LabIss | Agile model-based software development

1. QActors and process development
2. Models and meta-models
3. Domain Specific Languages

QActors and process development

QActor models should be introduced in the early stages of the software development process, to express in a concise way the overall *architecture* of a distributed system composed of a set of actors that use **high-level** message-oriented operations, by abstracting from the fact that they could work in a local environment (i.e. in a same JVM) or on different computational nodes.

The intent is to exploit the features of the QActor language to capture the **logic of the interaction** rather than the technological details of its implementation.

The possibility to execute a (simple) QActor model of a system as the result of the *problem analysis* phase can be very useful for a more productive interaction with the customer in order to better understand the requirements, define the product backlog and introduce proper functional test-plans already expressed as programs.

Logical architectures

Moreover, a system analyst could exploit the QActor modeling language to define a **logical architecture** of the system, to be used as a reference-point for software development.

Clearly, such a simple initial architectural model must be refined in the next steps of process development (mainly in the *project phase*) by gradually **zooming** into the details of each resource and of each actor.

The concepts embedded into the QActor language can express (running) models of software system bot at the 'macro' level (of a distributed, heterogeneous system) and at the 'micro' level (of each node composing the system) in a 'seamless' way.

The mapping between the logical level and the appropriate interaction-technology support(s) identified by the designer, can be done in a quite automated way by a proper software factory built around the QActor concepts, as formally defined in the **QActor meta-model**.

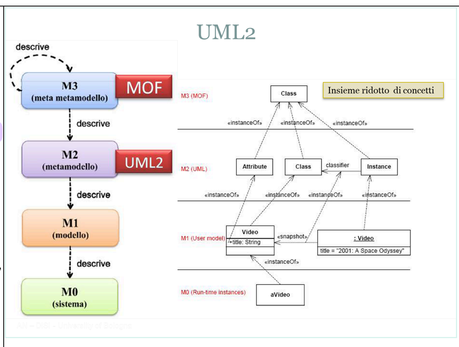
Models and meta-models

Metamodeling

The Object Management Group (OMG) has developed a metamodeling architecture to define the *UML*, called the **Meta-Object Facility (MOF)**, designed as a four-layered architecture.

The most prominent example of a Layer 2 MOF model is the *UML metamodel*, which describes the UML itself.

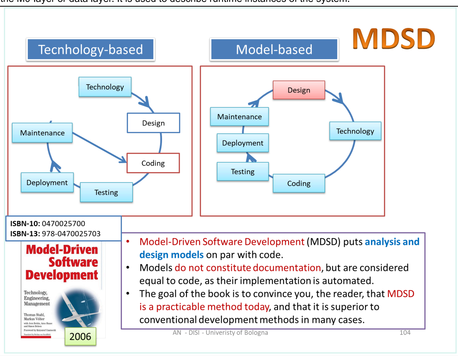
The UML language can be extended using a mechanism called *stereotyping*, but this has been criticized as being insufficient/untenable.



M2-models describe elements of the M1-layer, and thus M1-models. These would be, for example, models written in UML. The last layer is the M0-layer or data layer. It is used to describe runtime instances of the system.

Model Driven Software Development

See
- On semantics
- What models mean



EMF and Xtext

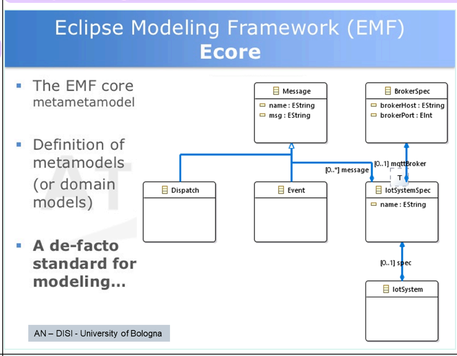
The Eclipse Modeling Framework (EMF) is a modeling framework and code generation facility for building tools and other applications based on a structured data model based on the EMF (core) standard.

Xtext is a framework for development of programming languages and domain-specific languages.

Xtext uses EMF models as the in-memory representation of any parsed text files. This in-memory object graph is called the abstract syntax tree (AST). Depending on the community this concepts is also called document object model (**DOM**), semantic model, or simply model.

With Xtext you define your language using a powerful grammar language. As a result you get a full infrastructure, including **parser, linker, typechecker, compiler** as well as **editing support** for Eclipse.

The Eclipse Modeling Framework



Definition of a custom language: example

```
grammar it.unibo.xtext.intro19.Iot
        with org.eclipse.xtext.common.Terminals
generate iot "http://www.unibo.it/xtext/intro19/Iot"
terminal VARID : ('A'..'Z'|'_'|'a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

IotSystem: "System" spec=IotSystemSpec ;
QualifiedName : ID ('.' ID)* ;

IotSystemSpec: name=ID
        (mqttBroker = BrokerSpec)?      //? means Optional
        (message    += Message    )*    //* means N>=0
;
BrokerSpec : "mqttBroker" host=STRING ":" port=INT ;

Message :  Event | Dispatch  ;

Event:    "Event"    name=ID  ":" msg = STRING  ;
Dispatch: "Dispatch" name=ID  ":" msg = STRING  ;
```

grammar it.unibo.xtext.Entities with org.eclipse.xtext.common.Terminals
generate entities "http://www.unibo.it/xtext/Entities"

EBNF notation

```
Model :         (imports+=Import)*
                (types+=Type)*      ;

Import :        'import' importURI = STRING;

Type:           SimpleType | Entity;

SimpleType:     'type' name = ID;

Entity :    'entity' name=ID ('extends' extends= [ Entity ] )? '{'
            properties+=Property*
            '}';

Property: 'property' name=ID ':' type= [ Type ] ( many ?= '[]')?;
```

The grammar is a collection of Rules.

Rules start with their name followed by : and ending with ;

The editor provides code completion and constraint checking for the grammars themselves

Domain Specific Languages

A Domain-specific languages (DSL) is a computer language specialized to a particular application domain. Examples of DSL are Gradle and SQL. This is in contrast to a general-purpose programming language (GPL), which is broadly applicable across domains.

DSL are specialized in solving a specific problem domain and can be very efficient at it. Moreover, by **incorporating knowledge** about that domain, DSLs can lead to more concise and more analyzable programs, promote *effective communication with customers*, better code quality and increased development speed.

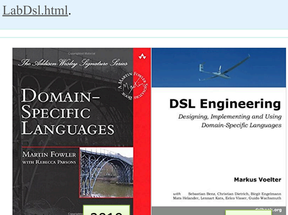
DSLs come in two main forms: external and internal.

- **External DSLs** are DSLs with their own syntax that's parsed independently of the host GPL. They requires work when integrating to an application written with a general programming language.
- **Internal DSLs** are written in the syntax of a general programming language. They are a particular form of API in a host general purpose language, often referred to as a *fluent interface* .

- Domain-specific Languages, by Javier Luis Cánovas Izquierdo, Inria
- Real World DSL, by Mario Fusco, Red Hat

For an example using Kotlin to define internal DSL see LabDsl.html.

Fowler presents effective techniques for building DSL, and guides software engineers in choosing the right approaches for their applications. This book's techniques may be utilized with most modern object-oriented languages; the author provides numerous examples in Java and C#, as well as selected examples in Ruby



ISBN-10: 0321712943
ISBN-13: 978-0321712943

ISBN-10: 1481218581
ISBN-13: 978-1481218580

This book provides a thorough introduction to DSL, relying on today's state of the art language workbenches.
The book provides details about the implementation of DSLs with lots of code. It uses three state-of-the-art but quite different language workbenches: JetBrains MPS, Eclipse Xtext and TU Delft's Spoofax.