Adaptive Object-Models: a Research Roadmap

Hugo Sereno Ferreira
INESC Porto
Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
hugo.sereno@fe.up.pt

Filipe Figueiredo Correia
Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
filipe.correia@fe.up.pt

Ademar Aguiar
INESC Porto
Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
ademar.aguiar@fe.up.pt

João Pascoal Faria
INESC Porto
Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
jpf@fe.up.pt

Abstract—The Adaptive Object-Model (AOM) is a meta-architectural pattern of systems that expose an high-degree of runtime adaptability of their domain. Despite there being a class of software projects that would directly benefit by being built as AOMs, their usage is still very scarce. To address this topic, a wide scope of concepts surrounding to Adaptive Object-Models need to be understood, such as the role of *incompleteness* in software, and its effects on system variability and adaptability, as well as existing metamodeling and metaprogramming techniques and how do they relate to software construction. The inherent complexity, reduced litera-ture and case-studies, lack of reusable framework components, and fundamental issues as those regarding evolution, frequently drive developers (and researchers) away from this topic. In this work, we provide an extensive review of the state-of-the-art in AOM, as well as a roadmap for empirical validation of research in this area, which underlying principles have the potential to alter the way software systems are perceived and designed.

Keywords-Architectural Structures and Viewpoints, Design Patterns, Families of Programs and Frameworks.
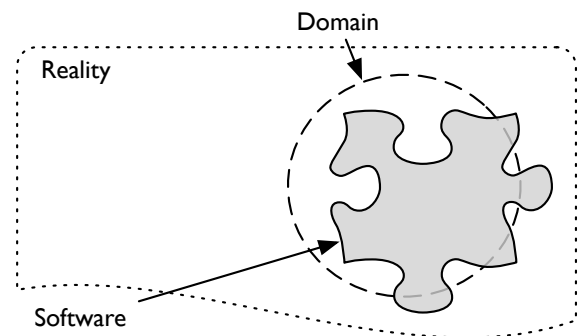
Figure 1. Software may be regarded as the crystallization of an abstraction that models a specific domain. Ideally, it would match the exact limits of that domain. But in practice: (i) those limits are fuzzy, (ii) software often imposes an artificial, unnaturally rigid structure, and (iii) reality itself keeps changing.

I. INTRODUCTION

The current demand for industrialization of software de-velopment is having a profound impact in the growth of software complexity and time-to-market [1]. Moreover, a lot of the effort in the development of software is repeatedly applied to the same tasks, despite all the effort in research for more effective reuse techniques and practices. Like in other areas of scientific research, the reaction has been to hide the inherent complexities of technological concerns by creating increasingly higher levels (and layers) of abstractions with the goal of facilitating reasoning, albeit often at the cost of widening the already existing gap between specification and implementation artifacts [2]. To make these abstractions useful beyond documentation and analytical and reasoning purposes [3], [4], higher-level models must be made exe-cutable, by systematic transformation [5] or interpretation [6] of problem-level abstractions into software implemen-tations. The primary focus of model-driven engineering (MDE) is to find ways of automatically animating such models, often used simply to describe complex systems at multiple levels of abstraction and perspectives and therefore

promoting them to first-class artifacts [2].

A. Incomplete by Design

A recurrent problem in software development is the difficulty of acquiring, inferring, capturing and formalizing requirements, particularly when designing systems where the process is highly-coupled with the stakeholders' perspective and the requirements often change faster than the imple-mentation. This reality is well known in industrial environ-ments, and is sometimes blamed upon incompleteness of the stakeholders' knowledge [7] — maintaining and evolving software is a knowledge intensive task that represents a significant amount of effort [8]. Consequently, once the analysis phase is finished and the implementation progresses, strong resistance to further change emerges, due to the mis-match between specification and implementation artifacts. Notwithstanding, from the stakeholder's perspective, some domains do rely on constant adaptation of their processes to an evolving reality, not to mention that new knowledge is continuously acquired, which lead to new insights of their own business and what support they expect from software.

Confronted with the above issues, some development methodologies (particularly those coined "agile") have intensified their focus on a highly iterative and incremental approach, accepting that *change* is, in fact, an invariant of software development [9]. For example, one of the most prominent books in agile methodologies — Kent Beck's "Extreme Programming Explained" [10] — use the phrase "embrace change" as its subtitle. This stance is in clear contrast with other practices that focus on *a priori*, time-consuming, rigorous design, considering continuous change as a luxurious (and somewhat dangerous) asset for the productivity and quality of software development.

Although the benefits of an up-front, correct and validated specification are undeniable — and have been particularly praised by formal methods of development, particularly when coping with critical systems — their approach is often recognized as impractical, particularly in environments characterized by continuous change. Likewise, the way developers currently cope with change often result in a BIG BALL OF MUD, where the systems will eventually face a total reconstruction, invariably impacting the economy [11]. Thus, software that is target of continuous change should be regarded as *incomplete by design*, or in other words, it needs to be constantly evolving and adapting itself to a new reality, and most attempts to freeze its requirements are probably doomed to fail (see Figure 1).

This notion, and the adequate infrastructure to support it, has roots that go as back as the history of Smalltalk and object-oriented programming in the dawn of personal computers [12]: *"Instead of at most a few thousand institutional mainframes in the world (...) and at most a few thousand users trained for each application, there would be millions of personal machines and users, mostly outside of direct institutional control. Where would the applications and training come from? Why should we expect an applications programmer to anticipate the specific needs of a particular one of the millions of potential users? An extensional system seemed to be called for in which the end-users would do most of the tailoring (and even some of the direct constructions) of their tools."*

B. Motivational Example

Figure 2 depicts small subset of the class diagram from real-world information system for a medical healthcare center. In summary, the medical center has `Patients` and specialized `Doctors`. Information about a patient, such as her personal information, `Contacts` and `Insurances`, are required to be stored. Patients go to the center to have `Appointments` with several `Doctors`, though they are normally followed by just one. During an appointment, several `Pathologies` may be identified, which are often addressed through the execution of medical `Procedures`.

In this example, we can begin to observe the *incompleteness* of these kind of information systems. For exam-ple, procedures, insurances, pathologies and contacts are depicted has having open-hierarchies (where each specialization may require different fields). Patients may not have all the relevant information recorded (e.g., critical health conditions) and foreseeing those missed formalizations, either the designer or the customer make extensive usage of an `observations` field. The system is also missing some domain notion, such that of Auxiliary Personnel, which would require a complete new entity. Maybe it will be revealed as relevant to store personal information of Doctors; actually, in the presence of this new requirements, a designer would probable make `Patients`, `Doctors` and `Auxiliary Personnel` inherits from a single abstraction (e.g., `Persons`). The healthcare center may also require the system to prevent doctors from performing procedures for which they are not qualified (e.g., through a specific constraint based on their specialization). In fact, it now seems evident that a doctor may have multiple specializations.

These are examples of requirements that could easily elude developers and stakeholders during the analysis process. What may seem a reasonable, realistic and useful system at some point, may quickly evolve beyond the original expectations, unfortunately after analysis is considered finished.

C. Accidental Complexity

Should the customer require the system to cope with these incomplete definitions, the designer would have to deliberately make the system *extensible* in appropriate points. Figure 3 shows the *refactored* elements of a particular solution that only addresses open inheritances and enumerations.

Compared to the initial design, the new one reveals itself as a much larger model. In fact, it is now more difficult to distinguish between elements that model the domain, from those that provide extensibility to the system. The result is an increase of what is defined as *accidental complexity* — complexity that arises in computer artifacts, or their development process, which is non-essential to the problem to be solved. In contract, the first model was much closer to that of *essential complexity* — inherent and unavoidable. This increase in accidental complexity was only caused by the specific approach chosen to solve the problem — in this case, recurrent usage of the TYPE-OBJECT pattern.

While sometimes accidental complexity can be due to mistakes such as ineffective planning, or low priority placed on a project, some accidental complexity always occurs as the side effect of solving any problem. For example, mechanisms to deal with out of memory errors are part of the accidental complexity of most implementations, although they occur just because one has decided to use a (von-neumann) computer to solve the problem. Because the minimization of accidental complexity is considered a good practice to any architecture, design, and implementation,
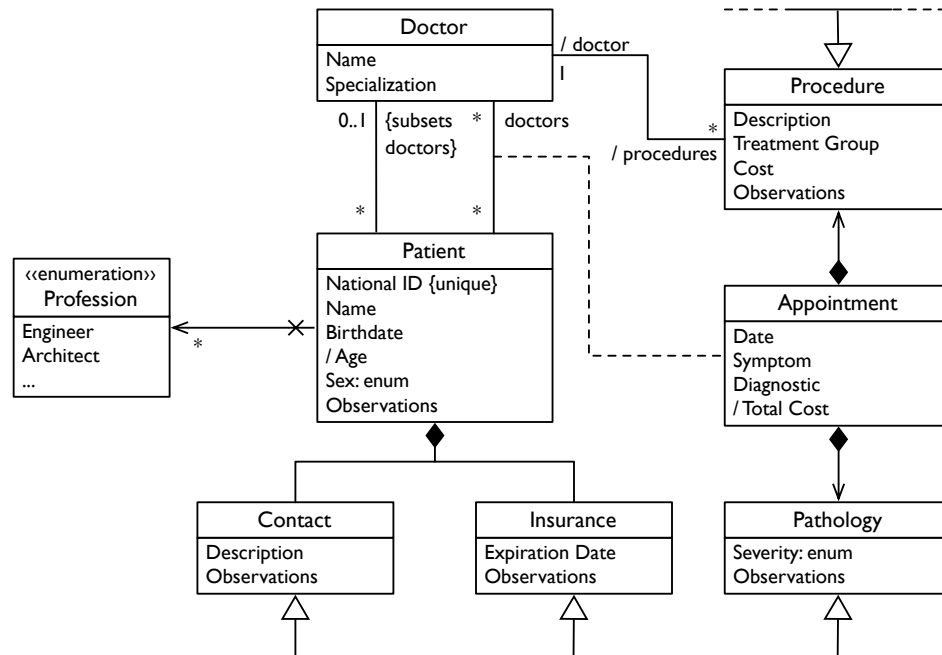
Figure 2.   Example of a domain diagram of an information system for a medical center. The horizontal dashed lines denote open (incomplete) inheritances. The dots inside the enumeration also denotes incomplete knowledge which should be editable by the end-user.

excessive accidental complexity is a clear example of an anti-pattern.

D. Designing for Incompleteness

While newer software engineering methodologies struggle to increase the ability to adjust easily to change of both the process and the development team, they seem to generally have a certain agnosticism regarding the form of the produced software artifacts (probably an over-simplification since agile methodologies, for example, recommended the *simplest design that works*, which addresses form, in a certain way). This doesn't mean they are not aware of this "need to change". In fact, iterative means several cycles going from analysis to development, and back again. Some are also aware of the BIG BALL OF MUD pattern (or anti-pattern, depending on the perspective); the practice of *refactoring* after each iteration in order to cope with *design debt* is specifically included to address that [13]. But the problem seems to remain in the sense that the outcome — w.r.t. *form* — of each iteration is mostly synthesized as if it would be the last one (albeit knowing it isn't).

Yet, if these systems are accepted and regarded as being *incomplete by design*, it seems reasonable to assume benefits when actively *designing them for incompleteness*. If we shift the way we develop software to *embrace change*, it seems a natural decision to deliberately design that same software to best cope with change. Citing the work of Garud et al.: *"The traditional scientific approach to design extols the virtues of completeness. However, in environments characterized by continual change (new solutions) highlight a pragmatic approach to design in which incompleteness is harnessed in a generative manner. This suggests a change in the meaning of the word* design *itself – from one that separates the process of design from its outcome, to one that considers design as both the medium and outcome of action."* [7]

This is in particular dissonance with the current approaches to software engineering, where most processes attempt to establish a clear line between designing and developing, specifying and implementing. Though it seems that, should we wish to harness *continual change*, that distinction no longer suits our purposes: *design should become both the medium and outcome of action*. Consequently, we are thus looking forward not just for a process to be effective and agile, but to what *form* should agile software take.

E. Article Structure

The remaining of the article is divided into three main sections. We will first present a literature survey of the state-of-the-art regarding several concepts and techniques used to harness the specification and construction of these *incomplete by design* systems. In Section III, we'll use return to the motivational example and delve into the Adaptive Object-Model meta-architectural pattern to further detail its related concerns. Section IV aims to summarize the current known open issues in the field, and discuss issues on research design and empirical validation for assessing
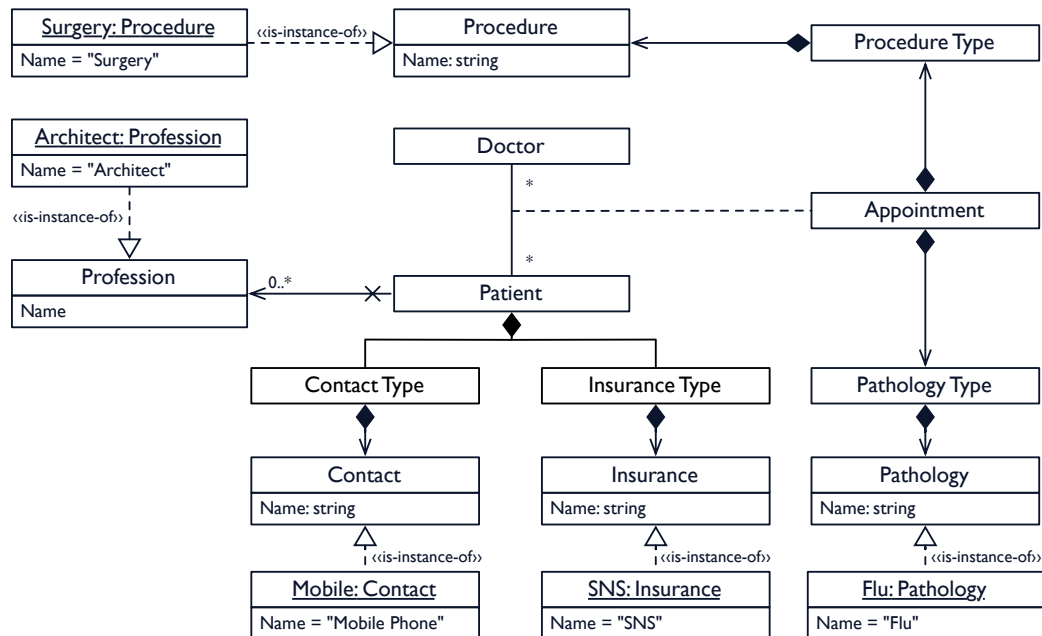
Figure 3. A *refactored* solution for the diagram in Figure 2, mainly depicting the elements that were changed/added for providing a mechanism to cope with open inheritance and enumerations. This example makes extensive use of the TYPE-OBJECT pattern (see Section III).

Adaptive Object-Models. We will finish this article by drafting some conclusions and pointing to future work.

II. STATE-OF-THE-ART

One way to design software able to cope with incompleteness is to encode the system's concepts into higher-level representations, which could then be systematically synthesized — desirably, in an automatic fashion — into executable artifacts, thus reducing the overall effort (and complexity) of changing it. An overview of the several concepts that will be approached in this section is shown in Figure 4.

A. Fundamentals

1) Variability: Software variability is the need of a software system or artifact to be changed, customized or configured for use in different contexts [14]. High variability means that the software may be used in a broader range of contexts (i.e., the software is more reusable). The degree of variability of a particular system is given by its *Variation Points*, or roughly the parts which support (re)configuration and consequently tailoring the product for different contexts or for different purposes. Variability is a well-known concept in Software Product Lines, which will be covered later.

2) Adaptability and Self-Adaptive Systems: While variability is given by context, the capability of software systems to react efficiently to changed circumstances is called Adaptability. The main difference relies on what has changed and what is being changed accordingly. The same software may

be reconfigured to be used in different contexts (e.g., by re-compiling with an additional component), and this provides Variability. The mechanisms that allow software to change its behavior (without recompiling) is called Adaptability. Adaptive systems may thus be defined as open systems which are able to fit their behavior according to either (or both) external or internal changes. The work by Andresen et al. [15] identifies some enabling criteria for adaptability in enterprise systems. Further work by Meso et al. [16] provides a insight into how agile software development practices can be used to improve adaptability.

Self-adaptation is a particular case of Adaptability, when software systems are empowered with the ability to adjust their own behavior themselves during run-time, in response to both their perception of the environment and itself [17]. Despite that in self-adaptation often (i) the change agent is the system itself in reaction to the external world, and (ii) the scope of adaptability is well-defined *a priori*, there is an extensive amount of research that still applies to systems that need adaptation, without having to adapt themselves.

B. Incompleteness

1) The Wiki Way: Earlier in 1995, Cunningham wrote a set of scripts that allowed collaborative editing of webpages inside the very same browser used to view them [18], and named this system *WikiWikiWeb*. He chose this word because of the analogy between its meaning (quick) and the underlying philosophy of its creation: a *quick-web*. Up until now, wikis have gradually become a popular tool on

Figure 4. Concept map representing the relationship between several areas, concepts and techniques related to Adaptive Object-Models.

several domains, including that of software development (e.g., to assist the creation of lightweight software documentation [19]) — they ease collaboration, provide generalized availability of information, and allow to combine different types of content (e.g., text, models, code) into the same infrastructures.

2) Characteristics: The overall success of wikis may be due to a set of design principles drafted by Cunningham that overall tend to embrace *change* and *incompleteness*, namely (i) Open, (ii) Incremental, (iii) organic, (iv) Universal, (v) Overt, (vi) Tolerant, (vii) Observable and (viii) Convergent. Their original definition may be looked up in [20].

3) Wikis and Incomplete Systems: Likewise, we may also regard the usage (and development) of a software systems as a team work. The underlying design and the knowledge that lead to it (e.g., requirements, use-cases, models) is mostly devised and shared between developers and stakeholders. Data is collaboratively viewed and edited among users. Oddly, not only these circles seem to be disjoint, but emergent knowledge from this collaboration seem to start and end towards the artifacts themselves, despite the fact they have (and are) built incrementally.

Incompleteness, once again, seems to be the key. Instead of regarding it as a defect, we should embrace it as the means through which the system evolves, thus fulfilling its function. WikiWikiWeb seemed to have realized this fundamental ideal, and for us it is reasonable to conjecture about attempting the same underlying principles [20] to other computational systems as well:

- **Open.** Should a resource be found to be incomplete or poorly organized, the end-user can evolve it as they see fit.
- **Incremental.** Resources can link to other resources,

including those who have not yet been brought into existence.
- **Organic.** Structure and content are open to editing and evolution. Evolution may be made more difficult if it is mandatory for information to strictly conform to a pre-established model.
- **Universal.** The same (or very similar) mechanisms for modifying data and model should be exposed by the system with no apparent distinction.
- **Overt.** End-user evolution should be made by non-programmers. The introduction of linguistic constructions (such as textual syntax) is usually required in order to provide formalization. However, such constructions may reveal unnatural, intrusive and complex to the end-user, thus model edition should be made as readily apparent (and transparent) as possible.
- **Tolerant.** Interpretable behavior is preferred to system halt.
- **Observable.** Activity should be exposed and reviewed by end-users, fomenting social collaboration.
- **Convergent.** Duplication is discouraged and removed by incremental restructuring and linking to similar or related content.

To reach expand these principles inherent to the *WikiWikiWeb* to other areas of software, we need a particular degree of adaptability — both to the developers and to the end-users — from the infrastructure which is neither commonly found, nor particularly easy to design.

C. Abstraction

Wikipedia [21] has several articles defining the concept of abstraction, depending on the scientific area: *"Conceptually, it is the process or result of generalization by reducing*

the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose. In mathematics, it is the process of extracting the underlying essence of a mathematical concept, removing any dependence on real world objects with which it might originally have been connected, and generalizing it so that it has wider applications or matching among other abstract descriptions of equivalent phenomena. In computer science, it is the mechanism and practice of abstraction reduces and factors out details so that one can focus on a few concepts at a time."

All definitions share one factor in common, i.e., that *abstraction* involves relinquishing some property (e.g., detail) to gain or increase another property (e.g., simplicity). For example, a common known use of abstraction is the level of programming language. Assembly is often called low-level because it exposes the underlying mechanisms of the machine with an high degree of fidelity. On the other end, Haskell is an high-level language, struggling to hide as much as possible the underlying details of its execution. The latter trades execution *performance* in favor of *cross-platform* and *domain expressiveness*.

In this sense, abstractions are never to be considered *win-win* solutions. For example, Joel Spolsky [22] observes a recurrent phenomena in technological abstractions called *Leaky Abstractions*, which occurs when one technological abstraction tries to completely hide the concepts of another, lower-level technology, and sometimes the underlying concepts *"leak through"* those supposed invisible layers, rendering them visible. For example, an high-level language may try to hide the fact that the program is being executed at all by a *von-neumann* machine. In this sense, although two programs may be functionally equivalent, memory consumption and processor cycles may eventually draw a clear separation between them. Hence, the programmer may need to learn about the middle and lower-level components (i.e., processor, memory, compiler, etc.) for the purpose of *designing a program that executes in a reasonable time*, thus breaking the abstraction. He goes as far as hypothesizing that *"all non-trivial abstractions, to some degree, are leaky"*. Hence, good abstractions are specifically designed to express the exactly intended details in a specific context, while relinquishing what is considered unimportant.

1) Metaprogramming: Metaprogramming consists on writing programs that generate or manipulate either other programs, or themselves, by treating code as data [23]. Historically, it is divided into two languages: (i) the meta-language, in which the meta-program is written, and (ii) the object language which the metaprogram produces or manipulates. Nowadays, most programming languages use the same language for the two functions [24], either by being homoiconic (e.g., Lisp), dynamic (e.g., Python) or by exposing the internals of the runtime engine through APIs (e.g., Java and .NET). Claims about the economic benefits in

terms of development and adaptability have been studied and published for more than twenty years [25], though its focus is on code-level manipulation and not on domain artifacts.

2) Meta-modeling: Supporting the use of models during runtime is an answer to high-variability systems [6], where the large semantic mismatch between specification and implementation artifacts in traditional systems can be reduced by the use of models, meta-models, and metadata in general. Metamodeling is thus the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for modeling a predefined class of problems [26] (i.e., a model to specify models).

3) Reflection: Reflection is the property of a system that allows to observe and alter its own structure or behavior during its own execution. This is normally achieved through the usage and manipulation of (meta-)data representing the state of the program/system. There are two aspects of such manipulation: (i) introspection, i.e., to observe and reason about its own state, and (ii) intercession, i.e., to modify its own execution state (structure) or alter its own interpretation or meaning (semantics) [24]. Due to this properties, reflection is a key property for metaprogramming.

4) Domain Specific Languages: A domain-specific language (DSL) is a programming or specification language specifically designed to suit a particular problem domain, representation technique, and/or solution technique. They can be either visual diagramming languages, such as UML, programatic abstractions, such as the Eclipse Modeling Framework, or textual languages, such as SQL. The benefits of creating a DSL (along with the necessary infrastructure to support its interpretation or execution) may reveal considerable whenever the language allows a more clear expression of a particular type of problems or solutions than pre-existing languages would, and the type of problem in question reappears sufficiently often (i.e., recurrent, either in a specific project, like extensive usage of mathematical formulae, or global-wise, such as database querying).

The creation of a DSL can be supported by tools such as AntLR [27] or YACC [28], which take a formalized grammar (e.g., defined in a meta-syntax such as BNF), and generate parsers in a supported target language (e.g., Java). Recently, the term DSL has also been used to coin a particular type of syntactic construction within a general purpose language which tends to more naturally resemble a particular problem domain, but without actually extending or creating a new grammar. The Ruby community, for example, has been enthusiastic in applying this term to such syntactic sugar [29].

Domain-specific languages share common design goals that contrast with those of general-purpose languages, in the sense they are (i) less comprehensive, (ii) more expressive in their domain, and (iii) exhibit minimum redundancy. Language Oriented Programming [30] considers the creation of special-purpose languages for expressing problems as a

standard methodology of the problem solving process.

5) Meta-Architectures: We have already seen several techniques used to address systems with high-variability needs. There is, nonetheless, differences between them. For example, some do not parse or interpret the system definition (meta-data) while it is running: Generative Programming and Metamodeling rely on code generation done at compile time. Reflection is more of a property than a technique by itself, and the level at which it is typically available (i.e., programming language) is inconvenient to deal with domain-level changes. Domain Specific Languages are programming (or specification) languages created for a specific purpose. They are not generally tailored to deal with change (though they could), and they do require a specific infrastructure in order to be executed.

Meta-architectures, or reflective-architectures, are architectures that strongly rely on reflective properties, and may even dynamically adapt to new user requirement during runtime. Pure OO environments, and MOF-based systems are examples of such architectures, as they make use of meta-data to create different levels that sequentially comply to each other. The lowest level in this chain is called the data level, and all the levels above the meta-data levels, but the line that separates them is frequently blurred as both are data.

D. Approaches

1) Software Product Lines: A software product line (SPL) is a set of software systems which share a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [31]. Software product line development, encompasses software engineering methods, tools and techniques for supporting such approach. A characteristic that distinguishes SPL from previous efforts is predictive versus opportunistic software reuse. Rather than put general software components into a library in the hope that opportunities for reuse will arise, software product lines only call for software artifacts to be created when reuse is predicted in one or more products in a well defined product line.

2) Naked Objects: Naked Objects takes the automatic generation of graphical user interfaces for domain models to the extreme. This software architectural pattern, first described in Richard Pawson's PhD thesis [32] which work includes a thorough investigation on prior known uses and variants of this pattern, is defined by three principles:

1) All business logic should be encapsulated onto the domain objects, which directly reflect the principle of *encapsulation* common to object-oriented design.
2) The user interface should be a direct representation of the domain objects, where all user actions are essentially creation, retrieval and message send (or method invocation) of domain objects. It has been

argued that this principle is a particular interpretation of an object-oriented user-interface (OOUI).
3) The user interface should be completely generated solely from the definition of the domain objects, by using several different technologies (e.g., source code generation or reflection).

The work of Pawson further contains some controversial information, namely a foreword by Trygve Reenskaug, who first formulated the model-view-controller (MVC) pattern, suggesting that Naked Objects is closer to the original MVC intent than many subsequent interpretations and implementations.

3) Domain-Driven Design: Domain-driven design (DDD) was coined by Eric Evans in his books of the same title [33]. It is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts, which encompasses a set of practices and terminology for making design decisions that focus and accelerate software projects dealing with complicated domains. The premise of domain-driven design is the following: (i) placing the project's primary focus on the core domain and domain logic, (ii) basing complex designs on a model, and (iii) initiating a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem.

4) Model-Driven Engineering: Model-Driven Engineering (MDE) is a metamodeling technique that strives to close the gap between specification artifacts which are based upon high-level models, and concrete implementations. A conservative statement claims that MDE tries to reduce the effort of shortening (not completely closing) that gap by generating code, producing artifacts or otherwise interpreting models such that they become executable [2].

Proponents of MDE claim several advantages over traditional approaches: (i) shorter time-to-market, since users model the domain instead of implementing it, focusing on analysis instead of implementation details; (ii) increased reuse, because the inner workings are hidden from the user, avoiding to deal with the intricate details of frameworks or system components; (iii) fewer bugs, because once one is detected and corrected, it immediately affects all the system leading to increased coherence; (iv) easier-to-understand systems and up-to-date documentation, because the design is the implementation so they never fall out of sync [6]. One can argue if these advantages are exclusive of MDE or just a consequence of "raising the level of abstraction" (see *Domain Specific Languages*).

Downsides in typical generative MDE approaches include the delay between model change and model instance execution due to code generation, debugging difficulties, compilation, system restart, installation and configuration of the new system, which can take a substantial time and must take place within the development environment [6]. Once

again, it doesn't seem a particular downside of MDE, but a general property of normal deployment and evolution of typical systems.

More interesting counter-points to MDE adoption will be addressed in Section IV. It seems worthwhile to note that the prefix *Model-driven* seems to be currently serving as a kind of umbrella definition for several techniques.

5) Model-Driven Architecture: Model-driven architecture (MDA) in an approach to MDE proposed by the OMG, for the development of software systems [34]. It provides a set of guidelines to the conception and use of model-based specifications and may be regarded as a type of domain engineering. It bases itself on the Meta Object Facility (MOF) [35], which main purpose is to define a strict, closed metamodeling architecture for MOF-based systems and UML itself, provides four modeling levels (M_3, M_2, M_1 and M_0), each conforming to the upper one (M_3 is in conformance to itself).
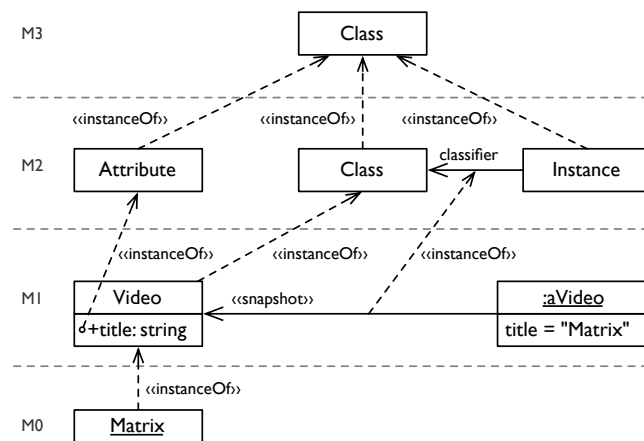
Figure 5. Layers of Abstraction in the Meta-Object Facility (MOF). Each level is in direct conformance to the upper level. The classes named *Class*, from both level 2 and 3, represent the same concept.

Like most of the MDE practices, MDA thrives within a complex ecosystem with specialized tools for performing specific actions. Moreover, MDA is typically oriented for generative approaches, using systematic offline transformation of high-level models into executable artifacts. for example, trying to answer MDA's objective of covering the entire gap between specification and implementation, xUML was developed. It is a UML profile which allows the graphical representation of a system, and takes an approach in which platform specific implementations are created automatically from platform-independent models, with the use of model compilers.

While some complex parts of MDA allow runtime adaptivity, developers seldom acquire enough knowledge and proficiency of the overall technologies to make them cost (and time) effective in medium-sized applications. Runtime adaptivity may be approached in different ways, including

the use of reflection, and the interpretation of models at runtime [6], covering the concept of *Adaptive Object-Model* (see section II-E3).

E. Infrastructures

1) Frameworks: Object-oriented frameworks are reusable designs of all or part of a software system described by a set of abstract artifacts and the way they collaborate [36]. They aim to provide both an infrastructure, through a COMPONENT LIBRARY and pre-defined interconnections among them, which may all be (re-)configured and extended to address different problems in a given domain. Good frameworks are able to reduce the cost of development by an order of magnitude. It should be stressed that software frameworks are more than just a collection of reusable components (also known as LIBRARY); a framework usually makes use of the *Hollywood Principle*[1] to promote high cohesion and low coupling in object-oriented designs, by ensuring a THREAD OF CONTROL.

2) Generative Programming: One common approach to address variability and adaptability is the use of Generative Programming (GP) methods, which transform a description of a system (model) based in primitive structures [37], into either executable code or code skeleton. This code can then be further modified and/or refined and linked to other components [38]. Generative Programming deals with a wide range of possibilities including those from Aspect Oriented Programming [39], [40] and Intentional Programming [41].

Because GP approaches focus on the automatic generation of systems from high-level (or, at least, higher-level) descriptions, it is arguable whether those act like a meta-model of the generated system. Still, the resulting functionality is not directly produced by programmers but specified using domain-related constructs. In summary, GP works as an off-line code-producer and not as a run-time adaptive system [42].

This technique typically assumes that (i) changes are always introduced by developers (change agents), (ii) within the development environment, and (iii) that a full transformation (and most likely, compilation) cycle is affordable (i.e., no run-time reconfiguration). When these premises fail to hold, generative approaches are easily overwhelmed [6].

3) Adaptive Object-Models: In search for flexibility and run-time adaptability, many developers had systematically applied code and design reuse of particular domains, effectively constructing higher-levels representations (or abstractions). For example, some implementations have their data structure and domain rules extracted from the code and stored externally as modifiable parameters of well-known structures, or statements of a DSL. This strategy gradually transforms some components of the underlying system into

---

[1] A well-known cliché response given to amateurs auditioning in Hollywood: *"Don't call us, we'll call you"*.

an Interpreter or Virtual Machine whose run-time behavior is defined by the particular instantiation of the defined model. Changing this model data thus results on the system following a different business domain model. Whenever we apply these techniques based upon object-oriented principles and design, we are using an architectural style, known as an Adaptive Object-Model [43].

Shortly, it is (i) a class of systems' architectures, (ii) heavily based on Metamodeling and Object-Oriented design, (iii) which often uses a Domain Specific Language, (iv) usually have the property of being Reflective, and (v) with the intent of exposing its configuration to the end-user. Because we are abstracting a set of systems and techniques into a common underlying architecture heavily based on object-oriented metaprogramming/metamodelling, we categorize the AOM as a meta-architecture.

The evolution of the core aspects of an AOM can be observed by the broad nomenclature used in the literature in the past couple of decades (e.g., Type Instance, User Defined Product Architecture, Active Object-Models and Dynamic Object Models). The concept of Adaptive Object-Model is inherently coupled with that of an architectural pattern, as it is an effective, documented, and prescriptive solution to a recurrent problem.

It should therefore be noted that most AOMs emerge from a bottom-up process [43], resulting in systems that will likely use only a subset of these concepts and properties, and only when and where they are needed. This is in absolute contrast with top-down efforts of specific meta-modeling techniques (e.g., Model-Driven Architecture) where the whole infrastructure is specifically designed to be as generic as possible (if possible, to completely abstract the underlying level).

The concepts of End-User Programming and Confined Variability — the capability of allowing the system's users to introduce changes and thus control either part, or the entire system's behavior — are significant consequences of the AOM architecture which aren't easily reconcilable with other techniques such as Generative Programming.

III. ARCHITECTURE AND DESIGN OF ADAPTIVE OBJECT-MODELS

The basic architecture of an Adaptive Object-Model is divided into three parts, or levels, that roughly correspond to the levels presented by MOF: M_0 is the operational level, where system data is stored, M_1 is the knowledge level where information that defines the system (i.e., the model) is stored, and M_2 is the design of our supporting infrastructure. M_0 and M_1 are variants of our system. M_2, the meta-model, is usually invariant — should it need to change, we would have to transform M_0 and M_1 to be compliant with the new definition.

A. Making the Structure Agile

In Section I, we have shown a refactored example which made use of a small set of patterns to introduce the desired

adaptability into the original system. As always, one key to good software design is two-fold: (i) recognize the things that will often change in a predictable way, and (ii) recognize what it rarely does not; it is the search for *patterns* and *invariants*.

B. The Type-Object Pattern

In the context Object-Oriented Programming and Analysis, the Type of an object is defined as a Class, and its Instance as an Object which conforms to its class. A typical implementation of our example system would hardcode into the program structure (i.e., source-code) every modeled entity (e.g., Patient). Would the system needed to be changed (e.g., to support a new entity) the source-code would have to be modified.

However, if one anticipate this change, objects can be generalized and their variation described as parameters, just like one can make a function that sums any two given numbers, regardless of their actual value. The TYPE OBJECT pattern, depicted in Figure 7, identifies the practice of splitting such a class in two: a Type called EntityType, and its Instances, called Entity.
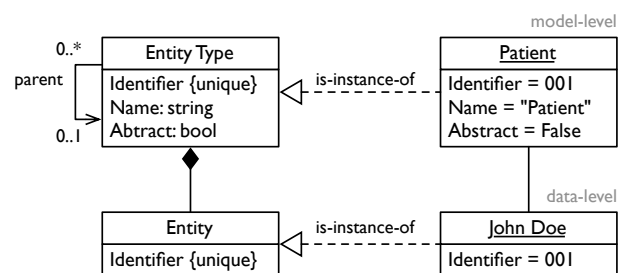

Figure 7. The TYPE-OBJECT pattern.

Using this pattern, Patients, Doctors, Appointments, etc. all become instances of Entity Types, therefore meta-data. The actual system data, such as a the patient John Doe, are now represented as instances of Entity. Because data and meta-data are values beyond the program structure, they can be changed during run-time.

We'll often extend and customize the original design of patterns to further enhance the model semantics. For example, by supporting the notion of inheritance through an optional relation between EntityTypes, which do incidentally solve the problem of open-inheritance. Provided sufficient mechanisms exists to allow the end-user customization of the model-level, new specializations (e.g., Procedures, Pathologies, etc.) can be added without modifying the source-code.

C. The Property Pattern

Similarly to the TYPE-OBJECT pattern, we face a similar problem with the attributes of an object, such as the Name and Age of a Patient, which are usually stored as values of fields of an object, which have been defined in its
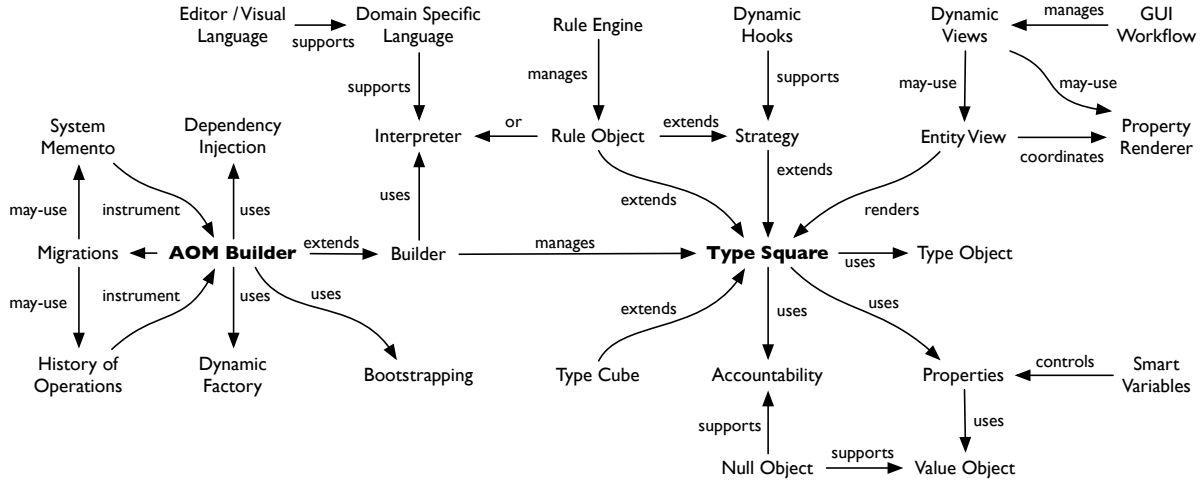
Figure 6.   Pattern map of design patterns and concepts related to Adaptive Object-Models. Adapted from [44].

appropriate classes. Once again, anticipation of this change leads to the PROPERTY pattern; a similar bi-section between the definition of a property and its corresponding value as depicted in Figure 8.

Using this pattern, the Name, Age, Birthday, Profession, and several attributes of the domain's entities become instances of Property Types, and their particular values instances of Property. Again, this technique solves the problem of adding (or removing) more information to existing entities beyond those originally designed.

D. The Type-Square Pattern

The two previous patterns, type-object and property, are usually used in conjunction, resulting in what is known as the TYPE-SQUARE pattern, which poses the core of an AOM. If we add the instantiations of these classes, we get the diagram depicted in Figure 9.

In this picture, the objects represent both the systems' data and model, while the classes represent our static, abstract infrastructure. The ability to represent an increasing number of different — and useful — systems is directly dependent on the power of the underlying infrastructure.

E. Revisiting the PROPERTY Pattern

Literature in AOM used to describe a form to capture relationship between different objects by using the AC-COUNTABILITY pattern [45]. But what exactly is a field, a relation or a property? Object fields, in OOP, are used to store either values of native types (such as int or float in Java) or references to other objects. They can also be either scalars or collections. Some OO languages (e.g., smalltalk) treat everything as an object, and as such do not make any difference from native types to references. Some also discard scalar values and instead use singleton sets. We may borrow these notions to extend the PROPERTY pattern in order to support associations between entities, provided we are able to state properties such as cardinality, navigability, role, etc. The actual differentiation between what is a scalar property and a relation, becomes an implementation detail. Figure 10 depicts such design.

One hook introduced between the abstraction and the underlying language is the use of the Native Type property in Entity Type, to allow any custom Entity Type to be directly mapped into a native type of the underlying language (such

Figure 8.   The PROPERTY pattern, which separates the definition of an object's property and its corresponding value.
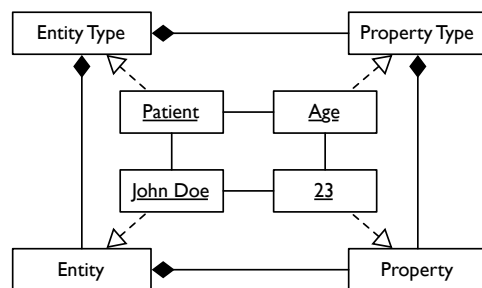

Figure 9. The TYPE-SQUARE pattern, which is a composition of the TYPE OBJECT and PROPERTY patterns.
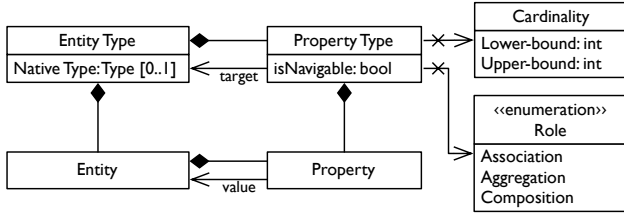
Figure 10. An extension of the TYPE-SQUARE pattern, using a variant of the ACCOUNTABILITY pattern.

as integers and strings).

There are also several logical restrictions related to the semantics of this design. For example, the lower and upper bound in cardinality should restrain the number of associations from a single Property to Entities. Likewise, Properties should only link to Entities which are of the same Entity Type as that defined in Property Type. The complete formalization of the semantics of the presented models is outside of the scope of this work.

F. Self-compliance

If our meta-model has enough expressivity, one can reach the point where the model can be represented inside itself. MOF is an example of such self-compliance by making M_3 a self-describing level. There are several reasons to make that design choice: (i) it makes a strict meta-modeling architecture, since every model element on every level is strictly in correspondence with a model element of above level, and (ii) the same mechanisms used to view, modify and evolve data can be reused for meta-data. A way to accomplish this is by introducing the notion of a Thing, that abstracts elements at any level. A Thing is thus an instance of another Thing, including of itself, such as depicted in Figure 11. This implies that during bootstrapping, the system would first need to load its own meta-model. When applicable, Entity Types are actually M_2 Entities, thus requiring a mechanism to inherently convert between them.
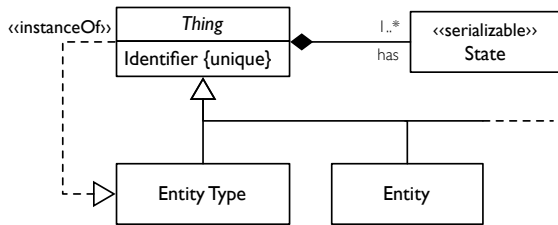
Figure 11. The THING pattern, which decouples the *Identity* of an object from its *State*.

G. Versioning

As is also shown by Figure 11, the identity of each instance is maintained as Things, while the respective details

are maintained as States. The decoupling of these two concepts may be leveraged to provide auditability capabilities, thus answering a common request when building information systems. Auditability may be reached by allowing users to access past versions of an instance, and thus to understand how such instance has evolved.

Figure 12 depicts an example of this mechanism. Two distinct values exist for the same Property Type, corresponding to two different changes the instance has been through over time.
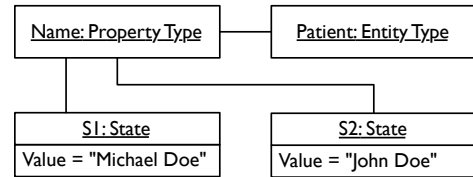

Figure 12. S1 and S2 are two different states — or snapshots — of the property name. Although a typical system usually stores only the most recent snapshot, some techniques rely on having the history of an object.

H. Making the Behavior Agile

In addition to structure, such as Entities, Properties and Associations, systems usefulness also relies on the ability to support rules and automatic behavior. Some examples of these include, but are not limited to, (i) Constraints, such as relationship cardinality, navigability, type redefinition, default values, pre-conditions, etc. (ii) Functional Rules, which include reactive logic such as triggers, events and actions, and (iii) Workflows.

During the instantiation of the object-model, after all types of objects and their respective attributes are created, there are some operations that can be applied to them. Some of these operations are simple Strategies, relatively immutable or otherwise parameterized, which can be easily described in the metadata as the method to be invoked along with the appropriate Strategy.

When the desired behavior reaches a certain level of complexity and dynamism, then a domain specific language can be designed using RULE-OBJECTS [46]. In this case, primitive rules are defined and composed together with logical objects, parsed into a tree structure to be interpreted during runtime. The use of patterns like SMART VARIABLES [47] and TABLE LOOKUP reveal useful for triggering automatic updates and validations of property values. More complex systems can make use of StateMachines and Workflows both for data and human-computer interaction.

A common problem that arises with the abstraction of rules, is that the developer may fall in the trap of creating (and then maintaining) a general-purpose programming language, which may result in increased complexity regarding the implementation and maintenance of the target application, far beyond what would be expected if those rules were

hardcoded [42]. It should be kept in mind that the goal is not to implement the whole application in a new language; just what changes.

I. Rules

Figure 13 depicts a design extending the RULE OBJECT pattern and presented in [48], which allows the definition of: (i) Entity Type invariants, which are predicates that must always hold, (ii) derivation rules for Property Types and Views, (iii) the body of Methods, (iv) guard-conditions of Operations, etc. Even some structural enforcement, such as the Cardinality and Uniqueness of a Property Type may be unified into conditions. Methods, which are used-defined Batches of Operations, may be invoked manually, or triggered by Events, thus providing enough expressivity to specify STATE MACHINES. This design ensures the capability of the system to enforce semantic integrity during normal usage and assisting model evolution.

J. Views

In the design presented so far, Views are regarded as Entity Types which have a derivation rule that returns a collection, and every property have a second derivation rule (often manipulating each item of that collection). They allow the existence of virtual Entity Types where their information is not stored by inferred, similar the querying mechanisms of relational databases (SQL). For example, we could consider a requirement for the system presented in Figure 2, that specifies a list of items composed by every doctor in the medical center, along with the number of high-risk procedures and their total cost. This would be represented by a new Entity Type ("High-Risk Treatments Income by Doctor") that iterates over Doctors and their Procedures. Since the latter is also a derived property, it should be specified as a rule of Doctor and so on.

K. Evolution

Has already discussed, structural integrity of the run-time model is asserted through rules stated in the meta-model. For example, Entities should conform to their specified Entity Type (e.g., they should only hold Properties to which its Property Type belong to the same Entity). Nonetheless, evolving the model may corrupt structural integrity. For example, when moving a mandatory Property Type to its superclass, if it doesn't have a default value, it can render some Entities structurally inconsistent. Some of these issues can be coped with, by foreseeing integrity violations and applying prior steps to avoid them (e.g., one could first introduce a default value before moving the Property Type to its superclass). Through careful annotation of model-level operations (e.g., by reducing the applicability scope through pre-conditions so that a well-formalized semantics is established), one can increase the confidence on maintaining structural integrity.

Another issue arises when parts of a composite evolution violate model integrity, although the global result would be valid. For example if a Property Type is mandatory, one cannot delete its Properties without deleting itself and vice-versa. This problem is solved by the use of transactions or change-sets, and by suspending integrity check until the end.

Semantic Integrity of a particular evolution, on the other hand, is much harder to ensure since it is not encoded as rules in the meta-model. State-based comparison of models have already shown this problem, because it is not always possible to just compare the results of an arbitrary evolution and accurately infer the performed operations. In our example, consider the scenario where a patient's age was being stored directly, but we now realize that having the birthdate is far less problematic. For that, the following modifications to the model are made: (i) rename Age to Date of Birth, (ii) reverse calculate it according to the current date, and (iii) move it to a the superclass Person.
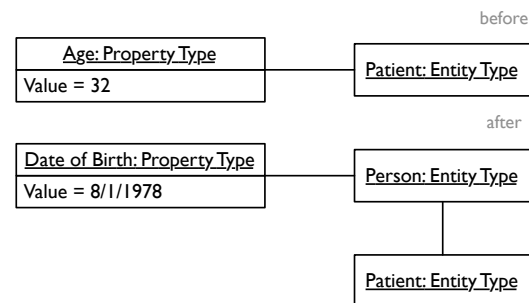

Figure 14. A model evolution example, where a property is both changed w.r.t. its semantics, and moved to the superclass.

Would we rely on the direct comparison of the initial and final models, a possible solution would be to delete the Age in Patient and create Date Of Birth in Person. Clearly, the original meaning of the intended evolution (e.g., that we wanted to transform ages to birth-dates) would be missed, and every data lost. Operation-based evolution solves this problem, and we can make use of the MIGRATION pattern [49] which express these changes through sequences of model-level operations that cascade into instance-level changes, as we'll see next.

L. (Co-)Evolution

Allowing (potentially collaborative) co-evolution of model and data by the end-user introduces a new set of concerns not usually found in classic systems. They are (i) how to preserve model and data integrity, (ii) how to reproduce previously introduced changes, (iii) how to access the state of the system at any arbitrary point in the past, and (iv) how to allow concurrent changes. All these concerns are directly related to traceability, reproducibility, auditability, disagreement and safety, and are commonly found, and coped with, on version-control systems.
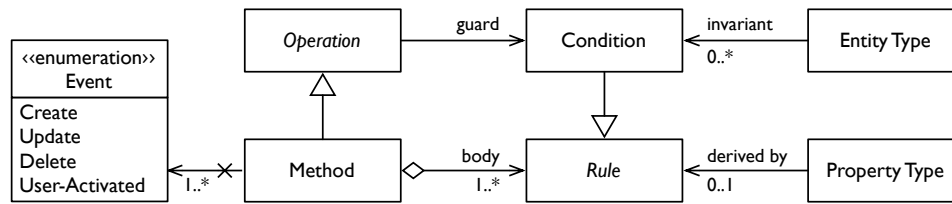
Figure 13.   The dynamic core of an Adaptive Object-Model framework. This particular design supports method definition, guard conditions, class invariants, derived rules and event triggers.

Typically, Evolution is understood as the introduction of changes to the model. Yet, the presented design so far doesn't establish a difference between changing data or meta-data; both are regarded as evolutions of Things, expressed as Operations over their States, and performed by the same underlying mechanisms as depicted in Figure 15. To provide enough expressivity such that semantic integrity can be preserved during co-evolution, model-level Batches operate simultaneously over data and meta-data.

Sequences of Operations may be encapsulated as Change-Sets, following the HISTORY OF OPERATIONS pattern [49], along with meta-information such as user, date and time, base version, textual observations, and data-hashes, etc. Whenever the system validates or commits a ChangeSet, the Controller uses the merge mechanism depicted in Figure 15 (similarly to the SYSTEM MEMENTO pattern [49]) by (i) orderly applying each Operation to create a new State, (ii) dynamically overlaying the new States onto the base version, (iii) evaluating and ensuring behavioral rules, and finally (iv) producing a new version.
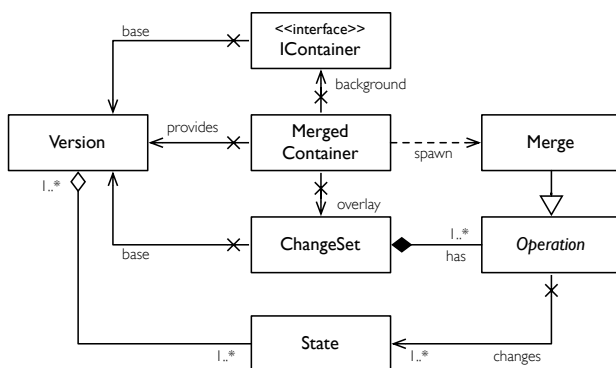

Figure 15.   The merging mechanism of an AOM framework, which overlays a changeset to provide a merged view (i.e., a snapshot) of a system.

M. Enduser (Co-)Evolution

In order for the end-user to perform arbitrary model evolutions, a sufficiently large library of (probably composable) operations should be provided. If the sequences of operations over data and meta-data are preserved, it becomes possible to recover past states of the system. It also opens way to solve concurrent changes to the model, by allowing the existence of multiple branches of evolution, and provide disagreement and reconciliation mechanisms [48].

N. Warehousing

So far we have been incrementally empowering an infrastructure — or meta-model — to describe most of our example application. One issue remains though: how should this metadata be represented, accessed, and stored?

We should have in mind that, by describing both the structure and behavior of the system, it is to be interpreted in three distinct phases: (i) during system initialization, a process also known as bootstrapping, (ii) during the construction of objects, a phase typically known as "instantiating the object-model", and (iii) during the assessment and execution of rules.

Warehousing thus aim to hide the details of persistency from the rest of the system, exposing and consuming data and meta-data (i.e., Things), and managing versioning (i.e., through Versions and States). Its behavior can be extended and modified through inheritance and composition, as by the DECORATOR pattern. Transient memory-only, direct data-base access, lazy and journaling strategies (e.g., using CACHES) are just a few examples of existing (and sometimes simultaneous) configurations.

O. Representation

We already discussed that because this information will be readily available for runtime manipulation (e.g., in a database or other external storage mechanism), and not hardcoded, it allows the business model to be updated and immediately reflected in the applications.

Options for storage and manipulation of meta-data include relational and object-oriented databases, Domain Specific Languages, custom XML vocabularies, etc. Direct serialization of the model — e.g., using native language primitives — may simplify system initialization. Domain Specific Languages and XML vocabularies may need the usage of the INTERPRETER and BUILDER patterns.

However, one of the most powerful abilities of an AOM is to allow the end-user himself to introduce (confined) changes to the model at runtime. This raises a number of concerns which the AOM literature does not commonly address, and that may require more elaborate strategies to deal with the

evolution of data and meta-data, and which will be discussed in Section IV.

P. Persistency

The use of object-oriented databases (OODB) simplifies persistency in AOM-based systems, due to the object-oriented nature of the meta-model. Other techniques include model-to-model transformations for relational models, use of the filesystem for storing serialized objects, or BLOBs in databases.

Still, persistency based on static-scheming, such as automatic generation of DDL code for specifying relational databases schema and subsequent DML code for manipulating data, which attempts a semantic correspondence between both models, significantly increases the implementation complexity, particularly when dealing with model co-evolution [49]. This has been long known as object-relational impedance mismatch [50], and evidence of such issues may be observed in the way Object-Relational Mapping (ORM) frameworks attempt to deal with these issues, often requiring knowledge of both representations and manual specification of their correspondence (e.g., Migrations in RoR).

Therefore, though persistency may seem a solved issue, the highly-dynamic nature of AOMs and their inherent mismatch with relational-models suggest that new alternatives need to be investigated and tested, specially when dealing with multiple versions and migrations of data and metadata [49].

Q. Thread of Control

In Figure 6, the AOM BUILDER serves as a Controller for the system, and its key responsibility is to orchestrate the several other components in the framework by establishing a thread of control. It bootstraps the system by loading the meta-model, and the necessary versions of the domain-model from the Warehouse. It manages data requests by interacting with the Warehouse. It also provides several HOOKS to the framework through CHAINS OF RESPONSIBILITY and PLUGINS (e.g., interoperability with third-party systems by allowing subscribers to intercept requests). It is the AOM BUILDER that establishes the THREAD OF CONTROL for an AOM-based system.

R. User-Interface

Adaptive Graphical User-Interfaces (GUI) for AOMs work through inspection and interpretation of the model and by using a set of pre-defined heuristics and patterns [51]. The work in [48] describes an example of a minimalistic workflow for an automated GUI, based on: (i) a set of grouped entry-points declared in the model, and further presented to the user grouped by Packages, (ii) list of the instances by Entity Type or View, which show several details in distinct columns, inferred from special annotations made in the model, (iii) pre-defined automated views inferred

by model inspection (edition and visualization) based on heuristics that consider the cardinality, navigability and role of properties, (iv) generic search mechanisms, (v) generic undo and redo mechanisms, (vi) support of custom panels for special types (e.g., dates) or model-chunks (e.g., user administration), through PLUGINS, etc.

This reactive user-interface also resembles a type of *offline mode*, similar to using version-control systems. User changes, instead of being immediately applied, are stored into the user Changeset, and sent to the main Controller (which would subsequently assert the resulting integrity of applying changes, and provide feedback on behavioral rules). The user can *commit* its work to the system when she wants to save it, review the list of Operations she has made, and additionally submit a descriptive text about her work.

Awareness of the system also makes use of several feedback techniques such as (i) graphics showing the history of changes, (ii) alerts for simultaneous pendent changes in the same subjects from other users, (iii) reconciliation wizards whenever conflicts are detected due to concurrent changes, etc.

S. Towards a Pattern Language

When building systems, there are recurrent problems which have proven, recurrent solutions, and as such are known as *patterns*. These patterns are presented as a three-part rule, which expresses a relation between a certain context, a problem, and a solution. A software design pattern addresses specific design problems specified in terms of interactions between elements of object-oriented design, such as classes, relations and objects [52]. They aren't meant to be applied as-is; rather, they provide a generic template to be instantiated in a concrete situation.

1) Categorization: The growing collection of AOM-related patterns which is forming a domain pattern language [49], [44], [53], is currently divided into six categories (i) Core, which defines the basis of a system, (ii) Creational, used for creating runtime instances, (iii) Behavioral, (iv) GUI, for providing human-machine interaction, (v) Process, to assist the creation of a AOM, and (vi) Instrumental, which helps the instrumentation:

- **Core.** This set of patterns constitutes the basis for an AOM-supported system. The patterns included in this category are Type Square, Type Object, Properties, Accountability, Value Object, Null Object and Smart Variables.
- **Creational.** These patterns are the ones used for creating runtime instances of AOMs: Builder, AOM Builder, Dynamic Factory, Bootstrapping, Dependency Injection and Editor / Visual Language.
- **Behavioral.** Behavioral patterns are used for adding and removing behavior of AOMs in a dynamic way.

They are Dynamic Hooks, Strategy, Rule Object, Rule Engine, Type Cube and Interpreter.

- **GUI.** User-interface rendering patterns have already been mentioned: Property Renderer, Entity View, and Dynamic View. Related to UI there's to add the GUI Workflow pattern.
- **Process.** Includes the patterns used in the process of creating AOMs. An AOM has usually much of a framework in it. The following patterns are good practices when building a framework as well as when building an AOM: Domain Specific Abstraction, Simple System, Three Examples, White Box Framework, Black Box Framework, Component Library, Hot Spots, Pluggable Objects, Fine-Grained Objects, Visual Builder and Language Tools.
- **Instrumental.** Patterns that help on the instrumentation of AOMs, namely, Context Object, Versioning, History and Caching.

2) Core Patterns:

- **Type Object.** As described in [54] and [55], a *TypeObject* decouples instances from their classes so that those classes can be implemented as instances of a class. Type Object allows new "classes" to be created dynamically at runtime, lets a system provide its own type-checking rules, and can lead to simpler, smaller systems.
- **Property.** The Property pattern gives a different solution to class attributes. Instead of being directly created as several class variables, attributes are kept in a collection, and stored as a single class variable. This makes it possible for different instances, of the same class, to have different attributes [45].
- **Type Square.** The combined application of the *Type-Object* and *Property* patterns result in the *TypeSquare* pattern [45]. Its name comes from the resulting layout when represented in class diagram, with the classes *Entity*, *EntityType*, *Attribute* and *AttributeType*.
- **Accountability.** Is used to represent different relations between parties, as described in [45] and [46], using an *AccountabilityType* to distinguish between different kinds of relation.
- **Composite.** This pattern consists of a way of representing part-hole hierarchies, by using the *Rule* and *CompositeRule* classes [54].
- **Strategy.** Strategies are a way to encapsulate behavior, so that it is independent of the client that uses it. Rules are Strategies, as they define behavior that can be attached to a given *EntityType* [54].
- **Rule Object.** This pattern results from the application of the Composite and Strategy patterns, for the representation of business rules by combining simpler elementary constraints [44].
- **Interpreter.** An AOM consists of a runtime interpretation of a model. The Interpreter pattern is used to extract meaning from a previously defined user representation of the model [54].
- **Builder.** A model used to feed an AOM-based system is interpreted from its user representation and a runtime representation of it is created. The Builder pattern is used in order to separate a model's interpretation from its runtime representation construction [54].

3) Graphical User Interface Patterns: The patterns in [53] focus specifically on User Interface (UI) generation issues when dealing with AOMs. In traditional systems, data presented in User Interfaces is usually obtained from business domain objects, which are thus mapped to UI elements in some way. In AOMs business objects exist under an additional level of indirection, which has to be considered. In fact, it can be taken into our advantage, as the existing meta-information, used to achieve adaptivity, can be used for the same purpose regarding user interfaces. User interfaces can this way be adaptive to the domain model in use.

- **Property Renderer.** Describes the handling of user-interface rendering for different types of properties.
- **Entity View.** Explains how to deal with the rendering of *EntityTypes*, and how *PropertyRenderers* can be coordinated for that purpose.
- **Dynamic View.** Approaches the rendering of a set of entities considering layout issues and the possibility of coordinating *EntityViews* and *PropertyRenderers* in that regard.

IV. RESEARCHING ADAPTIVE OBJECT-MODELS

The Adaptive Object-Model and its ecosystem is composed of architectural and design patterns that provide domain adaptability to Object-Oriented based systems. As patterns, they've been recurrently seen in the wild and systematically documented. However, we may argue there isn't enough scientific evidence of any specific benefits due to the lack of rigorous empirical validation. In this section, we raise several research questions about the benefits of AOMs, argue what metrics should be used to support common claims, point to what should be the baseline for such experiments, and underline the need to design them as repeatable packages for independent validation.

A. Epistemology

In order to understand the way software engineers build and maintain complex and evolving software systems, research needs to focus beyond the tools and methodologies. Researchers need to delve into the social and their surrounding cognitive processes vis-a-vis individuals, teams, and organizations. Therefore, research in software engineering may be regarded as inherently coupled with human activity, where the value of generated knowledge is directly dependent on the methods by which it was obtained.

Because the application of reductionism to assess the practice of software engineering, particularly in field research, is very complex (if not unsuitable), we claim that further research should be aligned with a pragmatistic view of truth, valuing acquired practical knowledge. That is, it should be used whatever methods will seem more appropriate to prove — or at least improve our knowledge about — the questions here raised, but mostly through the use of mixed methods, such as (i) (Quasi-)Experiments to primarily assess exploratory questions, which are suitable for an academic environment, and (ii) industrial Case-Studies, as both a conduit to harvest practical requirements, as to provide a tight feedback and application over the conducted investigation.

B. Fundamental Challenges

There are some fundamental questions directly inherited from the current research trends and challenges in the area of Model-Driven Engineering (MDE). A recent research roadmap by France et al. [2] states three driving issues: (i) what forms should runtime models take; (ii) how can the fidelity of the models be maintained; and (iii) what role should the models play in validation of the runtime behavior? Another survey by Teppola et al. [56] synthesize several obstacles related to wide adoption of MDE:

1) Understanding and managing the interrelations among artifacts: Multiple artifacts such as models, code and documentation, as well as multiple types of the same artifact (e.g., class, activity, state diagrams) are often used to represent different views or different levels of abstraction. Subsets of these overlap in the sense that they represent the same concepts. Often because they are manually created and maintained without any kind of connection, consistency poses a problem.

2) Evolving, comparing and merging different versions of models: The tools we currently have to visualize differences among code artifacts are suitable because they essentially deal with text. Models often don't have a textual representation, and when they do, it may not be the most appropriate to understand the differences and to make decisions, particularly if these are to be carried by the end-user.

3) Model transformations and causal connections: Models are often used to either (i) reflect a particular system, or (ii) dictate the system's behavior. The relationships between the system and its model, or between different models that represent different views of the same system, are called causal connections. Maintaining their consistency when artifacts evolve is a complex issue, often carried manually.

4) Model-level debugging: If the model is being used to dictate a system's behavior, enough causal connections must be kept in order to understand and debug a running application at the model-level.

5) Combination of graphical and forms-based syntaxes with text views: Developers and end-users have different preferences concerning textual syntaxes and graphical editors to view and edit models. To this extend, a complete correspondence between each strategy is currently not well supported.

6) Moving complexity rather than reducing it: Model-Driven Engineering is not a "silver-bullet" [57] and as such its benefits must be carefully weighted in context to assess whether the approach will actually reduce complexity, or simply move it elsewhere in the development process.

7) Level of expertise required: It is not clear if the interrelationships among multiple artifacts (which may have different formalisms), combined with the necessary (multiple) levels of abstraction to express a system's behavior actually eases the task of any given stakeholder to understand the impact and carry out a particular change, and to which extent current training in CS/SE courses is adequate.

C. Viewpoints

When researching Adaptive Object-Models, there are always two distinct viewpoints from where we can measure the benefits: (i) the developer viewpoint, which is actively trying to build a system for a specific use-case profile, and (ii) the end-user, which, when provided, will be evolving the system once delivered. The existence of an end-user as a change-agent, although always cited as a benefit of the use of AOMs, should not be taken lightly. What may seem as an excellent way to improve adaptivity to the well-trained developer, it may may reveal as an encumbrance to the end-user, or at worst, a designer's worst nightmare.

We thus suggest that some questions regarding end-user development should be: (i) either specifically researched in the area of AOMs, or (ii) borrowed from other fields of research:

1) End-user perception of the model: The way end-users see their systems is different from the abstraction the developer are used to. Understanding the differences between this two perspectives is essential to provide mechanisms in the user-interface that are suitable, and avoids an higher-level BIG BALL OF MUD.

2) Visual metaphors: We shouldn't expect the common end-user to actually type in a Domain Specific Language to express some new rules they want to insert in the system. Other kinds of visual metaphors should be considered as a proxy for the underlying rule engine. A more detailed discussion can be found in [58].

3) Evolving the model: A tentative, failed, evolution may be disastrous regarding the meaning of data. Mechanisms to recover from mistakes, though already useful to the developer, are paramount to the end-user.

D. Specific Challenges

Although the research in Adaptive Object-Models is a subset of the research in MDE, we think the following questions should be careful assessed and their answers would

contribute to the body of knowledge, particularly when choosing to use (or not) this pattern. Though the authors believe in the capability of Adaptive Object-Models to efficiently cope with several of the stated issues in software development, and this belief has been substantiated both by research on the wider area of MDE, as well as through the studies by the pattern community, we believe that we have much to gain if we could prove that an AOM, despite a pattern, is not an anti-pattern (i.e., an obvious, apparently good solution, but with unforeseen and eventually disastrous consequences):

1) Fitness for purpose: When is an AOM adequate to use? When should the use of an AOM be considered *over-engineering*. What metrics should we base our judgment for applicability?

2) Target audience: What type of developers are best suited for AOMs? Are current developers lacking in specific formation that hinders the usage and construction of AOMs? What about end-users? Are there specific profiles that could point to a more suitable audience?

3) Development speed indicator(s): What is the impact on the usage of AOMs during the several phases of the process? Do developers increase their ability to produce systems? How long is their *start-up* time?

4) Extensibility indicator(s): How easy is to extend an AOM-based system? How long does it take to HOOK a particular customization into the base architecture? Is this dependent on a specific implementation?

5) Quality indicator(s)?: What is the impact on software quality metrics when using AOMs? How does it affect Performance? How does it ensure Correctness? Is Consistency a major factor? What about the Usability of automatically-generated interfaces? How can we improve them?

E. Research Design

It is necessary to adequately define the experimental protocols which assess these claims in a rigorous and sound way. This includes a precise definition of the processes to be followed in industry case studies, as well as in the family of quasi-experiments to be performed in academic contexts. The design of experimental protocols for the industrial case studies, should attempt to cover the whole experimental process, i.e, from the requirements definition for each experiment, planning, data collection and analysis, to the results packaging. Discussion on guidelines for performing and reporting empirical studies have been recently going by the works of Shull et al. [59] and Kitchenham et al. [60]. The typical tasks and deliverables of a common experimental software engineering process can be found in [61].

F. (Quasi-)Experiments

(Quasi-)experiments conducted in an academic context should be randomized, multiple-group, comparison designs, which may be implemented as part of graduate student teams

lab work. One scenario would involve splitting the students into three groups. One group would act as a baseline and use any traditional development methodology and tools to construct and evolve a particular system. The second group would be mandated to develop an AOM-based solution. The third group would have direct access to a framework which already provides a specific infrastructure to build AOM-based systems.

There should be an evaluation of the base skills for every member. For example, was their academic track the same, or did they take courses that could influence the experiments outcome? In this case, it should be taken into consideration subjects such as Software Specification, Agile Methodologies, Formal Methods, Design Patterns, Object-Oriented Programming, Model-Driven Engineering, to mention a few, which could pose a direct influence. In order to guarantee that there is no significant statistical deviation on their base skills, researchers should also use of a subset of computer science related GRE (or similar) questions prior to conducting the selection. There's also the need to ensure that all groups share common skills with respect to metamodeling, compilers, interpreters, and architectural and design patterns, the AOM and its ecosystem, thus specific training them should be taken into consideration.

A (quasi-)experiment may assess several distinct claims, which could match into different phases. For example, researchers could make (quasi-)experiments aligned with two different phases: development and maintenance. In the first phase, a Requirements Specification Report, which would include detailed user stories and UML diagrams (or similar artifacts) that could semi-formalize a particular small system (e.g., around 50 model elements), would be given to all groups. Their task would be to implement a full system using their given technique and restrictions. The time available for pursuing the implementation should be based in effort estimations made by software-engineer professionals. Here, several metrics should be collected and assessed.

The second phase would be pursued after all the systems are finished. A series of small changes (e.g., 10 model elements each) would be separately handled to each group, thus accounting for a change in 50% (this number here is being used rather arbitrarily; the rationale is that it should reflect real-world profiles of high-variable systems). The implementation of each of these series should occur in a more strict laboratory environment (compared to the first phase), with the supervision of researchers and lecturers. For each series relevant data should be collected and assessed.

In order to improve confidence in the results, there should be a repetition of this experiment where the groups would (randomly) switch positions — e.g., the group which was working with the framework-based solution would switch to the baseline approach — for a second round.

G. Data Collection

We propose the usage of metrics such as time, correctness and complexity of the produced artifacts, but it remains the specification of tools and methods to collect the data during experimentation that reveal non-intrusive for practitioners. A possible technical solution to those using the framework would be to instrument it in order collect as much significant data as possible, including all steps of model evolution. The instrumentation of Integrated Development Environments (IDE) is also a possibility.

H. Independent Validation

The independent experimental validation of claims is not as common in Software Engineering as in other, more mature sciences. Hence, we stress the need to build reusable experimental packages that support the experimental validation of each claim by independent groups. The family of (quasi-)experiments should be performed in different locations, and lead by different researchers, but based on the same experimental package, in order to enhance the ability to integrate the results obtained in each of the quasi-experiments, and allow further meta-analysis on them.

V. Conclusion and Future Work

Software development face an increasing difficulty in acquiring, inferring, capturing and formalizing requirements, particularly in domains that rely on constant adaptation of their processes to an evolving reality, and thus what support they expect from software. This type of software is said to be *incomplete by design* and thus require a *design for incompleteness* approach. Agile processes have intensified their focus on a highly iterative and incremental approach, accepting and embracing the relevance of efficiently coping with *change*.

While methodologies struggle to make the process and the development team more suitable, we are looking forward to what form should agile software take. One example of such software is the *WikiWikiWeb*, which embrace *incompleteness*, by relying on fundamental principles such as organic, overt, tolerant and observable. We conjecture about attempting the same principles in other systems, and those based on Adaptive Object-Models reveal a good candidate.

The Adaptive Object-Model and its ecosystem is composed of architectural and design patterns that provide adaptability to systems based on object-oriented domain models. AOMs, Software Product Lines, Model-Driven Engineering, and Frameworks are all solutions for a common set of problems, such as increase software reuse and reduce time-to-market. But AOMs, by leveraging the concept of adaptability outside the development team, empower end-users to introduce (confined) changes to the model during run-time, and thus control themselves the evolution of their own tool.

As patterns, they've been recurrently seen and documented. However, this fact doesn't seem enough to provide sufficient scientific evidence of their benefits, both to the developer and to the end-user. This may be due to the lack of sufficient empirical validation published upon the use of AOMs, such as detailed case-studies and (quasi-)experiments. In this work, we have raised several research questions that address the benefits of AOMs. We have argued what metrics should be used to support these claims, and we have pointed to what should be the baseline for such experiments, including designing them as repeatable packages for independent validation.

References

[1] H. S. Ferreira, A. Aguiar, and J. P. Faria, "Adaptive object-modelling: Patterns, tools and applications," *Software Engineering Advances, International Conference on*, vol. 0, pp. 530–535, 2009.

[2] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 37–54.

[3] J. Arlow, W. Emmerich, and J. Quinn, "Literate modelling — capturing business knowledge with the uml," in *UML'98: Selected papers from the First International Workshop on The Unified Modeling Language*. London, UK: Springer-Verlag, 1999, pp. 189–199.

[4] J. Krogstie, A. L. Opdahl, and G. Sindre, Eds., *Advanced Information Systems Engineering, 19th International Conference, CAiSE 2007, Trondheim, Norway, June 11-15, 2007, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4495. Springer, 2007.

[5] M. Voelter, "A catalog of patterns for program generation," in *Proceedings of the Eighth European Conference on Pattern Languages of Programs*, Jun 2003.

[6] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe, "The architecture of a uml virtual machine," in *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2001, pp. 327–341.

[7] R. Garud, S. Jain, and P. Tuertscher, "Incomplete by design and designing for incompleteness," in *Organization studies as a science of design*, Marianne and G. Romme, Eds., 2007.

[8] N. Anquetil, K. M. de Oliveira, K. D. de Sousa, and M. G. Batista Dias, "Software maintenance seen as a knowledge management issue," *Inf. Softw. Technol.*, vol. 49, no. 5, pp. 515–529, 2007.

[9] L. Williams and A. Cockburn, "Guest editors' introduction: Agile software development: It's about feedback and change," *Computer*, vol. 36, pp. 39–43, 2003.

[10] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[11] B. Foote and J. Yoder, "Big ball of mud," in *Pattern Languages of Program Design*. Addison-Wesley, 1997, pp. 653–692.

[12] A. Kay, "The early history of smalltalk," *ACM SIGPLAN Notices*, Jan 1993.

[13] C. J. Neill and P. A. Laplante, "Paying down design debt with strategic refactoring," *Computer*, vol. 39, pp. 131–134, 2006.

[14] G. Jilles Van, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture*. Washington, DC, USA: IEEE Computer Society, 2001, p. 45.

[15] K. Andresen and N. Gronau, "An approach to increase adaptability in erp systems," in *Proceedings of the 2005 Information Recources Management Association International Conference*. Idea Group Publishing, May 2005, pp. 883–885.

[16] P. Meso and R. Jain, "Agile software development: Adaptive systems principles and best practices," *IS Management*, vol. 23, no. 3, pp. 19–30, 2006.

[17] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds., *Software Engineering for Self-Adaptive Systems*. Berlin, Heidelberg: Springer-Verlag, 2009.

[18] W. Cunningham, "WikiWiki," 1995. [Online]. Available: http://c2.com/cgi/wiki

[19] A. Aguiar, "A minimalist approach to framework documentation," Ph.D. dissertation, Faculdade de Engenharia da Universidade do Porto, Sep. 2003.

[20] W. Cunningham, "Wiki design principles." [Online]. Available: http://c2.com/cgi/wiki$?WikiDesignPrinciples

[21] Wikipedia, "Abstraction," July 2010. [Online]. Available: http://en.wikipedia.org/wiki/Abstraction

[22] J. Spolsky, "The law of leaky abstractions," Nov 2002. [Online]. Available: http://www.joelonsoftware.com/articles/LeakyAbstractions.html

[23] R. CAMERON and M. ITO, "Grammar-based definition of metaprogramming systems," *ACM Transactions on Programming Languages and Systems*, Jan 1984.

[24] W. Cazzola, "Evaluation of object-oriented reflective models," *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98), in 12th European Conference on Object-Oriented Programming (ECOOP'98)*, Jan 1998.

[25] L. Levy, "A metaprogramming method and its economic justification," *IEEE Transactions on Software Engineering*, Jan 1986.

[26] T. Stahl and M. Völter, "Model-driven software development: Technology, engineering, management," 2006.

[27] T. Parr and R. Quong, "ANTLR: A Predicated-LL(k) parser generator," *Journal of Software Practice and Experience,*, vol. 25, no. 7, pp. 789–810, July 1995.

[28] S. C. Johnson, "Yacc: Yet another compiler-compiler," Bell Laboratories, Tech. Rep., 1978.

[29] W. Schuster, "What's a ruby dsl and what isn't?" Jun 2007. [Online]. Available: http://www.infoq.com/news/2007/06/dsl-or-not

[30] M. P. Ward, "Language-oriented programming," *Software — Concepts and Tools*, vol. 15, no. 4, pp. 147–161, 1994.

[31] *Software product lines: practices and patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[32] R. Pawson, "Naked objects," Ph.D. dissertation, University of Dublin, Trinity College, Jun 2004.

[33] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, Aug 2003.

[34] OMG – MDA, "Model driven architecture." [Online]. Available: http://www.omg.org/mda/

[35] OMG – MOF, "MetaObject Facility." [Online]. Available: http://www.omg.org/mof/

[36] D. Roberts and R. Johnson, "Evolving frameworks: A pattern language for developing object-oriented frameworks," in *Proceedings of the Third Conference on Pattern Languages and Programming*, vol. 3, 1996.

[37] G. Roy, J. Kelso, and C. Standing, "Towards a visual programming environment for software development," *Proceedings on Software Engineering: Education & Practice*, Jan 1998.

[38] K. Czarnecki, "Overview of generative software development," *Unconventional Programming Paradigms (UPP)*, Jan 2004.

[39] G. Kiczales and E. Hilsdale, "Aspect-oriented programming," in *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2001, p. 313.

[40] A. Dantas, J. Yoder, P. Borba, and R. Johnson, "Using aspects to make adaptive object-models adaptable," *Research Reports on Mathematical and Computing Sciences*.

[41] K. Czarnecki and U. Eisenecker, "Generative programming: Methods, tools, and applications," p. 832, Jan 2000.

[42] J. Yoder, F. Balaguer, and R. Johnson, "Adaptive object-models for implementing business rules," *Urbana*, 2001.

[43] J. W. Yoder, F. Balaguer, and R. Johnson, "Architecture and design of adaptive object-models," *ACM SIG-PLAN Notices*, vol. 36, pp. 50–60, Dec. 2001.

[44] L. Welicki, J. W. Yoder, R. Wirfs-Brock, and R. E. Johnson, "Towards a pattern language for adaptive object models," in *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. New York, NY, USA: ACM, 2007, pp. 787–788.

[45] M. Fowler, "Analysis patterns: Reusable object models," 1997.

[46] A. Arsanjani, "Rule object: A pattern language for adaptive and scalable business rule construction," *Proceeding of PLoP*, 2000.

[47] J. Yoder, B. Foote, D. Riehle, and M. Tilman, "Metadata and active object models," *Conference on Object-Oriented Programming*, 1998.

[48] H. S. Ferreira, F. F. Correia, and A. Aguiar, "Design for an adaptive object-model framework: An overview," in *Proceedings of the 4th Workshop on Modelsrun.time, held at the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, October 2009.

[49] H. S. Ferreira, F. F. Correia, and L. Welicki, "Patterns for data and metadata evolution in adaptive object-models," *Proceedings of the Pattern Languages of Programs*, 2008.

[50] S. Ambler, *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[51] L. Welicki, J. W. Yoder, and R. Wirfs-Brock, "A pattern language for adaptive object models: Part i - rendering patterns," in *PLoP 2007*, Monticello, Illinois, 2007.

[52] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Elements of reusable object-oriented software," p. 395, Jan 1995.

[53] L. Welicki, J. Yoder, and R. Wirfs-Brock, "A pattern language for adaptive object models: Part i-rendering patterns," *hillside.net*, 2007.

[54] R. Johnson and B. Woolf, "Type object," *Addison-Wesley Software Pattern Series*, Jan 1997.

[55] J. Yoder, F. Balaguer, and R. Johnson, "Architecture and design of adaptive object-models," *ACM SIGPLAN Notices*, Jan 2001.

[56] S. Teppola, P. Parviainen, and J. Takalo, "Challenges in deployment of model driven development," *Software Engineering Advances, International Conference on*, vol. 0, pp. 15–20, 2009.

[57] J. B. F.P., "No silver bullet essence and accidents of software engineering," *Computer*, vol. 20, pp. 10–19, 1987.

[58] B. A. Nardi, *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA, USA: MIT Press, 1993.

[59] F. Shull, J. Singer, and D. I. Sjøberg, *Guide to Advanced Empirical Software Engineering*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[60] B. Kitchenham, H. Al-Khilidar, M. A. Babar, M. Berry, K. Cox, J. Keung, F. Kurniawati, M. Staples, H. Zhang, and L. Zhu, "Evaluating guidelines for reporting empirical software engineering studies," *Empirical Softw. Eng.*, vol. 13, no. 1, pp. 97–121, 2008.

[61] M. Goulao and F. B. Abreu, "Modeling the experimental software engineering process," in *QUATIC '07: Proceedings of the 6th International Conference on Quality of Information and Communications Technology*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 77–90.