Several implementation options can be chosen.
- **EML View**: shows the EML code for the pattern
Figure 5 – Multiple views on a pattern. New views can be attached to any pattern in the catalog. The multiple view system allows users to focus on a single aspect of a pattern according to their interest.

5. Experiences

In order to empirically prove our approach, we built the set of technologies and tools that we presented through this paper. To try them with real information, we specified the full GoF [GoF95] patterns catalog. We did not only specified the patterns but also the pattern language, the categories, the authors, the books, the refactorings and the object oriented principles on with patterns are founded. We also specified Larman’s GRASP pattern language [Larman99], and linked elements from the GoF pattern system with elements in the GRASP pattern language, making possible relation among elements in different pattern languages. To make the relationships richer, the related elements do not need to be patterns. We used EML to describe the following type of entities:

- Patterns
- Pattern Languages
- Categories
- Authors
- Books
- Object Orientation Concepts
- Abstraction Levels

As we said previously before, it is possible to create relationships between any of these concepts. In a relationship, concepts don’t have to be necessarily of the same type. For example, we related the GRASP Creator pattern with the “Creational” category of the GoF pattern system. We also related each pattern in GoF Creational category with the “Creator” pattern. The result was a very good combination of concepts in patterns at different level of abstraction (as shown in figure 5). Understanding the creator can help to understand the GoF Creational category and later on, each creational pattern.
We created several translators for the implementation level, allowing the code generation for the pattern in C#, VB.NET and Java. We also created a CRC Card creator that renders CRC card from the patterns specification. All this elements are views on the patterns. New views can be created on runtime, making possible focusing on different aspects of the pattern according users needs.

6. Conclusion: Benefits and Contributions

The creation of a meta-specification language for pattern specification and a catalog of patterns from different pattern languages described with this language (using the same syntactic and semantic constructs) is a clear step towards the solution of the stated problem. The creation of a web-based catalog visualization tool makes this knowledge available to the world, allowing searching, linking and using the patterns in the catalog.

The combination of Domain Specific Languages and Adaptive Object Model as presented in this work converge in the creation of a very expressive language and a flexible runtime platform. The entities expressed within this language form a web of concepts that can be dynamically evolved and extended founded on community knowledge.

Following, the main benefits and contributions of this research are listed:

- A meta-specification language to describe patterns and supporting at an abstract level
  - Patterns from different pattern languages can be described using the same semantic and syntactic elements
  - The concepts needed to correctly understand and/or classify a pattern (pattern language, category, principles, authors, etc.) can be specified using the same semantic and syntactic elements
  - Literary information can be dynamically extended and modified.
  - Behavior description is platform and programming-language independent
  - Classification and description can evolve easily

- A catalog of patterns, to establish the foundations of practical knowledge gathered from previous experiences
• Unique source of knowledge on diverse pattern languages and pattern systems
  • All patterns, pattern languages, categories, etc. are described using the same semantic and syntactic rules
  • Search facilities
  • Relational facilities. Possibility to link entities in different contexts
  • Built-in support for growing and evolution

  • A Web-enabled catalog browsing tool
    • Point of access to a unified repository of knowledge
    • Provide access to the catalog through a simple, accessible and easy to use.
    • Low cost of distribution
    • Multiples views on patterns
    • Allow discussing about patterns an concepts, making it “a living space”

7. Work in Progress and Future Work

As we stated previously, our research work is currently ongoing. We are still working enhancing the EML itself and our prototype of the catalog.

At the moment of this writing, we are planning or working on the following enhancements and extensions:

  ▪ **Classification**: we are currently working on several dynamic classification techniques, based on metadata, to enhance the catalog’s navigability

  ▪ **SOA interface**: in the first version of the prototype, we created a “proof of concept” SOA interface for pattern languages. That interface is not acceptable in a production system. Therefore, we are working on a better interface, so we can make this information accessible to any client.

  ▪ **Improve EML expressiveness for implementation definition**: EML can be further improved to be more expressive. This would imply supporting more data types, more programming statements, etc.

  ▪ **Search**: provide a better search UI, to allow users make advanced queries using the relationships between entities. Enhance the search engine to allow making richer queries to the catalog. Advanced search with inference facilities, in order to have a full-functional implementation of the “semantic gradient”.

  ▪ **EML validation**: finish the XSD (XML Schema) to validate EML pattern definitions and to formalize its usage.

  ▪ **Hyperbolic Tree based navigation**: use hyperbolic trees to navigate through the related entities stored in the catalog. The tree should be created dynamically using the entities and their relationships.

  ▪ **EML editor**: Create an editing tool for EML.

Even though the prototype is ambitious and implements advanced properties it doesn’t implement the whole model. It is a conceptual prototype that implements a great number of all the requisites established previously in [Welick05]. The main objective of the prototype was to prove and validate empirically the feasibility of the proposed model. We are currently working on enhancements in order to cope with all the requirements of the conceptual model.
Appendix A - EML Sample Cases

A.1 Describing a pattern using EML: Case Study

In this section we will show briefly how to describe patterns using EML. A full EML description of a pattern is very long. Therefore, we selected some meaningful examples of each of the sections of the pattern’s definition, so the reader can have an accurate idea of how an EML definition looks like.

A.1.1 Describing the entity identification

```xml
<entity
type="pattern"
id="GoF.Creational.AbstractFactory"
name="Abstract Factory"/>
```

Snippet 1 – First line of the definition of an entity. In this case, it is the first line of the definition of the abstract factory pattern.

A.1.2 Adding the Tags

Patterns are annotated with tags, which can be used for searching. The following fragment is a partial view of the tags for the Abstract Factory [GoF95] pattern:

```xml
<tags>
  <tag>Factory</tag>
  <tag>Object Creation</tag>
  <tag>Families</tag>
  <tag>Object Family</tag>
  <tag>GoF Pattern</tag>
  ...
</tags>
```

Snippet 2 – “Tags” section in Singleton’s EML pattern definition.

A.1.3 Establishing relationships

```xml
<context>
  <relationship
targetType="Source"
targetID="GoFBook"
type="isPublishedIn"/>
  <relationship
targetType="PatternLanguage"
targetID="GoF"
type="isContainedIn"/>
  <relationship
targetType="Category"
targetID="GoF.Creational"
type="isContainedIn"/>
  <relationship
targetType="PatternType"
targetID="Design Pattern"
type="isA"/>
  <relationship
targetType="OOPrinciple"
targetID="DIP"
type="conforms"/>
  ...
</context>
```

Snippet 3 – Definition of the relationships of a pattern.
A.1.4 Summary of the entity

The summary is a short phrase the represents the entity that is being described. In the following snippet, the summary section for the abstract factory pattern is shown.

```
<summary>Provide an interface for creating families of related or dependent objects without specifying their concrete classes</summary>
```

Snippet 4 – Summary of a pattern

A.1.5 Describing the template

The template contains the literary information for the entity being described.

```
<template>
  <intent>Provide an interface for creating families of related or dependent objects without specifying their concrete classes</intent>
  <problem>Families of related objects need to be instantiated</problem>
  <solution>Coordinate the creation of family of objects. Give a way to take the rules of how to perform the instantiation out of the client object that is using these created objects.</solution>
  <applicability>
    <useWhen>A system should be independent of how its products are created, composed, and represented.</useWhen>
    <useWhen>A system should be configured with one of multiple families of products.</useWhen>
    <useWhen>A family of related product objects is designed to be used together, and you need to enforce this constraint.</useWhen>
    <useWhen>You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.</useWhen>
  </applicability>
  ... Rest of the template section elided for simplicity
</template>
```

Snippet 5 – Describing the template of a pattern. A template is composed of a set of properties.

The template is composed of properties. Each node in the XML of the template definition is a property. There are some well known properties that the system knows how to handle.

A.1.5.1 Configuring the automatic property parsing

These well-known properties are described in metadata and are dynamically configurable. The following snippet shows a fragment of the configuration of the parser in order to handle the well-known properties.

```
<wellKnownProperties>
  <properties>
    <property
      id="consequences"
    <property
      id="relatedPatterns"
  </properties>
</wellKnownProperties>
```

Snippet 6 – Well-known properties for the template parser.

A.1.6 Describing the structure

The structure section contains the structural information of the pattern. It is divided in three parts:
- **Participants**: lists the participants of the pattern
- **Relationships**: relationships between the participants. The relationship types are inheritance, composition, aggregation, association and creation.
- **Responsibilities**: textual definition of the responsibilities of each participant

The next snippet shows the structure section for the Abstract Factory pattern:

```xml
<structure>
  <participants>
    <participant role="AbstractFactory" isAbstract="true" cardinality="1"/>
    <participant role="ConcreteFactory" isAbstract="false" cardinality="*"/>
    <participant role="AbstractProduct" isAbstract="false" cardinality="*"/>
    <participant role="ConcreteProduct" isAbstract="false" cardinality="*"/>
    <participant role="Client" isAbstract="false" cardinality="1"/>
  </participants>

  <relationships>
    <relationship type="inheritance"
      child="ConcreteFactory" parent="AbstractFactory"/>
    <relationship type="inheritance"
      child="ConcreteProduct" parent="AbstractProduct"/>
    <relationship type="creation"
      source="ConcreteFactory" target="ConcreteProduct"
      cardinality="1"/>
    <relationship type="association"
      source="Client" target="AbstractFactory"
      cardinality="1"/>
  </relationships>

  <responsibilities>
    <role name="AbstractFactory">
      <responsibility>Declares an interface for operations that create abstract product objects.</responsibility>
    </role>
    <role name="ConcreteFactory">
      <responsibility>Declares an interface for operations that create abstract product objects.</responsibility>
    </role>
    <role name="AbstractProduct">
      <responsibility>Declares an interface for operations that create abstract product objects.</responsibility>
    </role>
    <role name="ConcreteProduct">
      <responsibility>Implements the AbstractProduct interface.</responsibility>
    </role>
    <role name="Client">
      <responsibility>Uses only interfaces declared by AbstractFactory and AbstractProduct classes.</responsibility>
    </role>
  </responsibilities>
</structure>
```

Snippet 7 – “Structure” section in Abstract Factory’s EML pattern definition.

This structure makes it very easy to port patterns to EML because it allows focusing on each individual aspect of the structure separately. The user first identifies the participants and then establishes the relationships between them. Finally, he adds the responsibilities to each role.

### A.1.7 Describing the implementation level

As we stated previously, EML has a DSL to represent source code constructs at a high level of abstraction. Patterns that specify an implementation must have a “baseImplementation” section, which describes each participant and its basic implementation. A participant description contains three sections for describing its implementation: Properties, Constructors and Methods.
In the next snippet we will show how to define the base implementation section for the Singleton pattern:

```xml
<baseImplementation>
  <participant role="Singleton">
    <properties>
      <member scope="private,protected" name="instance"
             type="Singleton" isClass="true"/>
    </properties>
    <constructors>
      <signature scope="private,protected"/>
    </constructors>
    <behavior>
      <method name="getInstance">
        <signature scope="public" returns="Singleton"
                  isClass="true" isAbstract="false"/>
        <implementation>
          <if>
            <condition>
              <isNull variable="instance"/>
            </condition>
            <truePart>
              <createInstance variable="instance" type="Singleton"/>
            </truePart>
          </if>
          <return variable="instance"/>
        </implementation>
      </method>
    </behavior>
  </participant>
</baseImplementation>
```

Snippet 8 – “baseImplementation” section in Singleton’s EML pattern definition. The sample above represents the canonical implementation of the Singleton from the GoF book.

The basic implementation can be redefined, allowing a pattern having more than one concrete implementation. We can add additional implementations to a pattern using the “additionalImplementations” section. In this section, we can redefine or augment any of the participants defined in the “baseImplementation” section. Each particular implementation is described inside an “implementation” section.

In the next snippet we show how to add a “Double Check Lock” implementation to our Singleton pattern.

```xml
<concreteImplementations>
  <implementation name="Double Check Lock">
    <implementationInfo>
      <friendlyName>Double Check Lock Singleton</friendlyName>
      <description>Implementation of the Singleton using the Double Check Lock idiom</description>
    </implementationInfo>
    <instanceOf role="Singleton" name="SingletonMulti">
      <properties>
        <member scope="private" name="padlock"
               type="object" isClass="true"/>
      </properties>
      <behavior>
        <method name="getInstance" is="getInstance">
          <signature scope="public" returns="SingletonMulti"
                    isClass="true" isAbstract="false"/>
          <implementation>
            <if>
              <condition>
                <isNull variable="instance"/>
              </condition>
              <truePart>
                <createInstance variable="instance" type="SingletonMulti"/>
              </truePart>
            </if>
            <lock on="padlock">
              <if>
                <condition>
                </condition>
              </if>
            </lock>
          </implementation>
        </method>
      </behavior>
    </instanceOf>
  </implementation>
</concreteImplementations>
```
Snippet 9 – “additionalImplementation” section in Singleton’s EML pattern definition. The sample above represents the Singleton combined with Dual Check Lock.

**A.2 Another example: describing an concept in EML**

As we stated previously, EML can describe patterns and supporting concepts (pattern languages, categories, books, etc.). It is important to stress the word “entity” that we are using to refer to the root of the EML definition. An entity can be anything that can be described with EML.

In the following example, the “Design Patterns” book is described:

```xml
<entity
type="source"
id="GoFBook"
name="Design Patterns: Elements of Reusable Object Oriented Software">
  <context>
    <relationship
targetType="AuthorGroup" targetType="Gang of Four" type="wasWrittenBy"/>
  </context>
  <tags>
    <tag>GoF Book</tag>
    <tag>Book</tag>
    <tag>Gang of Four</tag>
    <tag>Design Patterns</tag>
  </tags>
  <summary><![CDATA[Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.]]></summary>
</entity>
```

This book isn't an introduction to object-oriented technology or design. Many books already do a good job of that...this isn't an advanced treatise either. It's a book of design patterns that describe simple and elegant solutions to specific problems in object-oriented software design. Once you understand the design patterns and have had an "Aha!" (and not just a "Huh?") experience with them, you won't ever think about object-oriented design in the same way. You'll have insights that can make your own
designs more flexible, modular, reusable, and understandable—which is why you're interested in object-oriented technology in the first place, right?

Notice that this description contains all the description level elements, but doesn’t contain any implementation level element. It also has a relationship to the authors of the book (in the context section).

A.3 Summary: Describing a full entity in EML

In the previous section we have seen how to represent individual sections of a pattern using EML. We have also seen how to represent a concept (an entity that is not a pattern) in EML. An entity is represented in a single file combining all of the sections presented previously.

Therefore, an EML entity definition has the following structure:

1. Tags *(optional)*
2. Context *(optional)*
   a. Relationships between entities
3. Summary
4. Template
   a. All the description level information of the pattern
      i. Problem
      ii. Solution
      iii. Motivation
      iv. Consequences
      v. Etc.
5. Structure *(optional)*
   a. Participants
   b. Relationships
   c. Responsibilities
6. Implementation *(optional)*
   a. Base Implementation
   b. Concrete Implementations
      i. Implementation

Appendix B – Views

In this appendix we will show some of the views that can be applied to the patterns in order to give to the reader a better idea on how patterns are shown in our tool.

B.1 Complete View

This view shows a great amount of information on a pattern. It includes the information in the template, source code, diagrams, tags, relationships, etc. In the next figure the complete view for the Abstract Factory pattern [GoF95] is shown.
This view also contains a section with links to places where more information on the pattern can be found (gathering sparse information and giving a centralized and contextualized point of access to it). The next figure shows the “More Information” section for the Abstract Factory pattern [GoF95].

There is another section included in the view aimed to show related patterns, allowing user to navigate through the contents of the catalog. Each related pattern (actually, related entity) includes a textual explanation of the relationship. The next figure shows the Related Patterns section for the Composite [GoF95] pattern.
CRC Cards View

CRC Cards can be generated for patterns that include in their meta-specification their participants with their respective responsibilities. The next figure shows the CRC View for the Template Method pattern [GoF95].

Source Code View

This view shows the source code generated from the EML meta-specification of a pattern. The source code view allows the user to select the destination language for the generated source code and the implementation to be used (as we stated previously, a meta-specification may contain several implementations). The next figure shows the language and implementation selectors for the Abstract Factory pattern.
Once the language and concrete implementation have been selected, the code can be generated. The next figure shows the source code view applied to the Abstract Factory pattern.

![Source Code view applied to the Abstract Factory pattern.](image)

This view also includes the expected results that may be obtained when the generated source code is executed. The next figure shows the expected results for the code generated above.

![Language and concrete implementation selector in the source code view.](image)

### Acknowledgements

We want to thank our shepherd Till Schuemmer. His valuable comments and suggestions helped us to improve this paper.

### References


