Core Patterns of Object-Oriented Meta-Architectures

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

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

Joseph Yoder
Refactory, Inc.
joe@joeyoder.com

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

ABSTRACT

Meta-architectures, also known as reflective architectures, are a specific type of software architectures that are able to inspect their own structure and behavior, and dynamically adapt at runtime, thus responding to new user requirements or changes in their environment. In object-oriented programming, these architectures rely on a small set of core concepts that provide them the means to describe themselves, thus becoming "closed". These three core patterns can be found in almost every object-oriented meta-architecture: EVERYTHING IS A THING, CLOSING THE ROOF, and BOOTSTRAPPING. By delving into the inherent problems they try to solve, and the forces that shape those problems, a developer will improve his ability to adequately make architectural and design choices to build and evolve systems with high-adaptability needs.

Keywords

Model driven software engineering, Adaptive object models, Design patterns, Meta-modeling, Meta-programming.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Patterns

1. INTRODUCTION

Meta-architectures, also known as reflective-architectures, are software systems architectures that rely on meta-data and reflection mechanisms in order to dynamically adapt, at runtime, to new (or changed) user requirements [3, 4, 19]. This is achieved by exposing the domain model as a first-class artifact able to be changed and shaped through the system itself.

1.1 Motivation

Back in 2005, some of the authors were leading a software project consisting on the construction of a geographical information system which would help to manage records of architectural and archeological heritage, their inventory and the associated business processes. Although our development methodology was a slight variant of *eXtreme Programming* [2], we were considerably restricted in applying some of the practices for this particular project: (i) it was bided, so the cost was fixed, (ii) we could not reduce the scope, although it was systematically enlarged, (iii) we could not have an on-site costumer, and (iv) the project needed to be considered a success[1].

Our problems began in the very first official meeting we had. Because the bid was made years before the official start of the project, the stakeholders' understanding had evolved since then. Therefore, the initial requirements were no longer a reflection of their current manual processes. Our contract enforced the delivery of a *requirement analysis* document which had to undergo validation before starting the development. And so we began the task of collecting requirements... for two years.

At a glance, this seems a good example of how it should not be done; two years collecting requirements smells like a good old *waterfall*. However, our mere presence was directly contributing to this status. We started to question things the stakeholders took for granted, and in the process of formalizing their practices, we uncovered inconsistencies which could not be solved promptly. This resulted in a series of analysis iterations, where the stakeholders had to re-think their goals, their processes, and their resulting artifacts, before we could even synthesize a coherent domain model.

At the end of those two years, and with a conceptual model of over two hundred concepts, we were strongly convinced of one thing: no matter how much time we invested in analysis, the resulting system would hardly ever be considered finished[2]. As an example, consider the following requirement: users needed to collect the physical properties of archeological artifacts found in excavations. At first, *length*, *width* and

[1] Even if it came to a point of non-profitability.
[2] Truthful to the agile principles.

height seemed a good measure. But some artifacts are highly irregular, like a three thousand year old *jar*. For these, *weight* and *material composition* are greatly more useful. Other artifacts, like *coins*, are very regular and rely on different properties, like *radius* and *thickness*. Then we have things in-between, such as *plates*. The more we categorized, the more complex and longer the hierarchy would become, without any confidence we would be able to cover all cases and exceptions. Our model was being haunted by **accidental complexity**[3], and a simple solution urged objects to be characterized by the end-user according to a pre-defined set of properties (which were not pre-defined at all). Of course users are no programmers, so they needed to add new properties and create new hierarchies *on-the-fly* from inside the application, without being explicitly aware of the underlying model.

Nowadays we have a name – Incomplete by Design [9] – and a software architecture – Adaptive Object-Models [19] – for this kind of systems. The irony of this story was that the final application converged to an AOM without its developers actually knowing it was one. Only some years later they have found literature on the subject, a fact that further validates the AOM as a pattern. We will make usage of this story throughout this paper to illustrate parts and pieces of our patterns.

1.2 Technical Description

Let's consider the information that some particular *Archeological Survey* in our system is named *Survey of the Coliseum*, and another one named *Survey of the Parthenon*, is called data for the purpose of using it as an information system for video renting. We could hypothetically take the set of objects that account for **data** (normally called **instances**) and name it the **meta-level-zero** (M_0) of our system.

The way we would typically model such a simple system in an object-oriented language would be to create a class named *Archeological Survey*, with an atribute named *Title*. This information is meta-data (it is data about data itself): it conveys a very crucial information, which is the data's structure (and meaning), for the purpose of specifying an executable program. We could draw a line around these things that represent information about other things – classes, properties, etc. – and call them **meta-level-one** (M_1), or simply **model**.

But, what exactly is a class, or a property? What is the meaning of calling a method, or storing a value? As the reader might have guessed, once again, there is structure behind structure itself – an **infrastructure** – and the collection of such things may be called the **meta-level-two** (M_2), or **meta-model** for short (i.e., a model that defines models), which is composed of *meta-classes*, *class factories*, and other similar artifacts.

This separating line between data, model and meta-model is blurred when speaking about **meta-data**, in the sense that everything is, ultimately, data; only its purpose is different. What may be considered the model in one context, may be seen as data in another, e.g., the compiler.

[3]Complexity that arises in computer artifacts, or their development process, which is non-essential to the problem to be solved.

In spite of this, it's worthwhile to establish a consistent vocabulary that allows us to reason about these topics. When we talk about *data* (or instances) we are referring to M_0 – bare information that doesn't provide structure. By *model* we are referring to M_1 – its information gives structure to data. By *meta-model* we are referring to M_2 – information used to define the infrastructure. And so on...

Ultimately, depending on the system's purpose, we will reach a level which has no layer above. This "top-most" level doesn't (yet) have a name; in MOF [14] it is called a **meta-meta-model**, due to being the third model layer[4]. This building up of levels (or layers), where each one is directly accountable for providing structure and meaning to the layer below is known as the **Reflective Tower**, a visual metaphor that can be observed in Figure 1.

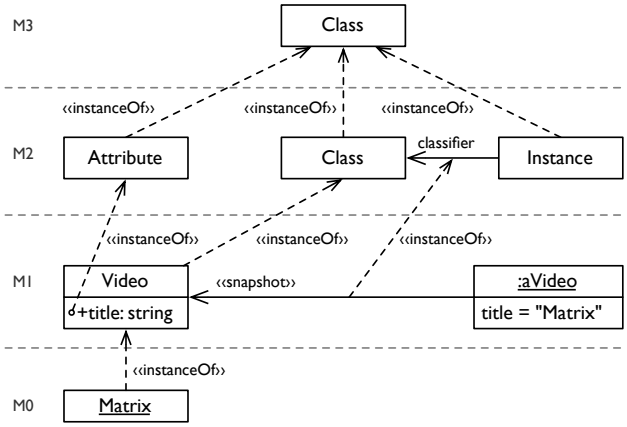

Figure 1: The Reflective Tower of a video renting system, showing four layers of data.

All this would not be very useful if it did not have a purpose. We already mentioned the compiler, whose task is to read a particular kind of information (known as **source code**) and translate it into a set of structures and instructions (known as a **program**), which would later be executed by a computer – a process known as **compilation**. The compiler acts as a processing machine: the input goes into one side, and the outcome comes from the other. Once the compiler has done its job, it is no longer required, and so it does not **observe** nor **interact** with the final program. Should we wish to modify the final program, we would need to change the source code and hand it again to the compiler.

Now let us suppose we wanted to add a new property to a *Video*, like the name of its *Director*, or create new sub-types of videos as needed, like *Documentary* or *TV Series*, each one with different properties and relations. In other words, what if we need to **adapt** the program as it is running? For that, we would need both to observe and interact with our running application, modifying its structure *on-the-fly* (the technical term is during **run-time**). The property of systems that allow this to be performed is called **Reflection**, i.e., the ability of a program to manipulate as data something representing the state of the program during its own execution. The two

[4]Would it be the sixth, we seriously doubt anyone would apply the same prefix five times.

mentioned aspects of such manipulation, observation and interaction, are respectively known as **introspection**, i.e., to observe and reason about its own state, and **intercession**, i.e., to modify its own execution state (**structure**) or alter its own interpretation or meaning (**semantics**).

The technique of using programs to manipulate other programs, or the running program itself, is known as **meta-programming**, and the high-level design of such system is called a **meta-architecture**. Granted, there has been some debate on the exact meaning of this Humpty-Dumpty word[5]. Joseph et al. defined it as *architectures that can dynamically adapt at runtime to new user requirements (a.k.a. "reflective architectures")* [19]. Ferreira et al. pointed to an *architecture of architectures, i.e. an abstraction over a class of systems which may rely on reflective mechanisms* [5]. This seemingly disagreement is due to the ubiquitous *meta* prefix, which can be understood as being applied to the word architecture (i.e., an architecture of architectures), or as a subset categorization (i.e., those architectures that rely on meta-* mechanisms). For the purpose of this work, we should consider **a meta-architecture as a software system architecture that heavily relies on reflective mechanisms**.

1.3 General Forces

The construction of this kind of systems is under the influence of a set of forces; concerns that should be weighted in order to achieve a good solution. Because the patterns described in this paper are deeply connected, most of them share a good amount of forces in common. Figure 2 shows a schematic relationship among some of the following forces that are generally relevant to meta-architectures.

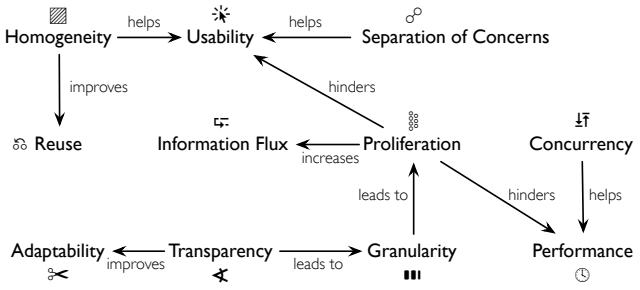

Figure 2: The relationship among forces of object-oriented meta-architectures.

1. **Transparency.** How much of the underlying system is available through reflection? In other words, to which degree does the infrastructure expose its own mechanisms for observation and manipulation? We may regard a system which is more transparent to improve *usability* in the sense that adds more power to it (hence, the user is able to do *more*). On the other side, a lot of transparency exposes details that can hinder its understandability, and consequently, its *usability*. Likewise, increasing transparency usually improves the system's adaptability, at the expense of an increase in granularity (which will, ultimately, hinder performance).

2. **Usability.** This is defined as *"the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use"* [10]. In this sense, meta-architectures[6], have two type of target users: (i) those which develop and evolve the infrastructure, and (ii) those who use the public facilities of the infrastructure to develop domain specific systems. Design choices may have different influences on the usability of different target users. This force is actually a result of several other forces, i.e., homogeneity, seperation of concerns, proliferation and transparency.

3. **Separation of Concerns.** This is a general design force the establishes the fact that a particular functionality of a systems should be the concern of a different component – in this case, a different level of the reflective tower. For example, M_1 should be reserved to only express domain-level concerns, but most systems regard it as immutable during runtime. Thus, accidental complexity arises when this level is tweaked by non-domain concerns which should belong to M_2. Having a clear separation of these levels reduces accidental complexity and thus helps on using the system (from the programmer's point of view).

4. **Concurrency.** Is a general counter-force to reflective meta-architectures, mainly due to integration mismatch (i.e., tight interconnection among different level artifacts, causal connection among entities to provide consistency in the meta-representation of the system, information flux among levels, etc.). Concurrency is mainly relevant due to *performance* and distributivity concerns, and has been a common issue in database design.

5. **Granularity.** Represents the smallest aspect of the base-entities of a computational system that are represented by different meta-entities, depending on the reflectivity scope — structural and/or behavioral. Typical granularity levels are classes, objects, properties, methods and method calls. The particular choice of the level of granularity is driven by its *transparency*, and has consequences on the resulting systems' object *proliferation* and *performance*.

6. **Proliferation.** Increasing the reflectivity *granularity*, e.g., by representing method calls as objects, leads to object proliferation, in the sense that more elements exist to represent the system's state. Likewise, more elements typically means more communication among them, increasing *information flux* and likely hindering overall *performance*.

7. **Information Flux.** Measures the amount of information that is exchanged between elements of a system to perform a desired computation. Depending on the meta-architecture design, instances typically exchange information with its class, classes with their meta-classes, and so on. The more objects the system

[5]From Lewis Carrol's *Down the Rabbit Hole*, when Humpty-Dumpty explains to Alice: *"when I use a word, it means just what I choose it to mean, neither more nor less"*.

[6]Particularly Adaptive Object-Models.

has to represent it's underlying state (both structural and computational), the more information is needed to be exchanged among them.

8. **Performance.** This is also a general engineering force that may mean short response time, high throughput, low utilization of computing resources, etc. A high *proliferation* of system elements may have an impact on performance, as more system resources will be required. On the other hand, the ability to perform *concurrent* computations usually improves performance.

9. **Adaptability.** Characterizes a system that empowers end-users without or with limited programming skills to customize or tailor it according to their individual or environment-specific requirements. The more transparent a reflective system is, the more it can be tailored.

10. **Reuse.** Is the ability of using existing artifacts, or knowledge, to build or synthesize new solutions, or to apply existing solutions to different artifacts. For example, one can reuse the persistency engine, typically tailored to persist data, to also persist model and meta-model elements. Reusing generally leads to a reduce of overall systems complexity and improves usability.

11. **Homogeneity.** A system's parts are more interchangeable if it's more homogeneous, leading to a higher capacity for reuse. A homogeneous system can also be understood a lot easier, contributing to its usability.

1.4 Pattern Thumbnails

TypeSquare [19] is one of the core patterns for adaptive object-models. It supports turning into meta-data level M_1 and the levels above it, so that the different layers may be handled by the system at runtime.

Figure 3 shows the three patterns here presented, how they relate with each other, and how they relate to TypeSquare, extending the pattern language for adaptive object-models [4, 7, 11, 12, 17, 18]. It is worth noting that the patterns in this language are classified according to more categories in addition to the *architectural* category, to which the patterns presented in this paper belong, such as *structural*, *behavioral*, *architectural*, *interaction*, *creational*, *evolution*, *construction* and *support* [4].

1. **Everything is a Thing.** Which unifies multiple representations of the same underlying concept into a single abstraction.

2. **Closing the Roof.** A pattern that encloses the structure and meaning of a meta-architecture by stopping the seemingly infinite escalation of meta-levels.

3. **Bootstrapping.** Which provides a way for enclosed structures to define themselves, by relying on a (small) set of basic definitions, upon which it's possible to build more complex structures.

In this paper we we use Christopher Alexander's pattern language (APL) format [1], instead of the more commonly

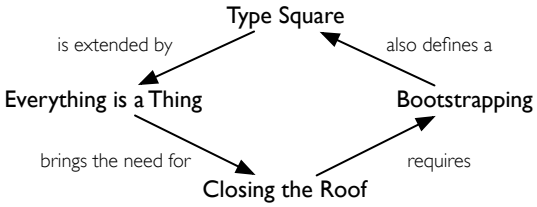

Figure 3: Patterns of object-oriented meta-architectures and their relationship.

used variants of the Gang of Four [8]. Although recognizing the several benefits of the latter, including a more methodological partitioning of the pattern, we feel that the APL form results in a more fluid, narrative-like structure.

Some typographical conventions are used to improve the readability of this document. Patterns names always appear in SmallCase style. Whenever referring to domain elements, e.g., class names, they are printed using `fixed-width` characters. If not otherwise specified, the graphical notation used complies to the latest versions of UML [16] and OCL [15] available at the date of publication (v.2.3).

1.5 Target Audience

The main goal of this paper is to present a collection of patterns that address fundamental concepts underlying meta-architectures. They are intended for those (developers) building or trying to understand the inner workings of such systems, among which may be (a) those whose interest is in the design of programming or specification languages, and (b) others that aim to improve their systems' adaptivity. We hope both find these patterns useful.

The secondary purpose of this work is to unveil some of the magic that seems to hover anything that is prefixed by *meta*. Tim Peters, a python guru, once said [13]:

> *"[Metaclasses] are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why)."*

While we respect, and to some extent agree with Tim's remarks, we have seen many developers scared away by this meme of "forbidden kingdom". The net result is a class of awfully designed systems and too many hours "reinventing the wheel", mainly due to the overall lack of education in the practical application of *meta* techniques. Should this paper help those 1% of users that actually need them, but don't (yet) know, then it has served its secondary purpose.

2. PATTERN I: EVERYTHING IS A THING

Also known as *Universal Object*, *Everything is an object*, *Meta-class*.

The system, with its several types of composing parts, needs to be adapted. Meta-architectures make use of elements available at runtime (i.e., models and meta-models) to specify the

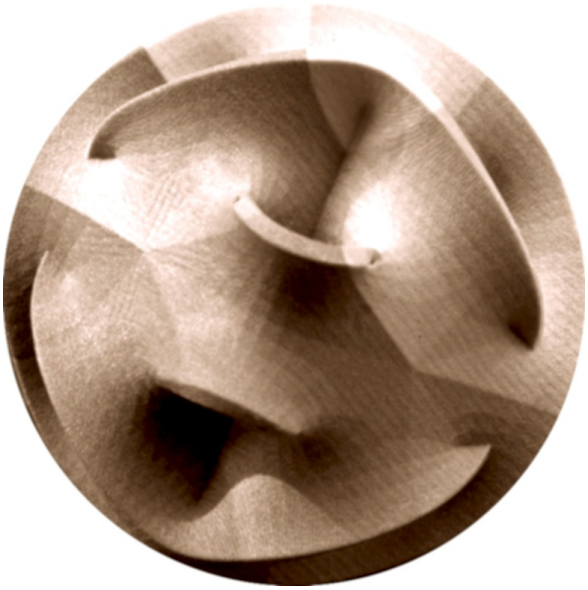

Figure 4: A 3D projection of a Calabi-Yau manifold from superstring theory – an hypothesis that explains every entity our universe is made of.

system's behavior. The system's data is observed and manipulated according to such elements, addressing concerns such as Persistency, Behavioral Rules, Graphical User-Interfaces, and Communications, among others.

Back to the story started in Section 1.1, our system began as a simple variant of the TypeSquare pattern that included attributes, relations, compositions, etc. In fact, it was heavily inspired in UML class diagrams and we were trying to automate as much as possible[7]. At first it was sufficient to store the model description (i.e., `EntityTypes`, `AttributeTypes`, etc.) in a separate XML file, and distribute it over client applications and load it at start-up. Truth be told, to modify the domain model we had to modify the XML file, so there was not that much run-time "adaptivity". There was also a "mapper", with the purpose of interpreting the XML file into runtime elements. Then we had a GUI engine which followed a set of heuristic rules and was able to automatically create a user interface by, like everything else, observing the system's definition.

The lack of **homogeneity** was a problem first spotted with the need to actually manipulate the domain model at runtime. Although easy to deal with, the round-trip to XML was ugly. Also, changing the name of an `AttributeType` or of an `EntityType` required specialized operations, such as `ChangeAttributeTypeName` or `ChangeEntityTypeName`, that established the degree of **transparency** at the model level, but what was really bugging us was that logic was duplicated all around. The GUI engine inferred the user interface to manipulate the data level, but the same rules were hardcoded for the manipulation of the model level. Then we realized that relying on XML to persist the model would not work well in a **concurrent** environment. We asked ourselves: why exactly do we have two types of representation (effectively, two ways of describing) for the same system? If we have the infrastructure to manipulate data, why don't we **reuse** it to manipulate meta-data? In other words:

How to represent all that needs to be reflected upon?

Clearly, we were lacking a fundamental, **unifying** principle. We have a system that observes and manipulates data, but it cannot do the same for meta-data? Why may we use a certain operation to change the attribute value of any instance, like setting the name of a person to John, but a different operation is required to change the name of a model element? Why could the data be stored in a warehouse, but needed the XML to store the model? We had **decoupled** the system from the domain, but we were coupled to what we believed to be a fixed structure; a false belief, since it soon needed to evolve. Our implementation pointed to a system that would need a large number of specific operations and components to manipulate the meta-level, and that number would increase in direct proportion with the system's **transparency**. What was so different between elements of M_0 and elements of M_1? The solution was right in front of us: the system knew how to manipulate *instances*, so we needed to make the elements of our model to also be *instances*.

Therefore, **make all system's elements specializations of a single concept, regardless of their model level.** These highly generic concepts are *Things* (or *Instances*, or *Objects*...). They are a single, unifying, primitive structure, as seen in Figure 5. To manipulate data or model elements, the system always relies on the manipulation of *Things*, that have a common set of basic capabilities for their own observation and manipulation. By **homogenizing** these concepts, the mechanisms that deal with such generalizations don't need to be specific to every kind of entity. For example, setting the name of a type is performed as setting the attribute called *name* of that instance. Consequently, this increases the degree of reflection **transparency** of the system.

Lets suppose that the persistency mechanism focuses on loading/saving States of *Things*, then the same mechanism can be reused for both levels (whether they are base-level objects, or types). This is also valid for graphical user-interfaces (GUI) and other features relying on the system's reflective properties. A known use is the Oghma framework [5, 4, 6], which is able to render a GUI for editing meta-levels. This GUI is dynamically generated using the same rules as those used for the base-level. For example, every enumeration is rendered as either a combo-box (if the property has an upper-bound cardinality of 1), or a check-list (for more than 1). Because the concept of enumeration is equal both in the user defined model (e.g., the gender of a person) and in the system's meta-model (e.g., the rule of an association), both are rendered in the same way.

The following code is a snippet of a C# unit-test, asserting

[7] In other words, we were trying to extract as much information as possible captured in UML diagrams, in order to automatically address several system's concerns, such as user-interface generation, data persistency, data constraints, etc.

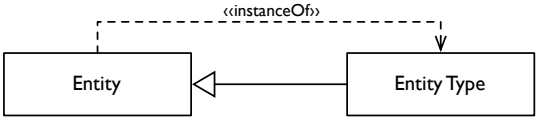

Figure 5: Extending the TypeSquare to implement this pattern, by making EntityType a specialization of Entity.

several properties that hold after implementing this pattern:

```
1 var m = new MetaModel();
2 var entity = m.OfType<Entity>().ByIdOrDefault("entity");
3
4 Expect(entity, Is.Not.Null, "There is an Entity named
      Entity.");
5 Expect(m.ToList(), Is.All.AssignableFrom<Thing>(), "
      Everything is a Thing.");
6 Expect(m.ToList().All(t => t.Meta.Identity == entity.
      Identity), Is.False, "There must be things beside
      Entities.");
7 Expect(m.Where(t => t.Is(entity)), Is.All.AssignableFrom<
      Entity>(), "All Things which have Entity as Meta are
       typed as Entities.");
8 Expect(name.Meta.Meta, Is.EqualTo(entity), "The Meta-Meta
       of any Thing is the Entity named Entity.");
```

Line 1 creates a new container loaded with the system's basic infrastructure. Line 2 finds and strongly types a `Thing` (more specifically, an element of the meta-model) with an Identifier named `Entity`. Line 4 and 6 are sanity checks. Line 5 states that everything that is defined inside the container derives from `Thing`. Line 7 checks that if the model says the meta of any `Thing` is an `Entity`, then the infrastructure ensures it is typed as one. Finally, line 8 checks for meta-circularity definition, which will be discussed in Section 3. The following snippet shows a usage of these features:

```
1 var car = entity.New<Entity>();
2 var attr = m.Get<Entity>("attributetype");
3 var c1 = car.New<Thing>();
4
5 Expect(c1.Violations, Is.Empty);
6
7 var vehicle = entity.New<Entity>();
8 var p = attr.New<AttributeType>(m, "name");
9 p.Owner = vehicle;
10 p.lowerBound = 1;
11
12 car.ParentEntity = vehicle;
13
14 Expect(c1.Violations, Is.Not.Empty);
```

Lines 1 – 3 create a new entity `car` and instantiates it. Line 5 verifies there are no violations for that instance. Lines 7 – 10 create a new entity `vehicle`, with a mandatory attribute. Line 12 changes the inheritance of `car`, and 14 checks that there is now a reported violation due to the mandatory parent entity attribute.

There are some **liabilities** to this pattern, which are direct consequence of the level of transparency. The model can be changed in many more ways than if we don't have specialized mechanisms to manipulate it. This results in an higher coupling between meta-levels, mainly due to an increase of **information flux**. For example, instead of a Type having a specialized field to hold its name, it would have to rely on holding it in a separate object (attribute), which is defined by its meta-type. The type would thus need to exchange information with the meta-type to access its own name. Considering that the model may change anytime, the same thing is even more evident with base-level objects. The fact that more objects are needed to hold basic properties of a system leads to what is known as object proliferation. Both information flux and object proliferation may contribute to a decrease in the overall performance of the system.

In set theory, everything is a set. In LISP, everything is a list. In the object-oriented world, everything is an object. Well, not quite everything – there are binary relations, function applications, and message passing. But the principle still applies, in the sense that there is a single, unifying, primitive aspect (set, list, object) defining the fundamental underlying structure.

Known uses of this pattern include the Meta Object Facility (MOF), pure Object-Oriented languages like Smalltalk, and adaptive object-model frameworks, such as Oghma [4, 5]. Both the Memento pattern and the History of Operations pattern can be used for storing the *States* of *Things* regardless of their underlying model level.

3. PATTERN II: CLOSING THE ROOF

Also known as *Self-Compliance*, *Rooftop*, *Idempotence*.

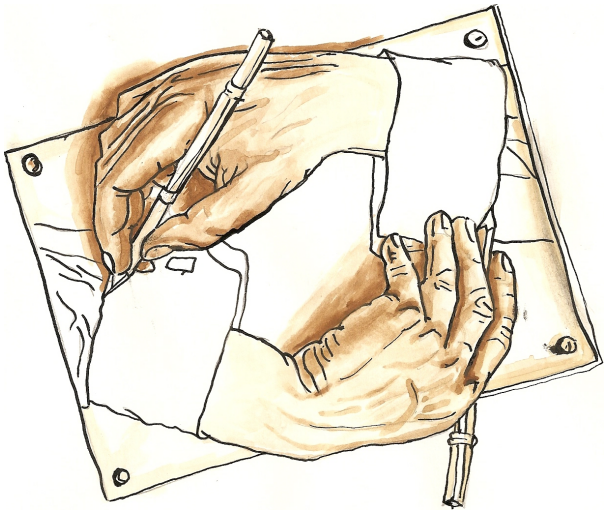

Figure 6: A famous painting by Escher, where two drawn hands seem to protrude out of the drawing, becoming the real hands that would be drawing one another.

By seeing the model as data, one can use Everything is a Thing to manipulate the several model levels using the same mechanisms. But, whenever we raise up a level, we find ourselves needing another (probably more abstract) level to describe it.

Continuing the story in Section 1.2, we implemented Ev-

erything is a Thing by making all objects directly inherit from Thing. Attributes and Relations were stored in slots (just like Ruby and Smalltalk), and accessed by Methods. Because the system needed to be adaptable, we designed it much like a dynamic language, so that when a method was invoked[8], the meta-class was responsible for dispatching it to the appropriate handler. This meant that the meaning of a method call of an instance was given by its meta-class.

We are dealing with three levels here: (M_1) the instance, (M_2) its class and (M_3) its meta-class. The semantics of a particular level is given by the level above. But what gives M_3 its meaning? As we were, M_3 was hardcoded in our application. In other words, at some point we decided that the concept of an "entity" or "class" would not change. But this isn't necessarily true. One simple example where extending M_3 makes sense is when defining a *static method*, i.e., a method that operates over a class and hence should belong to the meta-class. Should we need an M_4 to define a *fixed structure* so that we could adapt the M_3? This points towards a potentially unbounded number of levels (since each level requires an higher – more or equally abstract – level to describe it), thus resulting in a seemingly infinite escalation. In other words:

How do we stop a seemingly infinite escalation of meta-levels? We could devise a system were an infinite escalation of meta-levels would be consistent to its semantics, but, pragmatically, computations are more useful if they terminate. A solution that would establish an hard-coded structure, such as in the system's native programming language, could easily solve this problem at the expense of reducing both the overall level of transparency (i.e., this structure could not be reflected upon), and its homogeneity (since there would be a dependency on an external definition). Another way is to devise a stop condition able to halt the infinite recursion, although how does one establish an infinite number of levels? Maybe by induction, though this approach would be significantly more complex. Our main goal is to provide enough transparency among all levels so that they are both observable and changeable.

Therefore, **use a self-describing meta-level and make it the top-most layer of the reflective tower**. This meta-level needs to be expressive enough to define itself, but once this property is attained it could, in principle, expose all the necessary information to allows the whole system to adapt and evolve. This represents a very high level of transparency. You should make this top-most level as simple as possible, with the bare information needed to specify more extensive (and eventually more complex) lower levels.

This particular solution is also known as a **meta-circular** model (or a closed meta-model) where the primary representation of the model is a primitive model element in the model itself (a property related to **homoiconicity**). One primary advantage of this approach is not requiring (or depending on) an **external representation** of the system.

The main liability of this pattern is the potential for our system's logic to be trapped into infinite loops, since interpreting the self-describing level will imply that same level to be introspected. This may pose threats on the decidability of the meta-model, particularly when semantics-level reflection is provided. Bootstrapping and Lazy Evaluation may be used to solve such infinite dependencies.

[8]Or should we say *a message was sent*.

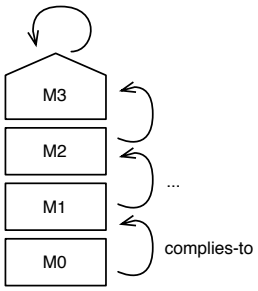

Figure 7: Closed Reflective Tower.

There are several known uses for this pattern, from model-driven (e.g. MOF) to grammar definitions (BNF). In UML-related models, the layer that describes UML per-se is the M_2, as depends on MOF — M_3 — which is self-compliant. Another common example of meta-circularity (although not always regarded as such) is the XSLT language: XSLT is expressed using XML, thus allowing XSLTs to be written that manipulate other XSLTs.

4. PATTERN III: BOOTSTRAPPING

There are no known alias.

Figure 8: An artist rendition of the *big-bang*, a cosmological theory that postulates the birth of the universe as a massive explosion of matter and energy from a singularity.

Using the Closing the Roof pattern implies the existence of a circular dependency in the top-most model level. In fact, due to the nature of reflective systems, it is very common for circular dependencies to appear throughout a system's design.

Continuing the story so far, when we decided to CLOSE THE ROOF so that we could bound the number of levels in our system, we were faced with another *chicken-and-egg* problem: if a model complies to itself, how does it come to exist? In fact, one cannot say that a model-level complies to another model-level (even to itself) if it doesn't yet exist. Actually, the creation of any model-level is usually constrained by the rules and specifications of higher levels; how can the elements of a level be instantiated, if they have themselves to be inspected for that purpose? Therefore, a first "blind" instantiation[9] of the model was needed to allow its self-definition.

How do we provide a model whose definition depends on itself? Or more generally, how do we start any process that depends on its own outcome? This often leads to a chicken-and-the-egg problem, where you may find yourself in need of definitions which may not yet have been defined, and in turn those definitions need whatever you are now defining. This is similarly to what happens when writing a dictionary, how do you write the meaning of a word if the words you'll use also require a meaning?

The first apparent solution to this problem was to hard-code a meta-model into the source code. This would move the instatiation issue to outside our running system. The first model would never be *given birth*; it was always assumed to exist, through well known, hard-coded rules.

The problem here was homogeneity; the code started to be polluted with conditional statements in the form of "if my meta-level is myself, then I'll do this special case"; in fact, a cross-cutting concern. We had also receded to have (again) two different types of descriptions; one using meta-data, and another using the host's source code. It was also starting to be clear that, should this level evolve, the code would have to be changed and recompiled. In fact, should we wish to ever make the system self-hosted, this would definitely represent an added difficulty. We actually started to experiment hard-coded definitions, but the sheer size of our model was starting to overwhelm us from the point of view of dealing with too many elements, definitions and constraints.

Therefore, **provide a minimalistic core of well-known elements from where you can build more complex constructions, and instantiate it at the very beginning of the system's execution.** Similarly to a dictionary, you start with a small set of existing resources and then proceed to create something more complex and effective. The smaller and simpler it is, the less the system will be bound to specific model elements, and the less likely the top-most level will need to change in the future. A thorough formalization of the core will benefit the system, as it will serve as a foundation for all the other levels. In order to help solving cyclic dependencies, lazy computational models may be used. Although maintaining the core small and simple, boot-strapping a system also requires a substantial degree of expressiveness, which will eventually result in increasingly richer levels.

The term bootstrap seems to have its roots on a metaphor derived from pull straps sewn onto the backs of leather boots with which a person could pull on their own boots (without outside help). The term was heavily to refer to the seemingly paradoxical fact that a computer cannot run without first loading its basic software, but to do so it needed to be running.

[9]In other words, an instantation assumed to be right, without any type of structural or constraint enforcement.

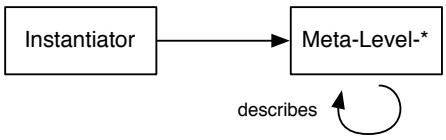

Figure 9: Bootstrapping a model level. The instantiator is the classic aristotelian *primum movens*, that gives "birth" to the first model-level.

The most common known uses of this pattern are programming languages and their compilers (e.g. Smalltalk and LISP). The advantages of starting with a small self-describing core to define the whole system are very patent in the following war story from Alan Kay:

> *"(...) the virtual machine, running inside Apple Smalltalk, was actually simulating the byte codes of the transformed image just five weeks into the project (...) Ten weeks into the project, we* crossed the bridge *and were able to use Squeak to evolve itself, no longer needing to port images forward from Apple Smalltalk. About six weeks later, Squeak's performance had improved to the point that it could simulate its own interpreter and run the C translator, and Squeak became entirely self-supporting."*

For model languages, the most well-known use is probably MOF, which began as a small subset of UML structural diagrams along with some constraints, and that is used to define the whole UML.

5. CONCLUSION

This paper presented three of the most central patterns for the design of a meta-architecture, which are a specific type of software architectures, able to inspect their own structure and behavior, and adapt them at runtime. They closely complement each other. EVERYTHING IS A THING allows to use the same representations strategy regardless of the model level, unifying these levels under the same concept. However, this brings the need of a upper level to every model level. CLOSING THE ROOF addresses this issue by creating a ciclic dependency on the upper-most level, that should thus comply to itself. This ciclic dependency is, however, not trivial to deal with. BOOTSTRAPPING addresses this problem by relying only on a minimal set of definitions, upon which more complex structures can be built.

6. ACKNOWLEDGMENTS

We would like to thank our shepherd Hans Wegener for all the support, patience, and valuable comments he has

handed during the course of this work. We would also like to thank all the participants of the *Frameworks & Environments* writer's workshop, led by Richard P. Gabriel: Brian Foote, Eduardo Guerra, Ricardo Lopez, Sebastian Günther, Thomas Cleenewerck, and Youngsu Son. Finally, we would like to acknowledge the Foundation for Science and Technology and ParadigmaXis, S.A., which have partially funded this work through the grants SFRH/BDE/33298/2008 and SFRH/BDE/33883/2009.

7. REFERENCES

[1] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Oct. 1977.

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

[3] F. F. Correia and H. S. Ferreira. Trends on adaptive object models research. 2008.

[4] H. Ferreira. *Adaptive Object Modeling: Patterns, Tools and Applications*. PhD thesis, University of Porto, Faculty of Engineering, 2010.

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

[6] H. S. Ferreira, F. F. Correia, A. Aguiar, and J. P. Faria. Adaptive object-models: A research roadmap. *International Journal On Advances in Software*, 3, 2010.

[7] H. S. Ferreira, F. F. Correia, and L. Welicki. Patterns for data and metadata evolution in adaptive object-models. In *PLoP '08: Proceedings of the 15th Conference on Pattern Languages of Programs*, pages 1–9, New York, NY, USA, 2008. ACM.

[8] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Professional, Nov. 1994.

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

[10] ISO 9241-11:1998. *Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability*. ISO, Geneva, Switzerland, 1998.

[11] R. W.-B. Leon Welicki, Joseph W. Yoder. The dynamic factory pattern. In *PLoP '08: Proceedings of the 15th Conference on Pattern Languages of Programs*, 2008.

[12] R. W.-B. Leon Welicki, Joseph W. Yoder. Adaptive object-model builder. In *PLoP '09: Proceedings of the 16th Conference on Pattern Languages of Programs*, 2009.

[13] M. Lutz. *Learning Python*. O'Reilly Media, Sept. 2009.

[14] OMG. MetaObject Facility (MOF), 2010. http://www.omg.org/mof/ [Accessed 5-July-2010].

[15] OMG. Object Constraint Language (OCL), 2010. http://www.omg.org/spec/OCL/ [Online; accessed 5-July-2010].

[16] OMG. Unified Modelling Language (UML), 2010. http://www.uml.org/ [Online; accessed 5-July-2010].

[17] L. Welicki, J. W. Yoder, and R. Wirfs-Brock. A pattern language for adaptive object models: Part i-rendering patterns. In *PLoP '07: Proceedings of the 14th Conference on Pattern Languages of Programs*, 2007.

[18] 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*, pages 787–788, New York, NY, USA, 2007. ACM.

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