The Lazy Semantics Pattern
on the context of 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

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

Joseph Yoder
Refactory, Inc.
joe@joeyoder.com

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. This paper describes the Lazy Semantics pattern, which arrises when there is the need to cope with expressions or model constructions where its meaning is initially undefined or subject to change.

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 [10]. This is achieved by exposing the domain model as a first-class artifact able to be changed and shaped through the system itself.

To understand this pattern the reader should have some background knowledge on Adaptive Object-Models (AOMs): an architecture for adaptive systems that relies on a representation of the system's domain model. This kind of system relies on meta-data and reflection mechanisms to adapt the system to changing business needs, by exposing its domain model as first-class artifacts that can be changed from within the system, at runtime.

AOMs are also sometimes denoted as meta-architectures, or reflective architectures [10]. When compared with traditional object-oriented systems, they climb one step further in the abstraction ladder, and promote the objects model to a first-class artifact to which all domain-dependent functionality can be derived from; like persistence mechanisms, validation of invariants, graphical user interfaces, etc.

Several patterns underlie this kind of architecture. Type Square [10] is the result of using Type Object [5] and Property [11] patterns, and is one of the key patterns in the creation of an AOM. This design supports defining model elements and their properties (respectively, the `EntityTypes` and `PropertyTypes` in Figure 1) and allows to instantiate them (as `Entities` and `Properties`).

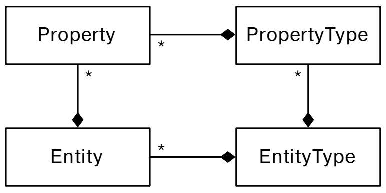

Figure 1: The Type Square pattern.

There is now a growing pattern language for Adaptive Object-Models (AOMs) [3, 2, 8, 9], that this paper will extend by introducing the Lazy Semantics pattern. This pattern is specially useful when implementing two patterns that have been previously descibed: Bootstrapping [3] and Closing the Roof [3].

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

used variants of the Gang of Four [4]. 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 SMALL CAPS 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 [7] and OCL [6] available at the date of publication (v.2.3).

1.2 Target Audience

The main goal of this paper is to present a pattern that address some fundamental concepts underlying meta-architectures. It is 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 this pattern useful.

2. LAZY SEMANTICS PATTERN *

Also known as *Deffered Semantics* and *Late Binding*.

When developing systems based upon meta-architectures, developers may find themselves in need of definitions which may not yet have been defined, and in turn those definitions need whatever they are defining. Or the definitions may change in time. This may very easily happen when BOOTSTRAPPING [3] and when CLOSING THE ROOF [3].

Let us consider three different examples where this pattern may occur, viz. (i) in dynamic language constructs, (ii) in meta-programming, and (iii) in meta-modeling.

1st Example

Consider the following Smalltalk code:

```
Vehicle wheels
```

What is the meaning of this code, i.e., its semantics? From a language point of view, we know it is sending the message `wheels` to the object `Vehicle`. We would expect that such evaluation would produce some behavior. But what if we cannot define the meaning of wheels until we evaluate the running state of the system? Or what if the messages semantics depend on a specific execution context?

2nd Example

Consider the following C# code:

```
public class Customer {
  public string CustomerID;
  public string City;

  public static IEnumerable<Customer>
       GetCustomersInLondon();
}
```

How would we implement the `GetCustomersInLondon()` method? If all the objects were stored in memory, we could do something like this:

```
foraech(var c in Customers) if (c.City == "London") yield
      return c;
```

But, would `Customer` be mapped into a relational database, we would probably want to do an SQL query like this:

```
select * from Customers where city = "London"
```

The problem is, how do we abstract `GetCustomersInLondon()` so that it could work with both (or probably more) modes? How could we adapt its semantics to the executing context?

3rd Example

Consider we are bootstrapping a meta-modelling based system (for example, a self-contained AOM, which is using the CLOSING THE ROOF and BOOTSTRAPPING patterns).

We may start be defining what is a Class, by saying that there's a class named `Class`, with a property `Name` that will hold the name of the class. But for that we first need to define what a property is. We could backtrack and start by defining that first: a property is a `Class` named `Property`. Wait!... We haven't defined what a class is yet.

This situation is usually known as the chicken-and-the-egg problem. Which came first? The class or the property? Sometimes we may need some definitions that have not yet been defined, and in turn those definitions need whatever we are now defining.

How can we make the system able to handle different meanings depending on its current known state? The system is expected to be consistent, that is, the semantics that each model-level establishes are expected to be enforced over its instances (i.e., the elements of the lower model-level). Enforcing *consistency* at all times may be hard, if at all possible, as it requires to find a sequence of actions that accounts for all dependencies. For example, there may be cyclic dependencies when BOOTSTRAPPING. If the system is made more *tolerant* to undefined constructions, consistency may not be assessed at all times. The following forces are to be considered: (i) transparency, in what concerns the degree of the system that is to be exposed by reflection; even if the system doesn't rely on CLOSING THE ROOF, the problem may occur in different circumstances, (ii) granularity, since by defining a string as an thing, we are homogenizing the infrastructure and thus we are lead to (iii) reuse, due to the preservation of existing mechanisms to deal with different meta-levels, (iv) convergence, in the sense that we may consider the system to be eventually consistent and (v) complexity, which is an overall concern to be reduced.

Therefore, **Enforce meaning only when absolutely required.** Pertain your system doesn't enforce any meaning up until it is absolutely necessary (e.g., your cyclic dependencies are resolved, or you have enough information to bind to a specific meaning). Change the execution from *call-by-value* or *call-by-reference* to *call-by-need.*

1st Example

Referring back to the previously presented example, let us imagine that the `start` message is not defined, i.e., the message could not be dispatched to a specific method. When this happens in smalltalk (or very similarly in Ruby), the interpreter reifies the wheels message into an object, and pass it as an argument of the `doesNotUnderstand:` message. The later could inspect the message and decide what to do with it. For example, it could decide that the method is a proxy to a relational table, and thus appropriately generate and query an SQL engine. In fact, while it could simply return the appropriate collection of wheels, it could also dispatch side-effects, like creating a new method that will specifically answer the wheels message.

In other words, the exact meaning (the semantics) of the wheels message is determined when it is needed. Moreover, since the precise meaning of the wheels message can change over time, it should only be determined when it is absolutely necessary. This is one of the reasons why dynamically typed languages are generally slower that strongly typed languages, since message passing cannot be transformed into a pointer derreferentiation.

2nd Example

Consider the following C# code, written in the LINQ DSL:

```
var q = from c in Customers
        where c.City == "London"
        select c;
```

The above expression is quite peculiar. Its type is `IEnumerable<T>`, and it will not be executed until it is absolutely needed, such as in the following snippet:

```
foreach (var c in q) Console.WriteLine("id = {0}, City =
     {1}", c.Id, c.City);
```

Here, the system will have to enumerate q. This will trigger some evaluation routines which will check the Customers collection. Let's consider that it was defined as:

```
[Table(Name="Customers")]
public class Customer {
  [Column(IsPrimaryKey=true)] public string CustomerID;
  [Column] public string City;
}

DataContext db = new DataContext("database.mdf");
Table<Customer> Customers = db.GetTable<Customer>();
```

In this case, the system will generate the appropriate SQL code by inspecting the LINQ expression, and pass it through the `DataContext`. It would then convert each row into a Customer instance, and yield it as items of the `IEnumerable<Customer>`. But, if the Customers was defined as a simple `List<Customer>`:

```
var Customers = new List<Customer> {
  new Customer() { Id = 1, City = "Paris" },
  new Customer() { Id = 2, City = "London" }
}
```

Then the DSL would be converted into a simple expression that would filter the collection of objects based on the given criteria:

```
var q = Customers.Where(c => c.city == "London")
```

Once again, the precise meaning of the LINQ expression cannot be known Ãă-priori. The system's state when the expression is executed will determine the exact semantics of that expression.

3rd Example

One solution to the scenario described in the 3rd example is to relax on the mandatory definitions. Instead of requiring every class to have names, we would first create the class `Class` without handing it any name. Next, we would create the class Property. We proceed to say that a class has a property, and so on. Sometimes infinite cycles can be worked around by carefully relaxing on the structure and choosing the appropriate sequence.

Another solution would be to relax on the meaning of properties. Instead of a class having a property named Name, it could have a slot accessed through a string identifier, that would hold its name. After defining what a property is, we could then resort to define that getting the values of a property consists on searching the slots for the property name. Hence, we have two different semantics for accessing a property: (i) a property is nothing but it's name looked up in the slots, or (ii) a property is a known-object that maps to values. The precise meaning of property changes over time, and depends on the state of the system.

One very similar pattern is Lazy Evaluation, although it focus on a different aspect. Lazy Evaluation may be resumed as delaying any kind of computation to the point where it is absolutely needed. For example, in the programming language Haskell, everything is lazily evaluated, allowing one to inductively define sets and only compute needed elements. For example, the following function defines an infinite list containing the fibonnaci series:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

In languages without lazy evaluation the computer would enter an infinite cycle trying to produce all the values of the list. Lazy evaluation doesn't require a flexible semantics. The above expression never changes its meaning (in fact, Haskell is statically typed).

Oghma, a framework supporting the development of AOM-based systems also makes use of Lazy Semantics to solve the cyclic dependencies that arise from the auto-compliance of the upper-most model level, achieved by Closing the Roof.

3. CONCLUSION

This paper explored the Lazy Semantics pattern and its applicability. Lazy semantics arrises when there is the need to cope with expressions or model constructions where its meaning is initially undefined or subject to change. Some examples were given in the context of Meta-architectures, also known as reflective architectures, which are a specific type of software architectures able to inspect their own structure and behavior, and dynamically adapt at runtime. This type of architectures are suitable to make the system able to respond to new user requirements or changes in their environment. Regarding Alexander's taxonomy of patterns, we would define the Lazy Semantics as one star, i.e., we have gathered enough empirical evidence of its usage, but we are still unsure that the true invariant is captured by the given ⟨*problem*, *forces*, *solution*⟩ triplet.

4. ACKNOWLEDGMENTS

We would like to thank our shepherd Norihiro Yoshida for all the support and valuable comments he has handed during the course of this work. We would also like to acknowledge the Foundation for Science and Technology and ParadigmaXis, S.A., which have partially funded this work through the grant SFRH/BDE/33883/2009.

5. REFERENCES

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

[2] 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.

[3] H. S. Ferreira, F. F. Correia, J. Yoder, and A. Aguiar. Core patterns of Object-Oriented Meta-Architectures. In *Proceedings of the Pattern Languages of Programs*, Reno, Nevada, USA, 2010. ACM.

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

[5] R. Johnson and B. Woolf. Type object. In *Pattern Languages of Program Design 3*, pages 47—65, 1997.

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

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

[8] 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.

[9] 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.

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

[11] J. W. Yoder, B. Foote, D. Riehle, and M. Tilman. Metadata and active object-models. Vancouver, British Columbia, Canada, 1998. ACM.