University of Jijel-Mohamed Seddik Benyahia
Faculty of Exact Sciences and computer science
Department of Computer Science

Option : Information Systems and Decision Support
Level : Master 1

Course Material

Module : Advanced Software Engineering

Written by:

Dr. Aissam Belghiat

School year: 2023 /2024

Table of contents

General introduction

Since the dawn of computing, the methods and techniques for developing computer systems have always been at the center of interest of different actors involved in it. These systems have become essential in all fields, in fact, they manifest a complexity that continues to increase, especially in their implementation. The objective was always to provide reliable, robust, efficient, and easy-to-use software.

This course is composed of **two main parts** developed through **five chapters**.

The first part of this course will be on Advanced Design Techniques. They play a crucial role in developing high-quality software systems. These techniques focus on creating scalable, maintainable, and flexible solutions that can easily adapt to changing requirements. In fact, students will explore essential methods and best practices to enhance their mastery of techniques of designing software. A key focus will be on design patterns, which offer reusable, proven solutions to common design challenges, helping to create software that is both robust and adaptable. Additionally, the course will delve into object-oriented design principles, such as the SOLID principles, which promote clean, maintainable, and scalable code. By understanding these advanced techniques, students will be better equipped to design software systems that are efficient, flexible, and easier to manage in the long term.

The second part of this course will be on Model Driven Engineering. In fact, the methods and techniques proposed in the literature were largely linked to the execution platforms, which posed many problems and obstacles each time we experiment with an evolution. Models have emerged as a solution thanks to their simplicity and flexibility. Models allow us to better understand the system we are developing, so they are completely independent of the execution platforms. The use of models at the different phases of development of a software system represents the basic idea of the approach proposed and supported by the OMG (Object Management Group), in this case MDA (Model Driven Architecture). MDA is based on the separation of concerns technique, which consists of separating the business aspects from the technical aspects using models. So for MDA, the classic development of a system can be replaced by a set of transformations between several models up to the generation of the code.

Both parts of the course are largely based on UML (Unified Modeling Language). Advanced Design Techniques use it to schematize the different solutions in the designs. The MDA approach uses it to model the different levels of abstraction. UML is fundamental and inevitable. It will be presented from the beginning to facilitate students to follow the rest of the course.

UML is a language designed to represent, specify, build and document software systems. Its two main objectives are the modeling of systems using object-oriented techniques, from design to maintenance, and the creation of an abstract language understandable by humans and interpretable by machines. UML is the result of the fusion of a set of other object-oriented graphical modeling languages and it has quickly become an inevitable standard. It allows to define the modeling elements as well as their semantics, it is considered as a powerful communication medium that facilitates the representation of object-oriented solutions. It offers a set of diagrams reflecting a particular aspect of the system, and undoubtedly the most important diagram in this language is the class diagram. The goal of this course is to teach our students Advanced Design Techniques and Model Driven Engineering, and specifically MDA.

The course is broken down into five chapters.

- **In the first chapter**, we will present the UML in detail, its characteristics, its diagrams and its advanced mechanisms. Mastering this language is essential for the rest of the course.
- **In the second chapter**, we discuss design patterns and how to use it.
- **In the third chapter**, we continue with object-oriented design principles.
- **In the fourth chapter**, the MDA approach will be presented in detail, showing the position of model transformation in it, the theoretical aspect of MDA must be mastered in this chapter.
- **The fifth chapter**, will present the approach based on graph transformation; the approach chosen in our course to implement model transformation. In the same chapter, a practical case study is proposed. This is the development of a transformation between two formalisms, the chosen source is the UML activity diagram, the chosen target is the finite state automaton. The case study is carried out using AToM^3.

This course of "Advanced Software Engineering", in the Master level, is related to the course "Software Engineering", in the License level. In fact, it can be considered as a continuation of the topics covered in the previous course, taking the reader to a more advanced level in designing reliable software. Hence, it is recommended that the reader starts by the "Software Engineering" course before learning this "Advanced Software Engineering" course. The subjects explained in the former represent the basis of the latter, and the reader should have the adequate background.

Chapter 1

Modeling with UML

In this chapter :

1.1. Introduction

UML (Unified Modeling *Language*) is a language for *visualizing*, *specifying*, *constructing* and *documenting* all aspects and artifacts of a software system **[OMG10]**. UML is the result of the merger of a number of other object-oriented graphical modeling languages (OMT, Booch and OOSE), and has rapidly become an essential standard. One of the ingredients of its success is the fact that it is a generic language. The set of concepts and views making up UML's semantics can be adapted to all design domains and problems. UML enables modeling elements and their semantics to be defined, it is considered a powerful communication medium that facilitates the representation of object-oriented solutions, it offers a set of diagrams reflecting a particular aspect of the system, and its graphical notation enables an object solution to be expressed visually, facilitating the comparison and evolution of solutions. However, it should not be forgotten that UML notation is an object modeling language, not an object method. The evolution of the UML language is controlled by the OMG, an international non-profit organization created in 1989 on the initiative of major companies (HP, Sun, Unisys, American Airlines and Philips).

In this chapter, we'll look at the fundamental concepts of the UML language and its various characteristics, then present the diagrams of this language in varying levels of detail. We're going to focus on the class diagram, as it will be over-used in the rest of the course, particularly as it will be adopted for the representation of meta-models and meta-meta-models in the MDA model-driven architecture, so we'll assume that the reader is not familiar with, others who master this diagram are invited to move on to the next.

1.2. History

In the 90s, a number of methods were developed to meet the needs of software documentation and specification (OMT, OOSE, Booch, etc.). These methods were very similar; the same basic concepts could appear differently in the notation accompanying these methods, leading to confusion in the interpretation made by their users. This diversity created a problem of interoperability between tools, which slowed down the deployment of object-oriented development methods. The ambition of UML, the first version of which was published in 1997 by the OMG, is to bring together in a single notation the best features of the various object-oriented modeling languages. The UML specification defines the fundamental concepts of the object, as well as other concepts useful in software modeling (use cases, etc.). The figure below shows the chronic evolution of UML.

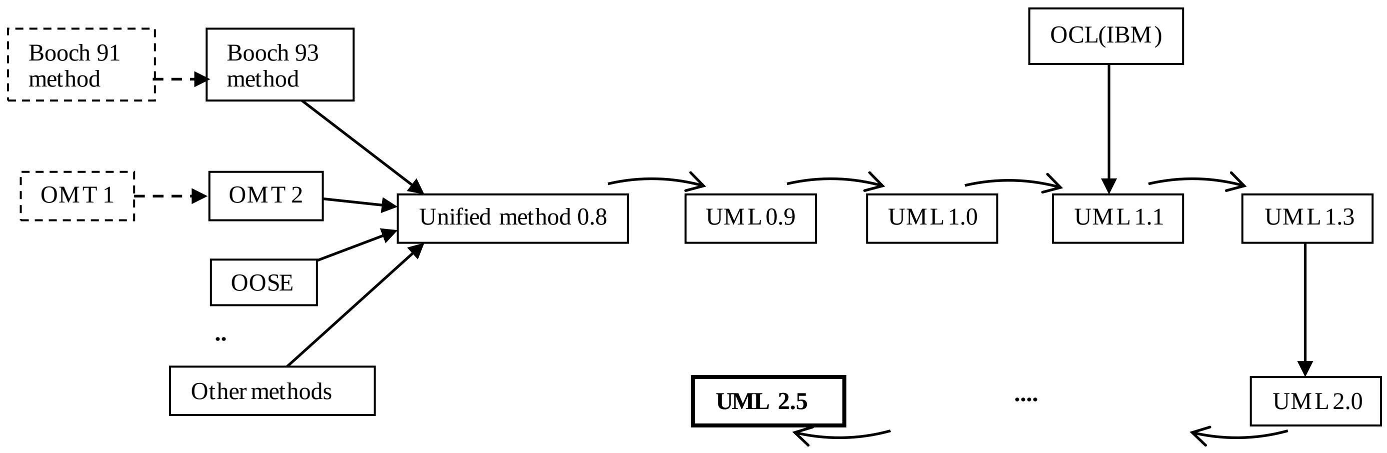


FIG. 1.1 - Historical development of UML.

1.3. Using UML

The UML language is a set of notations that have emerged in an attempt to standardize a multitude of modeling languages. Due to its many origins, UML has many different facets and is used in a wide variety of ways by different software development companies. It should be noted that when using UML, we must put in mind that it is not a method or a process, so to take advantage of UML, we need to use a process that is very well mastered by the UML tool and includes certain characteristics described later. UML can be used in three different ways **[Fow03]**:

- ✓ **UML as illustration:** With this usage, UML can be used to communicate certain aspects of the system. The basis of illustration is selectivity: the aim is to use illustrations as a means of communicating ideas and alternatives about the work to be done. Illustration is a rather informal and dynamic activity. Illustrations are also useful in documentation, where communication takes precedence over completeness.

- ✓ **UML as a blueprint:** In contrast to illustrations, blueprints strive for completeness. The central idea is that they are developed by a designer whose job is to construct a detailed specification of what the programmer will have to code. The design must be sufficiently complete that no decision is left to the programmer.
- ✓ **UML as a programming language:** use UML as a real programming language, with a much more precise and rigid syntax. This usage automatically produces executable code from a UML source.

The first is direct, where you start by writing a schema of the software in UML, then develop and implement based on this schema. This is classic direct design. The other is reverse engineering, in which UML is used as a tool of understanding rather than design.

1.4. Advantages and disadvantages of UML

1.4.1. The strengths of UML [Pie09]

- ✓ UML is a semi-formal, standardized language
  - improved precision
  - a guarantee of stability
  - encourages the use of tools

- ✓ UML is a powerful communication tool
  - It frames the analysis.
  - It facilitates the understanding of complex abstract representations.
  - Its versatility and flexibility make it a universal language.

1.4.2. The weaknesses of UML [Pie09]

- ✓ Putting UML into practice requires a learning curve and a period of adaptation.
  - Even if Esperanto is a utopia, the need to agree on common modes of expression is vital in IT.
- ✓ Process (not covered by UML) is another key to project success.
  - But integrating UML into a process is not trivial, and improving a process is a complex and time-consuming task.

- The authors of UML are well aware of the importance of this process, but the industrial acceptability of object modeling depends first and foremost on the availability of a high-performance, standard object analysis language.

1.5. Modeling with UML

This section is dedicated to the presentation of modeling with the UML language. The various diagrams and their uses are presented.

1.5.1. Model

A model is an abstract and simplified representation (i.e. one that excludes certain details) of an entity (phenomenon, process, system, etc.) in the real world, with a view to describing, explaining or predicting it. Model is synonymous with theory, but with a practical connotation **[Aud07]**.

1.5.2. Using models

Models have different uses **[COT09]**:

- They are used to circumscribe complex systems in order to dominate them. For example, it's unimaginable to build a rocket without using modeling to test the reactors, safety procedures, watertightness of the whole, etc.
- They optimize system organization. Modeling a company's structure in terms of divisions, departments, services, etc. provides a simplified view of the system, and thus ensures better management.
- They allow you to focus on specific aspects of a system without having to deal with irrelevant data. If we are interested in the structure of a system in order to factorize its components, there is no need to bother with its dynamic aspects. Using the UML language, for example, you can focus on the static description (via the class diagram) without worrying about other views.
- They can be used to describe requirements accurately and completely, without necessarily knowing the details of the system.
- They facilitate system design, including the creation of approximate or reduced-scale mock-ups.

- They make it possible to test a multitude of solutions at lower cost and in shorter timescales, and to select the one that solves the problems posed.

1.5.3. UML diagrams

A UML diagram is a graphical representation that focuses on a specific aspect of the model; it is a perspective of the model. Each type of UML diagram has a specific structure and conveys specific semantics. Combined, the different UML diagram types offer a complete view of the static and dynamic aspects of a system **[Pie09]**. An important feature of UML diagrams is that they support the notion of abstraction, which helps to simplify representation and better control complexity in the expression and elaboration of object solutions.

UML 2 offers us thirteen (13) diagram types representing as many distinct views to model particular system concepts. They fall into two main groups:

Structural or static diagrams

- ✓ class diagram
- ✓ object diagram
- ✓ component diagram
- ✓ deployment diagram
- ✓ package diagram
- ✓ composite structure diagram

Behavioral or dynamic diagrams

- ✓ use case diagram
- ✓ activity diagram
- ✓ state-transition diagram
- ✓ Interaction diagrams
  - ▪ sequence diagram
  - ▪ communication diagram
  - ▪ global interaction diagram
  - ▪ time diagram

The figure below shows the organization and classification of UML diagrams:

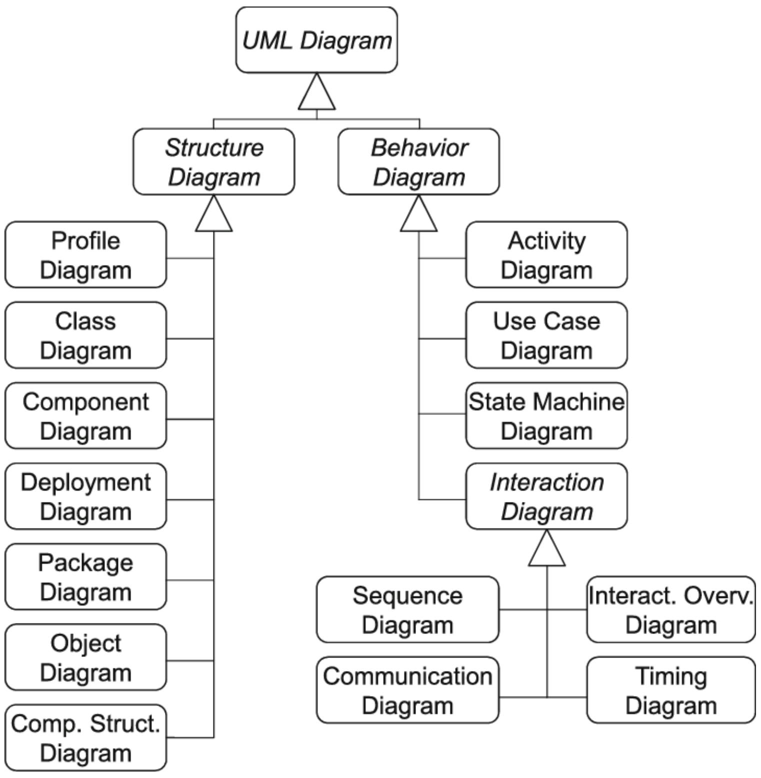


FIG. 1.2 - Classification of UML diagram types.

In the following, we present these diagrams at different levels of detail. We focus on the diagrams we need for the rest of the course **[COT09]**:

Class diagram

It represents the static description of a system, integrating the data and processing parts of each class. It is the pivotal diagram for all system modeling. We'll come back to it in more detail later.

Object diagram

The object diagram is used to illustrate complicated class structures by showing examples of instances.

Component diagram

The component diagram represents the various software components known to the operator for installing and troubleshooting the system. The aim here is to determine the structure of operating components such as dynamic libraries, database instances, distributed objects, software packages, applications, executables and so on.

Deployment diagram

The deployment diagram corresponds both to the physical structure of the computer network that supports the software system, and to the way in which the operating components are installed on it.

Package diagram

The package diagram represents the hierarchy of modules (categories) in a project and their relationships.

Diagram of composite structures

The composite structure diagram describes the composition of a complex object at run-time. This diagram introduces the notion of the structure of a complex object, as it appears in the run-time phase.

Use case diagram

The use case diagram is a model for gathering requirements in a software project. It provides a set of mechanisms and graphical elements for representing these requirements in a way that is simple and comprehensible to everyone involved.

A use case diagram is essentially made up of actors and use cases.

An actor is a representation of a role played by an external entity (a person, a process or a thing) that interacts with a system.

A use case is an externally visible unit of functionality. It performs an end-to-end service for the actor who initiates it, with a trigger, a sequence and an end. A use case therefore represents a service rendered by the system, with no further details on how it is realized. The behavior of use cases is expressed using dynamic diagrams, in particular interaction diagrams. The set of use cases constitutes a subject.

Actors and use cases are listed to show which actors participate in which use cases.

There are several relationships between the elements of a use case diagram.

- extension relationship: between two use cases, to indicate that the behavior described by one extends that of the other.
- the inclusion relationship: between two use cases, to indicate that the behavior described by one is included in the behavior of the other.
- the generalization relationship: between two use cases or between two actors, this is the relationship that exists in all UML diagrams.

<u>Example</u>:

University system diagram that shows student interactions with course registration and grade viewing features.

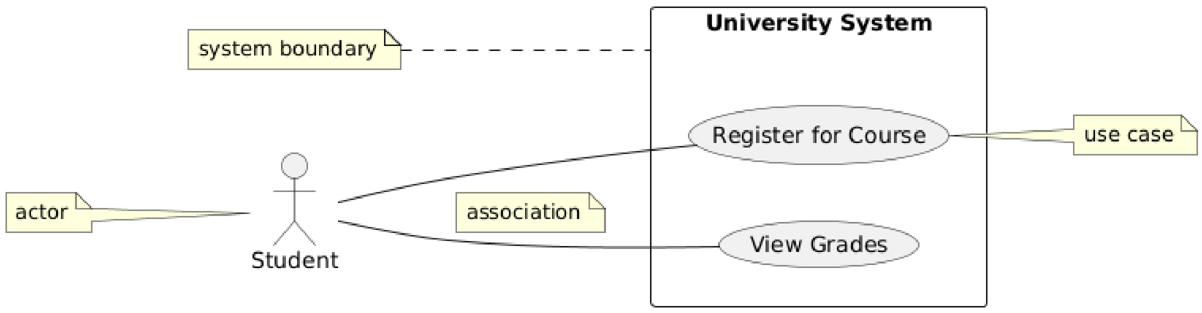


FIG. 1.3 - A use case diagram

Sequence diagram

The Sequence Diagram is an interaction diagram that models the behavior resulting from the interaction of all objects. It provides a set of mechanisms and graphical elements to describe these interactions in chronological order.

A sequence diagram is essentially made up of lifelines and exchanged messages.

An object lifeline represents its various states during interaction.

A message represents communication between lifelines, and has several types, such as sending a signal, invoking an operation and creating or destroying an instance. The diagram shows the temporal sequencing of messages, with time running from top to bottom.

For more complex situations, UML introduces combined interaction fragments. In fact, it proposes to break down an interaction into simple fragments called operands, then combine them by means of so-called operators in order to reconstitute the complexity of the system. These operators include

- alt: multiple alternative fragments (if then else)
- opt: optional fragment
- by: parallel fragment (concurrent processing)
- loop: fragment runs several times
- region: critical region (one thread at a time)
- neg: an invalid interaction
- break: represents exceptional scenarios
- ref: reference to an interaction in another diagram
- sd: complete sequence diagram fragment

Example:

A sequence diagram that models a university system where a student initiates a course registration request, and the system checks availability with the database before confirming the registration.

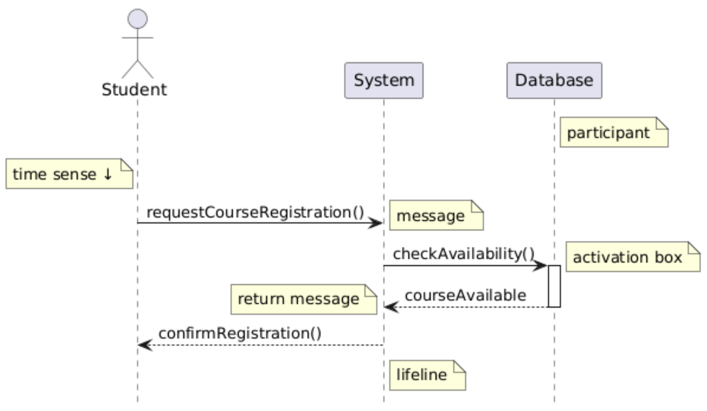


FIG. 1.4 - A sequence diagram

Communication diagram

A communication diagram is a graph whose nodes are objects and whose arcs (numbered according to chronology) represent exchanges between objects. Unlike a sequence diagram, it shows the spatial organization of the participants in the interaction, i.e. a representation of the spatial point of view. A communication diagram is particularly useful for illustrating a use case or an operation.

Example:

A communication diagram to illustrate the same previous interaction.

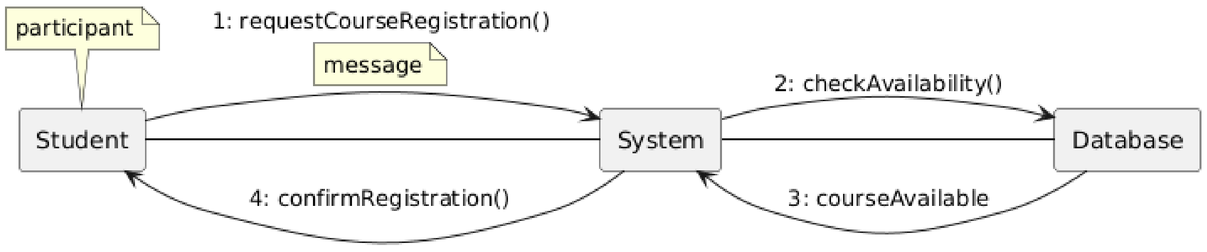


FIG. 1.5 - A communication diagram

Global interaction diagram

The global interaction diagram is a new diagram introduced by UML 2.0. It combines the notations of the sequence diagram with those of the activity diagram to account for the spatial organization of interaction participants. Global interaction diagrams define interactions through a variant of activity diagrams, in a way that provides an overview of control flows.

Time diagram

Time diagram is an interaction diagram that describes both the behavior of individual classifiers and the interactions of classifiers, focusing on the occurrence times of events that cause state changes.

State-transition diagram

The state-transition diagram represents the common life cycle of instances of the same class. In fact, it describes the internal behavior of an object using a finite-state automaton. It shows the states and actions an object undergoes during its life cycle in response to events. The diagram is not limited to specifying an object's behavior, but can also be used for use cases, methods, etc.

The state-transition diagram is made up of states and transitions.

A state represents a period in the life of an object during which it awaits an event or performs an activity. A state can be elementary, or composite, which may contain (i.e. envelop) sub-states.

A transition is a unidirectional link from one state (the source state) to another state (the target state). It can be labeled according to the syntax *Event [Condition] / Action*, which means that if an event occurs, an action will be executed if the condition evaluates to true. An event is something that occurs during the execution of a system. A condition can be set to true or false. An action is an activity to be completed.

State-transition diagrams provide advanced mechanisms for efficiently describing concurrent behavior through the use of orthogonal states. An orthogonal state is a composite state that contains more than one region, with each region representing an execution flow.

Example:

This state machine models a student's course registration status, switching between Registered and Not Registered, with transitions triggered by Register and Drop actions.

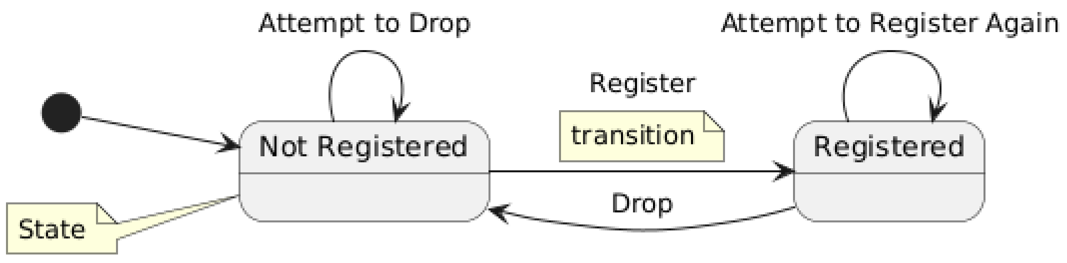


FIG. 1.6 - A state-transition diagram

Activity diagram

Activity diagrams are particularly well suited to modeling the flow of control and data streams. It provides a version of the sequence of activities and actions specific to an operation or use case. Its notation is relatively close to that of a state-transition diagram in its presentation, but its interpretation is significantly different.

The activity diagram is essentially made up of activities and transitions.

An activity specifies a behavior described by an organized sequence of units whose basic elements are actions. The most common types of action are : Action call (*call operation*), Action behavior (*call behavior*), Action send (*send*), Action accept event (*accept event*), Action accept call (*accept call*), Action reply (*reply*), Action create (*create*), Action destroy (*destroy*), Action raise exception (*raise exception*).

A transition materializes the passage from one activity to another. It is triggered when the source activity is completed, and immediately triggers the start of the target activity. Transitions therefore make it possible to specify the sequence of processes and define the flow of control.

State-transition diagrams provide the mechanism for partitions, often called *swimlanes*. They allow you to organize activity nodes in an activity diagram by grouping them together.

<u>Example:</u>

This activity diagram shows the flow of activities for course registration including actions, decision and junction points, and transitions that represent the sequence of steps in the process.

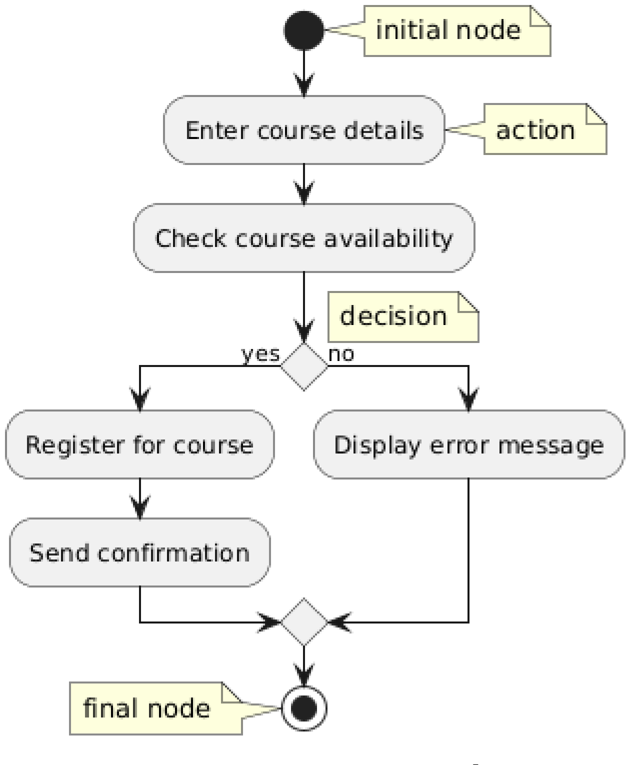


FIG. 1.7 - An activity diagram

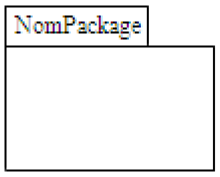
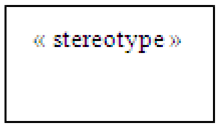
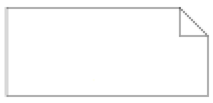
1.6. Class diagram

1.6.1. Presentation

A class diagram is a collection of static modeling elements (classes, packages, etc.), which shows the structure of a model. It abstracts from dynamic and temporal aspects **[Pie09]**. It can be instantiated into object diagrams to represent a specific context. For a complex model, several complementary class diagrams need to be constructed. It shows the internal structure of a system and provides an abstract representation of its objects, which interact with each other to realize use cases **[COT09]**. In this section, we present the elements that are essential for successful class diagram modeling, not all of them, but only the most important ones. For a full explanation, please refer to the OMG website on UML .[1]

1.6.2. General concepts

The UML elements discussed here are not diagram-specific, but general. The table below shows a brief description of these elements:

Element	Rating	Description
Package		A package is a grouping of model elements and diagrams. It allows you to organize model elements into groups. It can contain any type of model element: classes, use cases, diagrams, ... and even nested packages.
Stereotype		A stereotype is an annotation applied to a model element. It enables language to be adapted to specific situations. It is represented by a character string enclosed in quotation marks **(<< >>)**.
Note		A note contains textual information such as a comment, a method body or a constraint. Graphically, it is represented by a rectangle with a folded upper right corner.

TAB. 1.1 - General notions of the UML language.

[1] http://www.omg.org/uml

In addition to the concepts presented above, there are others such as :

Classifier

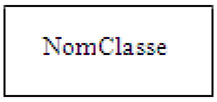
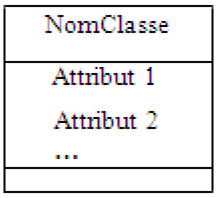
Generally speaking, the modeling elements that can have them are represented in Classifiers. A folder is a model element that describes a structural or behavioral unit **[Aud07]**. We often use the term " Classifier" because this notion also encompasses classes, interfaces, signals, nodes, components, subsystems and so on. The most important type of classifier is, of course, the class.

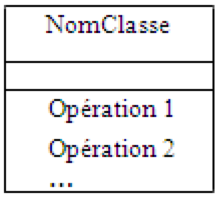
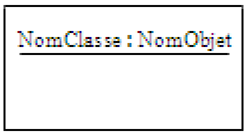
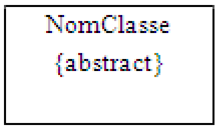
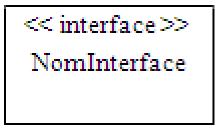
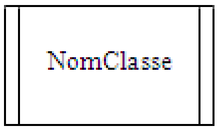
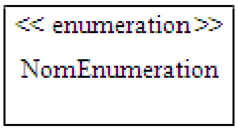
Name space

Namespaces are packages, classifiers and so on. A uniquely named element can be determined by its qualified name, which consists of the series of names of packages or other namespaces from the root to the element in question **[Aud07]**. In a qualified name, each namespace is separated by a colon (**::**). For example, if a package **B** is included in a package **A** and contains a class **X**, you need to write **A::B::X** to be able to use class **X** outside the context of package **B**.

1.6.3. The basic elements

The table below summarizes the basic elements of a class diagram:

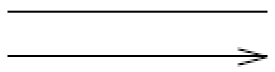
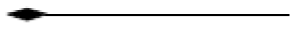
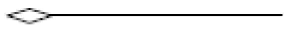
Element	Rating	Description
Class		*A class is an abstract type characterized by properties (attributes and methods) common to a set of objects, enabling the creation of objects with these properties* **[Pie09]**.
Attribute		Attributes define information that a class or object needs to know, each of which is defined by a name, data type, visibility and can be initialized. Attributes *are derived* i.e. can be calculated from other attributes and calculation formulas.

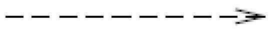
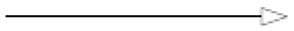
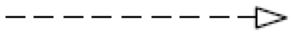
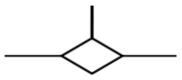
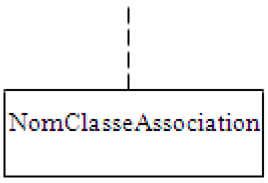
Operation		Operations are actions that a class knows how to perform. A method is said to be abstract when we know its header but not how it can be realized (i.e. we know its declaration but not its definition).
Object		An object is an instance of a class. A class is an abstract data type, characterized by properties (attributes and methods) common to all objects and enabling the creation of objects with these properties.
Abstract class		*A class is said to be abstract when it defines at least one abstract method, or when a parent class contains an abstract method not yet realized* **[Aud07]**.
Interface		A pure abstract class with only abstract methods is called an interface. An interface provides a total or partial view of a set of services offered by a class, package or component.
Active Class		A class is passive by default, saving data and offering services to others. An active class initiates and controls the flow of activities.
Listing		An enumeration is used to show a fixed set of values that have no properties other than their symbolic values.

TAB. 1.2 - Basic elements in a class diagram.

1.6.4. Relations between classes

Relationships between classes express semantic or structural links. The most commonly used relationships are association, aggregation, composition, dependency and inheritance. In most cases, these relationships are binary (linking two classes only). The table below summarizes the various relationships in a class diagram:

Element	Rating	Description
Association		*An association* represents a semantic relationship between the objects of a class, *a role* specifies the function of a class for a given association, *a cardinality (multiplicity)* specifies the number of instances participating in a relationship, there are two types of association: ***Association in active verb form:*** specifies the main reading direction of an association. ***Restricted navigability association***: Reducing the scope of the association is often done in the implementation phase, but can also be expressed in a model to indicate that instances of one class do not "know" instances of another.
Composition		Composition, also known as "composite aggregation", is a special kind of aggregation. This means that any composition can be replaced by an aggregation, which in turn can be replaced by an association. The only consequence is the loss of information.
Aggregation		This is a relationship between two classes, specifying that objects of one class are components of the other class. An aggregation relationship therefore makes it possible to define objects composed of other objects.

Dependency		A dependency is a one-way relationship expressing a semantic dependency between model elements.
Generalization		Inheritance is a mechanism for transmitting the properties of a class (its attributes and methods) to a subclass.
Realization		It is used to indicate that an implementation class implements one or more types.
Association n-ary		This is an association that links more than two classes.
Association class		An association can be refined and have its own properties, which are not available in any of the classes it links. As, in the object model, only classes can have properties, this association then becomes a class called a "class-association".

TAB. 1.3 - Relationships in a class diagram.

In addition to the relationships presented above, there are other associations such as:

Association with constraints

Adding constraints to an association, or between associations, provides more information, as it makes it easier to specify the scope and meaning of the association. Constraints are enclosed in braces and can be expressed in any language.

Derivative association

A derived association is conditional on, or can be deduced from, another association. Often a set of constraints, expressed in OCL, is added to a derived association to define the derivation conditions.

Qualified association

Allows you to select a subset of objects from all those participating in an association. The association restriction is defined by a key, which is used to select the targeted objects.

1.6.5. Creating a class diagram

Several approaches have been defined for building class diagrams. One commonly used approach is **[Aud07]** :

- **Find the classes of the domain under study:** This empirical step is generally carried out in collaboration with a domain expert. Classes generally correspond to concepts or nouns in the domain.
- **Find associations between classes:** Associations often correspond to verbs or verbal constructions, linking several classes. Note that some attributes are actually relationships between classes.
- **Finding class attributes:** Attributes often correspond to nouns or noun phrases. Adjectives and values often correspond to attribute values. You can add attributes at any stage of the project lifecycle (including implementation), and don't expect to find all the attributes as soon as you build the class diagram.
- **Organize and simplify the model:** by eliminating redundant classes and using inheritance.
- **Iterating and refining the model:** A model is rarely right the first time it is built. Object modeling is a non-linear, iterative process.

1.6.6. Example of a class diagram

This class diagram models a university system showing the relationships between students, courses, departments, and faculties. The diagram shows the main UML class diagram elements including classes with attributes and operations, several types of relationships (association, aggregation, composition, inheritance), multiplicity constraints, and dependencies between the entities.

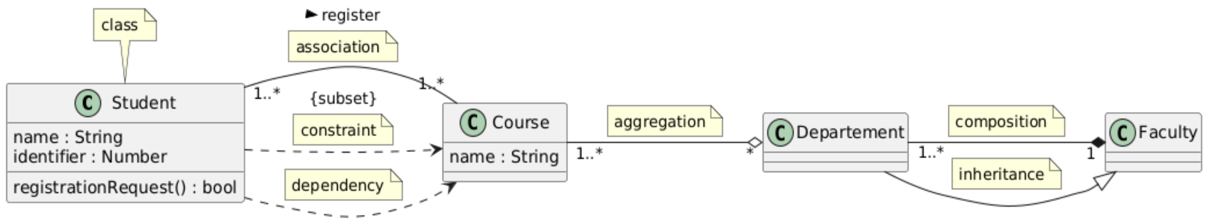


FIG. 1.8 - Example of a class diagram.

1.7. Object diagram

The static description of a system is based on a class diagram. This simplifies modeling by synthesizing common characteristics and covering a large number of objects. Sometimes, however, it is useful or even necessary to add an object diagram. An object diagram represents objects (i.e. instances of classes) and their links (i.e. instances of relationships) to give a fixed view of the state of a system at a given time **[Aud07]**. An object diagram can be used to **[Aud07]** :

- ✓ illustrate the class model by showing an example that explains the model;
- ✓ clarify certain aspects of the system by highlighting details that are imperceptible in the class diagram ;
- ✓ express an exception by modeling special cases or non-generalizable knowledge that are not modeled in a class diagram ;
- ✓ take a snapshot of a system at a given time.

The class diagram models rules, while the object diagram models facts. Often, the class diagram is used as a template to instantiate the classifiers to obtain the object diagram. The two main classes instantiated in the object diagram are classes and relationships.

<u>Example</u>:
These diagrams show a university enrollment system where the class diagram defines the structure with Student and Course classes, and the object diagram illustrates specific instances with actual data values. The class diagram establishes the one-to-many relationship pattern, and the object diagram demonstrates concrete examples of this relationship with student Alice enrolled in two courses.

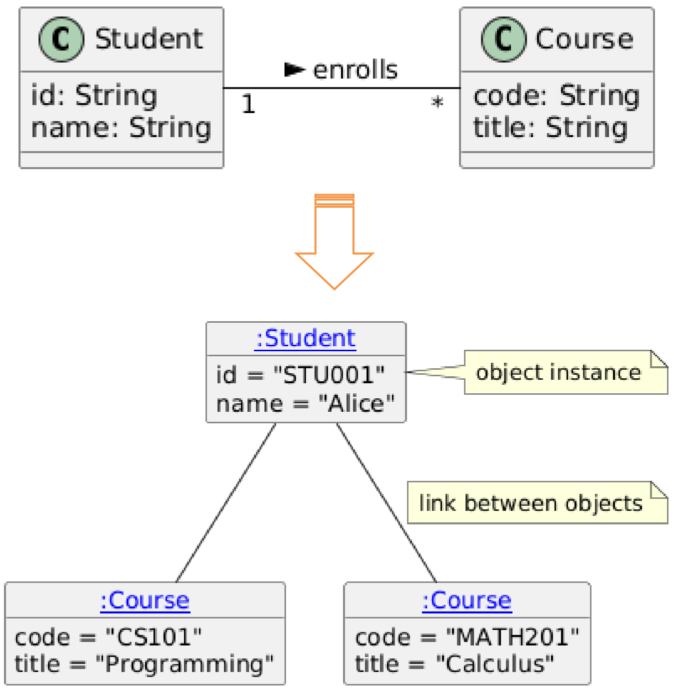


FIG. 1.9 - Example of a class and its object diagram.

1.8. Going further with UML

1.8.1. Meta-modeling with UML

Each element used in UML diagrams has its own syntax and semantics, defined by the language with a meta-model. A meta-model literally means model of the model. The UML meta-model defines the concepts available for modeling applications, i.e., it defines the language for specifying a model, whereas a model defines the language for describing an information domain. Each concept defined in an application is an instance of another concept defined in the meta-model. The UML meta-model has syntax and semantics, and is described in a combination of graphical notation, natural language and formal language **[OMG10]**:

Syntax: defined in the specification consists of: an *abstract syntax* and *a concrete syntax*.

- ✓ The abstract syntax given with a meta-model defining relationships between elements. It is presented in a UML class diagram showing the meta-classes defining the constructs and their relationships.
- ✓ Concrete syntax defining graphic notation.

Semantics: assigns meaning to syntactic expressions. It uses natural language and OCL[2] (*Object Constraint Language*).

- ✓ Natural language: UML's dynamic semantics are described in English.
- ✓ OCL: OCL rules are used to describe the static semantics of UML.

1.8.2. UML extension mechanisms

The UML standard provides concepts and notations, which have been carefully designed to address the modeling needs of typical software developmentprojects. However, due to the diversity of application domains, developers may require additional notations or features beyond those defined in the UML standard. For this reason, UML provides internal mechanisms for extending the language in a controlled way. These extension mechanisms include **[BM03, OMG10]**:

- ✓ **Stereotypes.** With a stereotype, we can create a new UML element from an existing one by defining its own special properties (providing its own set of *labeled values*), semantics (providing its own *constraints*), or notation (providing its own graphical representation or icon). We can think of a stereotype as a meta-type, because each one creates the equivalent of a new class in the UML meta-model.
- ✓ **Labeled values.** Each element in the UML has its own set of properties; for example, classes have names, attributes and operations. With stereotypes, we can add new elements to the UML, and with a tagged value, we can define a new type of property that can be attached to an element. We can think of a tagged value as metadata, because its value applies to the element itself, not its instances.
- ✓ **Constraints.** Everything in UML has its own semantics. With constraints, we can add new semantics to any element. Essentially, a constraint specifies a condition that must be true for one or more values in a model (or part). Constraints can be described by informal explanations (such as free text) or by OCL expressions.

A coherent set of extensions, defined for specific purposes using these extensibility mechanisms, constitutes a UML profile. In addition, it should be noted that it is possible to

[2] https://www.omg.org/spec/OCL/About-OCL/

extend the UML meta-model by defining new constructs using the specification technique described above.

1.8.3. Notion of UML profile

A profile is a standardization of an extended UML meta-model, and is consequently adapted to a business or middleware domain. For example, it can be used to specify that a UML class is in fact an EJB session. To compose a UML profile, you need to use stereotypes, tagged values and constraints.

A UML profile can contain :

- The elements selected in the reference meta-model.
- Extensions using different extension mechanisms.
- Semantic descriptions of extensions.
- Additional notations.
- Rules for validation, presentation and transformation.

UML has several standard profiles, including EJB, CORBA, SPEM and others. Almost all of these offer generation rules that make models productive. What's more, the literature is full of profiles proposed by researchers with the aim of providing languages adapted to the various fields in which they are working.

1.9. Software development process

UML is a language for representing models, but it does not define the process for building models. However, the literature on UML recommends that, to take advantage of UML, you follow a process with the following characteristics:

Iterative and incremental

The idea is simple: to model (understand and represent) a complex system, it's best to go about it in stages, refining the analysis in stages **[Pie09]**. The aim is to better master the unknowns and uncertainties that characterize complex systems.

Guided by the needs of system **users**

With UML, it's the users who guide the definition of models. The aim of the system to be modeled is to meet the needs of its users (users are the system's clients)**.**

Focus on software architecture

The right architecture is the key to successful development. It describes strategic choices which largely determine the qualities of the software (adaptability, performance, reliability...). We present Kruchten's architecture model **[Kru95]**, which proposes different, independent and complementary perspectives to define an architecture model. This view ("4+1") has strongly inspired UML :

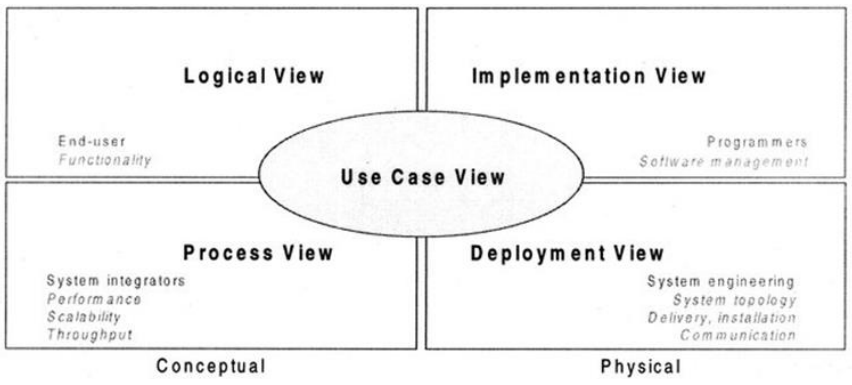


FIG. 1.10 - A Kruchten architecture model **[Kru95].**

- ✓ *use case view*: this shows the system's functionalities as perceived by external stakeholders;
- ✓ *logical view*: shows how functionalities are designed in the system in terms of static and dynamic structure;
- ✓ *realization view*: shows the organization of the code and its execution context;
- ✓ *process view*: shows the main elements of the system relating to process performance (decomposition of the system in terms of processes (tasks); interactions between processes (communication); synchronization and communication of parallel activities (threads)).
- ✓ *deployment view*: this view, very important in distributed environments, describes hardware resources and the allocation of software components to these resources.

According to the authors of UML, a development process that possesses these qualities should promote the success of a project. Examples of UML model development processes with these characteristics are the Rational Unified Process (RUP)[3] from IBM/Rationale, and the Y-cycle (2TUP - 2 Tracks Unified Process)[4] from Valtech.

[3] http://www-136.ibm.com/developerworks/rational/products/rup

[4] http://www.valtech.fr

1.10. Review

1.10.1. Exercise01

1. Name the different UML diagrams?
   - *Class diagram, Object diagram, Component diagram, Deployment diagram, Package diagram, Composite structure diagram, Use case diagram, Activity diagram, State-transition diagram, Communication diagram, Sequence diagram, Global interaction diagram, Time diagram.*
2. Is UML a language or a method? Why?
   - *language, it has no process*
3. Is UML a semi-formal or formal language? justify?
   - *semi-formal,*
   - *well-defined syntax, imprecise semantics (informal).*
4. Assuming you're going to develop a simple application, what's the minimum number of UML diagrams you can use in your design to have your system almost completely modeled?
   - *Use case diagrams, class diagrams, sequence diagrams (state/transition diagrams can also be added)*
5. Suppose you're doing a UML design for a complex system, you've found that the diagrams you're working on don't provide the mechanisms you need for this modeling, what should you do and how?
   - *I need to propose a UML extension using UML extension mechanisms (stereotypes, tagged values, Constraint).*
6. What is a software development process? Give three examples of processes.
   - *A process defines a sequence of steps, some of which are ordered, leading to the creation of a software system or the evolution of an existing system.*
   - *Cascade cycle, V-cycle, spiral cycle ...*
7. Why use OCL?
   - *OCL constraints provide a precise specification.*
8. Is it possible to generate a complete C++ application from a UML model?

   Yes, like Java code generation, but there are still limits that require developer intervention.

Chapter 2

Design patterns

In this chapter :

2.1. Introduction

Design patterns provide advanced conception techniques for software development, offering reusable solutions to common design problems. By encapsulating best practices, they help developers build scalable, flexible, and maintainable systems. These patterns serve as blueprints, guiding the structure and interaction of classes and objects to solve complex design challenges.

This chapter explores key design patterns, their applications, and how they can elevate the conception and architecture of robust software systems.

2.2. History

Design Patterns originated in the work of architect Christopher Alexander[1]. This work is a capitalization of experience that has highlighted patterns in building architecture.

On the same principle, in 1995, the book "Design Patterns -- Elements of Reusable Object-Oriented Software" by the GoF, Gang Of Four (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides), presented 23 Design Patterns. In this book, each Design Pattern is accompanied by examples in C++ and Smalltalk.

2.3. Definition

In software architecture, a Design Pattern is the description of a solution to a design problem. To be the subject of a Design Pattern, a solution must be reusable. A Design Pattern is said to be "proven" if it has been used in at least three cases **[Lon14]**.

2.4. Interests

Design Patterns improve development quality and shorten development times. Their application reduces coupling (points of dependency) within an application, brings flexibility, facilitates maintenance and generally helps to respect "good development practices".

[1] https://fr.wikipedia.org/wiki/Christopher_Alexander

2.5. Design pattern categories

There are three categories of design patterns: **Creation, Structure** and **Behavior**. We'll come back to them in detail below, explaining an example for each.

2.6. Scope of Design Patterns

The scope of Design Patterns concerns:

✓ **Class scope**

Focus on relationships between classes and their subclasses. Reuse by Inheritance is adopted.

✓ **Instance (Object) scope**

Focus on relationships between objects. Reuse by composition is adopted.

2.7. GoF Design Patterns

The book "Design Patterns: Elements of Reusable Object-Oriented Software" by authors Gamma, Helm, Johnson and Vlissides, who form the so-called GoF (Gang of Four), introduced 23 design patterns classified into the three design pattern categories and with different scopes **[Gam95]**.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight (195) Observer State Strategy Visitor

TAB. 2.1 - GoF design patterns.

2.8. Presentation of a Design Pattern

To present a design pattern, we need to follow this model **[You20] [Lon14]**:

- ✓ **Pattern name**
  - Used to describe the pattern, its solutions and consequences in a word or two.
- ✓ **Problem**
  - Description of application conditions. Explanation of the problem and its context
- ✓ **Solution**
  - Description of elements (objects, relationships, responsibilities, collaboration)
  - To design the solution to the problem; use class diagrams, sequence diagrams, etc.
  - Static and dynamic vision of the solution
- ✓ **Consequences**
  - Description of the results (induced effects) of applying the pattern to the system (positive and negative effects)

2.9. Prerequisites

2.9.1. Reminder

IT project requirements **[You20]**:

- ✓ Functional requirements (business needs)
- ✓ Technical requirements (performance, maintenance, security, portability, distribution, etc.)
- ✓ Financial requirements.

It is therefore very difficult to develop a software system that meets these requirements without drawing on the experience of others:

- ✓ Reuse design patterns
- ✓ Build applications on existing architectures (JEE, DotNet, etc.)

2.9.2. Object-oriented paradigm

Fundamental object-oriented concepts **[You20]**:

- ✓ Class
- ✓ Object
- ✓ Heritage

- ✓ Encapsulation
- ✓ polymorphism.

Why was the object-oriented approach created?

- ✓ To facilitate the **reuse of** other people's experience.
    - Instantiate existing classes (Composition)
    - Create new derived classes (Inheritance)
    - Frameworks reuse
- ✓ Create applications that are easy to maintain:
    - Applications closed to modification and open to extension
- ✓ Creating high-performance applications
- ✓ Creating secure applications
- ✓ Creating distributed applications
- ✓ Separating the different aspects of an application
    - Functional aspects
    - Presentation aspects (Web, Mobile, Desktop, etc.)
    - Technical aspects (transaction management, security, etc.)

2.9.3. Heritage and Composition

In object-oriented programming, **inheritance** and **composition (delegation)** are two ways of reusing classes.

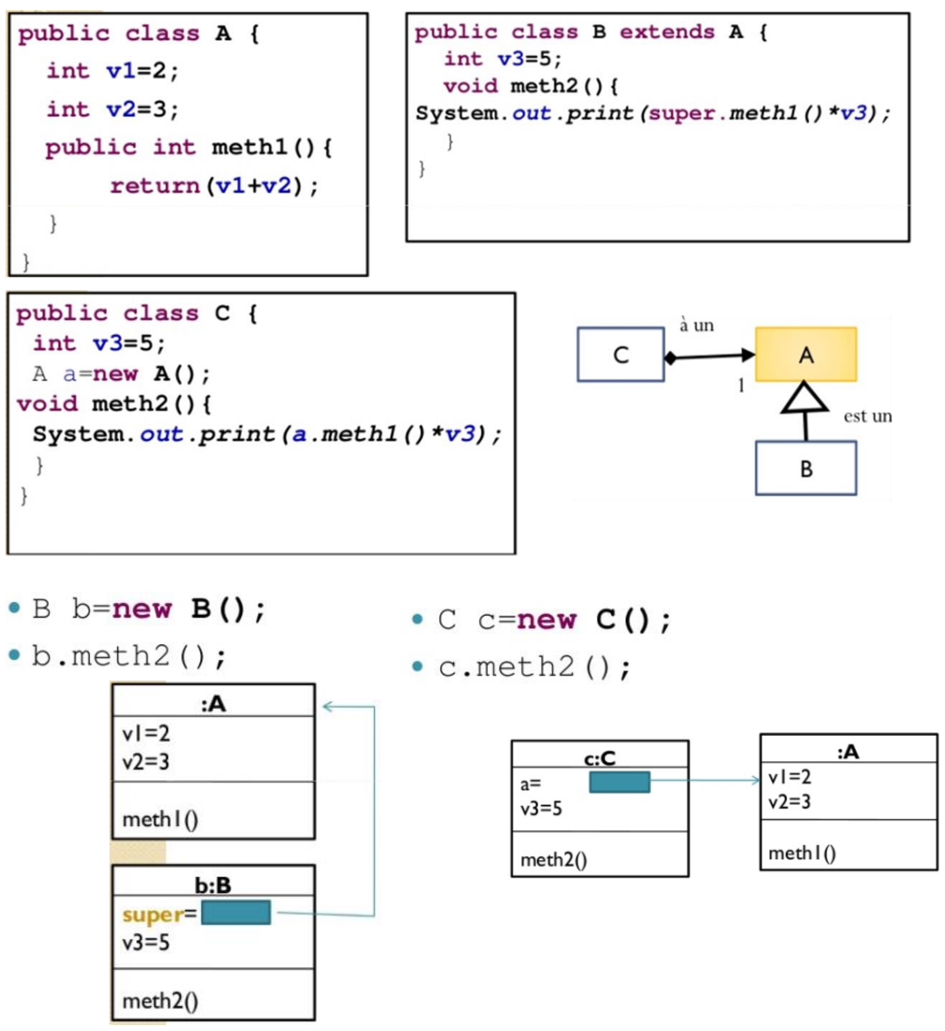
We talk about delegation (composition) when a class (client) **delegates** part of its activity to a class (server). Code is thus reused between classes.

Inheritance translates as "is one" or "a kind of".

Composition translates the term "has a" or "has many".

Let's look at an example between inheritance and "has a" type composition

<u>Example</u>:



```
public class A {
  int v1=2;
  int v2=3;
  public int meth1(){
       return(v1+v2);
  }
}
```

```
public class B extends A {
   int v3=5;
   void meth2(){
System.out.print(super.meth1()*v3);
   }
}
```

```
public class C {
 int v3=5;
 A a=new A();
void meth2(){
 System.out.print(a.meth1()*v3);
 }
}
```

- `B b=new B();`
- `b.meth2();`

- `C c=new C();`
- `c.meth2();`

FIG. 2.1 - Example of reuse through inheritance and composition **[You20]**

2.10. GoF tips

As a designer:

"program with interfaces, not implementations", i.e., limit all links between classes and replace them with links to interfaces.

"favor composition over inheritance", i.e., favor reuse by composition (delegation) instead of using inheritance.

The following explanation of different categories of GoF design patterns are mainly inspired by these references **[You20] [Lon14] [Dou99].**

2.11. Creative patterns

Description of how an object or set of objects can be created, initialized, and configured.

Isolation of the code relating to creation and initialization in order to make the application independent of these aspects.

Example: Abstract Factory, Builder, Prototype, Singleton.

2.11.1. Singleton

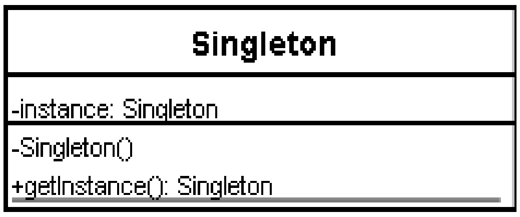


FIG. 2.2 -The Singleton UML diagram

Goals:

- ✓ Restrict the number of instances of a class to one and only one.
- ✓ Provide a method to access this single instance.

Reasons to use it:

- ✓ The class must have only one instance.
- ✓ This could be the case for a system resource for example.
- ✓ The class prevents other classes from instantiating it. It owns the only instance of itself and provides the only method to access that instance.

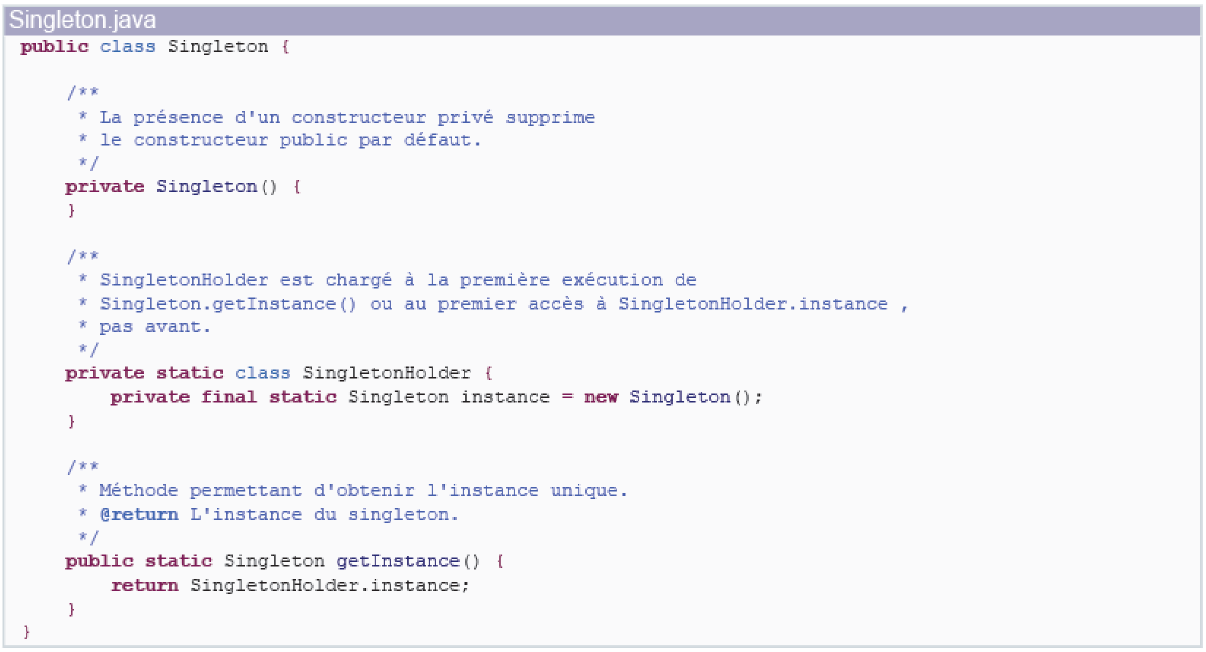
Result:

- ✓ The Design Pattern allows you to isolate the uniqueness of an instance.

Responsibilities:

- ✓ Singleton must restrict the number of its own instances to one and only one. Its constructor is private : this prevents other classes from instantiating it. The class provides the static getInstance() method which allows you to obtain the single instance.

JAVA implementation:



Singleton.java

```
public class Singleton {

    /**
     * La présence d'un constructeur privé supprime
     * le constructeur public par défaut.
     */
    private Singleton() {
    }

    /**
     * SingletonHolder est chargé à la première exécution de
     * Singleton.getInstance() ou au premier accès à SingletonHolder.instance ,
     * pas avant.
     */
    private static class SingletonHolder {
        private final static Singleton instance = new Singleton();
    }

    /**
     * Méthode permettant d'obtenir l'instance unique.
     * @return L'instance du singleton.
     */
    public static Singleton getInstance() {
        return SingletonHolder.instance;
    }
}
```

FIG. 2.3 - Java implementation of the Singleton **[Dou99]**

Practical Example

To not create connections to a database, assuming I have a "**Connection**" class. We don't want to detail the implementation of this class (ie the connection with the comic strip is fictitious). But what interests us is the creation of a single instance of this class. Then we just display a message, for example "**creating a connection**" to see how many instances of this class have been created. We use the DP Singleton to achieve this goal. We propose a Singleton " **SigletonConnexion** " which is responsible for creating the single instance.

In the Singleton:

- ✓ Declare a 'unique' static variable of type "**Connection**".
- ✓ Initialize this variable using the static block "**static {...}**".
- ✓ Afterwards, we declare the "**getInstace**" method which will return the unique variable (that's all it does).

Test how this Singleton works:

- ✓ Create a new test class
- ✓ To create a connection, we will no longer call **"new Connection"** , on the contrary, we must call the Singleton (the method which returns the single instance).
- ✓ Repeat the previous step a few times to see that there is only a single instance of the " **connection** " class

2.12. Structural patterns

Description of the way in which application objects must be connected in order to make these connections independent of future developments of the application.

Example: Adapter(object) , Composite , Bridge , Decorator , Facade , Proxy

2.12.1. Facade Pattern

Problem: Complex System

A subsystem is complex if it uses several interfaces.

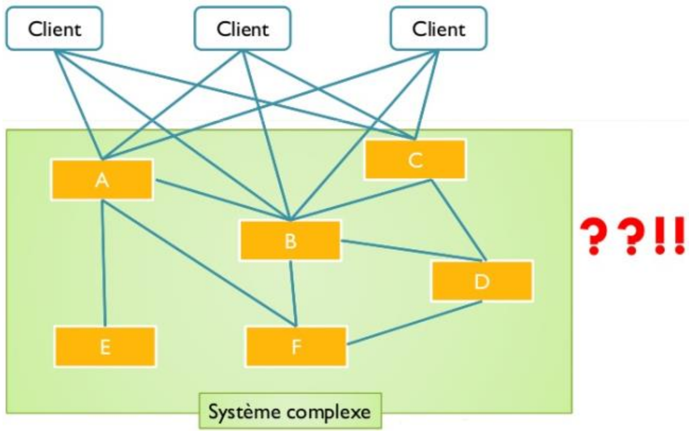


FIG. 2.4 - A complex system **[You20]**

Solution: Facade

Create a single facade for this subsystem.

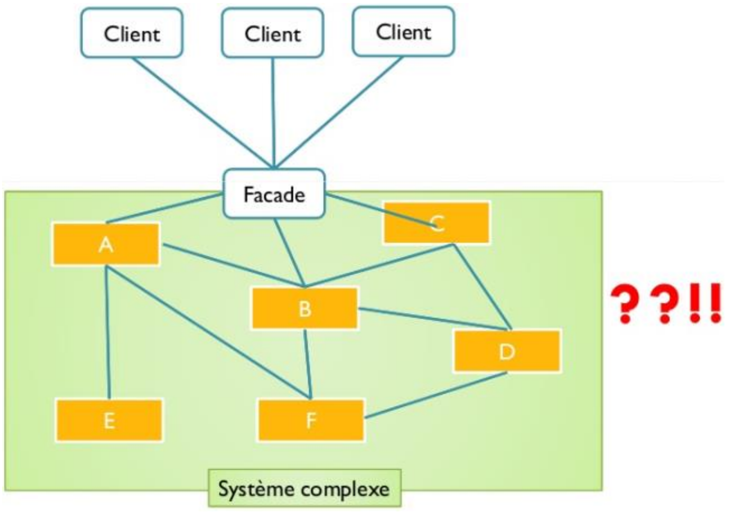


FIG. 2.5 - The solution to a complex system **[You20]**

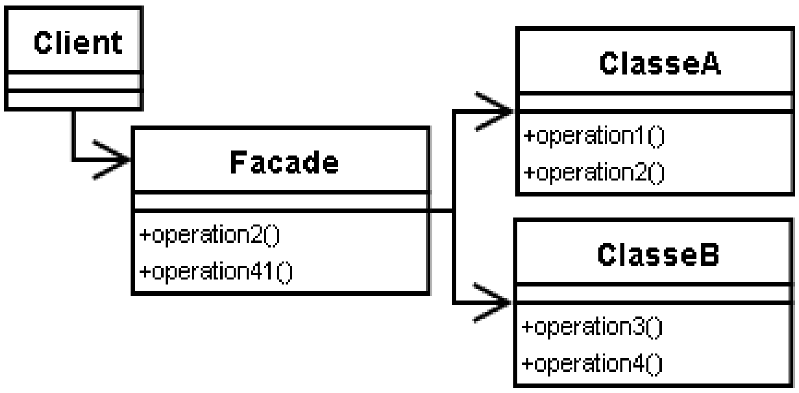


FIG. 2.6 - UML Façade diagram

Goals:

- ✓ Provide a single interface in replacement of a set of interfaces of a subsystem.
- ✓ Define a high-level interface to make the subsystem easier to use.

Reasons to use it:

- ✓ The system has a complex subsystem with several interfaces.
- ✓ Some of these interfaces present operations that are not useful to the rest of the system.
- ✓ This may be the case of a subsystem communicating with **measurement tools or of a database access** subsystem.
- ✓ It would be more judicious to use a single interface presenting only the useful operations.
- ✓ A single class, the facade, presents these truly necessary operations.

Result:

The Design Pattern makes it possible to isolate the functionalities of a subsystem useful to the client part.

Responsibilities:

- ✓ **ClassA** and **ClassB** : implement various features.
- ✓ **Facade** : presents certain functionalities. This class uses the implementations of **ClassA** and **ClassB objects** . It exposes a simplified version of the **ClassA - ClassB** subsystem .
- ✓ The client part uses the methods presented by the **Facade object** . There are therefore no dependencies between the client part and the **ClassA - ClassB** subsystem .

JAVA implementation:

ClasseA.java

```java
/**
 * Classe implémentant diverses fonctionnalités.
 */
public class ClasseA {

    public void operation1() {
        System.out.println("Methode operation1() de la classe ClasseA");
    }

    public void operation2() {
        System.out.println("Methode operation2() de la classe ClasseA");
    }
}
```

ClasseB.java

```java
/**
 * Classe implémentant d'autres fonctionnalités.
 */
public class ClasseB {

    public void operation3() {
        System.out.println("Methode operation3() de la classe ClasseB");
    }

    public void operation4() {
        System.out.println("Methode operation4() de la classe ClasseB");
    }
}
```

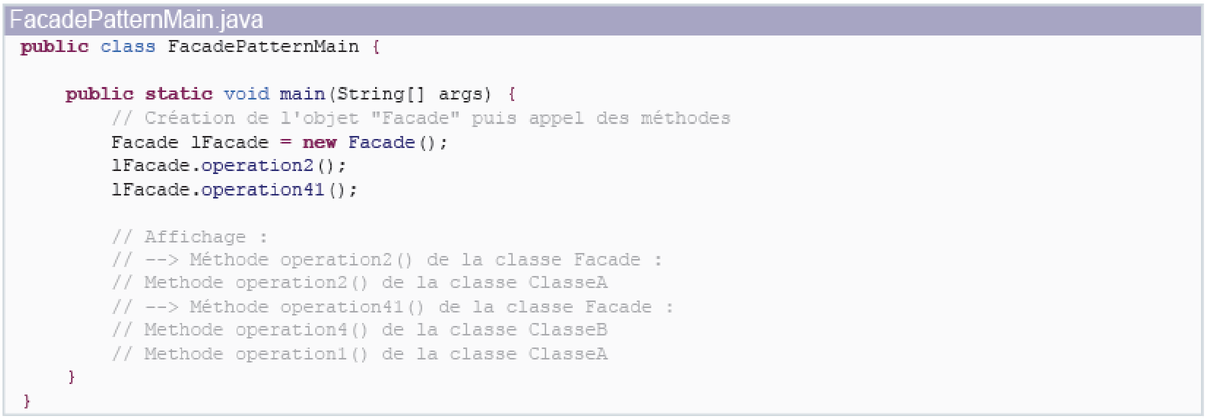
Facade.java

```java
/**
 * Présente certaines fonctionnalités.
 * Dans ce cas, ne présente que la méthode "operation2()" de "ClasseA"
 * et la méthode "operation41()" utilisant "operation4()" de "ClasseB"
 * et "operation1()" de "ClasseA".
 */
public class Facade {

    private ClasseA classeA = new ClasseA();
    private ClasseB classeB = new ClasseB();

    /**
     * La méthode operation2() appelle simplement
     * la même méthode de ClasseA
     */
    public void operation2() {
        System.out.println("--> Méthode operation2() de la classe Facade : ");
        classeA.operation2();
    }

    /**
     * La méthode operation41() appelle
     * operation4() de ClasseB
     * et operation1() de ClasseA
     */
    public void operation41() {
        System.out.println("--> Méthode operation41() de la classe Facade : ");
        classeB.operation4();
        classeA.operation1();
    }
}
```

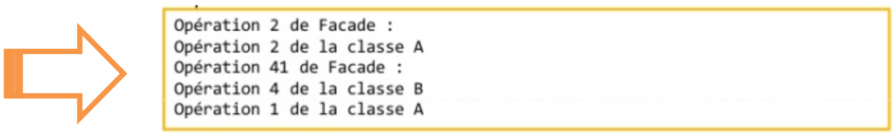


FacadePatternMain.java

```
public class FacadePatternMain {

    public static void main(String[] args) {
        // Création de l'objet "Facade" puis appel des méthodes
        Facade lFacade = new Facade();
        lFacade.operation2();
        lFacade.operation41();

        // Affichage :
        // --> Méthode operation2() de la classe Facade :
        // Methode operation2() de la classe ClasseA
        // --> Méthode operation41() de la classe Facade :
        // Methode operation4() de la classe ClasseB
        // Methode operation1() de la classe ClasseA
    }
}
```



```
Opération 2 de Facade :
Opération 2 de la classe A
Opération 41 de Facade :
Opération 4 de la classe B
Opération 1 de la classe A
```

FIG. 2.7 - The Java implementation of Façade **[Dou99]**

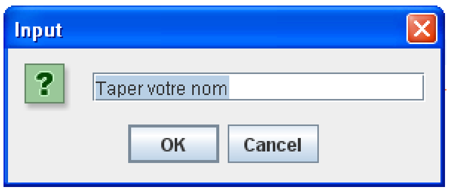
Practical Example 1

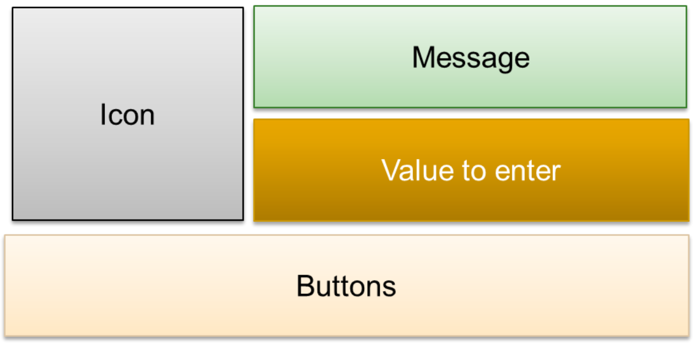
The **javax.swing.JOptionPane** class provides around twenty static methods allowing you to manipulate dialog windows of different types presenting the structure below. It is a facade which hides all the components constituting a modal dialog window and provides a simple interface to the client to manipulate this set.

Instruction:

String rep = JOptionPane.showInputDialog (myPanel , 'Type your name') ;

Brings up the window below and returns the user's response after input.





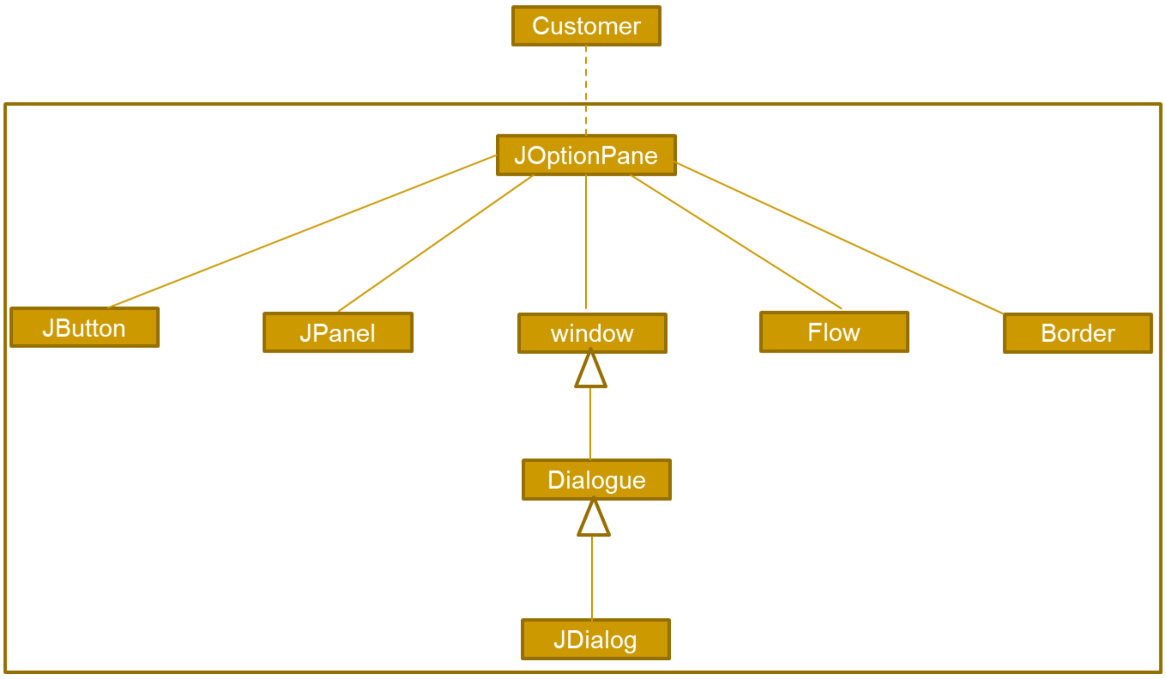
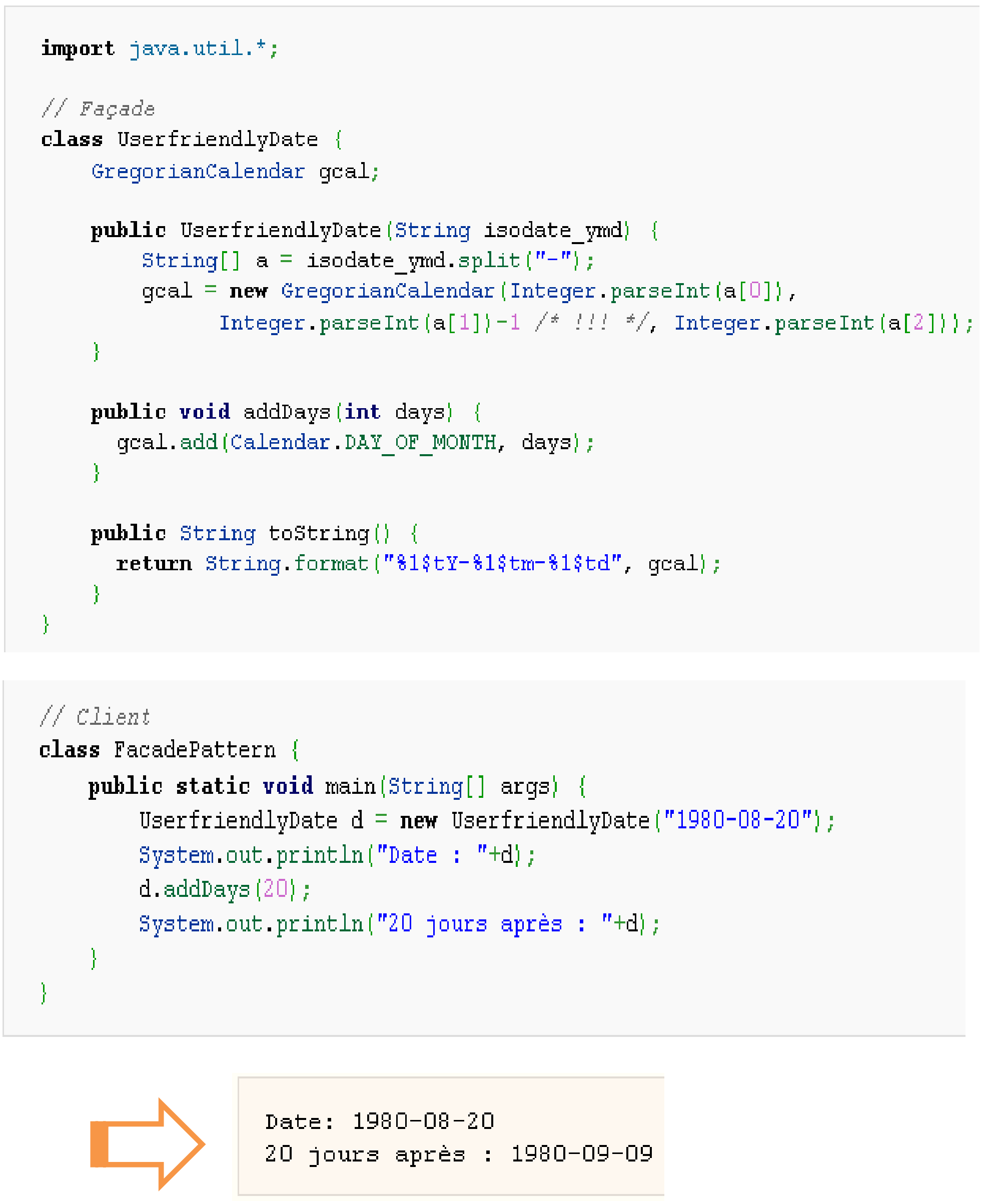


FIG. 2.8 - The javax.swing.JoptionPane Façade **[Lon14]**

Practical Example 2

The following example hides a complicated calendar management API behind a simpler facade. It displays:



```java
import java.util.*;

// Façade
class UserfriendlyDate {
    GregorianCalendar gcal;

    public UserfriendlyDate(String isodate_ymd) {
        String[] a = isodate_ymd.split("-");
        gcal = new GregorianCalendar(Integer.parseInt(a[0]),
              Integer.parseInt(a[1])-1 /* !!! */, Integer.parseInt(a[2]));
    }

    public void addDays(int days) {
      gcal.add(Calendar.DAY_OF_MONTH, days);
    }

    public String toString() {
      return String.format("%1$tY-%1$tm-%1$td", gcal);
    }
}
```

```java
// Client
class FacadePattern {
    public static void main(String[] args) {
        UserfriendlyDate d = new UserfriendlyDate("1980-08-20");
        System.out.println("Date : "+d);
        d.addDays(20);
        System.out.println("20 jours après : "+d);
    }
}
```

```
Date: 1980-08-20
20 jours après : 1980-09-09
```

FIG. 2.9 - The complicated calendar Facade

2.13. Behavioral Patterns

Description of interaction behaviors between objects. Manage dynamic interactions between classes and objects

Example: Strategy, Observer, Iterator, Mediator, Visitor, State

2.13.1. Template Method

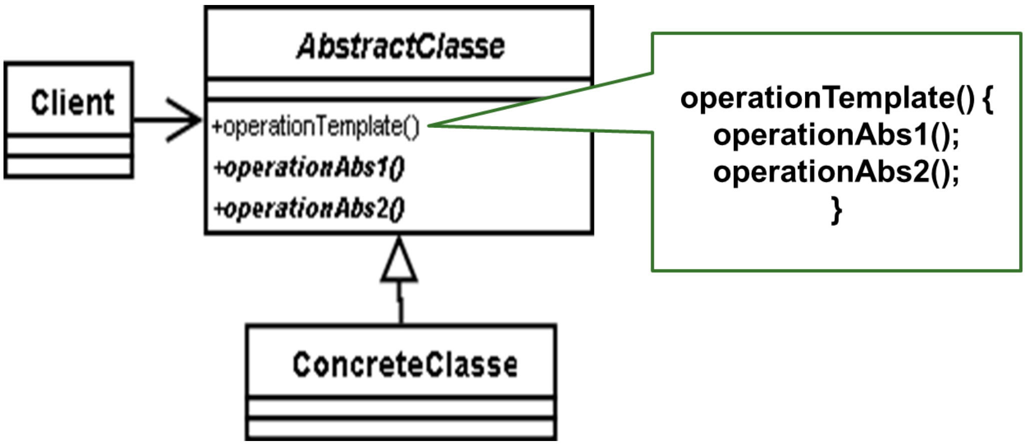


FIG. 2.10 - UML diagram of Template Method

Objectives :

Define the skeleton of an algorithm by delegating certain steps to subclasses.

Reasons to use it :

- ✓ A class **operates globally**. But **the details of its algorithm** must be specific to its subclasses.
- ✓ This may be the case for a computer document. The document has a global function where it is **saved**. To save, you always need **to open the file**, **write to it** and then **close it**. But, depending on the type of document, it will not be saved in the same way. If it's a word processing document, it will be saved as a sequence of bytes. If it's an HTML document, it will be saved as a text file.
- ✓ The general part of the algorithm (saving) is managed by the **abstract class** (document). The general part opens and closes the file and calls a write method. The specific part of the algorithm (writing to the file) is defined at the level of **concrete classes** (word processing document or HTML document).

Result:

- ✓ The Design Pattern is used to isolate the variable parts of an algorithm.

Responsibilities:

- ✓ **AbstractClass**: defines abstract primitive **methods**. The class implements the skeleton of an algorithm that calls primitive methods.

- **ConcreteClass**: is a concrete subclass of AbstractClass. It implements the methods used by the algorithm of **AbstractClass**'s **operationTemplate()** method.
- The client part calls the **AbstractClass** method that defines the algorithm.

JAVA implementation :

AbstractClasse.java

```java
/**
 * Définit l'algorithme
 */
public abstract class AbstractClasse {

    /**
     * Algorithme
     * La méthode est final afin que l'algorithme
     * ne puisse pas être redéfini par une classe fille
     */
    public final void operationTemplate() {
        operationAbs1();
        for(int i=0;i<5;i++) {
            operationAbs2(i);
        }
    }

    // Méthodes utilisées par l'algorithme
    // Elles seront implémentées par une sous-classe concrète
    public abstract void operationAbs1();
    public abstract void operationAbs2(int pNombre);
}
```

ConcreteClasse.java

```java
/**
 * Sous-classe concrète de AbstractClasse
 * Implémente les méthodes utilisées par l'algorithme
 * de la méthode operationTemplate() de AbstractClasse
 */
public class ConcreteClasse extends AbstractClasse {

    public void operationAbs1() {
        System.out.println("operationAbs1");
    }

    public void operationAbs2(int pNombre) {
        System.out.println("\toperationAbs2 : " + pNombre);
    }
}
```

TemplateMethodPatternMain.java

```java
public class TemplateMethodPatternMain {

    public static void main(String[] args) {
     // Création de l'instance
        AbstractClasse lClasse = new ConcreteClasse();
        // Appel de la méthode définie dans AbstractClasse
        lClasse.operationTemplate();

        // Affichage :
        // operationAbs1
        //          operationAbs2 : 0
        //          operationAbs2 : 1
        //          operationAbs2 : 2
        //          operationAbs2 : 3
        //          operationAbs2 : 4
    }
}
```

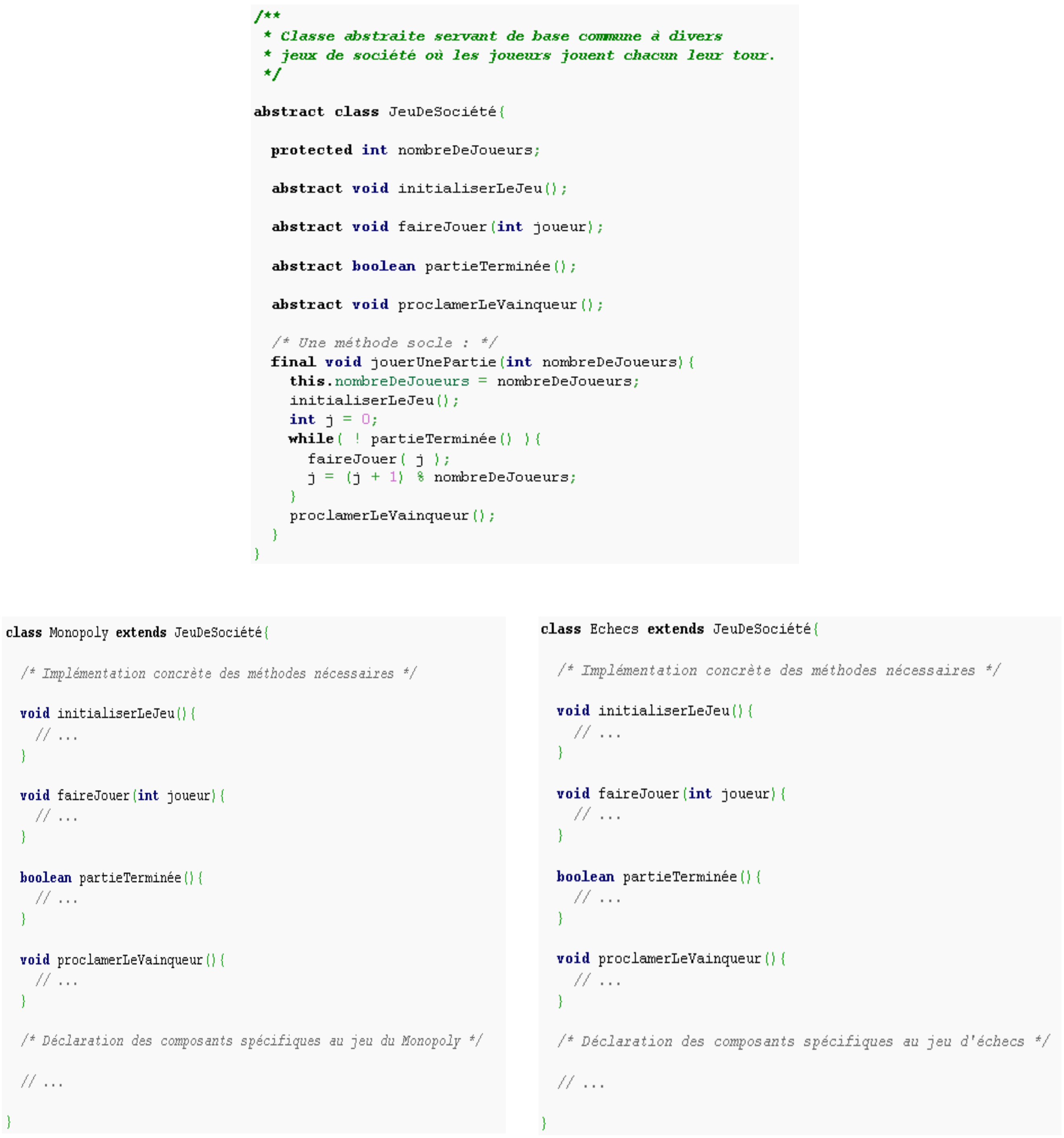
FIG. 2.11 - Java implementation of Template Method **[Dou99]**

Practical example 1

- ✓ This DP can be used in **data compression** algorithms.
- ✓ In fact, there are several data compression techniques, but they all happen to have a few things in common. I.e., all compression algorithms work in the same way, there's a common skeleton. But then there are variants within this algorithm. i.e. there are steps in the algorithm that can be done in different ways (versions), leaving the developer to create his own implementation.

Practical example 2

- ✓ In board games, after an initialization phase, each player takes a turn until the game is over. Rather than being repeated in every game, this basic skeleton can be encoded in an abstract "**BoardGame**" class, as a "**final void playAparty (int numberOfPlayers)**" method. The "**SocietyGame**" class also includes four abstract methods: "**void initializeTheGame()**", "**void makePlay(int player)**", "**boolean gameTerminated()**" and "**void proclaimTheWinner()**".
- ✓ These methods are encoded in subclasses corresponding to the different games.
- ✓ We can now derive this class to implement various games:



```
/**
 * Classe abstraite servant de base commune à divers
 * jeux de société où les joueurs jouent chacun leur tour.
 */

abstract class JeuDeSociété{

  protected int nombreDeJoueurs;

  abstract void initialiserLeJeu();

  abstract void faireJouer(int joueur);

  abstract boolean partieTerminée();

  abstract void proclamerLeVainqueur();

  /* Une méthode socle : */
  final void jouerUnePartie(int nombreDeJoueurs){
    this.nombreDeJoueurs = nombreDeJoueurs;
    initialiserLeJeu();
    int j = 0;
    while( ! partieTerminée() ){
      faireJouer( j );
      j = (j + 1) % nombreDeJoueurs;
    }
    proclamerLeVainqueur();
  }
}
```

```
class Monopoly extends JeuDeSociété{

  /* Implémentation concrète des méthodes nécessaires */

  void initialiserLeJeu(){
    // ...
  }

  void faireJouer(int joueur){
    // ...
  }

  boolean partieTerminée(){
    // ...
  }

  void proclamerLeVainqueur(){
    // ...
  }

  /* Déclaration des composants spécifiques au jeu du Monopoly */

  // ...

}
```

```
class Echecs extends JeuDeSociété{

  /* Implémentation concrète des méthodes nécessaires */

  void initialiserLeJeu(){
    // ...
  }

  void faireJouer(int joueur){
    // ...
  }

  boolean partieTerminée(){
    // ...
  }

  void proclamerLeVainqueur(){
    // ...
  }

  /* Déclaration des composants spécifiques au jeu d'échecs */

  // ...

}
```

FIG. 2.12 - Implementing board games and their derivatives

2.14. Review

2.14.1. Exercise01

1. What's the key to object-oriented success?
   - ➢ *Reuse*

2. What is delegation?
   - ➢ *We speak of delegation (composition) when a class (client) delegates part of its activity to a class (server). Code is thus reused between classes.*

3. What is a Design Pattern?
   - ➢ *In software architecture, a Design Pattern is the description of a solution to a design problem.*

4. What are the different categories of Designs Patterns? Give examples of each.
   - ➢ *Creation (Example: Abstract Factory, Builder, Prototype, Singleton), Structure (Examples: Adapter, Composite, Bridge, Decorator, Facade, Proxy), Behavior (Examples: Strategy, Observer, Iterator, Mediator, Visitor, State).*

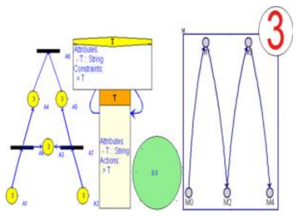
5. What are the benefits of applying Designs Patterns?
   - ➢ *Design Patterns improve development quality and shorten development times. Their application reduces coupling (points of dependency) within an application, brings flexibility, facilitates maintenance and generally helps to respect "good development practices".*

6. What two principles does GoF strongly recommend?
   - ➢ *"program with interfaces, not implementations".*
   - ➢ *"favor composition over inheritance"*

Chapter 3

Object-oriented design principles

In this chapter :



3.1. Introduction

Design patterns focus on specific problems. Despite their large number, they do not constitute a sufficient panoply to help develop complete solutions. Design principles, on the other hand, are fairly global and abstract precepts, aimed at facilitating the general design of systems that are easy to maintain and extend. By their very nature, they are far fewer in number than patterns, which makes them easier to assimilate. This leads some to consider them more "effective", from a pedagogical point of view, than design patterns.

The boundary between patterns and principles is not always clear. Some design patterns, such as KISS, YAGNI, or the first five GRASP patterns, correspond fairly closely to the definition of a design principle.

This chapter inspired by **[Lon14]** introduces the principles of object-oriented design.

3.2. SOLID design principles

SOLID is an acronym made up of the initials of the five design principles identified by Rober Martin in the early 2000s (Design Principles and Design Patterns).

- ✓ S (Single responsibility principle).
- ✓ O (Open-closed principle).
- ✓ L (Liskov substitution principle).
- ✓ I (Interface segregation principle).
- ✓ D (Dependency iversion principle).

SOLID are the most famous principles of object-oriented design. But there are other principles proposed by other authors such as inversion of control 'IOC', dependency injection 'DI', ...etc.).

3.3. Single Responsibility Principle (SRP)

Statement:

"There should never be more than one reason for a class to change. This is possible if the class focuses on a single responsibility, i.e. a single functional objective. It then has only one reason to evolve.

Discussion

If there are several reasons to modify a class, it's best to break it down into several more homogeneous classes, each designed around a single responsibility. In the event of a change, only the class concerned by the change will be impacted. Whereas with a class that mixes several responsibilities, changes in one are likely to impact the way the others are fulfilled. This principle is not easy to apply, in the absence of a precise method for identifying the responsibilities of a class and its granularity.

Example

We want to design an "Image" class. The first thing that comes to mind is its representation, display and storage in a file. A first possible design is to include everything in a single "Image" class. By doing so, the class is given several responsibilities. However, it is perfectly possible to manipulate images without displaying them (for example, in a command-line utility). Following the principle of single responsibility, we create three classes instead. "Image", responsible solely for managing its representation. A "DisplayImage" class responsible for display, and a "WarehouseImage" class responsible for saving and loading.

Links

The principle of single responsibility is similar to the idea of strong functional cohesion between modules.

3.4. Open-Closed Principle (OCP)

Statement

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification". As far as classes are concerned, this means that a class should be able to be modified/extended without having to touch its code. This does not, of course, apply to error correction, but only to functionality upgrades. In practice, modification will take the form of adding new classes, without modifying existing ones.

Discussion

This principle applies to all software entities. In addition to classes, it can also apply to methods and libraries. As far as classes are concerned, inheritance is the essential mechanism for implementing the principle. In the case of a library, for example, this principle is satisfied by adding a "new version" of a library element when it evolves, rather than modifying it. This allows customers who were using the old element to continue doing so.

Links

The Strategy and Method patterns exploit the open-closed principle in special cases. The case of a change/addition of algorithm for the Strategy pattern, the case of a change of part of a method for the Method pattern.

This is also the case for the various forms of Factory (the client is closed to the creation of new objects), Decorator (the target code is closed to the addition of new functionalities) and Visitor (the visited structure is closed to the addition of new algorithms).

3.5. Liskov substitution principle (LSP)

Statement

"If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T". (If for any object o1 of type S, there exists an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T".)

In simpler terms, any code that uses the objects of a class must be able to use the objects of its subclasses without creating difficulties.

Discussion

For Barbara Liskov, the fact that an object of subclass S is an object of superclass T must be examined from a behavioral point of view: the object of subclass S must always behave like the object of superclass T, and must not violate any of its expected characteristics. In other words, the question "Is an object S an object T?" must be replaced by the more precise question "For users, does an object S always behave in the same way as an object T?"

Inheritance respects this principle only if the derived class simply extends the parent class by adding functionality. Redefinition, on the other hand, changes the behavior of derived objects, which may no longer be substitutable for those of the parent class. Composition is therefore preferable to inheritance.

Links

The substitution principle is linked to the open-closed principle. By redefining a behavior in a subclass, we contradict the "contract" characterizing the class from which we derive. This superclass would then have to be modified to take account of this particular case. But this would violate the open-closed principle. In this case, it's better to forgo inheritance.

3.6. Dependency Inversion Principle (DIP)

Statement

*"High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions". (*High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions).

In other words, in a layered architecture, high-level modules must not depend on low-level modules. Modules at both levels must depend on abstractions.

Discussion

High-level modules, which describe the application's logic and business aspects, are often built from modules providing low-level services, such as those dedicated to building man-machine interfaces or communications.

While this may seem natural at first glance, it has two negative consequences.

- High-level modules need to be modified as low-level modules evolve.
- High-level modules (business logic) cannot be reused in a technical context other than the original one.

In a well-constructed architecture, a layer of abstraction - interfaces or abstract classes - must separate the levels.

Note that the dependencies are then reversed, from the lower-level modules to the upper-level interfaces (HMI and Com).

In this concept, the "business layer" becomes independent of the implementation of the lower layers. It depends only on their interfaces. It can therefore be reused in different contexts.

More generally, to reverse the dependency between two classes A and B, we introduce an IB interface (containing the methods of B that A can call) into a package made up of A and IB.

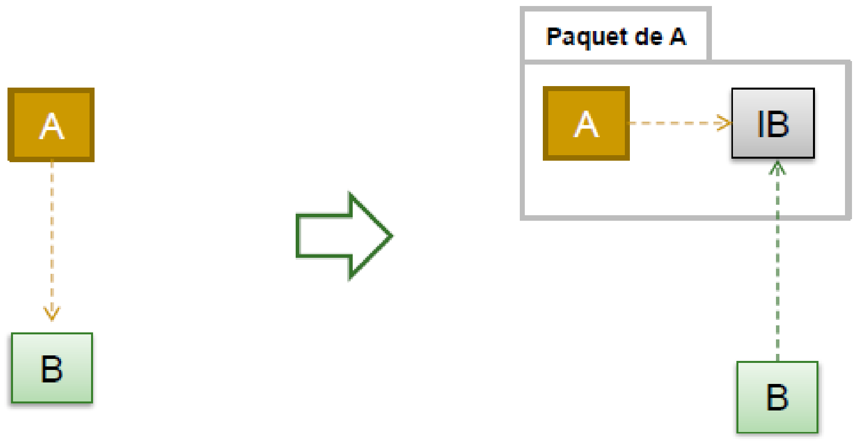


FIG. 3.1 - Dependency inversion

Links

Parton's method applies the principle of dependency inversion: abstract operations form the abstraction layer, and dependencies run from the lower to the upper layer.

3.7. Interface Segregation Principle (ISP)

Statement

"Clients should not be forced to depend upon interfaces that they do not use. (Clients should not be forced to depend upon interfaces that they do not use".) This somewhat abstract principle is often translated into more operational "advice": avoid "big interfaces" with lots of methods that are not always useful for all classes; split these "big interfaces" into more specialized, more coherent interfaces.

Discussion

A good practice in object programming is to use abstractions - interfaces, abstract classes - instead of concrete classes.

The existence of "large interfaces" with multiple methods is often a symptom of poor design. Either the concrete classes that implement them have little internal cohesion, or they only use part of the interface.

More specialized interfaces, corresponding to well-defined "roles", are preferable. Concrete classes implement one or more of these roles.

Links

There's an obvious link with the principle of single class responsibility. An interface, like a class, must correspond to a well-defined "role".

3.8. Review

3.8.1. Exercise01

1. What is SOLID? What are some examples?
   - *Design principles are fairly global and abstract precepts, aimed at facilitating the general design of systems that are easy to maintain and extend.*
   - *S (Single responsibility principle), O (Open-closed principle), L (Liskov substitution principle), I (Interface segregation principle), D (Dependency iversion principle)*

2. What's the difference between design principles and design patterns?
   Design patterns focus on specific problems. Design principles, on the other hand, are fairly general and abstract precepts.

3. Which SOLID design principle is the Singleton pattern related to?
   SOLID's Single Responsibility Principle.

4. Which SOLID design principle is the Facade pattern related to?
   The Facade pattern is closest to SOLID's Dependency Inversion Principle.

5. Which SOLID design principle is the Template Method pattern related to?
   SOLID's Open/Closed Principle.

6. How does the Liskov Substitution Principle impact OOP class design?
   Liskov's substitution principle stipulates that subclasses must be able to replace their base classes without altering the program's behavior. This ensures that subclasses respect the contracts of their parent classes. In OOP, this promotes consistency, reusability and reduces polymorphism errors by ensuring that subclasses behave in a predictable way.

Chapter 4

Model Driven Engineering

In this chapter :

4.1. Introduction

The creation and use of a large number of models enables flexible, iterative development. Each of these models contains part of the information needed to generate software applications. And to make models productive, we need to put them in relation to each other, and more specifically to their respective transformations, and that's thanks to model transformations.

In the context of MDE, model transformation plays a fundamental role and is considered one of the most promising techniques in this approach. The design process is reduced to a set of partially ordered model transformations. Each transformation takes models as input and produces others as output, until executable artifacts are obtained.

In this chapter, we'll start with an introduction to the concepts of MDE, MDA and model transformation, then go on to describe a few types of these transformations, along with their classification.

4.2. The MDE and MDA approaches

4.2.1. General principle

Model Driven Engineering (MDE)[1] is an approach that proposes the mechanization of the process that experienced engineers follow manually **[Bla05]**; it is more global and general than MDA.

Model Driven Architecture (MDA**) [OMG04]** is an approach proposed and supported by the Object Management Group (OMG). This approach can be seen as a variant of MDE for software engineering.

The MDA advocates the massive use of models and offers initial answers to how, when, what and why to model **[Bla05]**. Without claiming to be a modeling bible, listing all the best practices, it aims to highlight the intrinsic qualities of models, such as durability, productivity and consideration of execution platforms **[Bla05]**. MDA includes the definition of several standards, notably UML, MOF and XMI.

The key principle of MDA is the use of models in the various phases of an application's development cycle. The main objective of MDA is to develop perennial models, independent of the technical details of the execution platforms (J2EE, .Net, PHP or others), in

[1] MDE in English, IDM (Model Driven Engineering) in French

order to enable automatic generation of all application code and achieve significant productivity gains **[Bla05]**.

4.2.2. MDA architecture

The MDA can be summarized in the following pyramid, with four levels of abstraction **[Béz03]**:

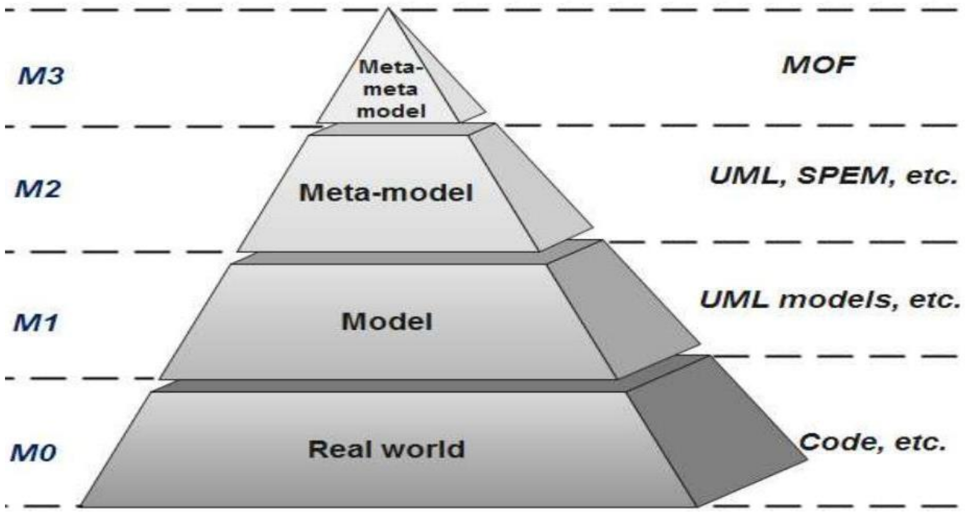


FIG. 2.1 - The four MDA abstraction levels.

- **Level M0:** Level M0 (or instance) corresponds to the real world. This level contains the user's actual information, instances of the M1 level model.
- **Level M1:** The M1 level (or model) is made up of information models. It describes M0 information. UML models, PIMs and PSMs belong to this level. The elements of a model (M1) are instances of the concepts described in a meta-model (M2).
- **Level M2:** The M2 level (or meta-model) defines the modeling language for M1 models. The UML metamodel is part of this level. UML profiles, which extend the UML metamodel, also belong to this level. Concepts defined by a meta-model are instances of MOF concepts.
- **Level M3:** In the MDA approach, level M3 (or meta-meta-model) is made up of a single entity called the MOF. The MOF is used to describe the structure of meta-models, and to extend or modify existing meta-models. The MOF is reflexive, i.e. it describes itself.

4.2.3. Standards

The MDA approach is based on a set of OMG standards (see figure below), including :

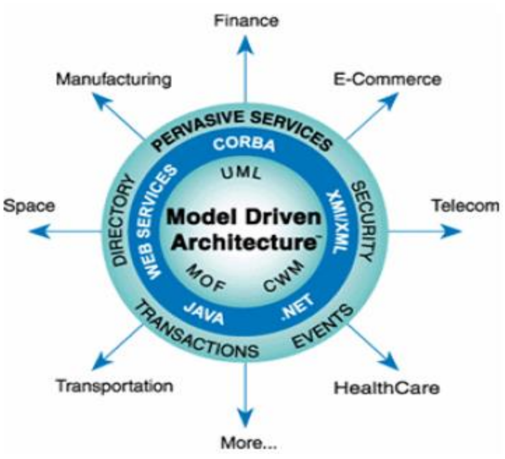


FIG. 2.2 - MDA standards **[OMG04]**.

- **The *Unified Modeling Language* (UML) [OMG10].**

  Described in the first chapter, this approach has largely inspired the MDA approach. The UML language is described by a MOF-compliant meta-model. In this way, a UML model can be serialized to XMI. But there are many other meta-models at the same level as UML. These include the CWM (Common Warehouse Metamodel), SPEM (Software Process Engineering Metamodel), SysML (Systems Modeling Language) and others.

- **The *Meta Object Facility* (MOF) [OMG06]**

  Each model is expressed by a formalism called the meta-model, which defines the concepts and relationships between them that are needed to express the models. To implement the MDA vision, particularly in terms of model relationships, it is essential to work not only at the model level, but also at the formalism level. This is why MDA advocates modeling the formalisms themselves, in order to provide a formalism that allows the expression of models of formalisms, and this is what we call meta-formalism (meta-meta-model). In MDA, the latter is the MOF (Meta Object Facility), which enables modeling formalisms to be expressed, which in turn enable models to be expressed. The MOF defines an abstract, extensible language for describing, defining and manipulating meta-models. It is also self-defining. Today, this is the cornerstone of the MDA approach.

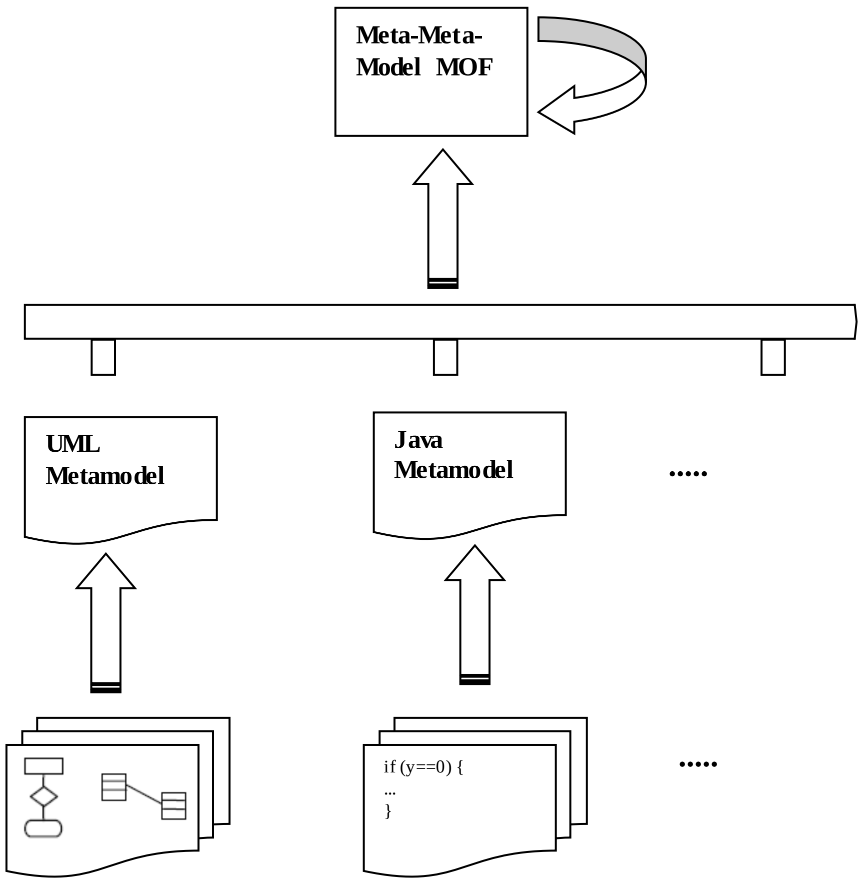


FIG. 2.3 - Model, meta-model (formalism) and meta-meta-model (meta-formalism**)** **[Bla05]**.

- **The *Common Warehouse Metamodel* (CWM) [OMG03].**

  Defines a *framework* for describing metadata concerning data sources, data transformations and data warehouse management processes. Its aim is to facilitate the exchange of metadata in distributed and heterogeneous environments.

- **The *XML Metadata Interchange* (XMI) standard [OMG05].**

  <u>Overview</u>

  Representing a model so that it could be manipulated was not available, given that models are abstract (and sometimes concrete) entities. Finding a way to express them in computer terms was an absolute necessity. The OMG, therefore, standardized XMI to enable these models to be represented by XML documents, thus enabling greater interoperability when exchanging models between tools.

  XMI is therefore an OMG standard for the exchange of UML metadata information using XML. It can also be used for any metadata whose meta-model can be expressed in MOF (Meta-Object Facility). In MDA, the most common use of XMI is in the exchange of UML models, although it can also be used to serialize models from other languages.

How it works

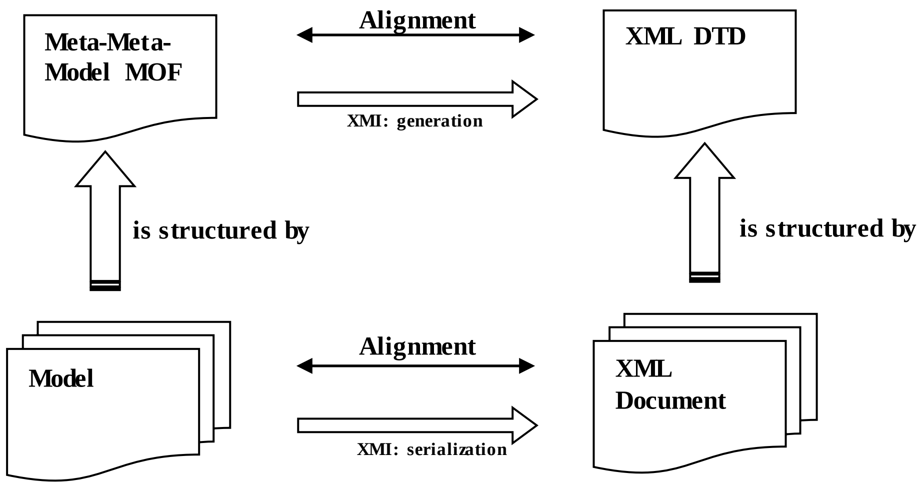


FIG. 2.4 - Alignment between meta-model/model and DTD/XML document **[Bla05]**.

The 4-level hierarchy exists outside MOF and UML, in technology areas other than OMG, such as XML :

- M0: system data
- M1: data modeled in XML
- M2: DTD / XML Schema
- M3: XML language

The OMG, therefore, took advantage of this by using the XML DTD and XML Schema tag structuring mechanisms to determine how to make such a representation. In fact, meta-models define model structures, while XML Schema DTDs define XML document structures. So, aligning meta-model/ XML DTD - Schema, and model/ XML documents will help define tag structures, which is how XMI works.

The representation of models in XML is always based on meta-models. Indeed, XMI defines a set of rules for automatically generating XML tag structuring from a meta-model, which in turn enables models to be represented by XML documents (this is called serialization).

4.2.4. Typology of models in the MDA approach

The OMG has defined a typology of models, as well as a set of transformation relationships that allow you to move from one to another. The four main types of model defined in the MDA approach are as follows **[BB02, Har08]**:

- **CIM (Computation Independent Model):** The CIM captures the requirements in terms of needs and describes the situation in which the system will be used. Its purpose is not only to help understand the problem, but also to establish a common vocabulary for a particular domain.
- **PIM (Platform Independent Model):** The PIM describes the system independently of the target platform on which it will run. It therefore presents a detailed functional view of the system, without any technical details.
- **PDM (Platform Description Model):** The PDM is the model that describes an execution platform. It provides a set of technical concepts representing the different parts of the platform and/or the services it provides.
- **PSM (Platform Specific Model):** The PSM is the result of combining the PIM and the PDM. It represents a detailed technical view of the system. It can exist at different levels of detail. In its most detailed form, it serves as the basis for generating the implementation.

4.2.5. Model transformation

A model transformation is a process whose inputs and outputs are models. It is based on a set of rules enabling the user to pass from one meta-model to another, by defining for each element of the source their equivalents among the elements of the target. These rules are executed by a transformation engine, which reads the source model, which must conform to the source meta-model, and applies the rules defined in the model transformation to arrive at the target model, which will itself conform to the target meta-model. The principle of model transformations is illustrated in the figure below:

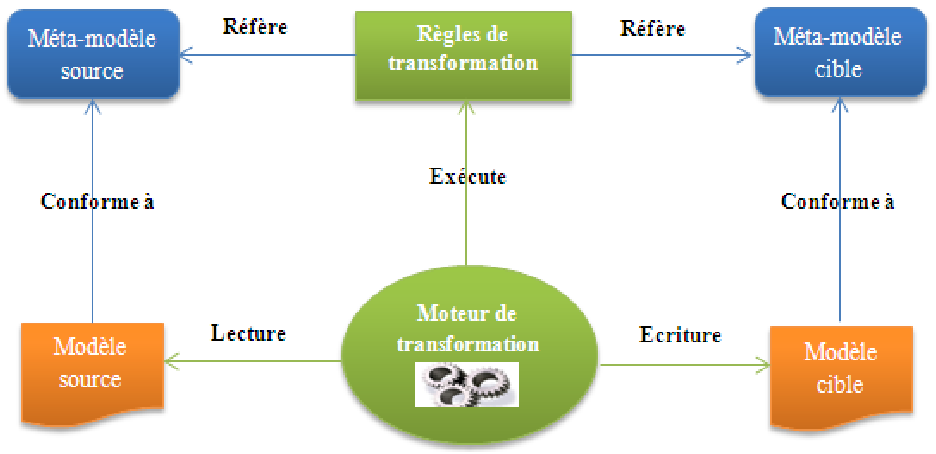


FIG. 2.5 - Principle of model transformations.

MDA specifies the various transformations that can take place at model level, as well as a proposed formalization of a transformation language. The possible transformations between these different types of models are shown in the figure below **[Har08]**:

- **PIM >> PIM and PSM >> PSM transformations:** PIM to PIM or PSM to PSM transformations aim to enrich, filter or specialize the model. They are model-to-model transformations **[CH06]**. They can be automated (or partially automated) in certain cases, such as translation into another language, but refinement transformations generally cannot.
- **PIM >> PSM transformation:** The transformation from PIM to PSM enables the PIM to be specialized according to the chosen target platform. It is only performed once the PIM has been sufficiently refined. This model-to-model transformation is based on information provided by the PDM.
- **PSM >> code transformation:** The transformation from PSM to implementation (code) is a model-to-text transformation **[CH06]**. The code is sometimes assimilated by some to an executable PSM. In practice, it is generally not possible to obtain the complete code from the model, and it is therefore necessary to complete it manually.
- **Reverse transformations PIM << PSM and PSM << code:** These transformations are reverse-engineering operations. This type of transformation poses many difficulties, but is essential for the reuse of existing code within the MDA approach.

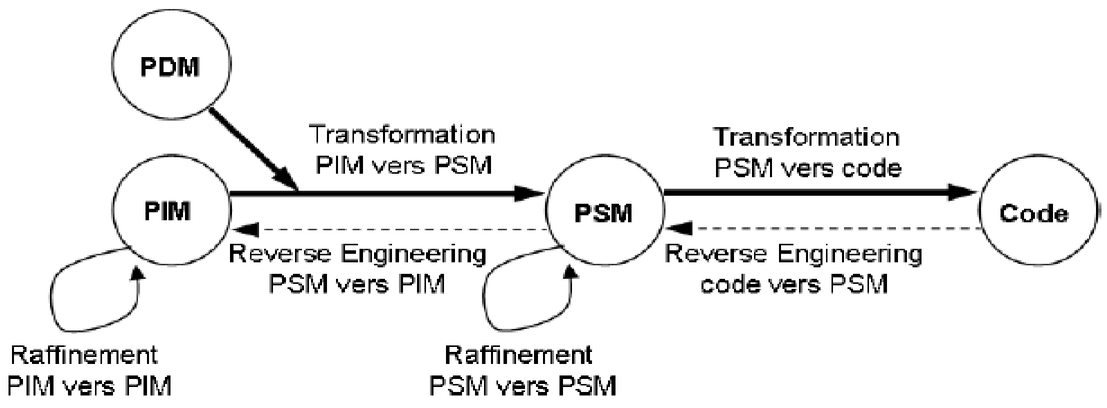

PDM
PIM
PSM
Code
Transformation PIM vers PSM
Transformation PSM vers code
Reverse Engineering PSM vers PIM
Reverse Engineering code vers PSM
Raffinement PIM vers PIM
Raffinement PSM vers PSM

FIG. 2.6 - Models and transformations in the MDA approach **[Har08]**.

4.2.6. Properties of model transformations

The main properties that characterize model transformations are: reversibility, traceability, reusability, scheduling and modularity **[FG05]**:

- **Reversibility:** a transformation is said to be reversible if it takes place in both directions. Example: Model to Text and Text to Model.
- **Traceability:** traceability provides information on the fate of model elements during the various transformations they undergo. In a model-driven engineering context, it is only natural that traceability information should be considered as a model. A model is therefore associated with each execution of a traced transformation. The definition of a trace metamodel makes it possible to structure the traces that will be generated by the traceability platform, and thus to handle them more effectively.
- **Reusability:** reusability enables transformation rules to be reused in other model transformations.
- **Scheduling:** Scheduling consists in representing the levels of nesting of transformation rules. Transformation rules can trigger other rules.
- **Modularity:** a modular transformation enables transformation rules to be better modeled by breaking down the problem. A model transformation language supporting modularity facilitates the reuse of transformation rules.

4.2.7. Classification of transformation approaches

There are many classifications of model transformation approaches, based on several points of view. In this section, we present a classification of model transformation approaches, inspired by the work of *CZarnneki* **[CH03]**, who decomposes model transformation into two categories: *model-to-code* transformations and *model-to-model* transformations.

Model-to-code transformation

This transformation proposes two transformational approaches. The first is the *Visitor-based approach*, while the second is the *Template-based approach*.

- **Visitor-based approaches:** transform the input model into code written in an output programming language. Visitors are added to the input model, reducing the difference in semantics between the model and the target language. The target code is obtained by traversing the model enriched by the visitors to create an output text stream. One example is the Jamda framework[2] .
- **Pattern-based approaches:** based on the use of meta-code fragments from the target code to access information from the source model. These approaches are currently widely used in MDA tools such as AndroMDA[3] (a code generator that uses Velocity's open technology[4] to write patterns).

Model-to-model transformation

Model-to-model transformations transform source models into target models, which may be instances of the same or different meta-models. Broadly speaking, there are several types of approach**:**

- **Direct manipulation approaches**: These are based on an internal representation of the source and target models, and a set of APIs for manipulating them. They are generally implemented as object-oriented structuring frameworks that provide a minimal set of concepts - in the form of abstract classes, for example. The implementation of the rules and their scheduling remain the responsibility of the developer.
- **Relational approaches**: These use declarative logic based on mathematical relationships. The basic idea is to specify the relationships between the elements of the source and target models by means of constraints. The use of logic programming is

[2] http://sourceforge.net/projects/jamda/
[3] http://www.andromda.org/
[4] http://jakarta.apache.org/velocity/

particularly well-suited to this type of approach. The transformations produced are generally bidirectional.

- **Approaches based on graph transformations:** These approaches exploit the work done on graph transformations **[AEH+99]**. Rules are no longer defined for single elements, but for model fragments: this is called *pattern matching*. Patterns in the source model that match certain criteria are replaced by other patterns in the target model. Patterns, or pattern fragments, are expressed either in the respective concrete syntaxes of the models or in their abstract syntaxes.
- **Structure-based approaches:** These approaches are divided into two phases. The first is to create the hierarchical structure of the target model. The second involves adjusting the attributes and references in the target model.
- **Hybrid approaches:** Hybrid approaches are the result of a combination of different techniques. Examples include approaches using both declarative and imperative logic rules, such as ATL[5] and XDE . [6]
- **Other approaches:** At least two other approaches should be mentioned for perfection: the transformation *framework* defined in the OMG's CWM specification and tree transformation using XSLT **[W3C99]**. The CWM transformation *framework* provides a mechanism for linking source and target elements, but the derivation of target elements must be implemented in some concrete language, which is not prescribed by CWM. In fact, CWM provides a general model, but no real mechanism for implementing model transformations. Since models can be serialized to XML using XMI, implementing model transformations using XSLT, which is a standard technology for transforming XML, seems very attractive.

[5] http://www.eclipse.org/atl/
[6] https://www.ibm.com/developerworks/rational/library/07/0731_kishore/index.html

4.3. Review

4.3.1. Exercise01

1. Why model? What will happen if I don't model?
    - ➢ *Simplify reality and better understand the system we're developing.*
    - ➢ *Potential for serious system errors.*
2. Why meta-model? What will happen if I don't meta-model?
    - ➢ *To define meta-models for certain models*
    - ➢ *No languages to define models*
3. What are the modeling levels defined by the OMG? Give an example for each one?
    - ➢ *The OMG defines 4 modeling levels*
        - ▪ *M0: real system, modeled system,*
        - ▪ *M1: model of the real system defined in a certain language, UML*
        - ▪ *M2: meta-model defining this language, UML*
        - ▪ *M3: meta-meta-model defining the meta-model, MOF*

2. MDA, MDE which is inspired by the other?
    - ➢ *MDA of MDE*
3. What is the basic idea behind the MDA approach?
    - ➢ *The key principle of MDA is the use of models in the various phases of an application's development cycle.*
4. Give an example of a model for each layer of the MDA architecture?
    - ➢ *M0>> real world, M1>> PIM, M2>> UML meta-model, M3>> MOF.*
5. What is a model transformation
    - ➢ *A model transformation is the process of moving from a source model to a target model, which requires a set of rules for moving from the source model's meta-model to the target model's meta-model.*
6. Why choose the class diagram to represent meta(meta)-models? Can I use other models?
    - ➢ *Class diagram: this is the UML diagram used to define elements/concepts (via classes) and their relationships (via associations).*
    - ➢ *yes, the E/R (entity/relationship) model.*

4.3.2. Exercise02

Statement: (Automated teller machines)

Design software to manage banking transactions, including operations carried out by human tellers and automated teller machines (ATMs).

Each bank provides its own computer to manage its own accounts and transactions. The various cash registers are owned by the different banks and communicate with the bank's computer. Cashiers enter account numbers and transaction data.

Cash registers communicate with a central computer that routes transactions to the appropriate bank. The automatic cash register accepts credit cards, interacts with the user, communicates with the central computer to complete the transaction, delivers the money and prints a receipt. Banks supply their own software for their own computers. So you'll only need to define the software for ATM and network management.

1. Describing the system in level M0
2. Describe the system in level M1 by :
    - a use case diagram (Distributor or Bank computer).
    - a class diagram.
    - a state-transition diagram.
3. Define the meta-model (level M2) for the last three diagrams?
4. Define the meta-meta-model (level M3) of the last meta-model?

Solution:

1. Describing the system in level M0
    - *Already done in the statement.*
2. Describe the system in level M1 by :
    - a use case diagram (Distributor or Bank computer).

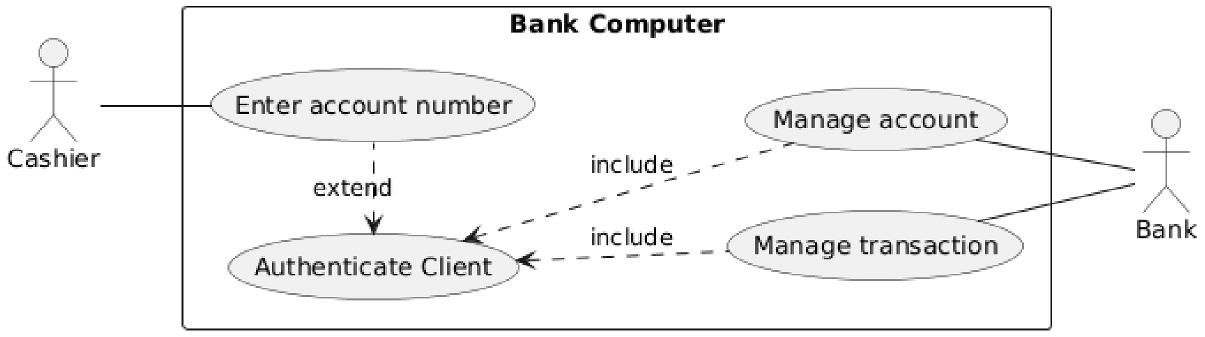


FIG. 2.7: Bank Computer use case diagram.

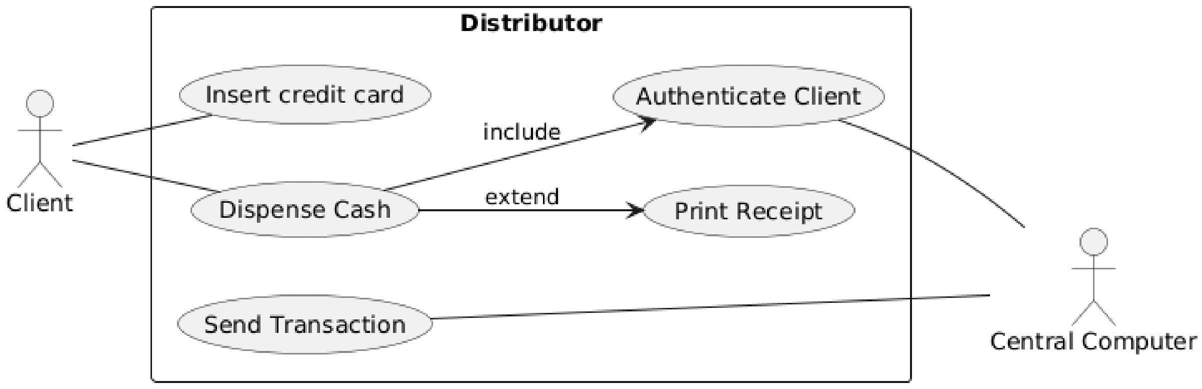


FIG. 2.8 - Distributor use case diagram.

- a class diagram.

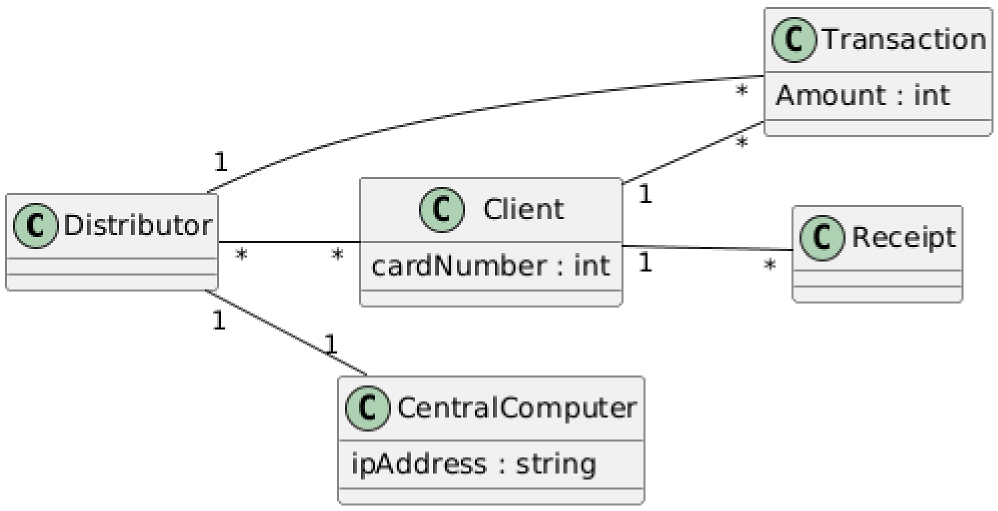


FIG. 2.9: Application class diagram.

- a state-transition diagram.

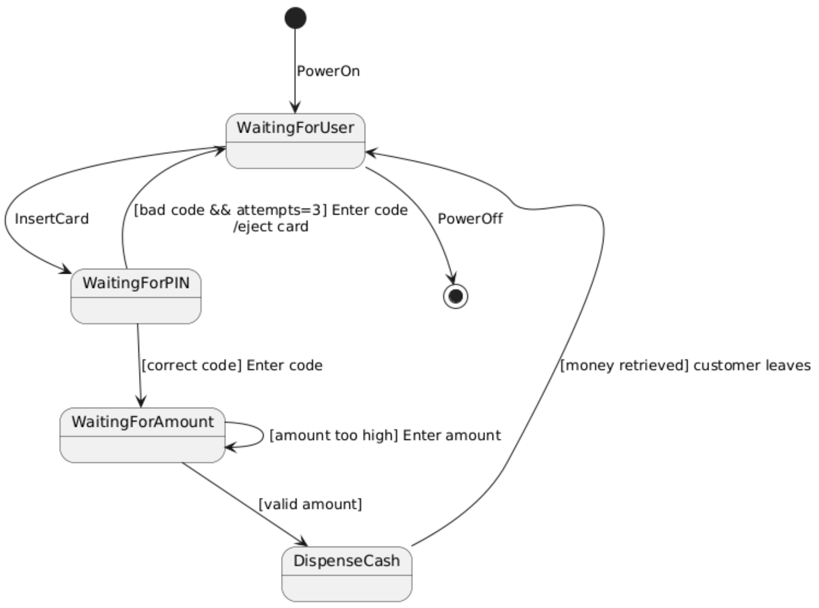

PowerOn
WaitingForUser
InsertCard
[bad code && attempts=3] Enter code /eject card
PowerOff
WaitingForPIN
[correct code] Enter code
[money retrieved] customer leaves
WaitingForAmount
[amount too high] Enter amount
[valid amount]
DispenseCash

FIG. 2.10 - State-transition diagram.

3. Define the meta-model (level M2) for the last three diagrams?

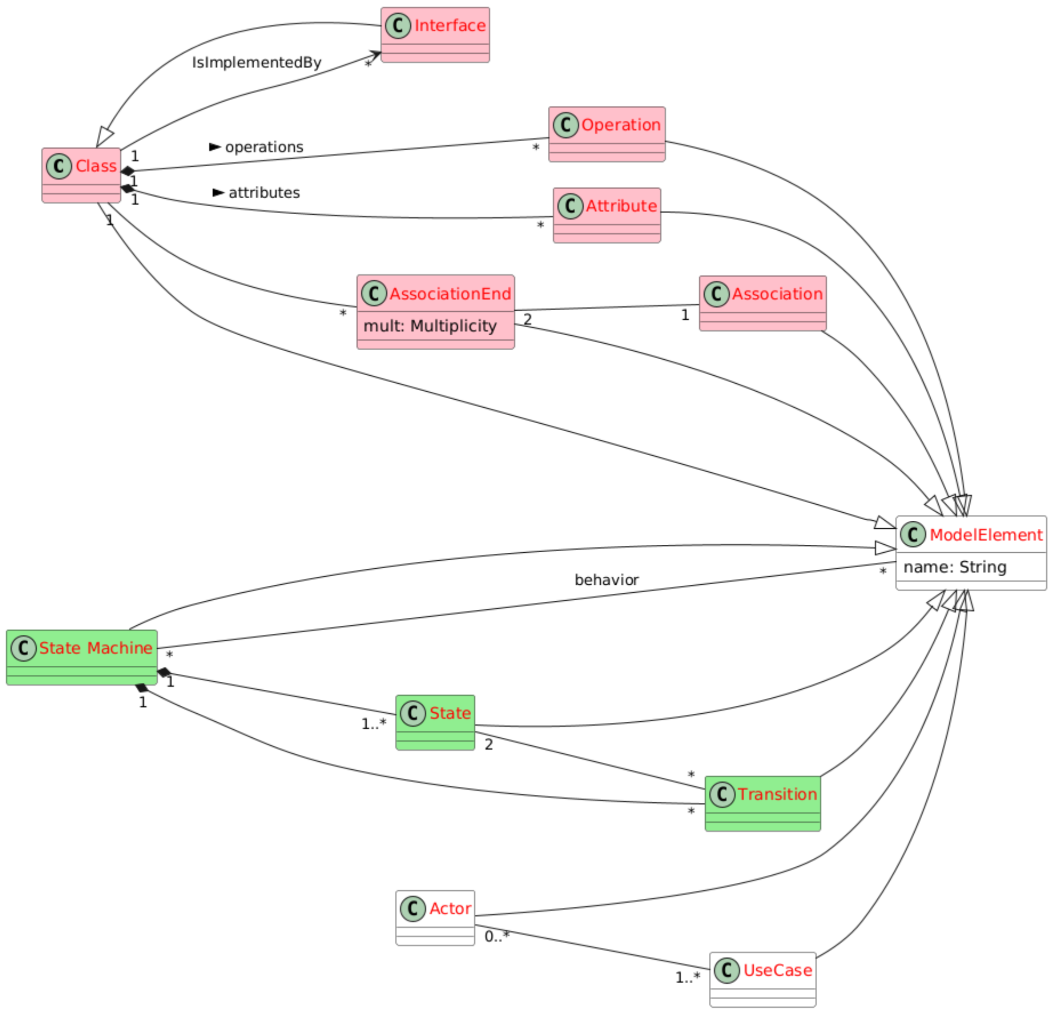


FIG. 2.11 - Simplified meta-model (level M2).

4. Define the meta-meta-model (level M3) of the last meta-model?

N.B. the meta-meta-model here is a part of the previous meta-model. Also, the meta-meta-model is reflexive, which means it defines itself, so no need to another level of modeling.

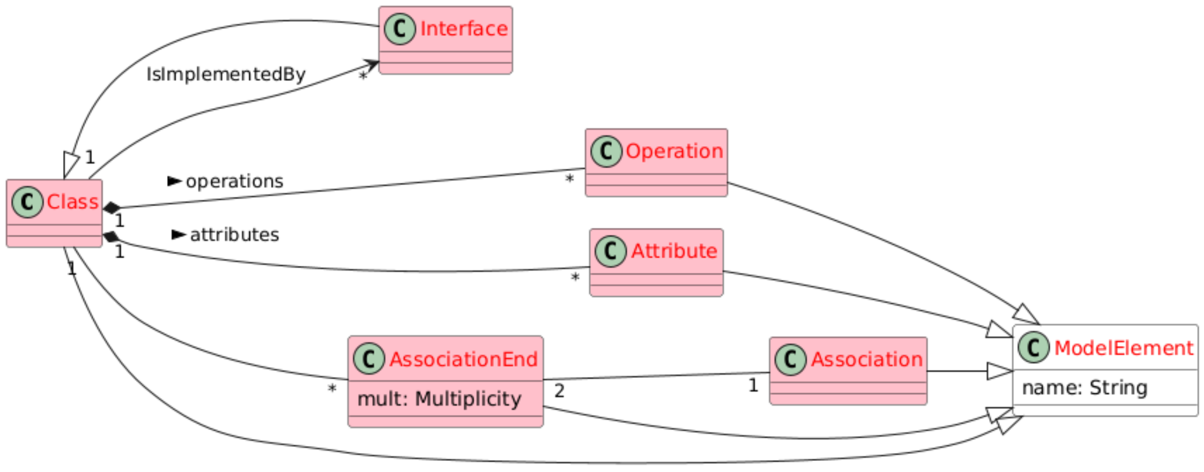
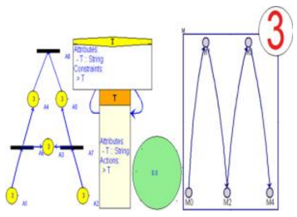


FIG. 2.12 - Simplified meta-model (level M3).

Chapter 5

Graph transformation

In this chapter :



5.1. Introduction

Graphs provide a simple and powerful approach to a variety of problems that are typical of computer science in general, and model-driven engineering in particular. In fact, for most of the latter's activities, a series of visual notations are proposed, which produce models that can be easily seen as graphs, and thus graph transformations are involved.

In this chapter we begin with a presentation of the basic concepts of graph transformation as a specific type of model transformation, and then go on to discuss our approach, which is based on $ATоM^3$ and includes the proposed class diagram meta-model and the developed graph grammar.

5.2. Graph transformation

Graph transformations have been widely used to express model transformations. In particular, visual model transformations can be naturally formulated by graph transformations, since graphs are well suited to describing fundamental model structures.

5.2.1. Principle of graph transformation

A graph transformation **[KA04, AEH+99, Roz99]** consists in applying a rule to a graph, and the part of the graph that corresponds to this rule will be replaced by another graph. This process is repeated until no rule can be applied.

The set of graph transformation rules constitutes the so-called graph grammar model. A graph grammar is a generalization, for graphs, of Chomsky's grammars. Each rule in a graph grammar consists of a left-hand side graph **(LHS)** and a right-hand side graph **(RHS)**.

So, graph transformation is the process of choosing a rule from the graph grammar, applying that rule to a graph and reiterating the process until no rule can be applied.

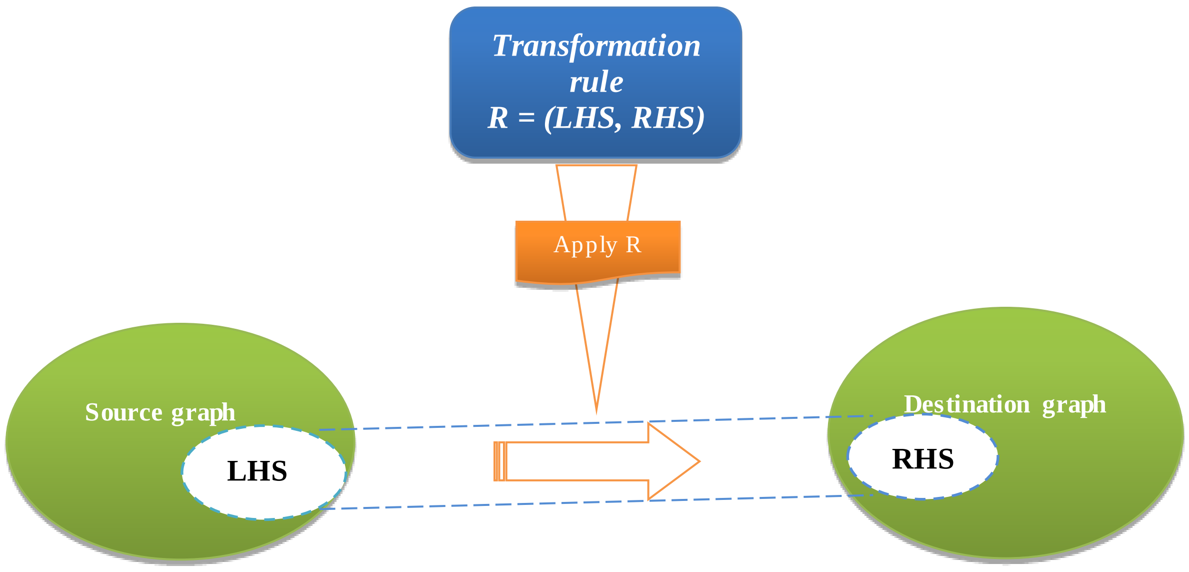


FIG. 5.1 - Principle of graph transformation.

5.2.2. Graph transformation tools

A number of graph transformation tools currently exist, including AGG **[AGG]**, FUJABA **[FUJ]**, ATOM^3 **[ATo]**, VIATRA **[VIA]**, GReAT **[GRe]** and others. Here, we have chosen ATOM^3 after studying the various tools and because of the advantages it offers. These advantages include

- ✓ simplicity,
- ✓ availability,
- ✓ it's multi-paradigm,
- ✓ and supports the two types of transformations used: model-to-code and model-to-model.

5.2.3. AToM tool presentation^3

AToM^3 **[ATo]** "**A** Tool for Multi-formalism and **Meta-Modeling**" is a visual tool for multi-formalism modeling and meta-modeling, written in Python[1] and running on various platforms (Windows, Linux, etc.). The two main tasks of AToM^3 are meta-modeling and model transformation. However, by extension, it can also handle simulation and code generation from elaborated models. For meta-modeling, AToM^3 supports visual modeling

[1] http://www.python.org/

using the Entity/Relationship formalism or the UML class diagram formalism. This means that in AToM^3, we can choose between these two formalisms to meta-model a new formalism. Once we've developed the meta-model for the model in question, AToM^3 can automatically produce a visual modeling environment, in which we can build and edit the new models. For the second task, model transformation, AToM^3 uses graph grammars to express the transformation (see figures below).

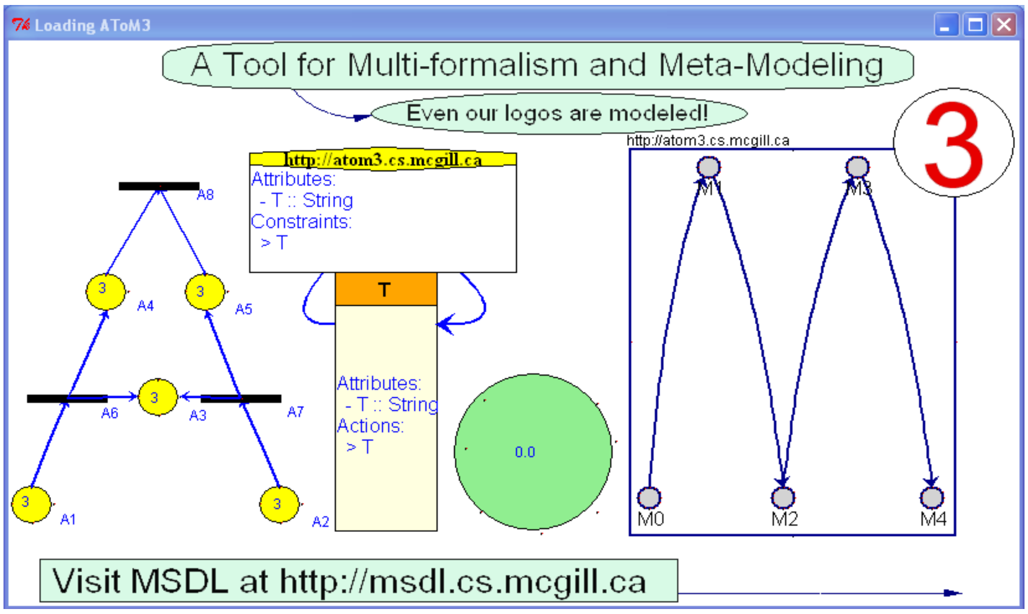
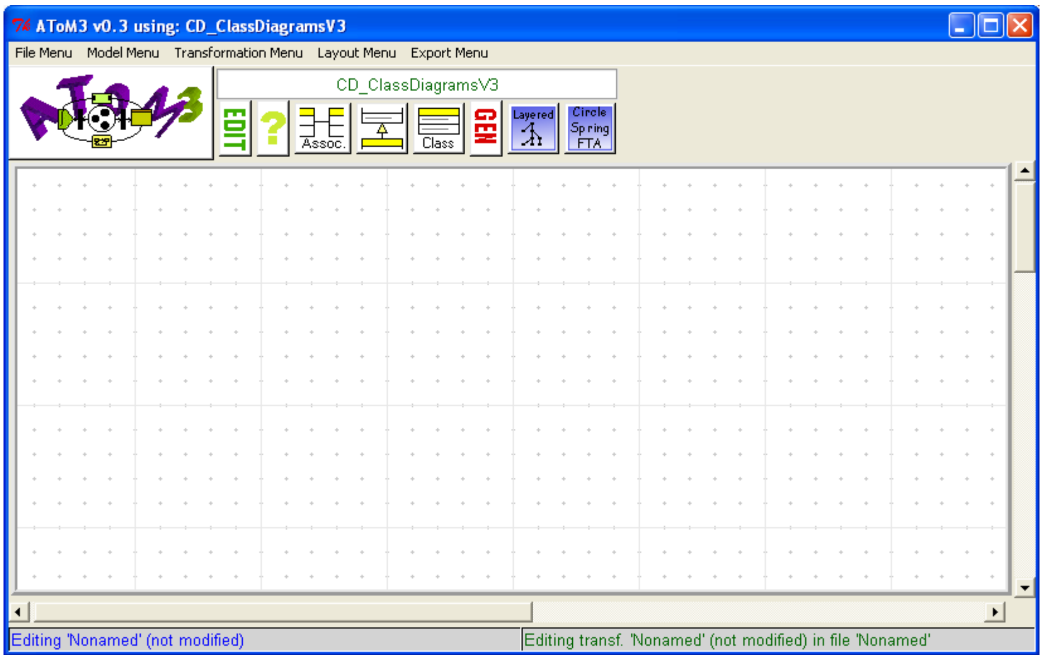

Loading AToM3
A Tool for Multi-formalism and Meta-Modeling
Even our logos are modeled!
http://atom3.cs.mcgill.ca
Attributes:
- T :: String
Constraints:
> T
T
Attributes:
- T :: String
Actions:
> T
0.0
A1
A2
A3
A4
A5
A6
A7
A8
M0
M1
M2
M3
M4
3
Visit MSDL at http://msdl.cs.mcgill.ca

FIG. 5.2 - Overview of the AToM^3 tool.



Editing 'Nonamed' (not modified)

Editing transf. 'Nonamed' (not modified) in file 'Nonamed'

FIG. 5.3 - The AToM^3 tool canvas.

5.2.4. Development of an AToM^3 transformation

To develop a model transformation in AToM^3 between two modeling formalisms, the following three essential steps must be followed:

- ✓ Define the source formalism meta-model.
- ✓ Define the meta-model of the target formalism.
- ✓ Propose the graph grammar.

5.2.5. Meta-modeling in AToM^3

We're going to look at the creation and editing of meta-models and templates in AToM^3. But first (and again), what is a meta-model? A meta-model "is a precise definition of the constructs and rules required to create semantic models". In other words, a meta-model describes what is allowed in an environment where we can create models. Consequently, in order to create valid models, we need to respect the reality of their environment: its rules and constructs. As a very simple example, let's suppose we're describing a meta-model with the English language for creating sets of integers (this isn't a complete example, we wanted to give the reader something concrete):

- This meta-model:

  Valid constructions: A set of integers, appearance: {a1, a2, ...}
  rules: any integer i is > 0

- Allow these kinds of models:

  {1,2,3,10,100}; {2}; etc.

- But not those:

  {-1,5}; (4,5); 4,5,6,7; 34; etc.

As you can see, we need a language (or formalism) to describe the meta-models that need to be defined in a meta-meta-model. In AToM^3 , a meta-meta-model for the Entity-Relationship formalism (ditto Class Diagram) provides us with an environment for creating and editing meta-models.

To learn how to build models/meta-models, you still need to master the modeling hierarchy in AToM^3 . The table below summarizes this hierarchy.

Meta-Meta level	-Entity-Relationship meta-model -Metamodel Class diagram
Meta level	-Entity-Relationship Modeling Environment -Modeling environment Class diagram.
Model level	Custom FormatsModeling Environment (created at the meta level)
Models	Different model instances (created at model level)

TAB. 5.1 - The modeling hierarchy in AToM^3.

5.2.6. AToM^3 graph grammar

The most important step in the process of implementing a graph transformation on AToM^3 is the construction of grammar rules. This requires a good understanding of both languages. The graph grammar recognizes **portions of the source graph** as independent entities, and then transforms them into equivalent **portions of the target graph**.

The graph grammar is composed of (see FIG. 5.3):

- ✓ An initial action
- ✓ Several rules
- ✓ A final action

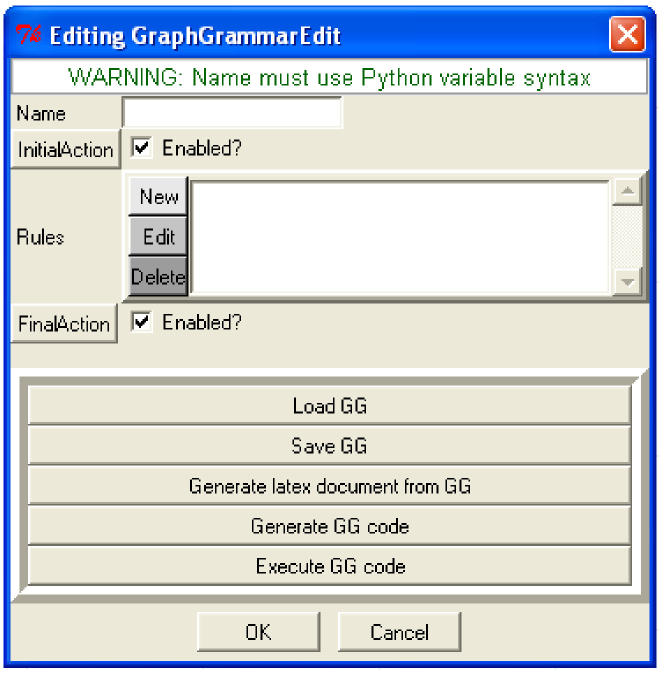
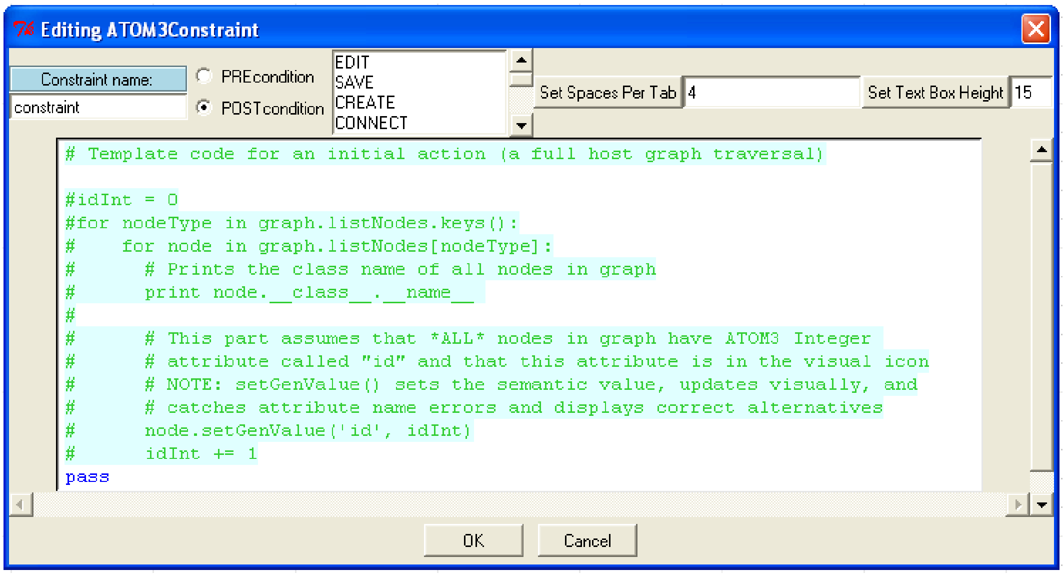


FIG. 5.4 - Components of a graph grammar.

- **Initial action:** The graph grammar has a single initial action.

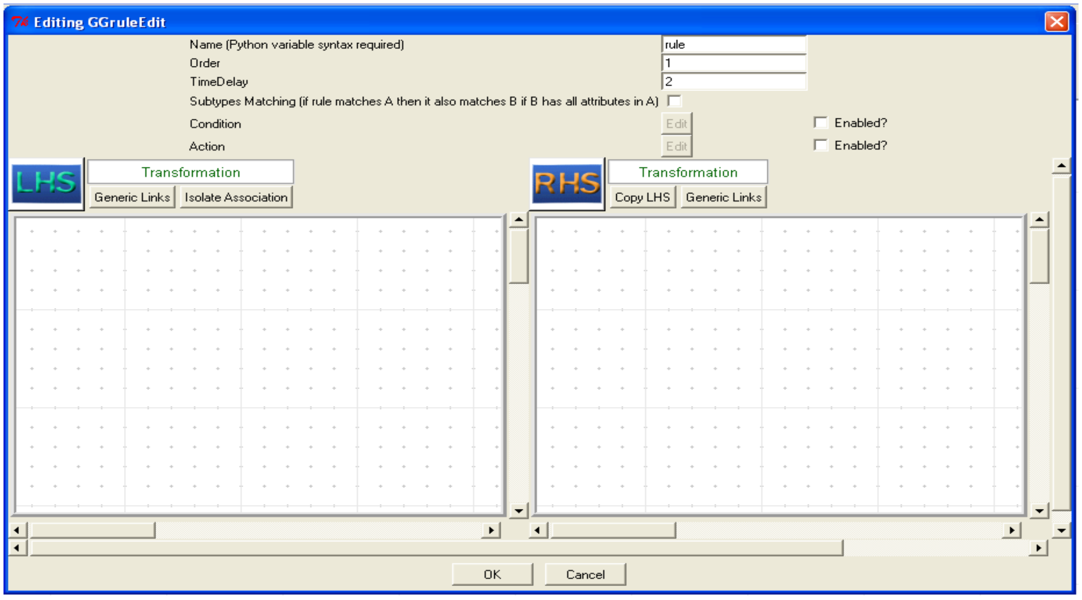


```
# Template code for an initial action (a full host graph traversal)

#idInt = 0
#for nodeType in graph.listNodes.keys():
#    for node in graph.listNodes[nodeType]:
#       # Prints the class name of all nodes in graph
#       print node.__class__.__name__
#
#       # This part assumes that *ALL* nodes in graph have ATOM3 Integer
#       # attribute called "id" and that this attribute is in the visual icon
#       # NOTE: setGenValue() sets the semantic value, updates visually, and
#       # catches attribute name errors and displays correct alternatives
#       node.setGenValue('id', idInt)
#       idInt += 1
pass
```

FIG. 5.5 - The initial action.

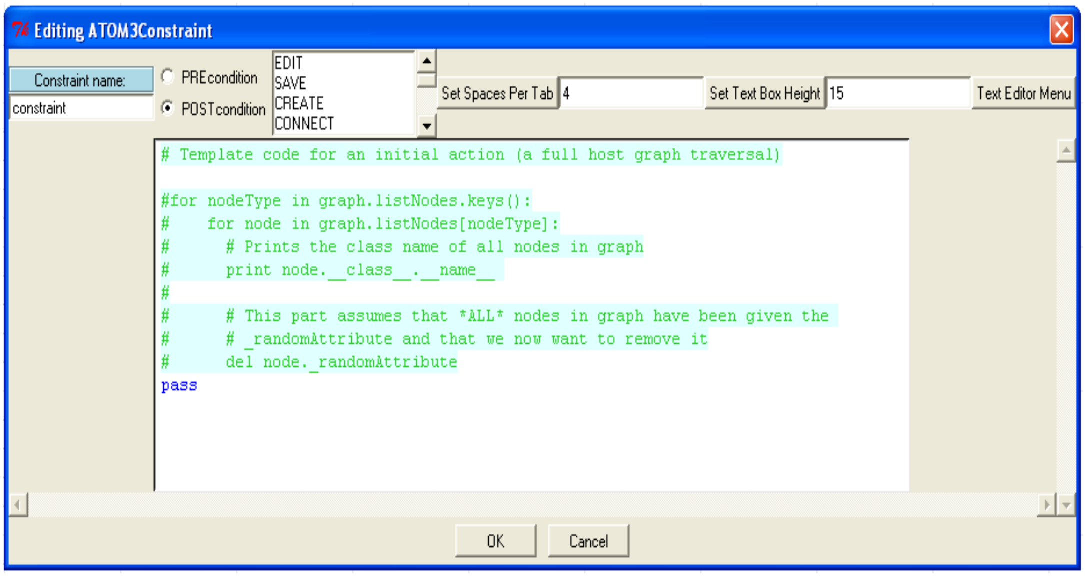
- **Rules:** The graph grammar has several rules. Each rule has the following form:


Editing GGruleEdit

Name (Python variable syntax required) rule
Order 1
TimeDelay 2
Subtypes Matching (if rule matches A then it also matches B if B has all attributes in A)
Condition Edit Enabled?
Action Edit Enabled?

FIG. 5.6 - A ruler.

- **Final action:** The graph grammar has a single final action


Editing ATOM3Constraint

Constraint name: constraint
PREcondition [✓ POSTcondition]
EDIT
SAVE
CREATE
CONNECT
Set Spaces Per Tab 4
Set Text Box Height 15
Text Editor Menu

```
# Template code for an initial action (a full host graph traversal)

#for nodeType in graph.listNodes.keys():
#    for node in graph.listNodes[nodeType]:
#      # Prints the class name of all nodes in graph
#      print node.__class__.__name__
#
#      # This part assumes that *ALL* nodes in graph have been given the
#      # _randomAttribute and that we now want to remove it
#      del node._randomAttribute
pass
```

FIG. 5.7 - The final action.

5.3. Case study : Transforming UML activity diagrams into finite-state automata

In this simple case study, we're going to develop a transformation from UML activity diagrams to finite-state automata. We'll just consider the basic notions of both source and target formalisms, to simplify the transformation and let the reader focus on the nitty-gritty of how to do it, so we won't dwell on these notions and their detailed syntax and semantics.

5.3.1. Transformation scenario

This transformation is carried out by automatically generating a finite-state automaton from a UML activity diagram described in the AToM framework^3 , this solution is implemented in AToM^3 , and takes place in several stages (see FIG. 3.8):

❶ • Graphic description of the activity diagram in ATOM^3.

❷ • This activity diagram conforms to the activity diagram meta-model developed in AToM^3.

❸ • Apply the graph grammar (*ActivityDiagram2FSM*) to the activity diagram.

❹ • A finite-state automaton is automatically generated.

❺ • This finite-state automaton conforms to the finite-state automaton meta-model developed in AToM^3.

❻ • The two meta-models themselves conform to the *CD_classDiagramsV3* meta-model from AToM^3.

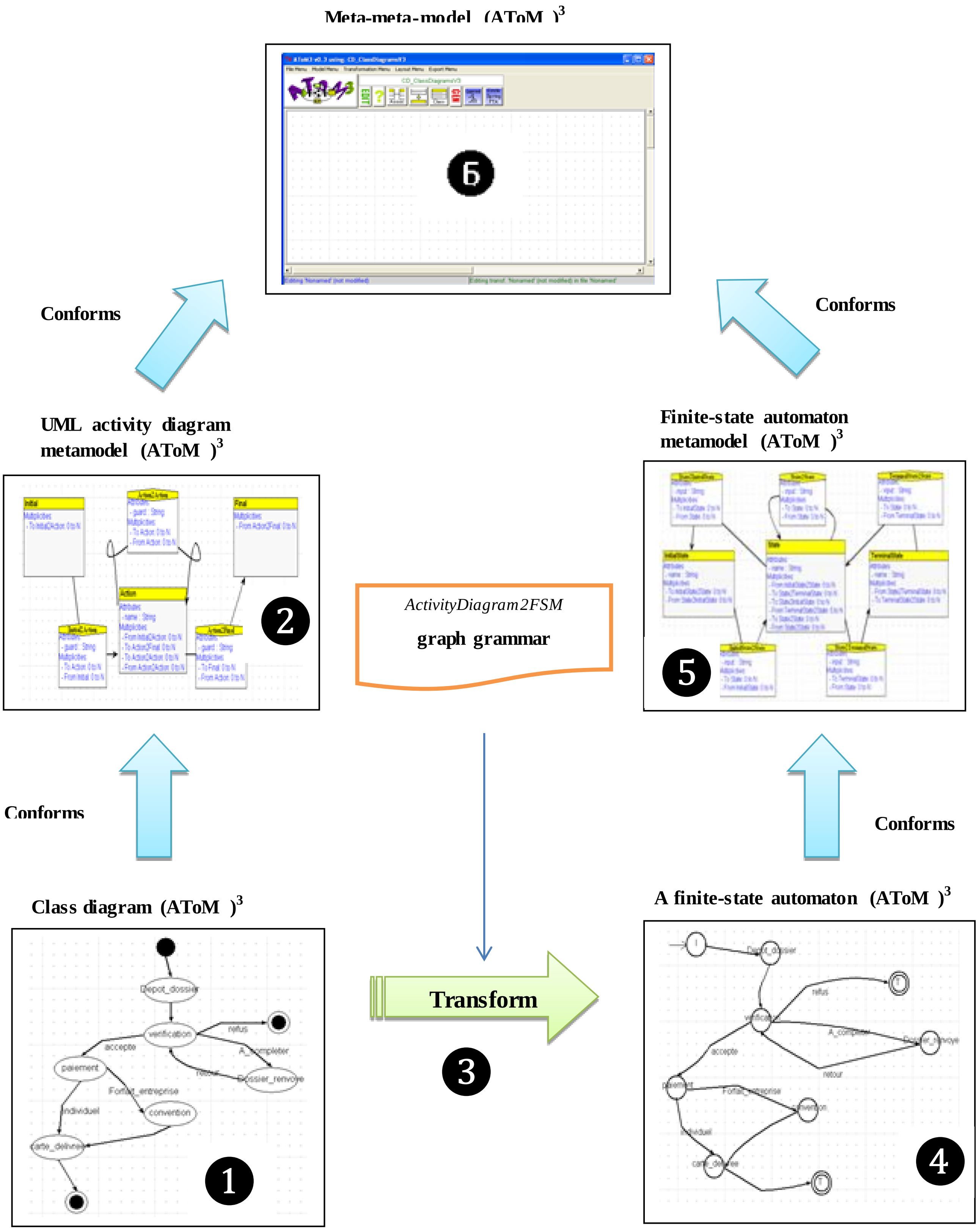


FIG. 5.8 - Transformation scenario.

5.3.2. Activity diagram meta-model

Our proposed meta-model is called "ADMM_META", and is made up of 3 classes and 3 associations developed by the meta-formalism (*CD_classDiagramsV3*). It provides a tool integrating AToM[3], which offers the necessary elements for modeling activity diagrams. The abstract syntax of this meta-model is shown in the figure below.

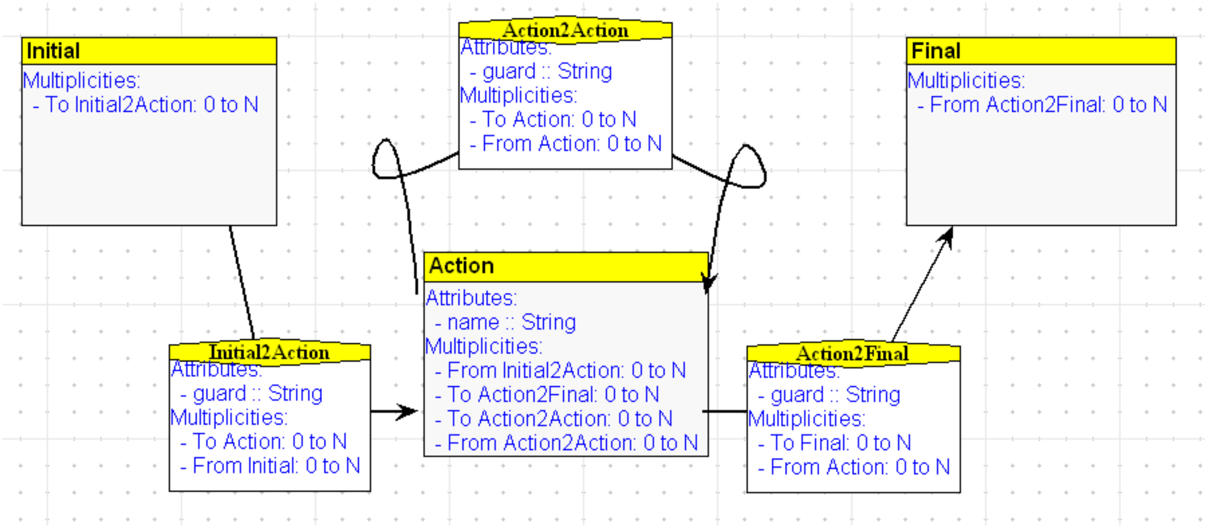
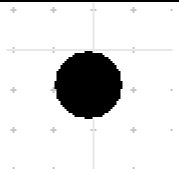
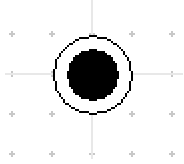
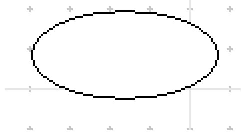
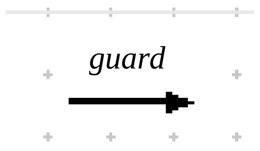


FIG. 5.9 - Activity diagram meta-model.

The table below explains the different entities of the meta-model and their concrete syntax:

Element	Graphical representation
Initial" class: represents the pseudo-initial state of the activity diagram, with no attributes.	
The "Final" class: represents the pseudo-final state of the activity diagram; it has no attributes.	
The "Action" class: represents an action in the activity diagram, and has a ***"name"*** attribute to give the action a name.	
The "Initial2Action" association: models the transition between the initial pseudo-state and	

the action in an activity diagram. ***Action2Final*" *association*:** models the transition between action and pseudo-final state in an activity diagram. ***Action2Action*" *association*:** models the transition between action and action in an activity diagram. They all have a "***guard***" attribute to represent the guard of a transition.	

TAB. 5.2 - Entities of the ADMM_META meta-model.

Once the "ADMM_META" metamodel has been proposed, all that remains is to generate it by clicking on the red "Gen" button. The generated metamodel contains the set of classes modeled as buttons, ready to be used for eventual modeling of an activity diagram. The generated environment is illustrated in the figure below:

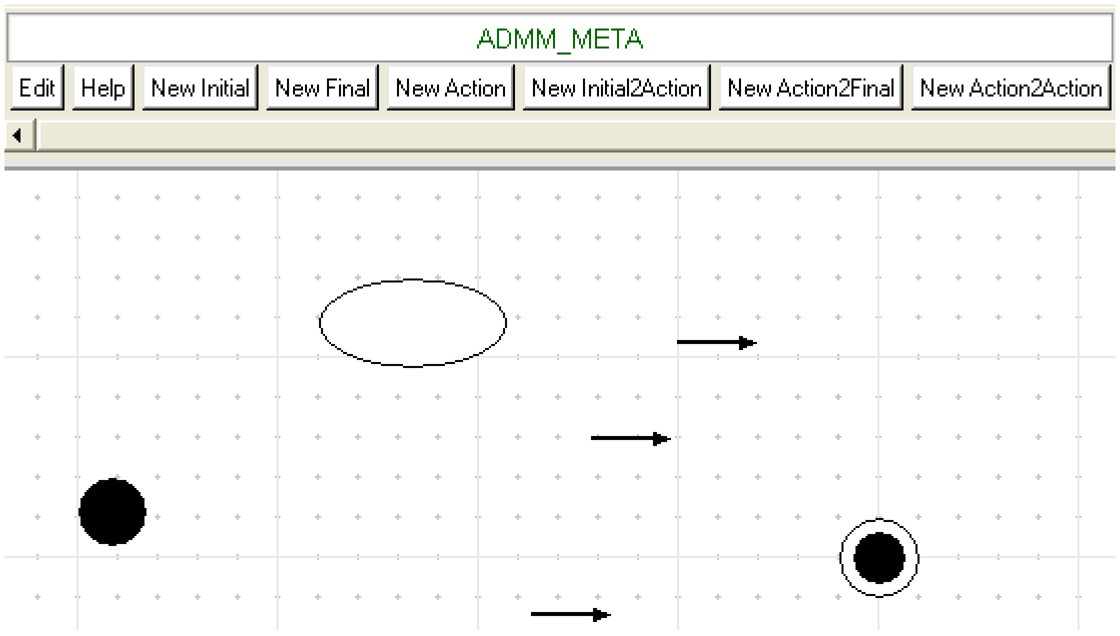


FIG. 5.10 - AToM activity diagram environment .[3]

5.3.3. Finite-state automata meta-model

Our proposed meta-model is called "FSMMM_META", and is made up of 3 classes and 5 associations developed by the meta-formalism (*CD_classDiagramsV3*). It provides a tool integrating AToM[3], offering the necessary tools for modeling finite-state automata. The abstract syntax of this meta-model is shown in the figure below.

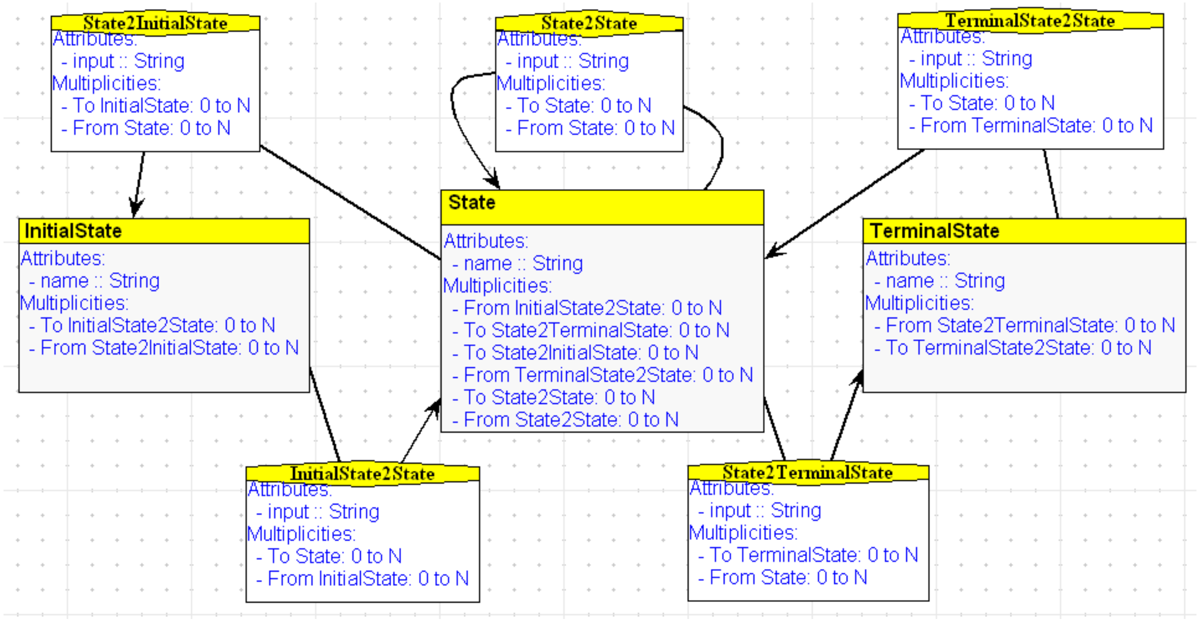
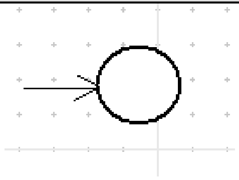
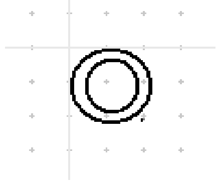
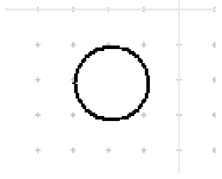
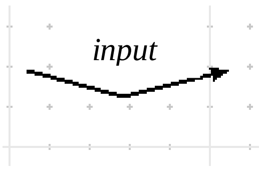


FIG. 5.11 - Finite-state automata meta-model.

The table below explains the various entities of the meta-model and their concrete syntax:

Element	Graphical representation
The "InitialState" class: represents the initial state of the finite-state automaton, and has a "***name***" attribute to attach a name to it.	
The "TerminalState" class: represents the terminal (acceptance) state of the finite-state automaton, and has a "***name***" attribute to attach a name to it.	

The "State" class: represents a state of the finite-state automaton, and has a "***name"*** attribute to attach a name to it.	
InitialState2State" and ***"State2InitialState***" ***associations***: model the transition between an initial state and a state, and vice versa, in a finite-state automaton. ***State2TerminalState***" and ***"TerminalState2State***" ***associations***: model the transition between a state and a terminal state, and vice versa, in a finite-state automaton. ***The "State2State***" ***association***: models the transition between one state and another in a finite-state automaton. They all have an "***input"*** attribute to represent the label of a transition.	

TAB. 5.3 - Entities of the FSMMM_META metamodel.

Once the "FSMMM_META" metamodel has been proposed, all that remains is to generate it by clicking on the red "Gen" button. The generated metamodel contains the set of classes modeled as buttons, ready to be used for a possible finite-state automaton modeling. The generated environment is illustrated in the figure below:

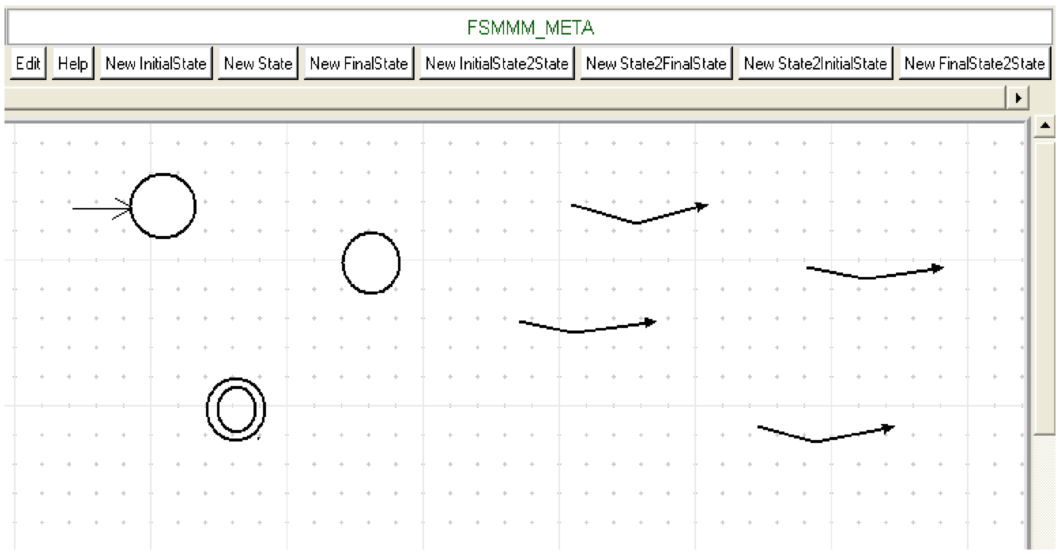


FIG. 5.12 - Finite-state machine environment under AToM .[3]

5.3.4. Processing rules

The transformation from UML activity diagrams to finite-state automata requires us to propose some correspondence rules between the two formalisms. This proposal represents a crucial step in the construction of our transformation; in fact, it implies a good understanding of both formalisms. The table below summarizes this passage.

UML activity diagram	Finite-state automaton
Initial pseudo-state	Initial state
Final pseudo-state	Terminal state
Action	Status
A guard	A label

TAB. 5.4 - Proposed transformation rules.

5.3.5. Graph grammar

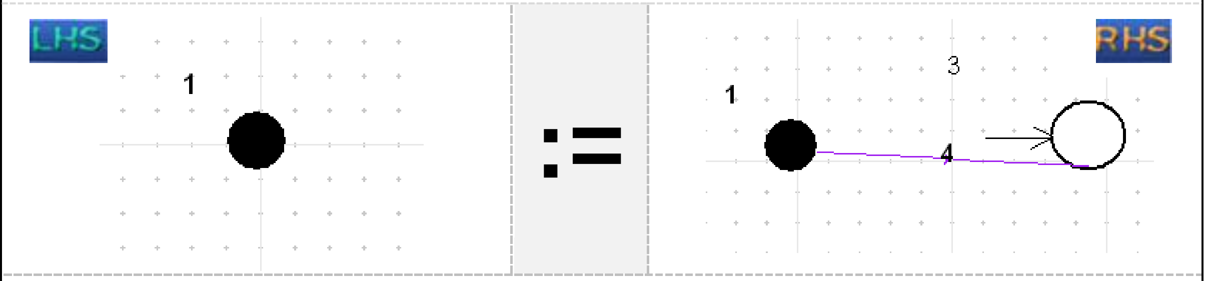
To implement the rules seen previously, we propose a graph grammar composed of several rules. The table below shows the details of this implementation, each rule has a name, a priority, a condition, an LHS, an RHS and an action.

Name : Initial2Initial

Priority : 1

Condition :

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return not hasattr(node, "Initial2Initial_ok")
```



Action :

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
node.Initial2Initial_ok = True
```

Description :

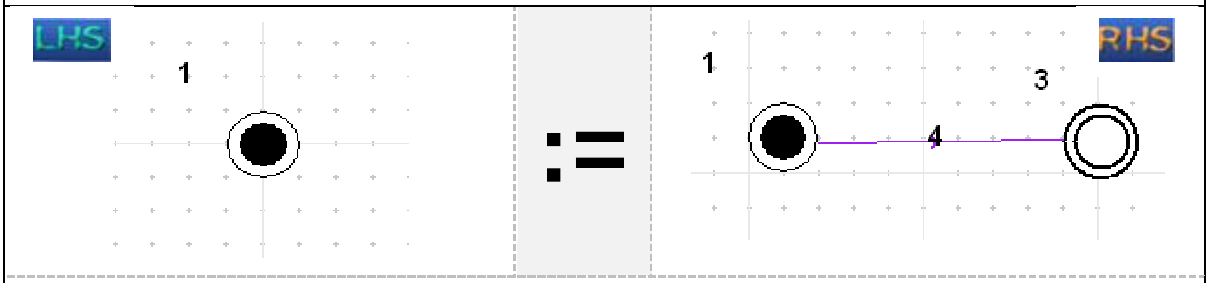
The initial pseudo-state in an activity diagram is transformed to an initial state in the finite state machine

Name : Final2Terminal

Priority : 2

Condition :

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return not hasattr(node, "Final2Terminal_ok")
```



Action :

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
node.Final2Terminal_ok = True
```

Description :

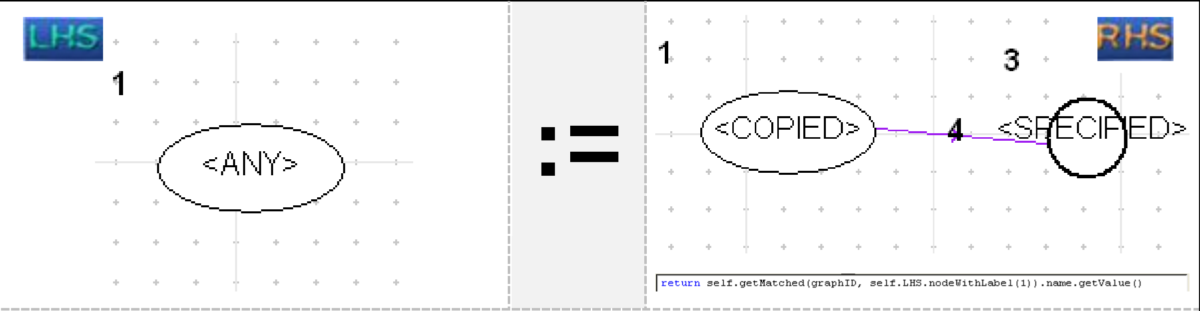
The pseudo-final state in an activity diagram is transformed to a terminal state in the finite state machine

Name : Action2State

Priority : 3

Condition :

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return not hasattr(node, "Action2State_ok")
```



```
return self.getMatched(graphID, self.LHS.nodeWithLabel(1)).name.getValue()
```

Action :

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
node.Action2State_ok = True
```

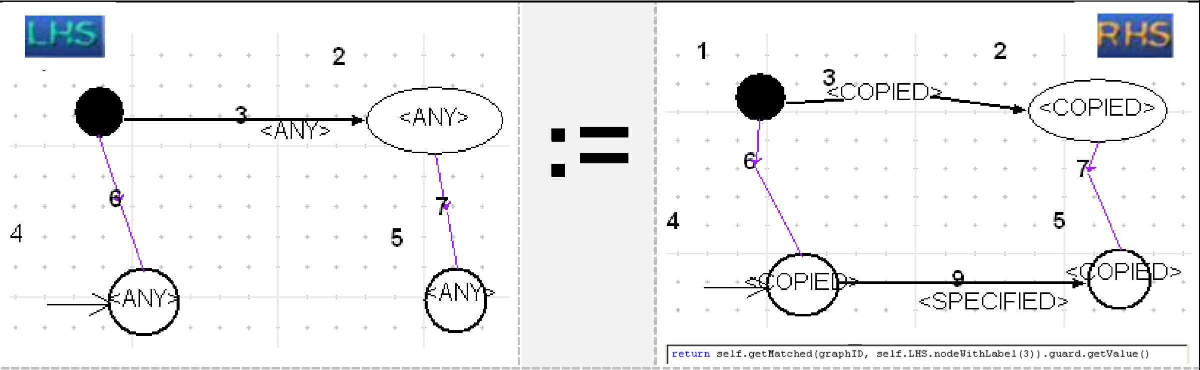
Description :
The action in an activity diagram is transformed to a state in the finite state machine. A code is added on this state (node 3) in the RHS to transmit the name of the action to it (see the RHS).

Name : Initial2Action

Priority : 4

Condition :

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return not hasattr(node, "Initial2Action_ok")
```



```
return self.getMatched(graphID, self.LHS.nodeWithLabel(3)).guard.getValue()
```

Action :

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
node.Initial2Action_ok = True
```

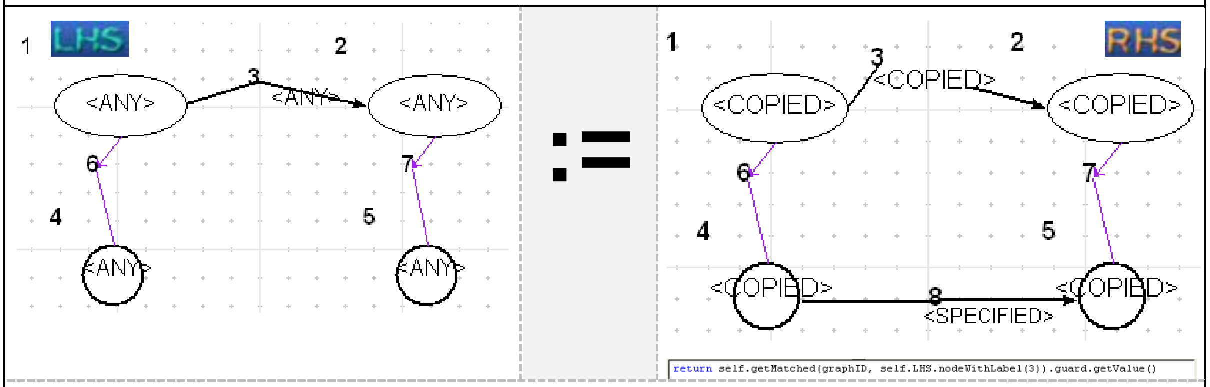
Description :
An initial pseudo-state and an action in an activity diagram are transformed to an initial state and a state in the finite state machine and they are associated with a transition. A code is added on this transition (node 9) in the RHS to retrieve its label from the guard of node 3 (see the RHS).

Name : Transition2Transition

Priority : 5

Condition :

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return not hasattr(node, "Transition2Transition_ok")
```



```
return self.getMatched(graphID, self.LHS.nodeWithLabel(3)).guard.getValue()
```

Action :

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
node.Transition2Transition_ok = True
```

Description :

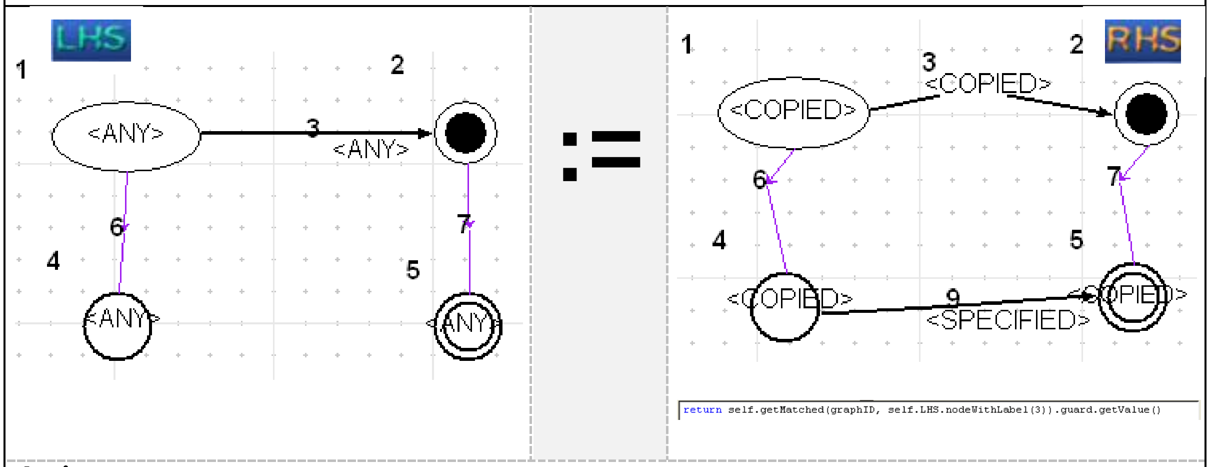
A transition between two actions in an activity diagram is transformed to a transition in the finite state machine. Code is added on this transition (node 8) in the RHS to retrieve its label from the guard of node 3 (see the RHS).

Name : Action2Final

Priority : 6

Condition :

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return not hasattr(node, "Action2Final_ok")
```



```
return self.getMatched(graphID, self.LHS.nodeWithLabel(3)).guard.getValue()
```

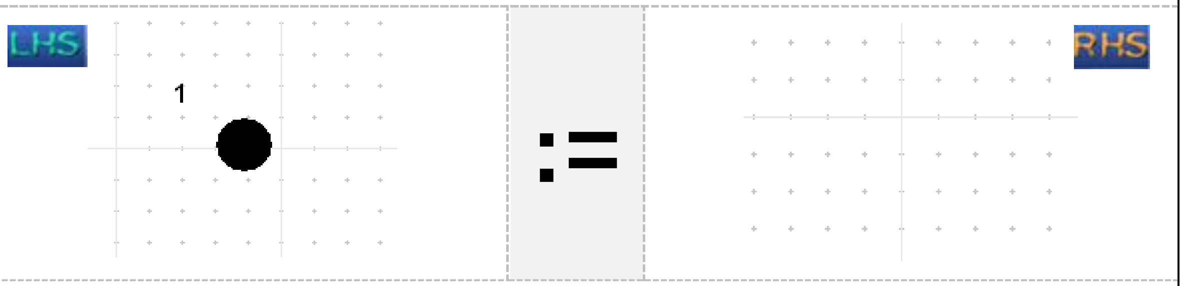
Action :

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
node.Action2Final_ok = True
```

Description :
A transition between an action and a final state in an activity diagram is transformed into a transition in the finite state machine. Code is added on this transition (node 9) in the RHS to retrieve its label from the guard of node 3 (see the RHS).

Name : InitialDelete

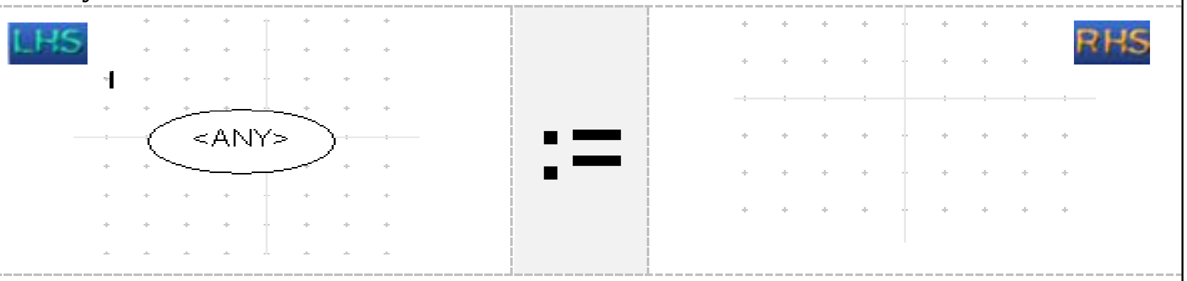
Priority : 7



Description :
Deleting an initial state from the activity diagram. This rule has no condition or action.

Name : ActionDelete

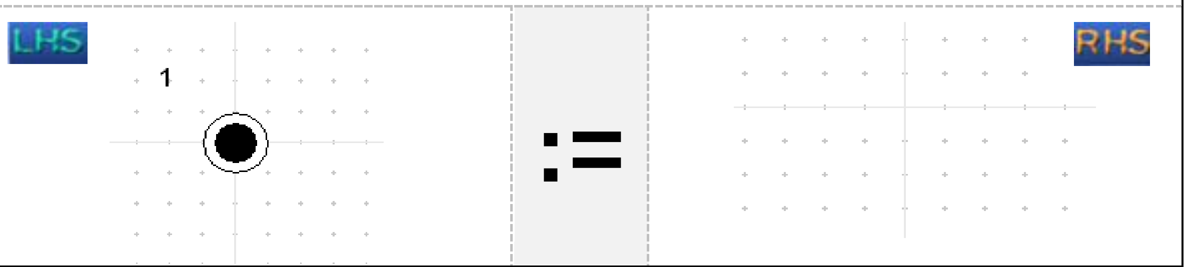
Priority : 8



Description :
Removing actions from the activity diagram. This rule has no condition or action.

Name : FinalDelete

Priority : 9



Description :
Deleting a final state from the activity diagram. This rule has no condition or action.

TAB. 5.5 – The developed graph grammar.

5.3.6. Example

In order to be able to concretize the usefulness of the defined graph grammar and of the transformation in general, we tried to apply it on the example of activity diagram presented in the figure below. It should be noted that this example does not claim to be exhaustive but it groups together the minimum of the elements of an activity diagram.

Taking the example of the activity diagram below:

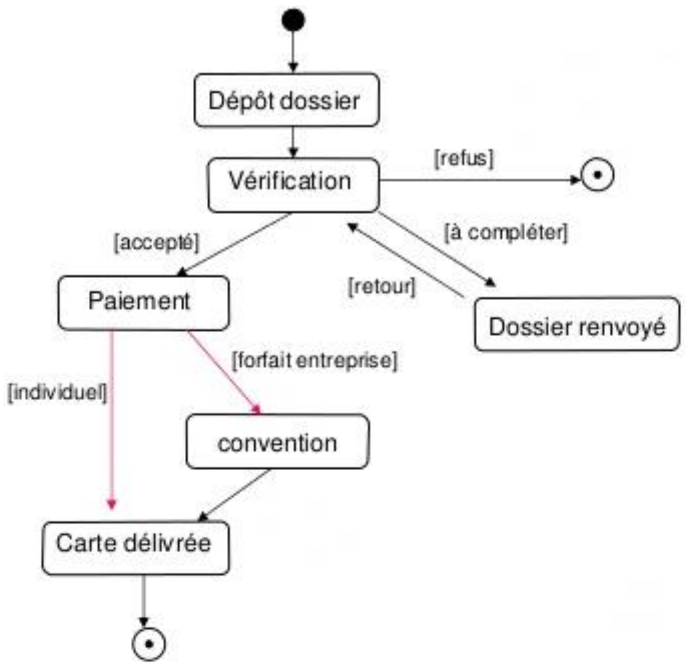


FIG. 5.13 – Activity diagram example.

Using our integrated tool, we implement this diagram as follows:

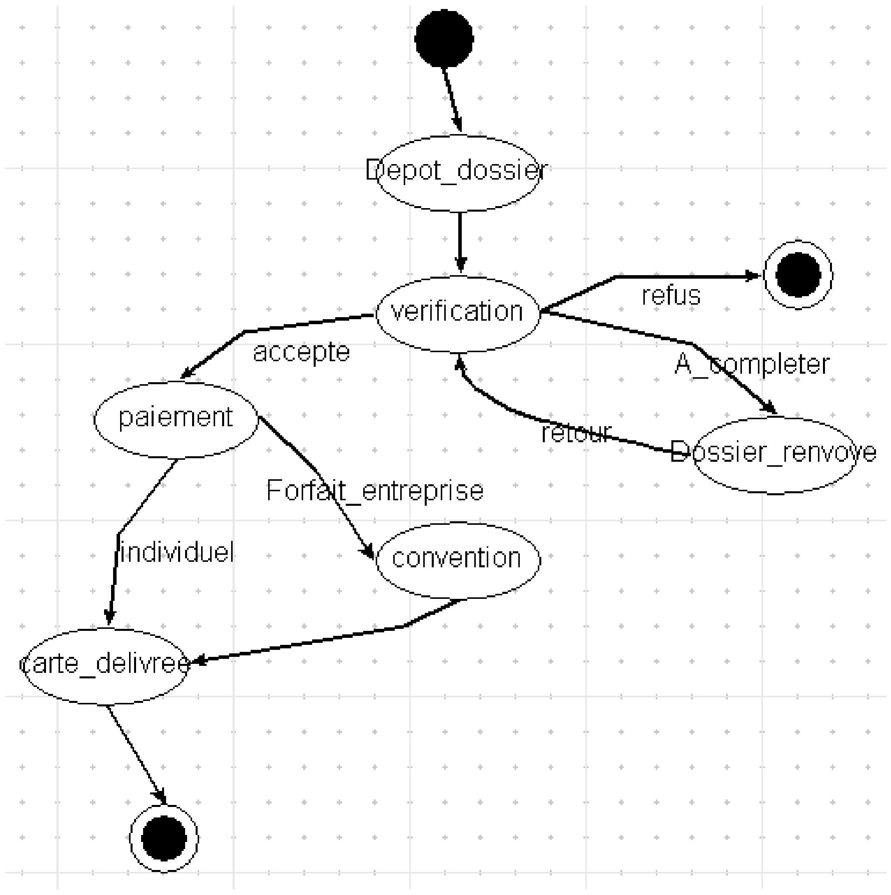


FIG. 5.14 –The activity diagram in the AToM^3 integrated tool.

Starting the execution of our graph grammar, we obtain the intermediate graph below. We can clearly see that this is a mixture of elements from both the source and target formalisms.

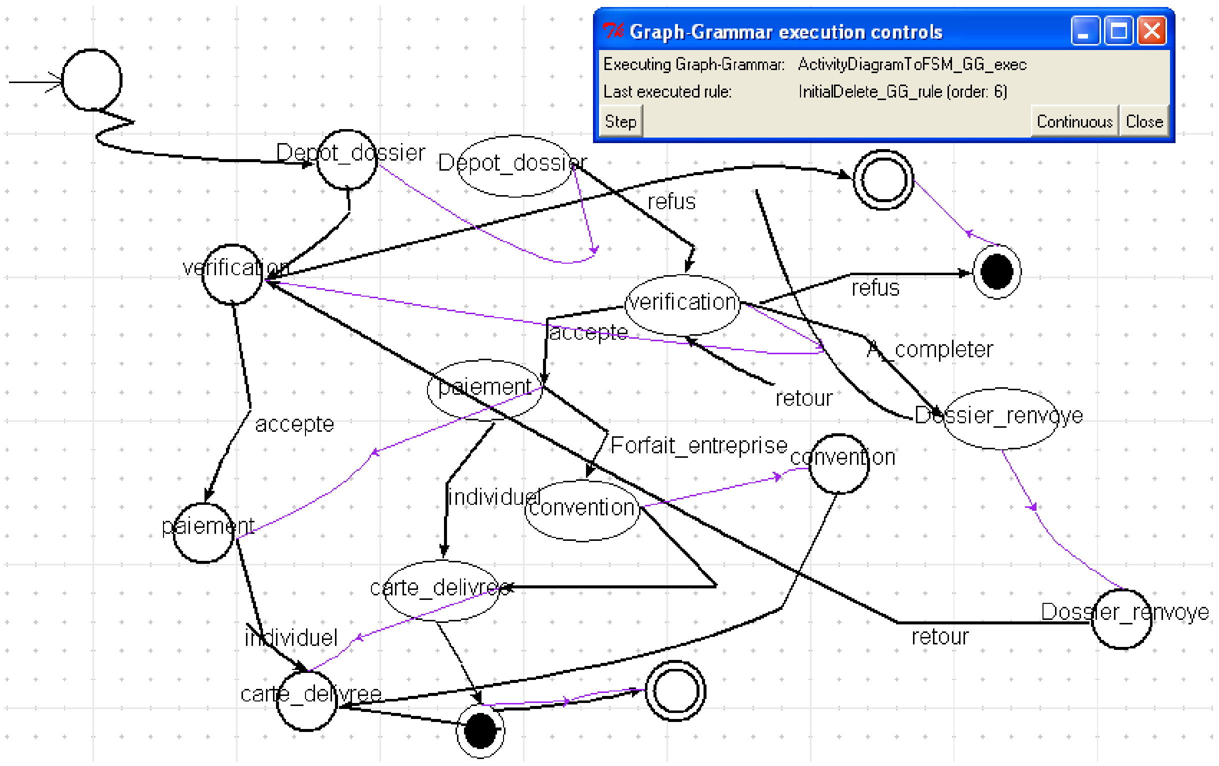


FIG. 5.15 - Intermediate graph of the example during execution.

After we complete the execution of the graph grammar we obtain the finite state automaton represented in the figure below:

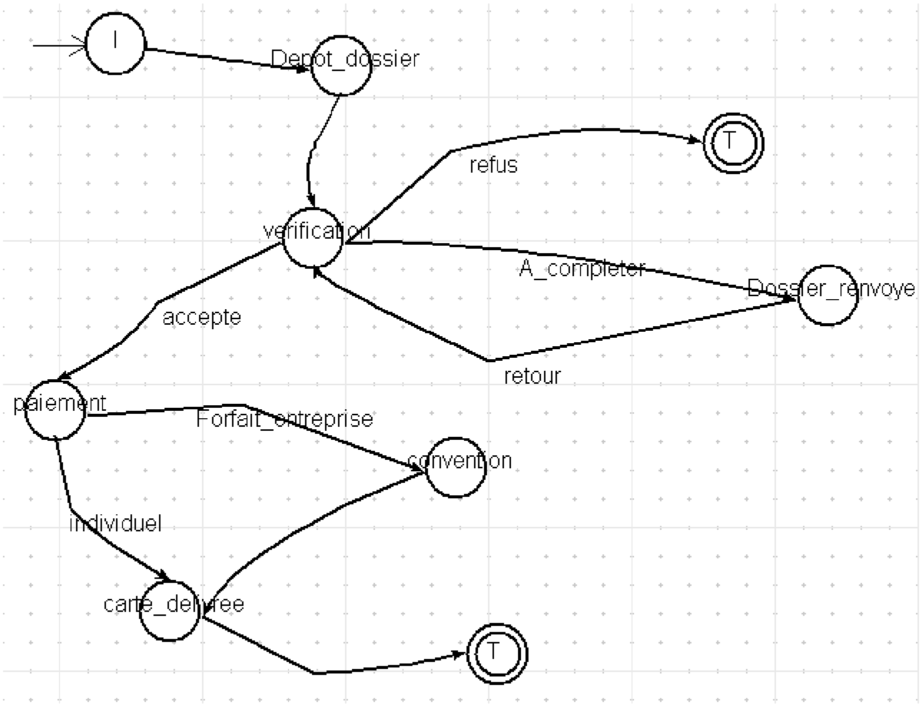


FIG. 5.16 –The resulted finite state automaton.

5.4. Review

5.4.1. Exercice01

1. Give the principle of graph transformation ?
   - *Graph transformation is the process of choosing a rule among the rules of the graph grammar, applying this rule to a graph and iterating the process until no rule can be applied.*
2. What is the difference between model transformation and graph transformation?
   - *Graph transformation is an implementation of model transformation.*
3. When should I choose graph transformation?
   - *When we have graphical models.*
4. I would like to do a model transformation in AToM^3 between two modeling formalisms, what should I do?
   - *define the meta-model of the source formalism.*
   - *define the meta-model of the target formalism.*
   - *propose the graph grammar.*
5. What are the types of transformations that can be performed in AToM^3?
   - *model to model.*
   - *model to text.*

Bibliography and Webography

[AEH+99]	M. Andries, G. Engels, A. Habel, B. Hoffmann, H-J. Kreowski, S. Kuske, D. Pump, A. Schürr and G. Taentzer**,** "*Graph transformation for specification and programming*"**,** Science of Computer programming, vol 34, NO°1, pages 1-54, Avril 1999.
[AGG]	AGG. Home page: http://tfs.cs.tu-berlin.de/agg/
[ATo]	AToM^3. Home page: http://atom3.cs.mcgill.ca/
[Aud07]	Laurent Audibert, "UML 2", http://www-lipn.univ-paris13.fr/audibert/pages/enseignement/cours.htm, 2007.
[BB02]	Bézivin, J. et X. Blanc: "*Promesses et interrogations de l'approche MDA*". Développeur Référence – Septembre 2002.
[Bel12]	A. Belghiat, "*Transformation des modèles UML vers des ontologies OWL*", Mémoire de Magister, Université de M'sila, Algérie, 2012.
[Bel17]	A. Belghiat, " *Une approche de spécification et de vérification des systèmes logiciels à base d'agents mobiles en utilisant UML mobile et π-calcul* ", Thèse de doctorat, Université de Constantine2, Algérie, 2017.
[Béz03]	Bézivin J., "*La transformation de modèles*", Ecole d'Eté d'Informatique CEA EDF INRIA : cours #6, INRIA-ATLAS & Université de Nantes, 2003.
[Bla05]	Xavier Blanc, "*MDA en action Ingénierie logicielle guidée par les modèles*", 2005.
[BM03]	E. Belloni, C. Marcos, "*Modeling of mobile-agent applications with UML*". In Proceedings of the Fourth Argentine Symposium on Software Engineering (ASSE 2003) (Vol. 32, pp. 1666-1141).
[CH03]	K. Czarnecki and S. Helsen, "*Classification of Model Transformation Approaches*", OOPSLA‟03 Workshop on Generative Techniques in the Context of Model-Driven Architecture**,** 2003.
[CH06]	K. Czarnecki, S. Helsen, "*Feature-based survey of model transformation approaches*", IBM SYSTEMS JOURNAL, VOL 45, NO 3, 2006.
[COT09]	Benoît Charroux, Aomar Osmani, Yann Thierry. "*UML 2 Pratique de la modélisation*" - 2e édition, 2009.
[FG05]	P. Farail, P. Gaufillet, "*TOPCASED Un environnement de développement Open Source pour les systèmes embarqués*", Airbus France, 2005.
[Car05]	E. Cariou, "*Ingénierie des modèles*", Cours de master, Université de Pau et des Pays de l'Adour, France, 2015.
[Fow03]	Martin Fowler, "*UML Distilled - Third Edition - A Brief Guide to the Standard Object Modeling Language*". Number ISBN: 0-321-19368-7 in The Addison-Wesley Object Technology Series, 2003.

[FUJ]	FUJABA. Home page: http://www.fujaba.de/.
[GRe]	GReAT. Home page: http://www.escherinstitute.org/Plone/tools/suites/mic/great/
[Har08]	Cécile Hardebolle, "*Composition de modèles pour la modélisation multi-paradigme du comportement des systèmes*", Thèse de Doctorat soutenu en 2008 à l'université de Université Paris-Sud XI, 205 pages.
[KA04]	G. Karsai, A. Agrawal, "*Graph Transformations in OMG's Model-Driven Architecture*", Lecture Notes in Computer Science, Vol 3062, 243-259, Springer Berlin / Heidelberg, juillet 2004.
[Kru95]	P. Kruchten "*Architectural blueprints - the "4+1" view model of software architecture*". IEEE Software 12(6), pages 42-50, 1995.
[OMG03]	OMG, "*Common Warehouse Metamodel (CWM)", version 1.1*. http://www.omg.org/technology/documents/formal/cwm.htm. Mars 2003.
[OMG04]	Object Management Group (OMG), *"Model Driven Architecture (MDA)"*, http://www.omg.org/mda. 2004.
[OMG05]	OMG, "*XML Metadata Interchange (XMI)", version 2.0.1*. http://www.omg.org/spec/XMI/2.0.1/. Juillet 2005.
[OMG06]	OMG, "*Meta Object Facility (MOF)", version 2.0*. http://www.omg.org/mof/. Janvier 2006.
[OMG10]	OMG, "*Unified Modeling Language (OMG UML) Superstructure*", *version* 2.3, http://www.omg.org/spec/UML/2.3/Superstructure. May 2010.
[Pie09]	Laurent Piechocki , "*UML, le langage de modélisation objet unifié*", Date de publication : 22/10/07, Date de mise à jour : 14/09/09.
[Roz99]	G. Rozenberg, "*Handbook of Graph Grammars and Computing by Graph Transformation*", Vol.1. World Scientific, 1999.
[VIA]	VIATRA. Home page: http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html
[You20]	M. Youssefi, "Cours Design Patterns". ENSET, Université Hassan II Casablanca, 2020.
[Lon14]	J. Lonchamp, "Conception d'applications en Java/JEE: Principes, patterns et architectures". Dunod. 2014.
[Dou99]	J. M. Doudoux, "Developpons en JAVA", 1999.
[Gam95]	E. Gamma, "Design patterns: elements of reusable object-oriented software", Pearson Education India. 1995.
[W3C99]	World Wide Web Consortium. James Clark, "*Extensible Stylesheet Language Transformation (XSLT) version 1.0*", W3C Recommendation 16 November 1999. Disponible en ligne http://www.w3.org/TR/xslt.