Université de Montréal

Modélisation et implémentation des patrons de conception

par
Yousra Tagmouti

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

juin, 2008

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Modélisation et implémentation des patrons de conception

présenté par:

Yousra Tagmouti

a été évalué par un jury composé des personnes suivantes:

Jian-Yun NIE
président-rapporteur

Aboulhamid El Mostapha
directeur de recherche

Yann-Gaël Guéhéneuc
codirecteur

Victor Ostromoukhov
membre du jury

Mémoire accepté le

RÉSUMÉ

Les patrons de conception sont des solutions communes à des problèmes de conception récurrent. Ils représentent un vocabulaire commun à l'expertise des concepteurs et favorisent le transfert de connaissances entre les concepteurs experts et novices.

À ce jour, plusieurs travaux ont été réalisés afin d'explorer le potentiel des patrons de conception pour améliorer la conception des systèmes. Cependant, la plupart de ces travaux se limitent aux systèmes purement logiciels. Très peu de travaux ont tenté d'intégrer les patrons de conception dans d'autres types de systèmes et plus généralement dans d'autres domaines de l'informatique, pour exploiter leurs avantages.

Nous contribuons à combler cette lacune en un travail de mise en correspondance entre des architectures matérielle et des patrons de conception orientés objet. Le travail de correspondance matérielle/logicielle se fait dans le cadre du projet Esys.net, qui a pour objectif la création d'un environement de modélisation et simulation de systèmes matériels, Esys.net.

Dans le cadre de cette collaboration, l'objectif de notre travail consiste à offrir à cette mise en correspondance matérielle/logicielle un environnement de modélisation afin de prendre en charge, d'une part, la modélisation des patrons et d'autre part, leurs implémentations en générant automatiquement du code Esys.net à partir des modèles de ces patrons.

Ainsi, nous avons (1) étendu le méta-modèle PADL, dédié à la modélisation des patrons de conception orientés objet, en un méta-modèle plus riche que nous avons nommé MIP et ce dans le but de pouvoir modéliser les différents aspects reliés aux patrons de conception orientés objet ; (2) étendu le méta-modèle MIP en un méta-modèle MIP$\mathcal{C}\sharp$ incluant les constituants du langage $\mathcal{C}\sharp$ qui différent de Java ; (3) modélisé douze patrons de conception avec le méta-modèle MIP et MIP$\mathcal{C}\sharp$; (4) construit les générateurs de code, *SimpleJavaGenerator*, *CSharpGenerator* et *JavatoCSharpGenerator*, permettant de générer du code Java et $\mathcal{C}\sharp$ à

partir des modèles de patrons définis dans l'étape précédente ; (5) construit un méta-modèle Esys.net afin de structurer ses différentes composantes ; (6) établit une correspondance entre les constituants du méta-modèle MIP et ceux du méta-modèle Esys.net afin de faciliter la génération de code Esys.net à partir des modèles de patrons ; (7) construit le générateur de code Esys.net et son application à un système de régulateur de vitesse implémenté en Esys.net que nous avons modélisé avec le méta-modèle Esys.net et généré le code Esys.net lui correspondant.

Mots clés : Patrons de conception, Esys.net, méta-modèle, PADL, MIP, MIP$\mathcal{C}\sharp$

ABSTRACT

The design patterns are common solutions to recurring design problems. They represent a common vocabulary to the expertise of designers and promote knowledge transfer between expert and novice designers.

So far, several studies have been performed to explore the potential of design patterns to improve the design of systems. However, most of them are limited to purely software systems. Very few studies have attempted to integrate design patterns in other types of systems and more generally in other areas of computing, to exploit their advantages.

We contribute to fill this gap by mapping hardware architectures with object oriented designs using design patterns. This mapping of hardware/software is realised in the context of the Esys.net project, which aims to create a modeling and a simulation environment of hardware systems.

Through this collaboration, the objective of our work is to provide to this matching hardware/software modeling an environment to support the modeling of patterns and their implementations generating automatically Esys.net code from these patterns.

Our contribution can be described as follow : (1) extending the meta-model PADL for modeling design patterns to a more developed meta-model (called MIP) that can model more aspects of design patterns ; (2) creating MIP$\mathcal{C}\sharp$ that is an extension of MIP by including components of $\mathcal{C}\sharp$ programming language ; (3) modeling 12 design patterns with MIP and MIP$\mathcal{C}\sharp$; (4) designing code generator *SimpleJavaGenerator*, *CSharpGenerator* and *JavatoCSharpGenerator* for $\mathcal{C}\sharp$ and Java coding from our 12 design patterns ; (5) designing a meta-model Esys.net to assemble its components ; (6) doing a model transformation between the MIP and Esys.net to assist code generating from a design pattern model ; (7) creating a code generator Esys.net, which we applied to a cruise control system modeled using meta-model Esys.net.

Keywords : Design patterns, meta-model, Esys.net, PADL, MIP,

$\mathbf{MIP}\mathcal{C}_\sharp$

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES SIGLES

COGENT	Code Generation Template
CGI	Common Gateway Interface
DEES	event-driven simulation
ESL	Electronic System level
Esys.net	Embedded System .NET
GOF	Gang of Four
HDL	Hardware description language
HTML	Hypertext Markup Language
LASSO	Laboratoire d'analyse et de Synthèse des Systèmes Ordinés
MIP	Modélisation et implémentation des patrons de conceptions
PADL	Pattern and abstract-level description language
RTL	Register transfert level event-driven simulation
UML	Unified Modeling Language
VHDL	Very high speed integrated circuit Hardware Description Language

XMI	XML Metadata Interchange

(dédicace) à ma maman et mon papa.

REMERCIEMENTS

Je remercie, tout d'abord, mes parents d'avoir cru en moi, de m'avoir encouragé tout au long de mon cursus et d'avoir sacrifié autant pour mon éducation et mes études. J'adresse tous mes remerciements aux personnes qui m'ont apporté leur soutien et aide et qui ont contribué à l'élaboration de ce mémoire.

Je remercie tout d'abord, Monsieur Aboulhamid El Mostapha et Monsieur Guéhéneuc Yann-Gaël, mes directeurs de recherche, d'avoir pris de leurs temps pour m'aider et m'encourager et je les remercie surtout pour leur joie de vivre «contagieuse» et leur bonne humeur.

Je remercie également tous les membres des laboratoires Geodes et LASSO pour leur aide et les enrichissantes pauses cafés. Enfin, j'adresse mes plus sincères remerciements à tous mes proches et mes amis qui m'ont toujours soutenu et encouragé au cours de mes études.

CHAPITRE 1

INTRODUCTION

1.1 Contexte

Plusieurs travaux ont tenté d'exploiter les avantages des patrons de conception en domaine du génie logiciel. Afin d'apporter notre contribution à un effort d'exploitation plus large, nous avons travaillé en collaboration avec des membres de l'équipe LASSO (*laboratoire d'analyse et de Synthèse des Systèmes Ordinés*) qui travaillent en particulier sur la conception de système logiciel et matériel, dans l'un des volets du projet Esys.net. Esys.net offre un environnement de développement et de modélisation d'architectures matérielles dans le cadre d'application .net. Esys.net est basé sur le langage de programmation orienté objet $\mathcal{C}\sharp$ et a permis la création d'un environnement de modélisation polyvalent en poussant les limites d'un langage de description matériel, *SystemC* qui permet une modélisation des systèmes au niveau comportemental. Cet environnement prendra en charge la modélisation et simulation de systèmes matériels/logiciels, afin de faciliter à l'utilisateur le passage d'une description d'un modèle abstrait à la réalisation du prototype correspondant.

Le volet du projet Esys.net où nous intervenons a pour objectif de faire correspondre des patrons de conception orientés objet à des architectures matérielles. Dans le cadre de cette collaboration, l'objectif de notre travail est, d'une part, d'offrir à cette représentation matérielle/logicielle un environnement de modélisation afin de prendre en charge la représentation des patrons et, d'autre part, d'assurer leurs implémentations en générant automatiquement du code Esys.net à partir de chaque modèle de patron.

1.2 Intérêts et objectifs du projet

Dans ce qui suit, nous allons décrire les objectifs que nous nous sommes assignés dans le cadre de ce projet de recherche ainsi que l'intérêt qu'ils présentent pour la

communauté scientifique.

1.2.1 Intégration des patrons de conception dans le domaine matériel

Les patrons de conception sont des solutions communes à des problèmes de conception récurrent. Considérés comme des solutions efficaces à l'usage et éprouvées par les concepteurs les plus expérimentés, ils permettent à ces derniers de partager leurs connaissances avec les plus novices d'entre eux. Dans le domaine du génie logiciel, plusieurs travaux se sont intéressés à l'étude des patrons de conception et leurs documentations, l'un des plus connus est le catalogue de Gamma et al. [GHJV94]. Cette équipe appelée GOF, pour Gang Of Four a, en fait, contribué à l'émergence des patrons de conception et leur adoption définitive dans la communauté du génie logiciel. Gamma et al. ont défini 23 patrons de conception et les ont classés en trois catégories, selon leurs rôles et leurs domaines d'application :

- les patrons créateurs, décrivant la manière de créer les différents objets ;
- les patrons structurels, permettant la création de nouvelles fonctionnalités par compositions entre classes et objets ;
- les patrons comportementaux, décrivant, d'une part les relations entre les classes et d'autre part la distribution des responsabilités entre elles.

À ce jour, de nombreux travaux tentent d'exploiter la force de ces patrons pour résoudre des problèmes de conception récurrent. Cependant, tous ces travaux se limitent aux architectures logicielles.

Afin d'exploiter les patrons de conception dans d'autres domaines que le génie logiciel, des membres de l'équipe LASSO ont tenté de trouver des correspondances entre des architectures matérielles et les patrons de conception définis dans le catalogue de Gamma et al. Car une telle correspondance pourrait donner naissance à de nouvelles applications intéressantes pour la synthèse de code. Ceci est d'autant plus pertinent que, dans le domaine matériel, la synthèse de l'orienté objet - concept sur lequel se basent la plupart des patrons de conception logiciels - connaît un vif intérêt de la part de la communauté scientifique [SSM01], [JSQK07].

1.2.2 Modélisation des patrons de conception

Plusieurs travaux ont tenté de donner aux patrons de conception une représentation simple et manipulable. Parmi les techniques de représentation les plus utilisées, on retrouve la méta-modélisation. *«Un modèle est une abstraction d'un système qui devrait être plus simple que celui-ci. Un mèta-modèle est la spécification d'une abstraction, i.e, d'un ou plusieurs modèles. Cette spécification définit un ensemble de concepts importants pour exprimer des modèles, ainsi que les relations entre ces concepts. Le méta-modèle définit la terminologie à utiliser pour définir des modèles»* [Mar04].

Dans le cadre de cette maîtrise, notre modélisation s'appuie sur le méta-modèle PADL (*Pattern and abstract-level description language*), défini par Albin et Guéhéneuc [AACG02]. Notre choix a été motivé par le fait que (1) PADL est développé au sein de notre équipe ; (2) a atteint une certaine maturité du fait que sa conception remonte à huit ans ; (3) est dédié au patron de conception ; (4) comprend un aspect génération de code qui nous intéresse pour assurer l'implémentation des patrons de conception.

1.2.3 Implémentation des patrons de conception

À la différence des algorithmes, qui s'attachent à décrire formellement la méthode de résolution d'un problème particulier, un patron de conception décrit le procédé à suivre pour résoudre un problème de conception. Mais au-delà d'une description du problème, de sa solution et du contexte dans lequel cette solution peut être considérée, la tâche de l'implémentation est laissée aux programmeurs.

Par conséquent, l'utilisateur peut trouver l'implémentation d'une solution, à partir de cette description difficile, et ce, même si des modèles de patrons incluent des fragments de codes illustrant la façon dont on peut implémenter le patron. Certains utilisateurs pourraient n'avoir aucune difficulté à traduire un modèle en code, mais cette opération peut devenir difficile ; surtout quand elle est répétitive. En effet, un changement de conception pourrait exiger une nouvelle implémentation

du code, car différents choix de conceptions dans le modèle peuvent engendrer des codes très différents.

C'est donc dans le but de prendre en charge l'implémentation des patrons que plusieurs travaux tentent de créer des environnements de modélisation de patrons [GAJM00], des langages d'implémentation dédiés aux patrons [RiC01], des outils de génération automatique de code [BFVY96], des modèles d'implémentation possible pour différents patrons [AACG02]. C'est dans le même but que nous allons modéliser des patrons de conception orientés objet et assurer leurs implémentations en générant automatiquement différents types de code à partir de chaque modèle.

1.3 Contribution

Nous allons présenter, dans ce qui suit, les éléments de notre contribution que nous avons structurés en cinq parties :

1. La première partie consiste à étendre le méta-modèle PADL (*pattern and abstract-level description language*) dédié à la modélisation des patrons en un méta-modèle plus riche, MIP (*modélisation et implémentation des patrons de conception*), afin de modéliser les différents aspects des patrons de conception orientés objet. Étendre le méta-modèle MIP en un méta-modèle plus riche MIP$\mathcal{C}\sharp$ incluant les constituants du langage $\mathcal{C}\sharp$ qui sont différents de ceux de Java.

2. La deuxième partie consiste à construire 12 modèles de patron basés sur les méta-modèles MIP et MIP$\mathcal{C}\sharp$. Dans la mesure où l'équipe LASSO s'intéresse aux patrons de conception orientés objet ; et particulièrement, les 23 patrons définis par GOF, notre modélisation s'intéressera également aux mêmes types de patrons.

3. La troisième partie consiste à assurer l'implémentation des patrons modélisés dans la deuxième partie, en générant du code automatiquement à partir des modèles de patron. Pour ce faire, nous avons implémenté trois générateurs de code indépendants, mais reliés : (1) Dans la mesure où PADL et MIP

sont basés sur le langage Java, le premier générateur implémenté est *SimpleJavaGenerator* qui génère du code Java. (Cette étape nous a permis de vérifier si, en nous basant sur le méta-modèle MIP, nous sommes capables de représenter toutes les entités des patrons orientés objet et de générer le code leurs correspondants) ; (2) dans la mesure où Esys.net est basé sur le langage C♯, le deuxième générateur implémenté est un générateur C♯, *CsharpGenerator*. (Cette étape nous a permis d'enrichir notre environnement avec un méta-modèle MIPC♯, afin de pouvoir décrire toutes les structures qui sont différentes entre Java et C♯) ; (3) le troisième générateur est une fusion des deux précédents, car il en résulte un générateur *JavatoCsharpGenerator* qui est capable de reconnaître des constituants C♯ et de leur donner une équivalence en Java. Il est important de construire ce type de générateur, car tous les générateurs doivent utiliser le même modèle de patrons pour générer différents types de codes.

4. Dans la quatrième partie, nous avons construit un méta-modèle dédié à la bibliothèque Esys.net dans le but de structurer tous ses constituants et faciliter son intégration dans notre environnement.

5. Dans la mesure où les constituants du méta-modèle Esys.net sont différents de ceux de MIP, nous proposons dans cette partie, une transformation de modèle basée sur les rôles entre les constituants du méta-modèle Esys.net et ceux de MIP. Cette transformation nous permettra d'utiliser un générateur de code tel que les précédents pour générer du code Esys.net.

6. Enfin, après la construction du générateur Esys.net nous l'avons appliqué ainsi que le méta-modèle Esys.net à un réel cas d'étude. Cette application consiste en la modélisation d'un régulateur de vitesse avec les constituants du méta-modèle Esys.net et assurer son implémentation en générant le code Esys.net lui correspondant en utilisant le générateur de code Esys.net.

Dans la mesure où le projet PADL est partagé par toute notre équipe, nous n'avons pas modifié le méta-modèle PADL directement, mais plutôt défini chacune

de nos contributions dans un projet indépendant de telle manière à ce que PADL soit le noyau de tous les projets. L'utilisateur aura ainsi la possibilité d'utiliser PADL indépendamment des autres projets, ou de télécharger chaque projet défini dans le cadre de ce travail et le relier à PADL, pour plus de fonctionnalités dans la modélisation et l'implémentation des patrons de conception.

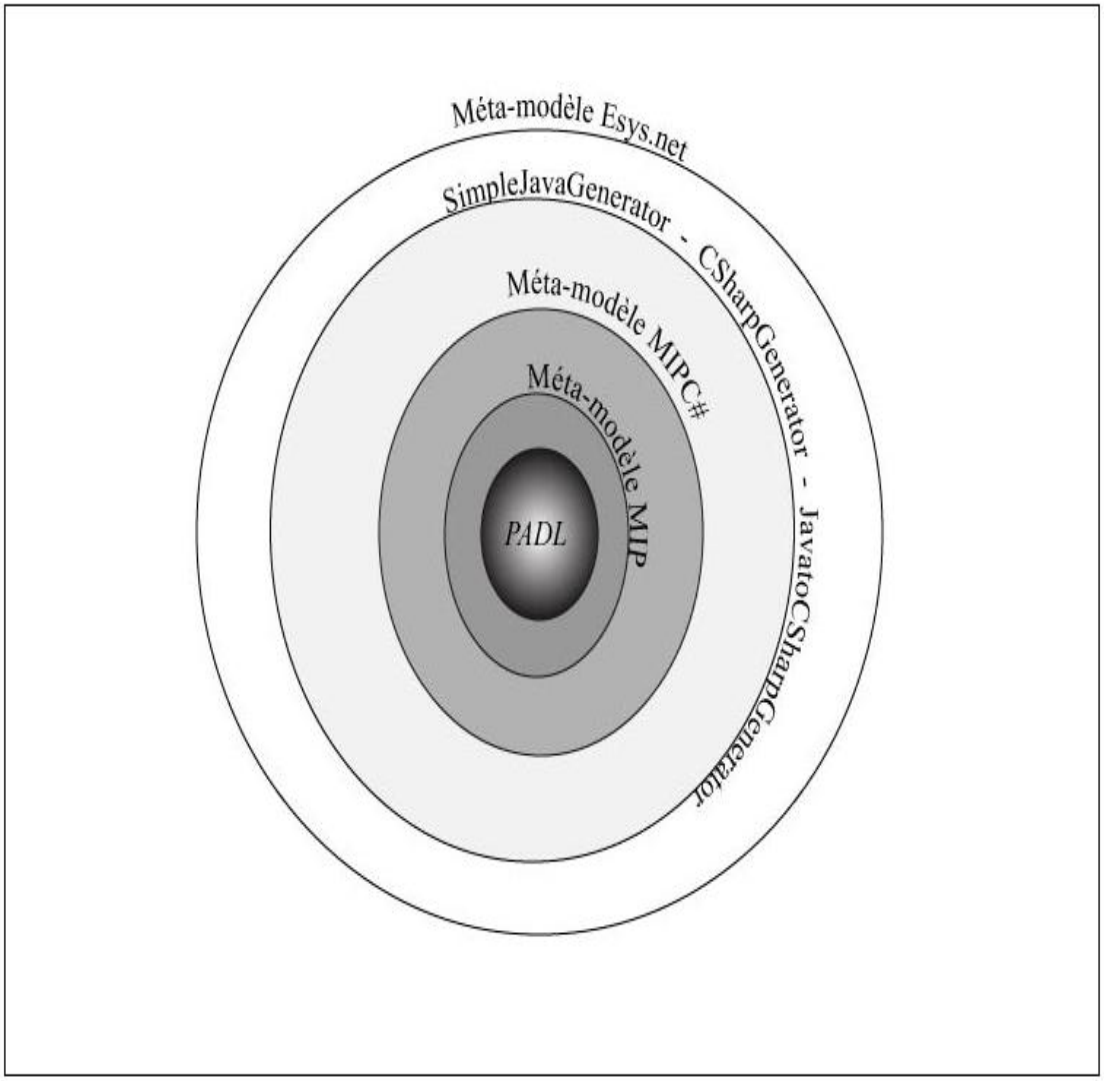

FIG. 1.1 – *Structure des projets ajoutés à PADL*

1.4 Structure du mémoire

La suite de ce mémoire s'articule comme suit : le deuxième chapitre présente l'état de l'art sur les travaux qui touchent à la modélisation des patrons, la génération automatique de codes à partir des patrons de conception et à la transformation de modèles. Nous consacrons, également, dans le cadre de ce chapitre, une patrie à la présentation des travaux liés à l'intégration des patrons de conception orientés objet dans le domaine matériel et le langage Esys.net. Le chapitre 3 présente le méta-modèle PADL utilisé pour la modélisation des patrons, les modifications que nous avons apportées à l'environnement, ainsi que la modélisation des patrons de conception du GOF. Le chapitre 4 est, quant à lui, consacré à la génération de code basée sur les modèles de patrons décrits au chapitre 3. Dans le chapitre 5, nous introduisons un méta-modèle consacré à la bibliothèque Esys.net, ainsi que la modélisation de deux patrons du GOF *Singleton* et *Observer* avec le méta-modèle Esys.net dans le but de reproduire la solution de chaque patron. En nous basant sur les rôles des constituants du *Singleton* et *Observer* dans la modélisation, nous avons établi nos choix de transformation entre les méta-modèles Esys.net et MIP. Dans le chapitre 6, nous présentons le générateur de code Esys.net et nous l'appliquons ainsi que le méta-modèle Esys.net à la modélisation d'un système matériel qui est un régulateur de vitesse. Finalement, nous conclurons ce travail et présenterons les travaux futurs dans le chapitre 7.

CHAPITRE 2

ÉTAT DE L'ART

Les travaux présentés dans ce chapitre portent sur deux axes principaux, chacun de ces axes est relié à nos objectifs, à savoir la modélisation des patrons de conception orientés objet et leur implémentation en générant du code Java, $\mathcal{C}\sharp$ et Esys.net à partir des modèles de patrons.

Nous allons consacrer ce chapitre à la présentation (1) du méta-modèle PADL que nous avons utilisé pour la modélisation et implémentation des patrons de conception orientés objet ; (2) des travaux reliés à la modélisation et implémentation des patrons de conception, dans le but de nous tenir au courant des différents moyens de modélisation et implémentation disponibles et justifier également le choix du méta-modèle PADL en le comparant à chacun des travaux passés en revue ; (3) des travaux reliés à l'intégration des patrons de conception dans les environnements d'architecture matérielle ; (4) de Esys.net afin d'introduire le lecteur à cet environnement ; (5) de quelques techniques de transformation de modèle que nous avons utilisés pour mettre en correspondance les constituants du méta-modèle Esys.net et le méta-modèle MIP.

2.1 Le méta-modèle PADL

PADL est le méta-modèle que nous avons choisi pour la modélisation et implémentation de nos patrons de conception.

Un modèle correspond à une image abstraite d'un système, pour que cette image soit fidèle au système qu'elle représente, elle doit inclure tous les constituants qui permettent la compréhension de ce dernier. La modélisation se fait dans le but de simplifier la conception d'un système en faisant abstraction de tous les détails non indispensables à cette compréhension. Un méta-modèle est d'un niveau d'abstraction plus élevé qu'un modèle, car il définit un ensemble de concepts qui

permettent de créer un modèle, ainsi que les interactions entre ces concepts.

Albin et Guéhéneuc [AACG02] ont tenté d'outiller un environnement de développement intégré « *Visual Age* » pour Java en lui adjoignant un catalogue recensant l'ensemble des patrons du GOF [GHJV94] et deux assistants dédiés à leur application et leur détection. Pour ce faire, les auteurs utilisent les techniques de méta-modélisation en créant le méta-modèle PADL qui se résume à un ensemble d'entités : classes et interfaces et d'éléments : méthodes, champs, etc. Cet ensemble de constituants permettent d'obtenir, par composition, la description d'un patron de conception.

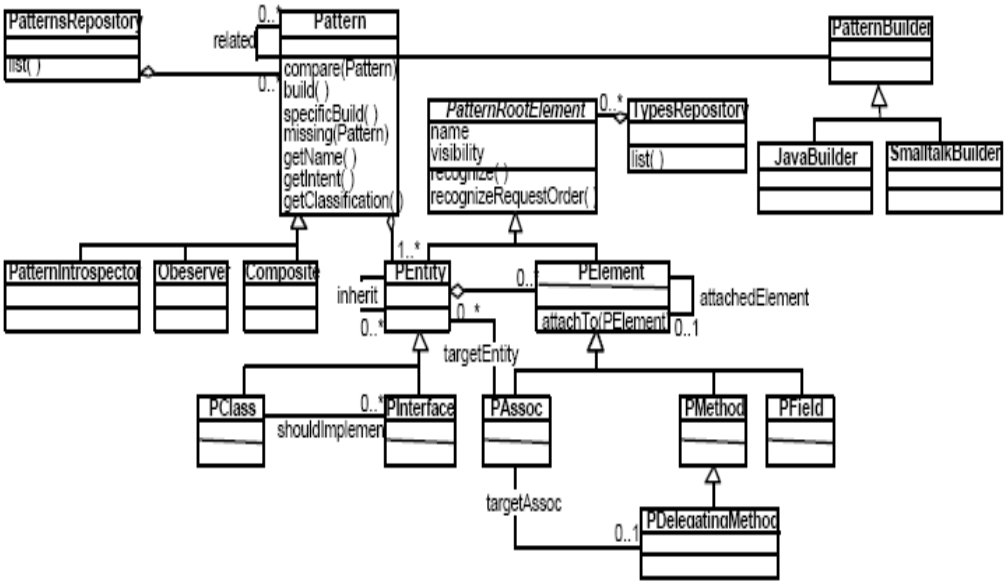

FIG. 2.1 – *Schéma UML du méta-modèle PADL (figure extraite de [AACG02])*

Suivant la sémantique du méta-modèle, chaque patron prend l'instance de la classe *Pattern* ou l'une de ses sous-classes et sera composé d'un ensemble de constituants correspondant aux classes et interfaces représentant la notion de participants qu'on retrouve dans le GOF. Chaque entité sera composée d'un ensemble d'éléments

correspondant aux méthodes, champs et paramètres.

La figure 2.2, présente les étapes d'utilisation du méta-modèle PADL :

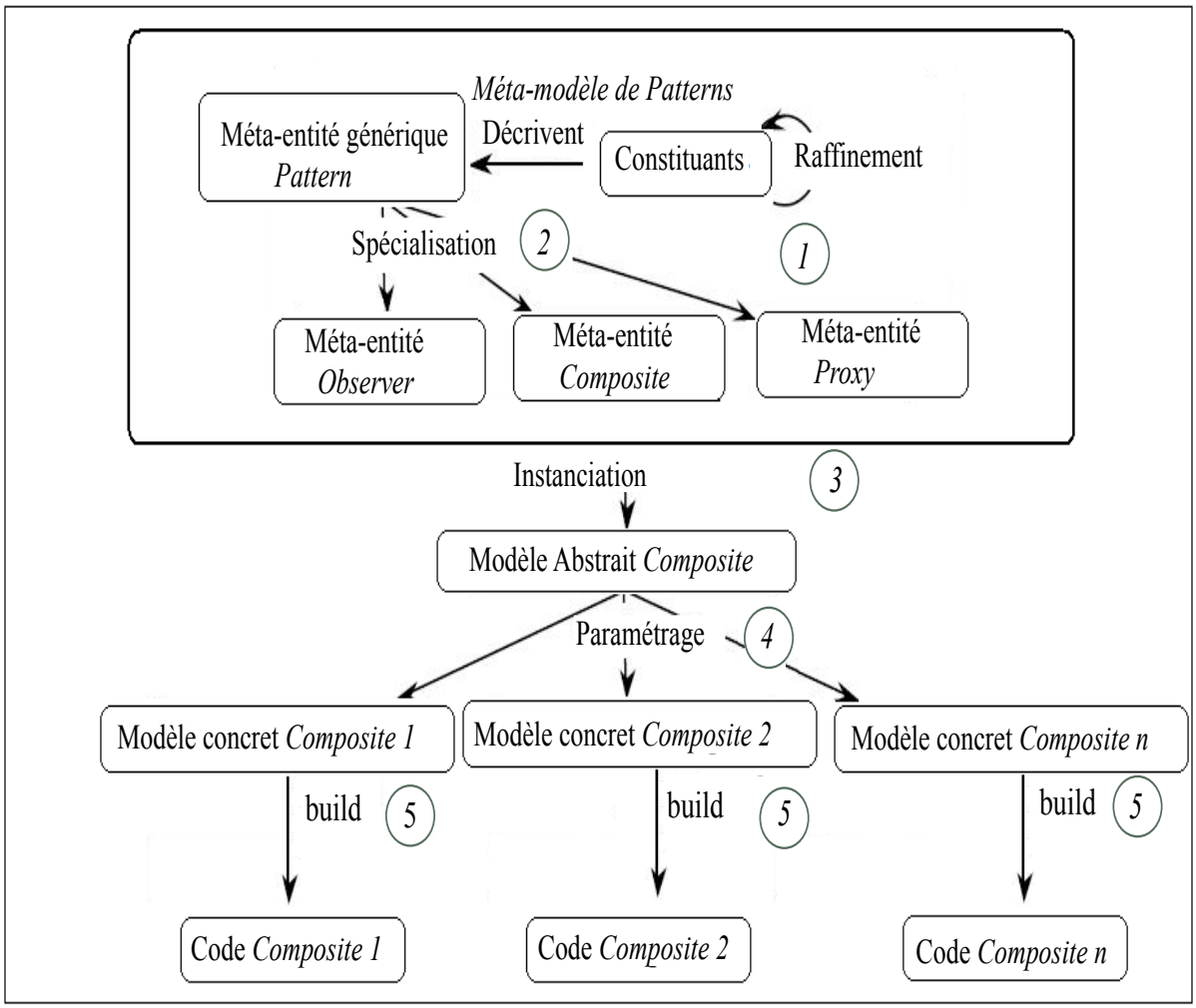

FIG. 2.2 – *Schéma général d'utilisation du méta-modèle PADL (figure extraite de [AACG02])*

1. La première étape consiste en la définition de la méta-entité générique *Pattern*. Cette définition est réalisée en deux étapes : (1) tout d'abord, le méta-modèle est raffiné en y rajoutant tous les constituants structurels et comportementaux nécessaires à la modélisation d'un patron donné. Ces constituants peuvent être utilisés par la suite pour la modélisation d'autres patrons.
2. La deuxième étape consiste en la description de la méta-entité spécialisée, qui se fait en suivant la sémantique définie par le méta-modèle. La méta-entité spécialisée est une particularisation de la méta-entité générique *Pattern* et joue deux rôles, décrire les modèles de patrons qui pourront en être issus et fixer les règles d'interactions avec les instances du patron qu'elle représente. La structure de la méta-entité spécialisée *Composite*, par exemple, correspond à la structure du patron *Composite* qu'on retrouve dans le GOF et de toutes les informations informelles représentant le *leitmotiv* [Ede99] et qui correspondent à la solution du patron. Le comportement de la méta-entité spécialisée *Composite* décrit les services permettant l'adaptation des instances de cette méta-entité à un contexte donné, cela comprend la définition des méthodes spécifiques au patron. La méta-entité spécialisée correspond à une classe Java que l'utilisateur peut instancier selon ses besoins de modélisation.
3. La troisième étape consiste en la définition du modèle abstrait du patron qui est obtenu par instanciation de la méta-entité spécialisée correspondante, c'est-à-dire en instanciant la classe Java correspondante à cette dernière.
4. La quatrième étape consiste en la définition du modèle concret du patron à partir de son modèle abstrait. Le modèle concret représente l'adaptation du modèle abstrait à un contexte donné et ce par paramétrage de ses constituants telles que les méthodes.
5. La dernière étape est celle de la génération de code qui est déclenchée par l'appel d'une méthode de la classe *Pattern*. Cette méthode donne l'équivalent en Java à chaque constituant visité dans le modèle.

Le but du méta-modèle PADL est d'obtenir un langage de description de pa-

trons dont chaque constituant peut être traduit dans un langage de programmation et être détecté dans une implémentation donnée. Ainsi, le modèle est utilisé à la fois dans le processus de génération et de détection des patrons. Cependant, dans le cadre de notre travail, nous nous intéresserons uniquement au processus de génération de code, que nous allons utiliser pour assurer l'implémentation de nos patrons de conception. Nous avons ainsi utilisé le méta-modèle PADL pour la modélisation et implémentation des patrons de conception orientés objet.

2.2 Modélisation et implémentations des patrons de conception

Cette partie est consacrée à la présentation des travaux reliés à la modélisation et implémentation des patrons de conception.

2.2.1 Une revue des travaux intégrant les patrons de conception

Ghizlane El Boussaidi et al. [BM05] ont fourni une classification des différents travaux reliés aux patrons de conception. Certains de ces travaux s'intéressent à la représentation des patrons et d'autres à leur automatisation en les intégrant dans des outils ou à des environnements de développement.

Durant leur recherche, les auteurs ont remarqué qu'il existe plusieurs approches pour représenter les patrons de conception, qu'ils ont classées en trois catégories :

1. La première famille d'approches représente les patrons d'une façon implicite et ne les dissocie pas de leur application. Cette approche permet une représentation plus rapide des patrons, car ces derniers sont présentés en même temps que leur application. Cependant, ce type de représentation ne permet pas une manipulation du patron, car sa représentation est incluse dans son application.
2. La deuxième famille d'approches propose une représentation explicite des patrons, mais ne sépare pas le mécanisme de leurs applications de cette représentation. Cette approche est plus intéressante que la représentation implicite, car elle donne à l'utilisateur la possibilité de modéliser les patrons.

Cependant, le mécanisme d'application des patrons est toujours associé à cette modélisation, ce qui ne donne toujours pas à l'utilisateur la liberté de manipuler un modèle de patron.

3. La dernière famille d'approches propose une représentation explicite du patron et son mécanisme d'application est indépendant du patron lui-même. Ce type de représentation est avantageux par rapport aux deux précédents, car il donne à l'utilisateur la liberté de manipuler son modèle de patron, ce qui pousse ce dernier à se familiariser avec les patrons de conception et mieux saisir le comportement, la structure, l'intention, etc. de chaque patron qu'il désire modéliser. Cette approche permet également la spécialisation d'un modèle de patron construi afin de l'adapter à un contexte particulier.

Suivant cette classification nous avons choisi une modélisation explicite des patrons de conception. Dans ce type de modélisation, la plupart des travaux proposent des méta-modèles pour la description des patrons de conception. Les techniques de méta-modélisation consistent en la définition d'un ensemble de méta-entités élémentaires qui permettent d'obtenir par composition la description d'un modèle de patron. Dans le domaine des patrons de conception, les méta-modèles sont utilisés pour permettre la description des différents aspects des patrons telle que la structure, le comportement, l'intention, etc. Cependant, chacun des méta-modèles traite d'un aspect de patron différent. En effet, à cause de l'aspect informel des patrons de conception, il est difficile qu'un méta-modèle puisse donner une interprétation à tous les aspects reliés aux patrons. De ce fait, plusieurs travaux ont défini des méta-modèles dans le but de capturer un ou plusieurs aspects reliés aux patrons.

2.2.2 Un framework pour la documentation des patrons de conception

Le paradigme orienté objet a permis d'améliorer le développement logiciel, en éliminant dans le code la description d'une action à réaliser de façon linéaire, mais dans un ensemble cohérent appelé «objet». En permettant ainsi d'écrire du code

souple et réutilisable [Pas02].

Les *frameworks* font partie des techniques permettant d'exploiter pleinement les avantages de l'orientés objet. *«Un framework est un espace de travail modulaire. C'est un ensemble de bibliothèques, d'outils et de conventions permettant le développement d'applications. Il fournit suffisamment de briques logicielles et impose suffisamment de rigueur pour pouvoir produire une application aboutie et facile à maintenir»* [WS]. Afin de permettre la structuration et l'exploitation des patrons de conception, les auteurs ont défini un *framework* permettant essentiellement la documentation des patrons de conception.

Bien que le *framework* développé par les auteurs permet l'exploitation des patrons de conception et a la possibilité de manipuler un patron de conception dans un modèle, la génération de code permettant l'implémentation des patrons n'est pas abordée. De ce fait, nous avons choisi le méta-modèle PADL pour la modélisation et implémentation des patrons de conception.

2.2.3 Protocole de définition des méta-entités Pattern

Depuis l'émergence des patrons de conception dans le domaine du génie logiciel, de nombreux travaux se sont intéressés à leur représentation et formalisation dans le but de faciliter leur compréhension et ainsi leur utilisation. Ces efforts sont motivés par le fait que l'utilisation des patrons de conception aide à la conception des applications et à leur documentation [R.J, OQC]. Cependant, du fait qu'un patron de conception n'est qu'un modèle, chaque utilisateur peut créer sa propre version du patron selon l'application en développement.

Lors de la conception d'une application, l'intégration d'un patron peut se résumer à cloner un modèle de patron en créant et renommant les constituants du patron, telles que les classes, les interfaces, les méthodes, etc. ou à intégrer un patron à une application existante, et ce, en documentant du code existant ou encore en le structurant en suivant la sémantique d'un patron donné. Il faut donc, commenter le code ou le modifier afin de l'adapter au patron utilisé. Dans les deux cas, le travail est souvent répétitif et peut mener à des erreurs, lorsque la hiérarchie des

classes est modifiée et les méthodes ajoutées pour intégrer le patron désiré. Après l'adaptation de l'application pour y intégrer un patron de conception il reste au concepteur à vérifier que sa représentation du patron est adéquate à un patron décrit à l'aide de plusieurs pages de diagrammes, d'exemple, de code, etc. [HJY].

Dans ce travail [RBF00], les auteurs prennent en charge l'intégration des patrons dans la conception d'un système. Pour ce faire, les auteurs ont pour but d'aider l'utilisateur à l'instanciation et la vérification des patrons de conception. Les auteurs ont isolé les différentes entités nécessaires à la définition des patrons de conception dans un protocole, appelé «*le protocole de définition des méta-entités Pattern*». Les auteurs considèrent le modèle d'un patron de conception comme un *méta-pattern*. Une version concrète du patron est alors une instanciation du *méta-pattern* et représente une personnalisation des entités du patron pour une application particulière. Le terme de *méta-pettern* a été emprunté à [Pre94] qui propose une réification partielle des patrons au niveau structurel et qui a été repris par les auteurs afin de prendre en compte d'autres aspects d'un patron et permettre aussi sa manipulation (instanciation, vérification ou documentation). Ainsi, chacun des constituants du *méta-pattern* comprend des informations structurelles et comportementales qui lui sont propres. Si, par exemple, l'entité du *méta-pattern* correspond à un générateur de méthode il peut définir la signature de ses instances, la manière de l'instancier, etc.

En se basant sur le protocole de définition des méta-entités *Pattern*, les auteurs ont développé un outil permettant la manipulation, l'instanciation et la vérification des patrons de conception, mais également la définition de nouveaux patrons. La version actuelle de l'outil construit par les auteurs offre des fonctionnalités d'instanciation et de vérification. À partir d'un diagramme UML (*Unified Modeling Language*) au format XMI (*XML Metadata Interchange*) l'outil retourne un nouveau fichier XMI contenant l'instance créée. Durant cette phase d'instanciation, l'utilisateur interagit avec l'outil en lui donnant par exemple, le nombre d'instances qu'il désire créer lorsqu'une classe peut avoir plusieurs instances. Cependant, l'opération de vérification n'est pas encore développée par les auteurs. L'outil

développé est fourni avec un catalogue de *méta-pattern*, ainsi après avoir retenu un patron du catalogue ou conçu un patron de conception correspondant à son besoin, le développeur peut instancier ce patron ou vérifier la justesse du patron concret.

La méta-entité *Pattern* proposée par les auteurs s'intéresse à l'instanciation et la vérification du patron. Cependant, l'aspect génération de code n'est pas abordé. Bien qu'en nous basant sur le travail des auteurs, nous pouvons modéliser des patrons de conception. Toutefois, le méta-modèle PADL nous offre une double perspective à savoir la modélisation des patrons de conception et leur implémentation en générant du code à partir des modèles de patron.

2.2.3.1 Génération automatique de code à partir des patrons de conception

Budinsky et al. [BFVY96] proposent un outil qui donne un accès rapide à la description des patrons de conception, organisé sous forme de pages HTML (*Hypertext Markup Language*). Cet outil englobe une représentation des 23 patrons du catalogue du GOF, en se basant sur le formalisme des patrons de conception de Gamma et al. [GHJV94]. Chaque section du patron (*intention*, *motivation*, etc.) est affichée dans une page séparée. Dans la figure 2.7, on retrouve une page HTML qui présente l'intention du patron *Composite*.

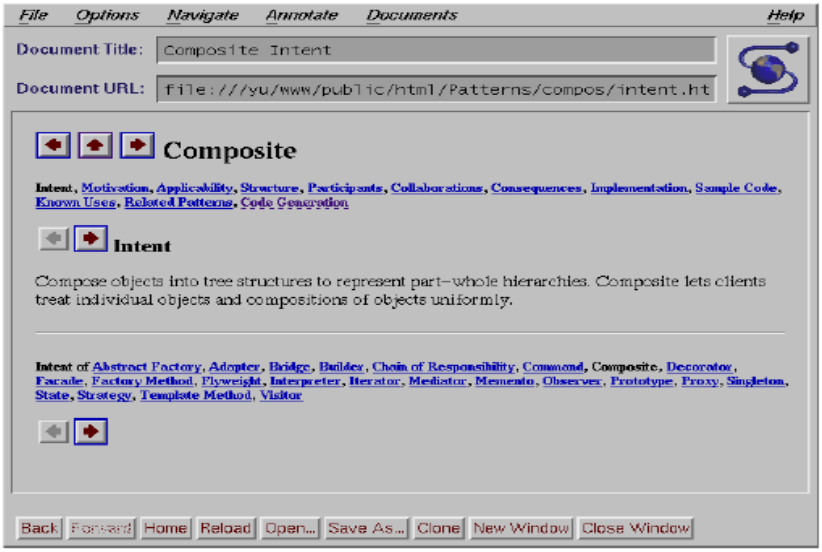

FIG. 2.3 – *Page HTML montrant la motivation du patron Composite (figure extraite de Budinsky et al. [BFVY96])*

En plus de ces pages, une page de génération de code est associée à chaque patron figure 2.4. Dans cette page, l'utilisateur peut rentrer des informations qui permettront une implémentation personnalisée du patron. Ces informations se limitent au nom des classes du patron et quelques compromis d'implémentation.

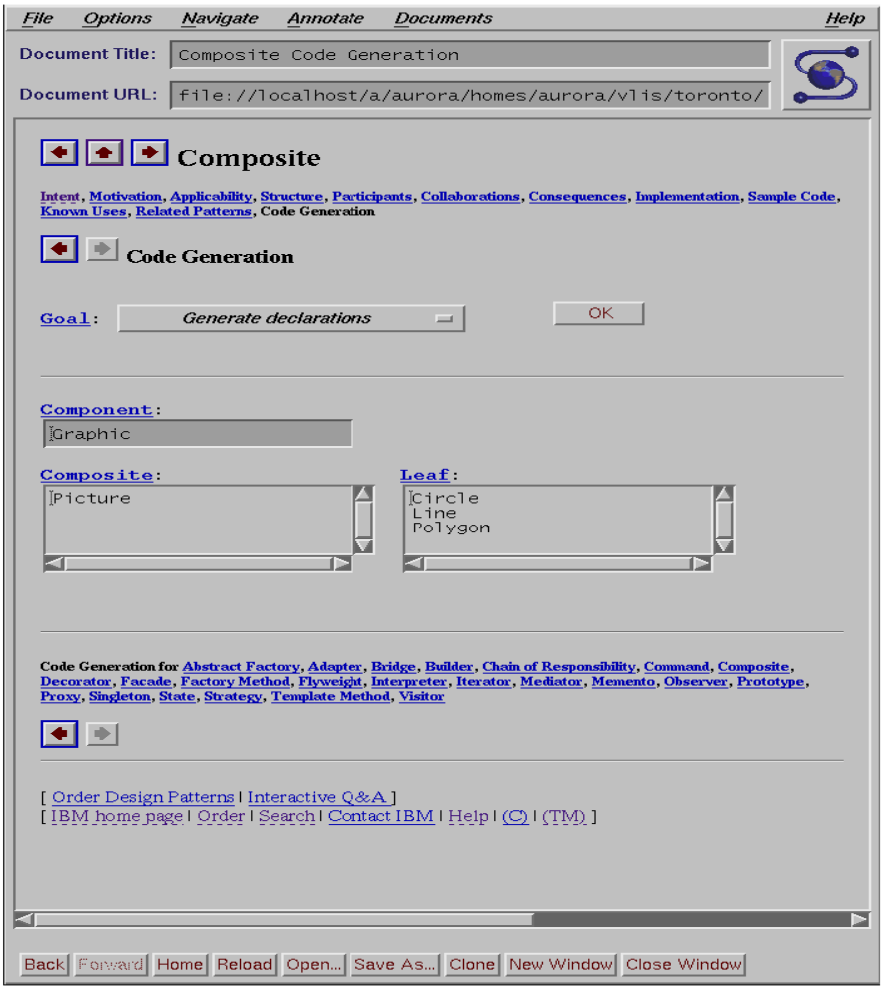

FIG. 2.4 – *page représentant le code correspondant au patron Composite (figure extraite de Budinsky et al. [BFVY96])*

L'utilisateur navigue ainsi entre ces pages jusqu'à ce qu'il trouve la solution qui lui convient.

L'architecture de l'outil se compose de trois parties :

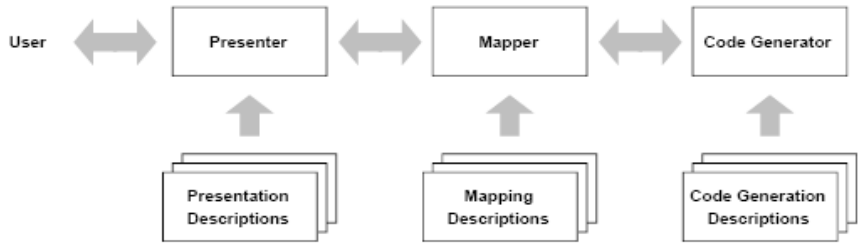

FIG. 2.5 – *Architecture de l'outil (figure extraite de Budinsky et al. [BFVY96])*

- Le présentateur *PRESENTER* qui est une sorte de navigateur chargé d'afficher les pages de description des patrons de conception écrites en HTML ;
- Le générateur de code qui interprète la spécification dans la page de génération. L'interpréteur est appelé *COGENT* (*Code Generation Template*) et génère le code en C++ ;
- Le *Mapper* qui, quant à lui, spécifie comment l'interface utilisateur et le générateur de code coopèrent. Le *Mapper* reçoit des informations du browser Web, par des commandes CGI (*Common Gateway Interface*), les traite et les transmet au générateur de code ;

Cette approche a le mérite d'être rapide. Cependant, elle ne prend pas en charge la représentation explicite des patrons. Elle permet de les représenter d'une façon textuelle sous forme de pages HTML, sans toutefois, proposer un modèle indépendant et manipulable. De plus, la génération de code n'est intégrée à aucun environnement de conception. En effet, les morceaux de code C++ engendrés apparaissent dans une page HTML et doivent être copiés et insérés à l'intérieur du code source de l'application en développement.

2.3 L'application des patrons de conception orientés objet à la conception d'architectures matérielles

Charest et Aboulhamid [CM08] proposent l'application des patrons de conception orientés objet à la conception d'architectures matérielles, car une telle union pourrait donner naissance à de nouvelles applications intéressantes pour la synthèse de code. Ceci est d'autant plus pertinent que, dans le domaine matériel, la synthèse de l'orienté objet, concept sur lequel se basent la plupart des patrons de conception logiciels, connaît un vif intérêt de la part de la communauté scientifique [SSM01], [JSQK07]. Les auteurs partent du principe qu'une variable orientée objet peut correspondre en matériel à un registre, en appliquant ainsi les constituants des patrons de conception orientés objet à la conception d'architectures matérielles.

Les auteurs [CM08] s'intéressent aux patrons orientés objet et plus précisément les patrons définis dans le catalogue du GOF, avec la motivation qu'il pourrait y avoir un catalogue recensant un ensemble de solutions mises en place par des concepteurs expérimentés *«Hardware Design Patterns»*. Cependant, l'intention des auteurs n'est pas de définir ce catalogue, mais de montrer certaines relations (même au niveau théorique) entre les solutions des patrons de conception orientés objet et les architectures matérielles, en permettant ainsi le partage de ses solutions entre les concepteurs expérimentés et novices.

Prenons, par exemple, un circuit C qui prend trop de temps à s'exécuter, figure 2.6. Pour un concepteur matériel la solution classique pour accélérer le circuit de données est de le découper en sous circuit (c1, c2, c3, etc.), chacun des sous-circuits pouvant exécuter un petit ensemble de données en parallèle.

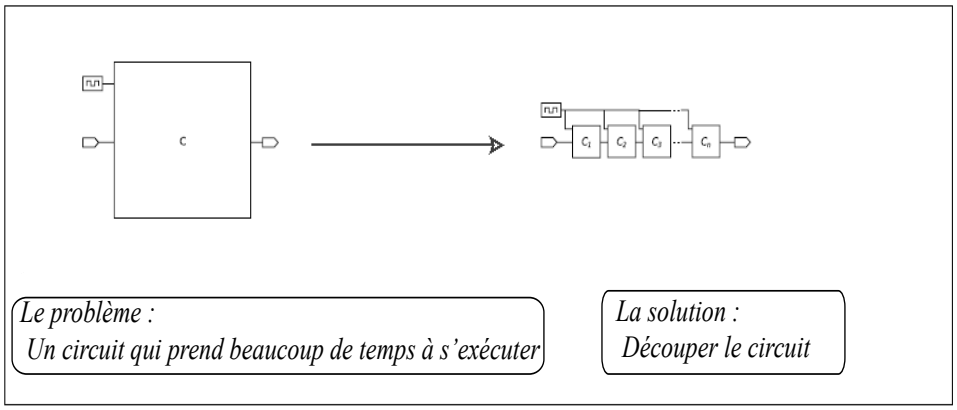

FIG. 2.6 – *La solution à un problème de conception d'architecture matérielle*

Afin de structurer leur approche, les auteurs ont défini une classification dans leurs correspondances entre les patrons de conception logiciels et les architectures matérielles :

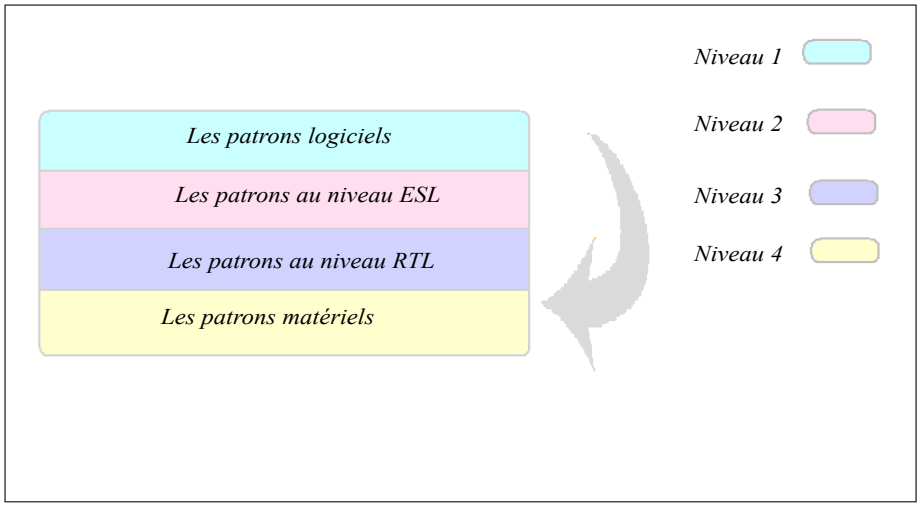

FIG. 2.7 – *Les niveaux de correspondance des patrons de conception orientés objet*

– **Les patrons logiciels :** correspondent aux patrons orientés objet et plus précisément ceux définis dans le catalogue du GOF. Les patrons de conceptions logiciels représentent la base de la correspondance définie par les auteurs, dans le sens où le but des auteurs est de faire correspondre des archi-

tectures matérielles à ces patrons logiciels ;

- **Les patrons aux niveaux ESL :** la correspondance à ce niveau, se fait entre les patrons logiciels définis au niveau 1 et les patrons appliqués au niveau conceptuel. De manière générale, la conception et la vérification au niveau du système électronique ESL (*Electronic System level*) offre des outils et des méthodologies qu'on peut utiliser pour décrire et analyser des circuits à un niveau d'abstraction élevé. La modélisation au niveau ESL se fait dans des environnements où le concept orienté objet est disponible. Comme pour une utilisation des patrons orientés objet au niveau conceptuel, les patrons utilisés au niveau ESL peuvent être très proches des patrons orientés objet si ce ne sont pas les mêmes ;
- **Les patrons au niveau RTL :** la correspondance à ce niveau se fait entre les patrons logiciels et un ensemble de patrons qui peuvent être appliqués à des problèmes de modélisation au niveau RTL (*Register transfert level event-driven simulation*). Au niveau RTL le comportement d'un circuit est défini en termes d'envois de signaux ou de transferts de données entre registres et les opérations logiques effectuées sur ces signaux. L'application des patrons au niveau RTL et à ses niveaux inférieurs, est plus éloignée de l'application des patrons logiciels reliés au paradigme orienté objet ;
- **Les patrons matériels :** peuvent se résumer à des patrons de conception électroniques qui optimisent la conception électronique d'un système matériel et favorisent la réutilisation des constituants d'un système.

  La principale idée de la correspondance est de transposer le concept des patrons logiciels existant qui représentent le niveau de conception le plus haut, (niveau 1 de la figure 2.7) vers chacune des couches inférieurs, en passant ainsi du logiciel à ESL à RTL pour arriver finalement au matériel. Présentement les auteurs se placent au niveau ESL en faisant correspondre des patrons de conception logiciels à des systèmes matériels au niveau conceptuel.

  Avant de procéder à la correspondance entre les systèmes au niveau ESL et les patrons logiciels, les auteurs ont commencé par établir des traductions

entre les constituants du paradigme orienté objet tels que les classes, le polymorphisme, l'héritage, etc. vers des constituants conceptuels d'architectures matérielles.

Dans la mesure où l'un de nos objectif est d'offrir à cette correspondance un environnement de modélisation et d'implémentation prenant en charge les patrons logiciels ainsi que ceux résultant de la mise en correspondance matérielles/logicielles, nous nous plaçons également au niveau ESL pour reproduire le comportement des patrons de conception orientés objet, en utilisant des constituants du méta-modèle Esys.net qui sera présenté au chapitre 7.

2.4 La bibliothèque Esys.net

La bibliothèque Esys.net a été développée par James Lapalme en 2003 [LAN+04] au sein du laboratoire LASSO et est le fruit d'une collaboration de ce dernier avec des professeurs et des étudiants d'autres laboratoires à l'école polytechnique de Montréal. Esys.net offre un environnement de développement et de modélisation d'architectures matérielles dans le cadre d'applications .net de Microsoft. En effet, Esys.net a été développé en respectant les standards de Microsoft et a ajouté à l'environnement .net des concepts fondamentaux à la modélisation d'architectures matérielles tels que les signaux, les modules, les événements, etc. L'attachement d'Esys.net à l'environnement .net qui a une grammaire de programmation très riche, rend la programmation avec Esys.net flexible.

Le projet Esys.net est né pour répondre aux lacunes des langages de développement de systèmes matériels HDL (Hardware description language) tels que Verilog et VHDL (Very high speed integrated circuit Hardware Description Language) qui, bien qu'ils soient toujours d'actualité dans le domaine, ne permettent pas une exploration large des modèles à décrire et la modélisation des systèmes logiciels.

Afin d'apporter une amélioration au niveau de la modélisation des systèmes matériels, Esys.net a poussé les limites d'un langage de description de matériel

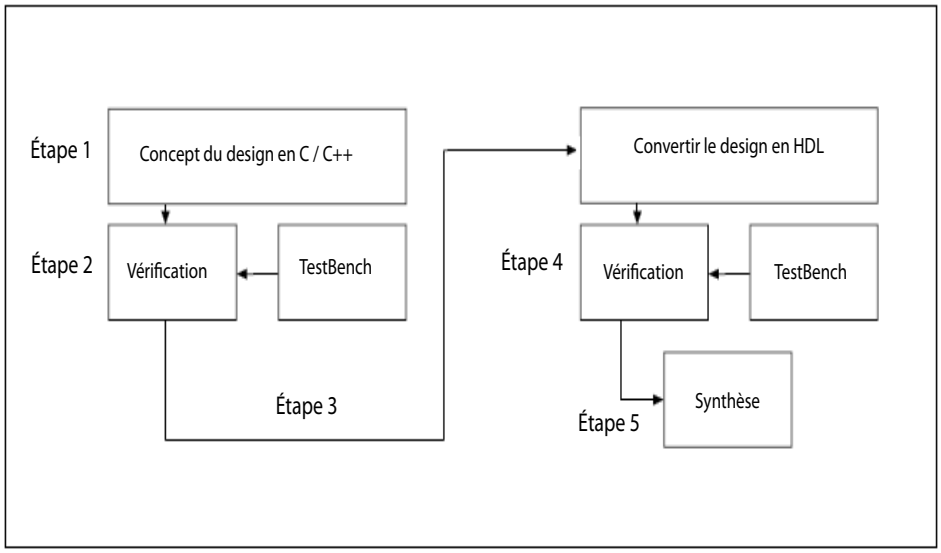

FIG. 2.8 – *Méthodologie de développement utilisé jusqu'à ce jour (figure extraite de [Gir05]*

qui permet une modélisation des systèmes au niveau comportemental (à savoir *SystemC*) et la création d'un environnement plus polyvalent. Le choix de *SystemC* parmi tous les autres langages de modélisation matérielle est motivé par le fait que *SystemC* présente de nombreux avantages sur ses précédents. Cependant, *SystemC* a aussi ses inconvénients : (1) la librairie de *SystemC* est basée sur le langage C++ qui est un langage très vaste. De ce fait, il est parfois difficile d'appréhender le comportement du simulateur ; (2) *SystemC* n'est pas assez riche, ce qui ne permet pas la description de tous les constituants d'un système lors de sa modélisation avec *SystemC* ; (3) le principal inconvénient de *SystemC* réside dans le fait qui'il ne possède pas un mécanisme d'introspection permettant la conception d'outils de synthèse efficace.

L'environnement Esys.net supporte tous les langages respectant le standard ECMA [Mic] tels que J♯, $\mathcal{C}\sharp$, $\mathcal{C}++$, VB.net, etc. et l'infrastructure .net peut y être utilisée. Esys.net adopte le concept orienté objet et sa structure est proche du *SystemC*. l'environnent de développement de Esys.net présente des avantages

sur ses concurrents. La figure 2.8 présente le processus suivi par le concepteur pour décrire le modèle d'un système matériel dans les différents environnements de modélisation connus à ce jour :

- le développeur fait la description du modèle correspondant à un système matériel dans l'environnement C ;
- le modèle est ensuite vérifié à l'aide d'un *testBench* qui est une entité qui n'a ni entrée ni sortie et qui est destinée uniquement à la simulation ;
- l'utilisateur convertit par la suite son modèle manuellement dans un langage HDL ;
- une fois le modèle converti, il est vérifié à l'aide d'un *testBench* ;
- le modèle passe, finalement, à la synthèse qui consiste à compiler la description fonctionnelle d'un circuit à l'aide d'un outil de synthèse ;

La conversion du modèle réalisée à l'étape 3 de ce processus peut être difficile, couteuse et susceptible de générer plusieurs erreurs d'implémentation dans le modèle. Dans le but d'améliorer la méthodologie, Esys.net propose un environnement qui supporte toutes les étapes de la figure 2.9, ainsi le concepteur pourra se familiariser avec son unique environnent de développement sans avoir à migrer vers d'autres environnements à chaque étape de la modélisation. Le concepteur aura aussi la possibilité de raffiner librement son modèle et le sectionner en plusieurs composantes distinctes. L'environnement Esys.net permet ainsi la programmation à différents niveaux et avec une abstraction très élevée pour une programmation plus rapide figure 2.9.

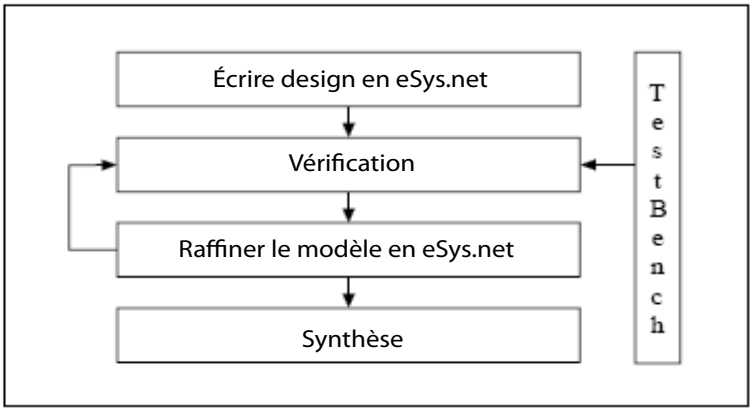

FIG. 2.9 – *Méthodologie de esys.net (figure extraite de [Gir05])*

La puissance de la méthodologie Esys.net se résume dans : (1) le développement des modèles de systèmes matériels complexes avec très peu de code ; (2) une simulation facile ; (3) la possibilité d'appliquer les mêmes *testBench* à plusieurs niveaux de modélisation pour gagner ainsi du temps. Esys.net est composé d'un ensemble de classes abstraites correspondant aux constituants d'un système matériel [Lab98], tels que les classes, les interfaces, les modules, les événements, etc. Ces différents concepts seront présentés plus en détail lors de la construction du méta-modèle Esys.net au chapitre 6 de ce mémoire.

2.5 Les techniques de transformation de modèle

Krzysztof et al. [CH03] proposent une taxonomie, dans le but de définir une classification des différentes approches de transformations de modèles. Pour leur classification, les auteurs proposent un modèle de fonctionnalité permettant de structurer les différents choix de conception des approches de transformation de modèles. Dans le cadre de ce modèle de fonctionnalité, les auteurs classifient les approches passées en revue en deux grandes catégories : (i) les approches *Model-To-code* qui permettent d'établir une correspondance entre une patrie du code d'un

programme et une entité du modèle représentant le modèle du code et vice versa ; (ii) les approches *Model-To-Model* consistent quant à elles à faire correspondre deux entités d'un modèle en se basant sur leurs rôles, leurs structures, etc. Dans le cadre de notre travail nous utiliserons une approche de transformation *Model-To-Model*, entre les constituants du méta-modèle Esys.net et ceux du MIP, en nous basant sur leurs rôles.

CHAPITRE 3

EXTENSION DU MÉTA-MODÈLE PADL

Nous avons choisi le méta-modèle PADL pour modéliser 12 motifs de conceptions du catalogue du GOF et assurer la génération du code leur correspondant, par la construction de trois générateurs de code. Le choix du méta-modèle PADL a été motivé par le fait que PADL : (1) est développé au sein de notre équipe ; (2) a atteint une certaine maturité, car sa conception remonte à huit ans [AACG02] ; (3) est dédié aux patrons de conception. De plus, PADL comprend déjà un aspect génération de code qui nous intéresse, tout particulièrement, pour assurer l'implémentation des patrons de conception.

Chaque patron de conception propose une solution pour résoudre un problème et la manière de mettre en place cette solution se résume au comportement et à la structure de la solution proposée [GHJV94]. Cependant, le méta-modèle PADL présente certaines limites qui nous empêchent de décrire le comportement de la solution des patrons de conception, car il ne permet de décrire que leurs structures. Or, dans notre modélisation, nous avons besoin de donner à la solution des patrons une représentation complète, incluant le comportement de la solution.

Nous allons consacrer cette section à la présentation (1) des limites du méta-modèle PADL ; (2) d'un exemple de modélisation du motif de conception *Singleton*, afin d'illustrer les limites de PADL ; (3) des modifications que nous avons apportées au méta-modèle PADL dans un méta-modèle MIP, ainsi que l'interprétation des constituants que nous lui avons ajoutés pour permettre une description complète des patrons. Dans le but d'illustrer l'intérêt des modifications apportées, nous reprendrons la modélisation du patron de conception *Singleton* en utilisant les constituants ajoutés à PADL. Dans la mesure où le méta-modèle PADL et MIP sont tous deux basés sur le langage Java, lors de la construction du générateur de code $\mathcal{C}\sharp$, nous avons étendu le méta-modèle MIP en un méta-modèle MIP$\mathcal{C}\sharp$. Ce méta-modèle va inclure tous les constituants faisant la différence entre les langages Java et $\mathcal{C}\sharp$.

La figure suivante résume notre contribution dans cette section.

3.1 Limites du méta-modèle PADL

Albin et Guéhéneuc [AACG02] ont défini un méta-modèle appelé PADL, dans le but d'obtenir un langage de description dédié aux patrons de conception, dont chacun des constituants peut être traduit dans un langage de programmation et être détecté dans une implémentation donnée. La figure 3.1 résume la structure du méta-modèle PADL.

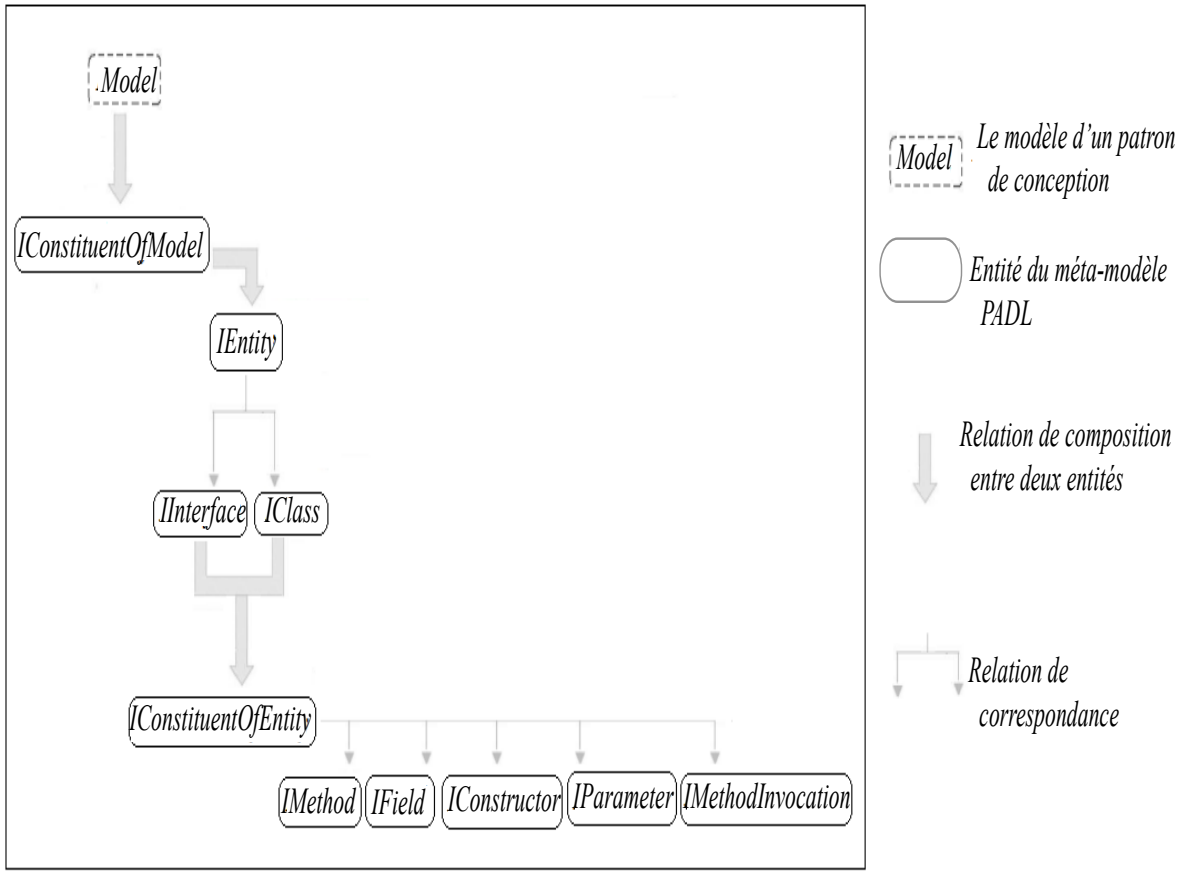

FIG. 3.1 – *Schéma illustrant la structure du méta-modèle PADL*

En commençant par le niveau d'abstraction le plus haut, on trouve le *«Model»* qui représente un modèle de la solution d'un patron :

- chaque modèle *«Model»* est constitué de plusieurs entités *«IEntity»* correspondant aux classes et aux interfaces qui participent à la solution d'un patron ;
- chaque entité *«IEntity»* est composée de constituants d'entités *«IConstituentOfEntity»* qui, dans PADL correspondent à un ensemble de méthodes, de champs, de constructeurs, d'invocations de méthodes et de paramètres.

Suivant la sémantique du méta-modèle PADL, figure 3.1, chaque modèle est une instance de la classe *Model* et est constitué d'un ensemble *d'entités* : classes, interfaces et de constituants d'entités : méthodes, champs, etc. Cet ensemble d'entités et leurs constituants permettent d'obtenir, par composition, la description de la structure des patrons de conception du GOF. Cependant, la profondeur du méta-modèle PADL s'arrête au niveau des méthodes et des champs, ce qui ne donne pas la possibilité de décrire le contenu des méthodes, mais seulement leurs signatures.

3.2 Modélisation du Singleton avec le méta-modèle PADL

Ainsi, bien que le méta-modèle PADL permet de décrire la structure de la solution d'un patron et les interactions entre ses différents participants, le fait que l'on ne puisse pas décrire le contenu des méthodes peut fausser la sémantique de la solution. En effet, le corps d'une méthode, dans certains patrons de conception décrit le comportement du patron lui-même, dans le cas par exemple des patrons *State* et *Stratégie* [WS], dont les structures sont identiques mais le comportement différent.

Afin de mieux saisir les limites de la modélisation du méta-modèle PADL, nous allons présenter un exemple où nous modéliserons le patron de conception *Singleton* du GOF en suivant la sémantique du méta-modèle PADL. Nous avons choisi de modéliser le *Singleton* parce qu'il n'est composé que d'une seule classe et que la modélisation de sa solution se résume à la description de son comportement.

Singleton est un motif de conception orienté objet et fait partie du catalogue de Gamma et al. [GHJV94], (cf. page 127 du GOF). *Singleton* a pour objectif de

restreindre l'instanciation de sa classe statique *«Singleton»* à une seule instance. Pour ce faire, le *Singleton* adopte un comportement qui se résume au comportement de la méthode *«GetInstance()»* et du constructeur de la classe *«Singleton»*. La méthode *«GetInstance()»* est l'unique méthode de la classe *«Singleton»* et inclut dans son corps une structure conditionnelle qui se charge de vérifier si l'instance unique de la classe *«Singleton»* existe et si c'est le cas, de retourner cette instance, sinon d'en créer une autre. L'instance de la classe *«Singleton»* est représentée par un champ privé et statique. Le constructeur de classe *«Singleton»* est également privé, afin de s'assurer que cette dernière ne puisse être instanciée autrement que par la méthode contrôlée, *«GetInstance()»*.

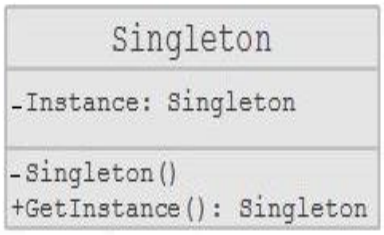

FIG. 3.2 – *Schéma UML représentant la structure du patron de conception Singleton*

En suivant la sémantique du méta-modèle PADL et la syntaxe du langage qu'il définit nous pouvons illustrer la modélisation du patron *Singleton* avec le méta-modèle PADL par le biais de la figure 3.3.

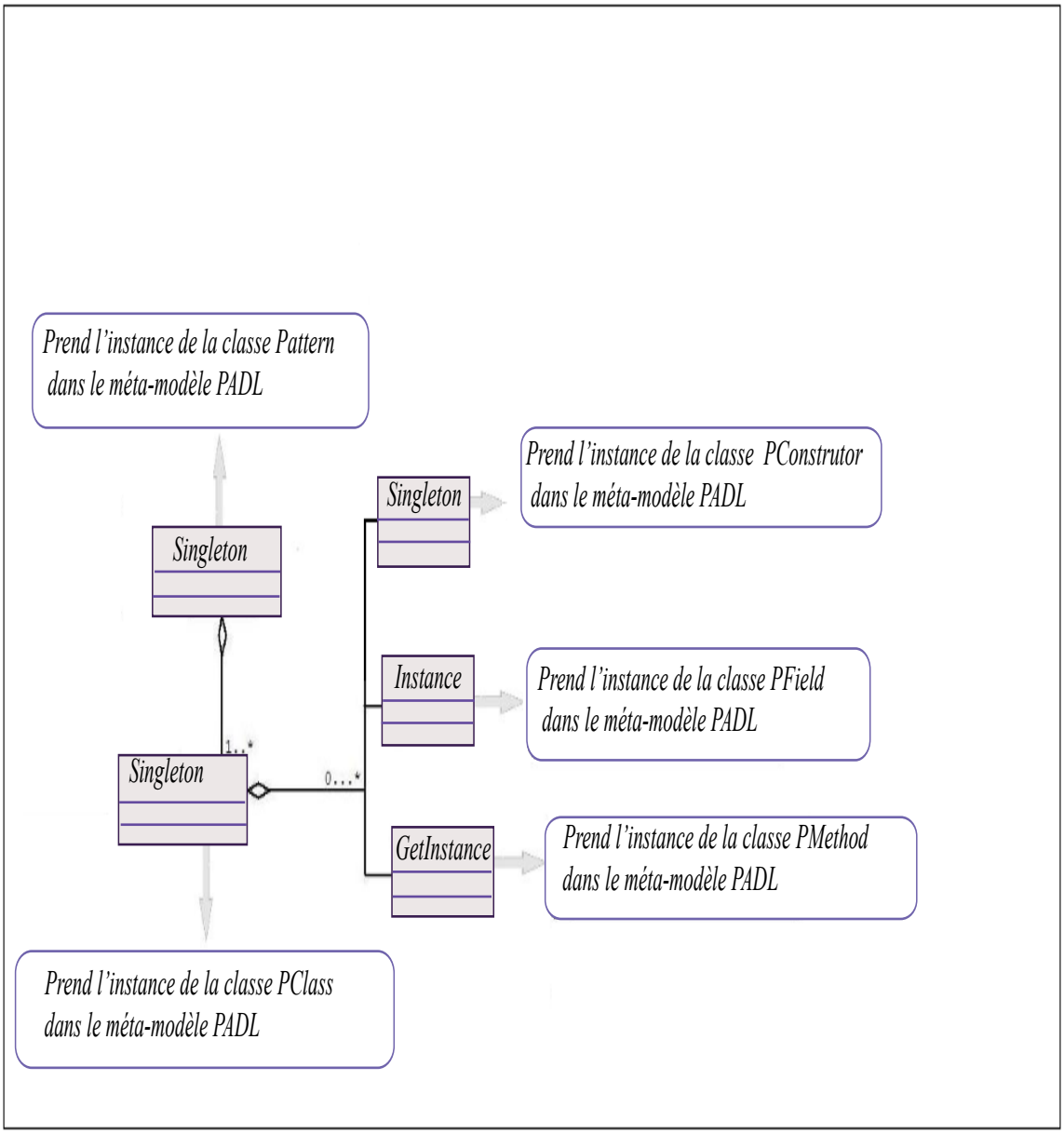

FIG. 3.3 – *Modélisation du patron de conception Singleton avec le méta-modèle PADL*

En commençant par le niveau d'abstraction le plus haut, on retrouve :

- *Singleton* qui est une sous-classe de la classe *Model* dans le méta-modèle PADL, figure 3.1. La classe *«Singleton»* joue le rôle de conteneur et représente la solution du patron *Singleton* ;
- La solution *Singleton* est composée d'une seule classe portant le même nom et qui est une instance de la classe *IClass* dans PADL ;
- la classe *«Singleton»* est composée de trois objets : (1) *«GetInstance()»*, qui représente l'unique méthode de la classe *«Singleton»* et qui constitue une instance de la classe *IMethod* dans PADL ; (2) *«Instance»*, qui représente l'unique instance de la classe *«Singleton»* et qui constitue une instance de la classe *IField* dans PADL ; (3) *«Singleton»*, qui représente le constructeur de la classe *«Singleton»* et qui constitue une instance de la classe *IConstructor* dans PADL.

Le modèle de la figure 3.3 ne donne pas une interprétation complète de la solution du patron *Singleton*, car sa structure pourrait correspondre à n'importe quelle autre classe. En effet, bien que la modélisation PADL du *Singleton* puisse décrire ses constituants et les relations entre eux, le comportement du *Singleton* correspondant au comportement de la méthode *«GetInstance()»*, ne peut être décrit dans le modèle.

3.3 Modifications apportées au méta-modèle PADL

Dans le but de rendre possible la modélisation complète du patron de conception *Singleton* et des autres patrons pour lesquels les comportements des méthodes sont importants, nous avons étendu le méta-modèle PADL en lui rajoutant des constituants décrivant les méthodes, pour obtenir un méta-modèle plus riche que nous avons nommé MIP. MIP est défini de manière indépendante, mais relié au méta-modèle PADL ; c'est-à-dire que l'utilisateur a la possibilité, soit d'utiliser uniquement le méta-modèle PADL pour décrire la structure des patrons de conception, soit d'utiliser le méta-modèle MIP étendant le méta-modèle PADL, figure 1.1, pour

modéliser des patrons de conception au niveau structurel et comportemental.

Dans la mesure où le méta-modèle MIP est dédié à la modélisation des patrons de conception orientés objet, nous avons restreint les composantes des méthodes ajoutées aux seuls éléments nécessaires à la modélisation des patrons de conception définis dans le catalogue du GoF.

La figure 3.4 reprend la structure de la figure 3.1 afin de présenter les constituants ajoutés aux méthodes :

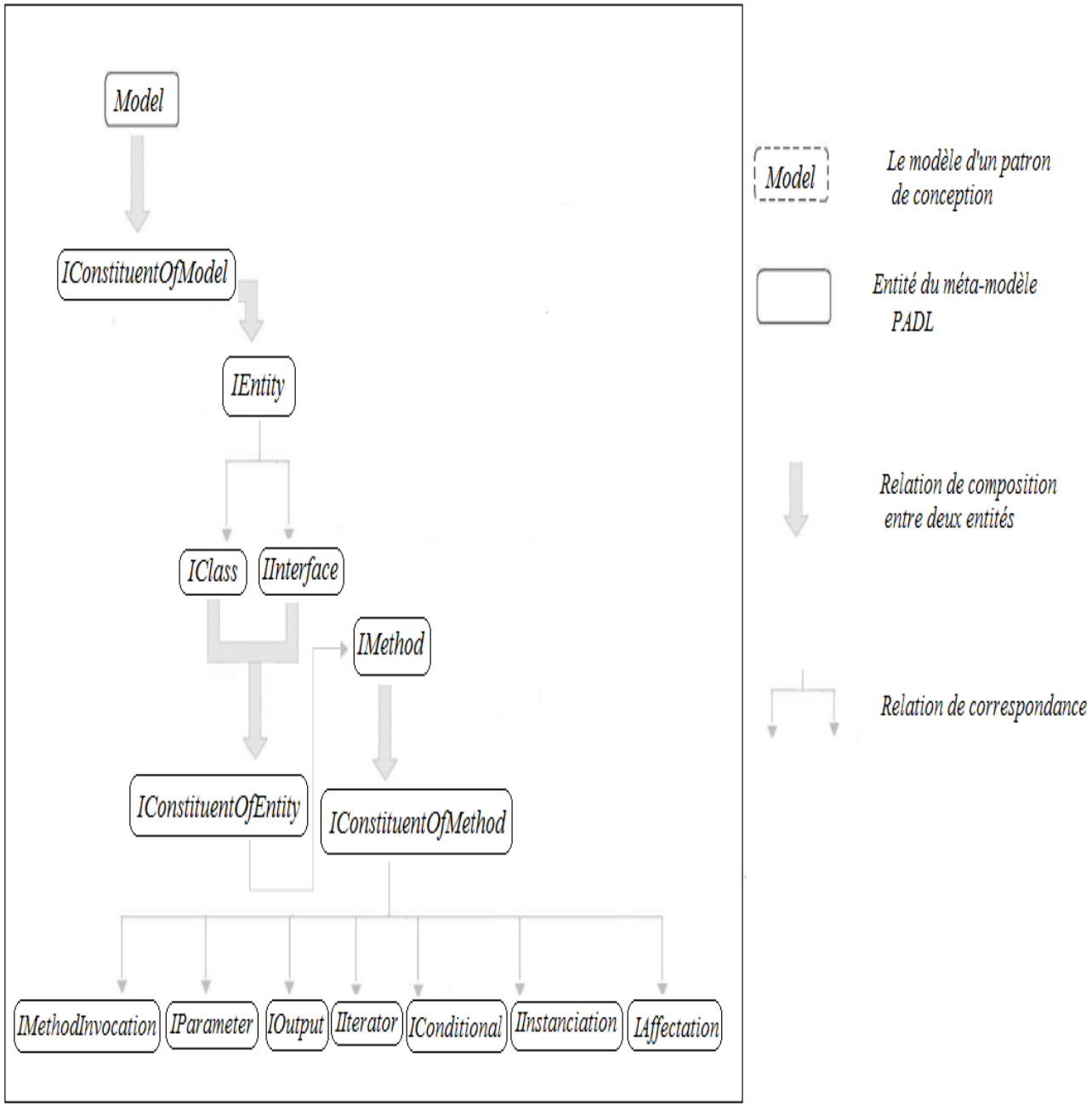

FIG. 3.4 – *Les constituants des méthodes ajoutés au méta-modèle PADL*

- *IInstanciation :* correspond à l'instanciation d'une classe ;
- *IAffectation :* correspond à l'affectation permettant d'attribuer une valeur à une variable, un champ, *IField* ou un paramètre, *IParameter* ;
- *IConditional :* correspond aux structures conditionnelles pour tester si une condition est vraie ou non ;
- *IIterator :* correspond aux structures itératives ou boucles pour répéter une instruction, soit un nombre précis de fois, soit tant qu'une condition donnée est vraie ;
- *IOutPut :* correspond à la méthode *Println* en Java ou *Console.WriteLine* en C♯. *IOutPut* prend en paramètre une chaîne de caractères et permet de l'afficher sur la sortie standard.

La figure 3.5 donne une présentation plus détaillée des différents constituants du méta-modèle MIP, leurs hiérarchies et les principales méthodes régissant leur utilisation.

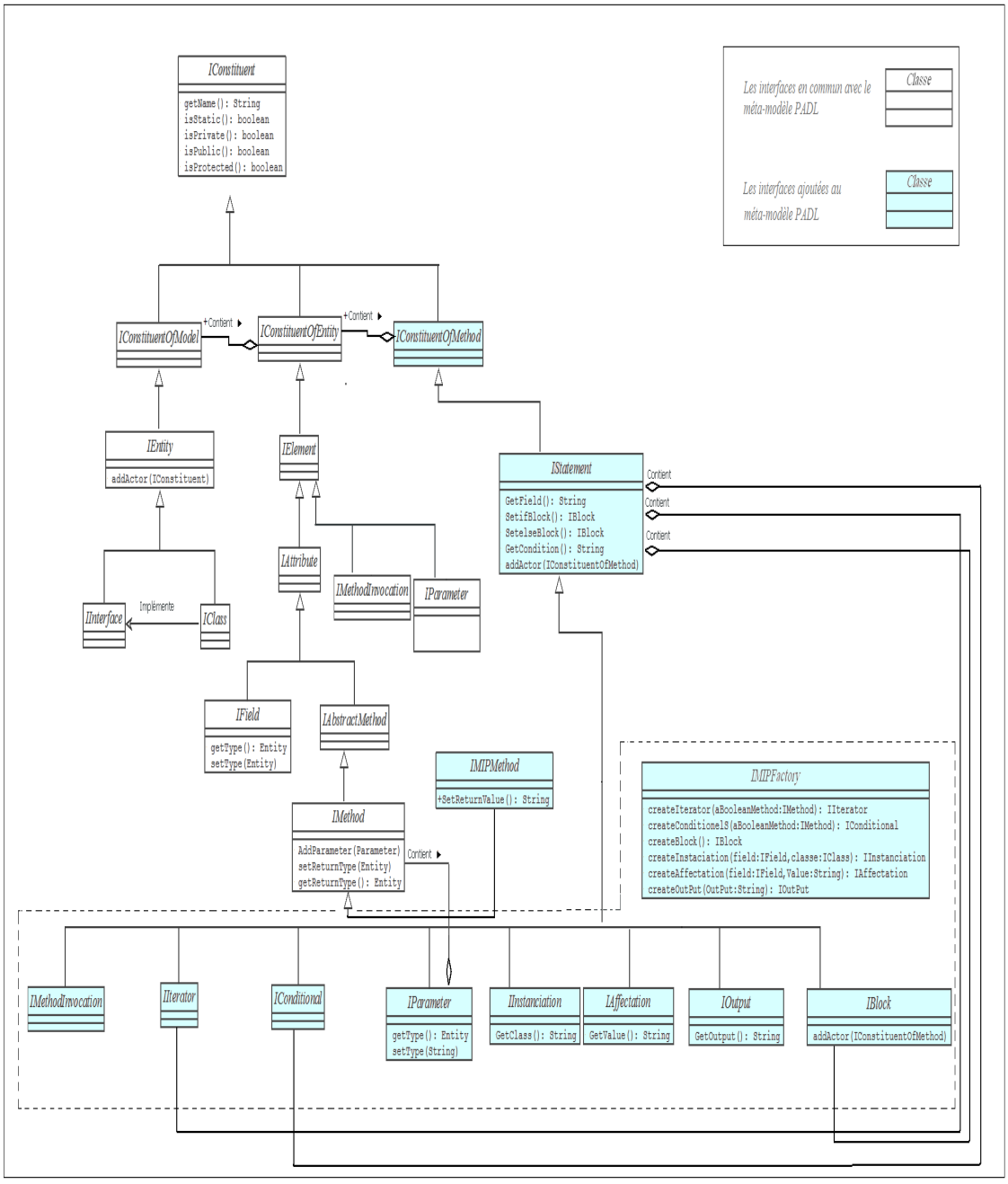

FIG. 3.5 – *Schéma UML du méta-modèle MIP*

La structure des classes sur la figure 3.5 représente la hiérarchie des interfaces dans notre méta-modèle, car l'implémentation du méta-modèle PADL ainsi que celle du MIP se décompose en deux parties : (1) les interfaces, qui sont structurées dans la hiérarchie suivant le principe de constituants présenté dans la figure 3.4 et (2) les classes d'implémentation leur correspondant, avec possiblement une hiérarchie différente dépendamment de nos besoins en factorisation de code. En commençant par le niveau d'abstraction le plus haut, on retrouve :

- l'interface *IConstituent* qui constitue l'une des principales classes du méta-modèle PADL. Cette classe implémente plusieurs méthodes utilisées par l'ensemble de ses sous-classes, parmi lesquelles, par exemple, *isProtected()*, *isPublic()*, *isPrivate()* , *isStatic()*, etc. Ces méthodes définissent les modificateurs d'accès qui sont des mots clés permettant de spécifier l'accessibilité d'un membre ou d'un type ;
- l'interface *IConstituentOfModel* peut contenir des instances de la classe *IConstituentOfEntity*, ainsi que les instances de toutes ses sous-classes. Par exemple une classe, instance de *IClass*, peut contenir un champ, instance de *IField*. Cette composition est exprimée par la méthode *addActor()* définie dans la classe *IEntity* ;
- l'interface *IMIPMethod*, représente les méthodes dans le méta-modèle MIP. *IMIPMethod* est une sous-classe de *IMethod* et permet d'étendre les fonctionnalités des méthodes du méta-modèle PADL, en définissant la méthode *returnValue()*, qui permet à une méthode de retourner une valeur. En effet, les méthodes n'avaient aucune implémentation dans le méta-modèle PADL, elles ne retourneraient alors aucune valeur ;
- l'interface *IStatement* est une sous-classe de la classe *IConstituentOfMethod*, c'est la classe mère de tous les constituants des méthodes ajoutées par MIP. Nous avons placé la classe *IStatement* à ce niveau d'abstraction parce qu'elle définit toutes les méthodes que ses sous-classes ont en commun. Par exemple, la méthode *addActor()* est utilisée par les classes *IBlock*, *IConditional* et *IIterator* et permet d'établir des compositions entre les objets de type *ICons-*

tituentOfMethod ;

– l'interface *IMIPFactory* englobe toutes les méthodes permettant la création des objets du méta-modèle MIP.

– les constituants des méthodes sont représentés par les classes suivantes :

1. *IBlock* permet de définir un block qui contient un ensemble d'objets de type *IConstituentOfMethod*. La création d'un objet de type *IBlock* est exprimée par la méthode *createBlock()* de la classe *IMIPFactory*.

   ```
   IBlock createBlock()
   ```

   Cette méthode ne prend aucun paramètre, car les objets *IBlock* ne représentent qu'un conteneur d'instruction.

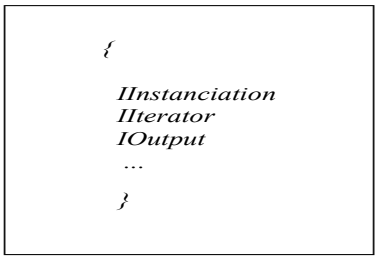

FIG. 3.6 – *La structure d'un block*

Ce block peut être ajouté par la suite à une structure conditionnelle ou itérative.

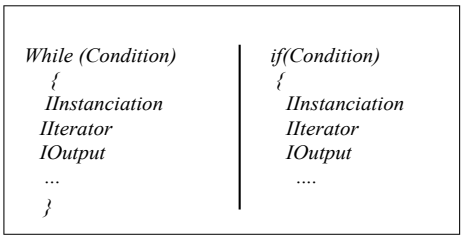

FIG. 3.7 – *La structure d'un block*

2. *IConditional* et *IIterator* représentent respectivement les structures conditionnelles et itératives. La création des objets de types *IIterator* et *IConditional* est exprimée par les méthodes *createConditional()* et *createIterator()* de la classe *IMIPFactory*.

```
IConditional createConditionel
(final IConstituent booleanMethod)
IIterator createIterator(final IMethod booleanMethod)
```

   Ces méthodes prennent en paramètres des méthodes retournant un boolean qui représente la condition d'arrêt.

3. *IInstanciation* permet d'exprimer l'instanciation et sa création est représentée par la méthode *createInstaciation()* de la classe *IMIPFactory*.

```
createInstaciation(final IField field, final IClass class)
```

   La méthode *createInstaciation()* prend en paramètres un objet de type *IField* qui représente l'instance et un objet de type *IClass* qui représente le type de l'instance.

4. *IAffectation* permet d'exprimer l'affectation et sa création est représentée par la méthode *createAffectation()* de la classe *IMIPFactory*. La méthode *createAffectation()* prend en paramètre un objet de type *IField* et une chaîne de caractères qui représente la valeur affectée à l'objet *IField*. Le fait que la valeur du champ *IField* soit représentée par une chaîne de caractères est une limite de notre méthode, car la valeur du champ entrée par l'utilisateur apparaîtra tel que dans le code généré. Cependant dans le méta-modèle PADL et MIP, les primitives (types de données) n'existent pas encore.

```
createAffectation(final IField field, final String value)
```

5. *IOutput* permet d'afficher une chaîne de caractères sur la sortie standard et sa création est exprimée par la méthode *createOutput()* qui prend en paramètres une chaîne de caractères représentant la chaîne à afficher.

```
public IOutPut createOutPut(final String Output)
```

6. Les paramètres, *IParameter* et les invocations de méthodes, *IMethodInvocation* existaient dans le méta-modèle PADL. Ils ont été déplacés de leur place initiale comme sous-type de *IConstituentOfEntity* à *IConstituentOfMethod*, afin de rassembler tous les constituants de méthodes en un tout cohérent. En revanche ce changement n'affecte pas la sémantique du méta-modèle PADL.

3.4 Modélisation du Singleton avec le méta-modèle MIP

Afin d'illustrer l'intérêt de l'extension du méta-modèle PADL, nous allons modéliser à nouveau la solution du patron de conception *Singleton*, en utilisant cette fois-ci, les constituants du MIP. La figure 3.8 récapitule la description du patron de conception *Singleton* avec le méta-modèle MIP :

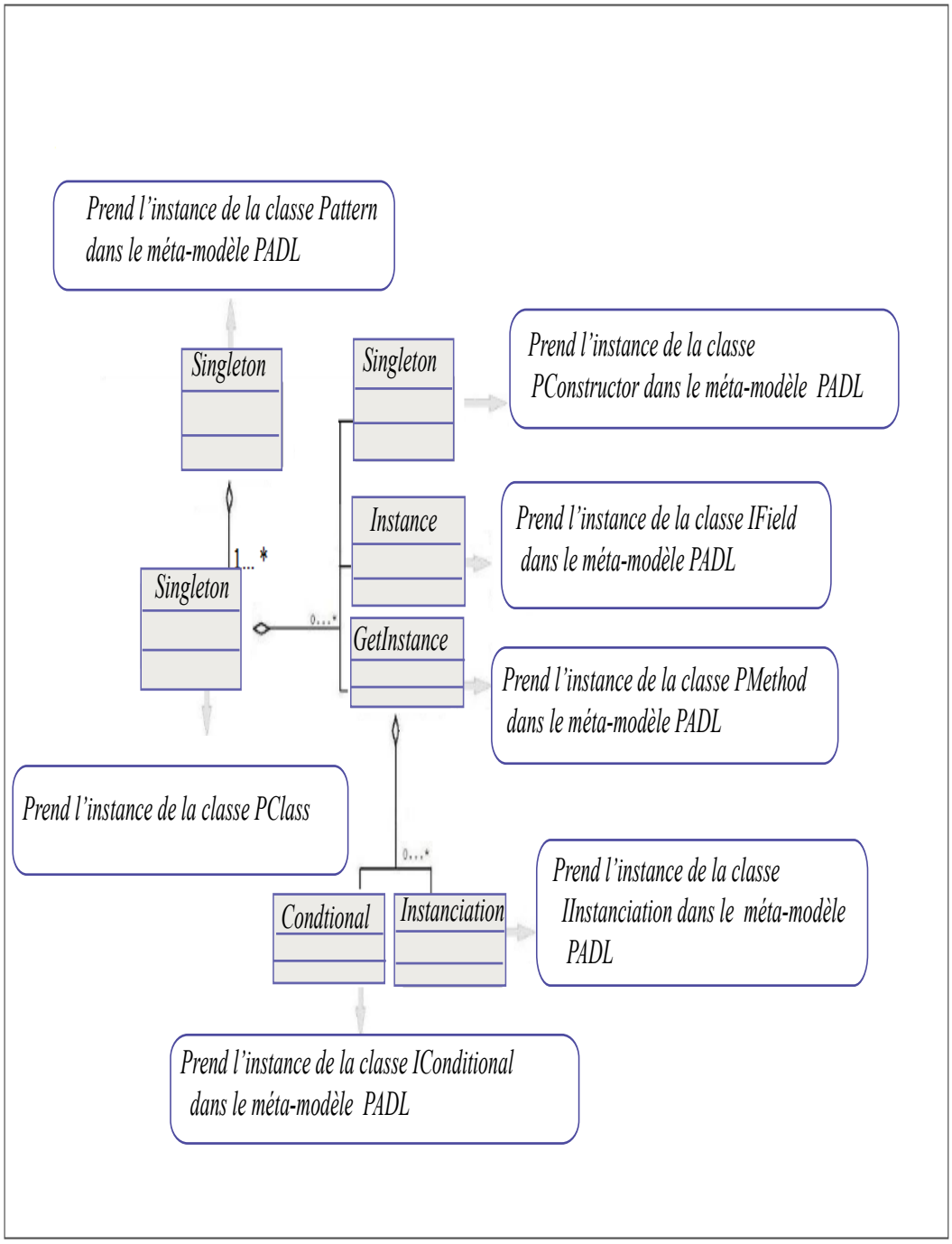

FIG. 3.8 – *Modélisation du patron de conception Singleton avec le méta-modèle MIP*

Nous avons ajouté une structure conditionnelle *IConditional* et une instanciation *IInstanciation* à la méthode *GetInstance()*. La structure conditionnelle se charge de vérifier si l'instance de la classe *«Singleton»* existe, si c'est le cas elle retourne cette instance. Sinon, elle en crée une en utilisant *IInstanciation* qui se charge de créer une instance de la classe *«Singleton»*.

À la différence du modèle de la figure 3.3, le modèle de la figure 3.8 donne une interprétation de la structure et du comportement du patron de conception *Singleton*.

3.5 Le méta-modèle MIP$\mathcal{C}\sharp$

Dans la mesure où le méta-modèle MIP est basé sur le langage Java, au moment où nous avons construit le générateur de code $\mathcal{C}\sharp$, il a fallu étendre MIP avec un méta-modèle MIP$\mathcal{C}\sharp$ en lui ajoutant tous les constituants qui différent entre les deux langages (Java et $\mathcal{C}\sharp$). Cette extension est necessaire, car la grammaire du langage $\mathcal{C}\sharp$ est plus riche que celle de Java. En effet, bien que les deux langages soient très proches, le langage $\mathcal{C}\sharp$ offre des constituants supplémentaires comme les propriétés, les indexeurs, etc. figure 3.10.

MIP$\mathcal{C}\sharp$ constitue une spécialisation du méta-modèle MIP ; c'est-à-dire que l'utilisateur à la possibilité soit, d'utiliser le méta-modèle MIP pour modéliser des patrons de conception avec des constituants Java et de générer le code Java leur correspondant, soit d'utiliser le méta-modèle *MIP*$\mathcal{C}\sharp$, (figure 1.1) afin de modéliser les patrons de conception en utilisant des constituants Java et les constituants propres à $\mathcal{C}\sharp$. La figure 3.9 représente la structure du méta-modèle *MIP*$\mathcal{C}\sharp$.

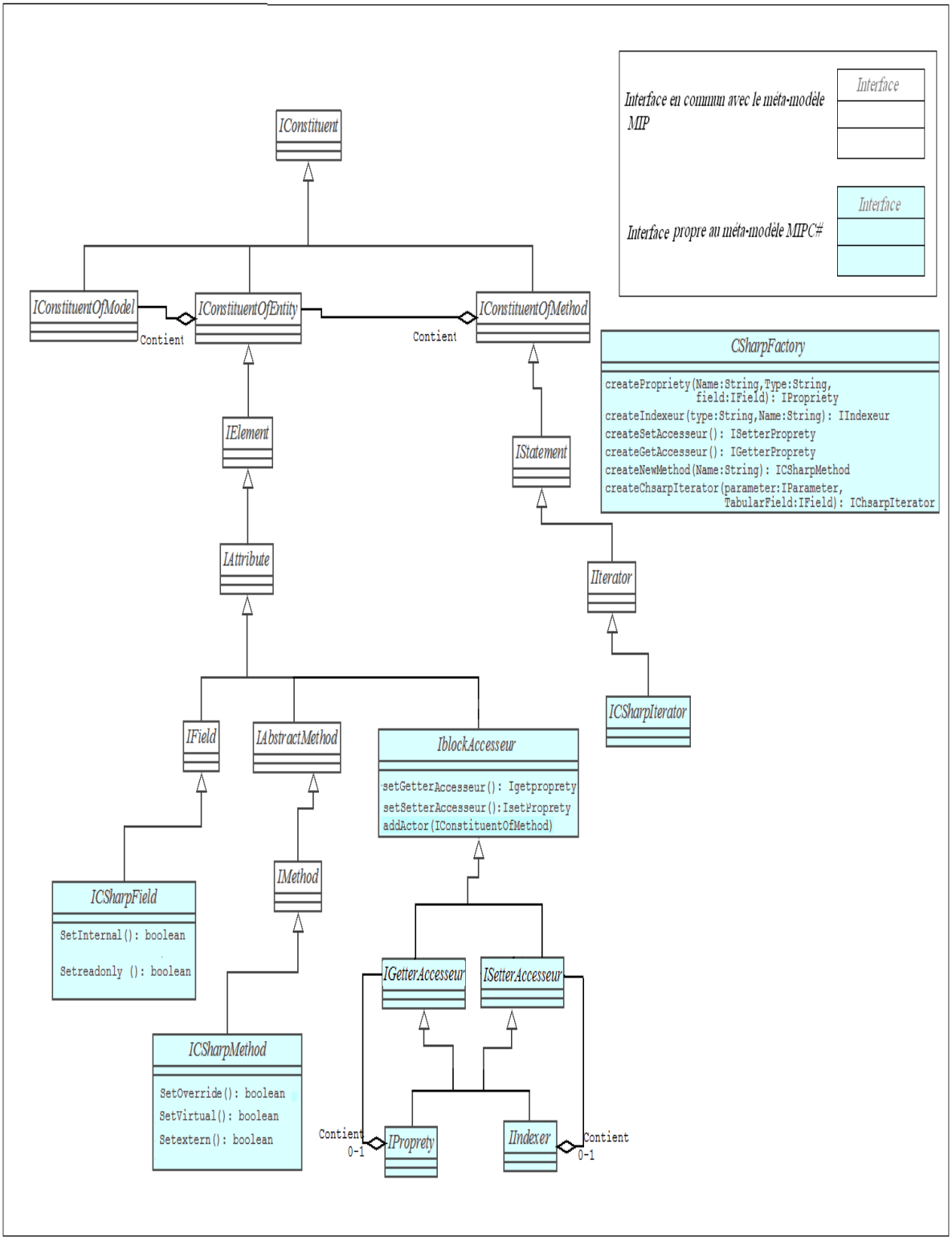

FIG. 3.9 – *Schéma UML du méta-modèle MIPC♯*

La structure des classes sur la figure 3.9, représente la hiérarchie des interfaces dans le méta-modèle MIP$\mathcal{C}\sharp$, car comme pour l'implémentation des méta-modèles MIP et PADL, l'implémentation de MIP$\mathcal{C}\sharp$ se décompose en deux parties : (1) les interfaces, suivant la hiérarchie présentée dans la figure 3.4 et (2) les classes d'implémentation :

- l'interface *ICSharpFactory* englobe toutes les méthodes permettant la création des constituants du méta-modèle MIP$\mathcal{C}\sharp$;
- l'interface *ICsharpField* représente les champs d'une classe et hérite de la classe *IField*, ainsi de toutes les méthodes correspondant aux constituants propres au langage Java et partagés par le langage $\mathcal{C}\sharp$. Nous définissons un type champ, *ICsharpField* propre au langage $\mathcal{C}\sharp$, car les modificateurs d'accès des champs entre Java et en $\mathcal{C}\sharp$ diffèrent. La création d'un objet de type *ICsharpField* est exprimée par la méthode *createField()*, de la classe *ICSharpFactory* ;

  ```
  ICSharpField createField(final String Name,final IClass type,
  final int cardinality)
  ```

  Cette méthode prend en paramètre : (1) une chaîne de caractères, *Name* qui représente le nom du champ ; (2) un paramètre, *type*, de type *IClass* représentant le type du champ ; (3) un paramètre de type entier, *cardinality* correspondant à la cardinalité du champ. Les modificateurs d'accès propres au langage $\mathcal{C}\sharp$ sont accessibles par les méthodes (i) *setInternal()*, qui limitent l'accès à un champ au membre de sa structure déclarative, ce qui permet à un groupe de composants de coopérer de manière privée sans s'exposer au reste du code de l'application ; (ii) *setReadOnly()* qui permet de définir un champ en lecture seule. Un champ en lecture seule peut uniquement se voir assigné une valeur pendant son initialisation ou dans un constructeur ;
- l'interface *ICsharpMethod* spécialise *IMIPMethod*, hérite de cette dernière et ainsi de toutes les méthodes correspondant aux constituants propres au langage Java et partagés par le langage $\mathcal{C}\sharp$. Les objets de type *ICsharpMethod* représentent les méthodes propres au langage $\mathcal{C}\sharp$ et qui diffèrent des méthodes

du langage Java par leurs modificateurs d'accès. Les nouveaux modificateurs d'accès sont accessibles par les méthodes : (1) *setOverride()* permet à une méthode de fournir une nouvelle implémentation à un membre hérité d'une classe de base ; (2) *setVirtual()* permet à une méthode d'être substituées dans une classe dérivée ; (3) *setExtern()* permet de déclarer une méthode implémentée en externe du code de sa déclaration. La création d'un objet de type *ICSharpMethod* est exprimée par la méthode *createnewMethod()*, de l'interface *ICSharpFactory* ;

```
ICSharpMethod createNewMethod(final String aName)
```

Cette méthode prend en paramètre une chaîne de caractères, *aName* représentant le nom de la méthode ;

– l'interface *IProprety* représente des membres de classes, les *Propriétés* qui permettent la lecture, l'écriture ou le calcul des valeurs de champs privés. Elles représentent des méthodes spéciales, *accesseurs* qui donnent l'accès aux données privées d'une classe.

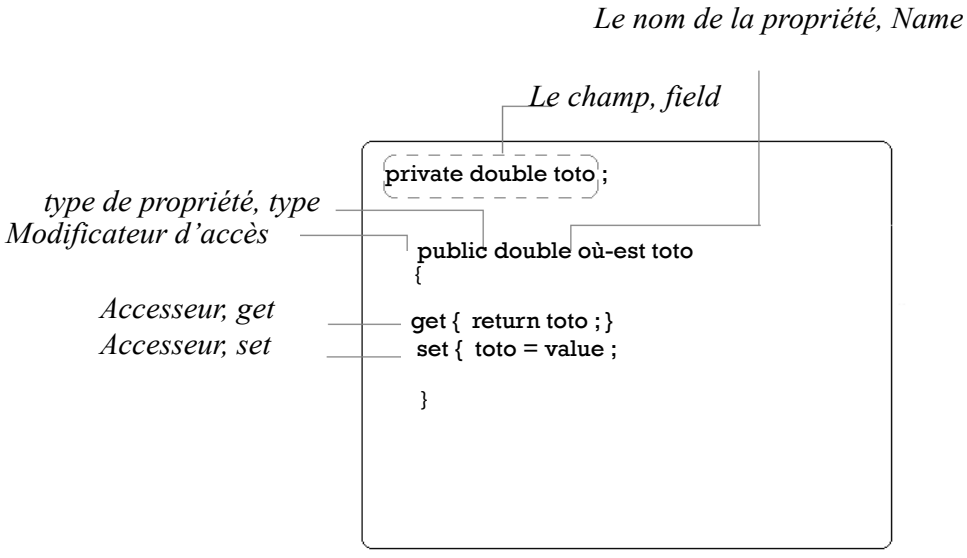

FIG. 3.10 – *La structure des propriétés*

La création d'un objet de type *IProprety* est définie par la méthode *createPropriety()*, de l'interface *ICSharpFactory*

```
IPropriety createPropriety(final String name, final String type,
final IField field)
```

Cette méthode prend en paramètre : (1) une chaîne de caractères, *name* correspondant au nom de la propriété ; (2) une chaîne de caractères, *type* correspondant au type de la propriété et (3) un objet de type *IField*, *field*, représentant le champ sur lequel la propriété va opérer. Une propriété peut être de trois types, en lecture seule, *get* ou en écriture seule, *set*, ou en lecture/écriture, (figure 3.10). Un accesseur de propriété *get* permet de retourner la valeur du champ, *field* et l'accesseur *set* est utilisé pour assigner une nouvelle valeur au champ. Le mot clé *value* sert à définir la valeur assignée par l'accesseur *set*.

– l'interface *IIndexer* représente des membres de classes, qui permettent d'indexer les instances d'une classe de la même façon que des tableaux. Les indexeurs présentent la même forme que les propriétés, *IProprety* et se déclarent de façon similaire à une propriété, dont le nom est *this*, (figure 3.11). Tout comme pour une propriété, la présence des accesseurs *get* et «*set*» indique si l'indexeur est en lecture seule, écriture seule ou les deux. À la différence des propriétés, les indexeurs prennent des paramètres.

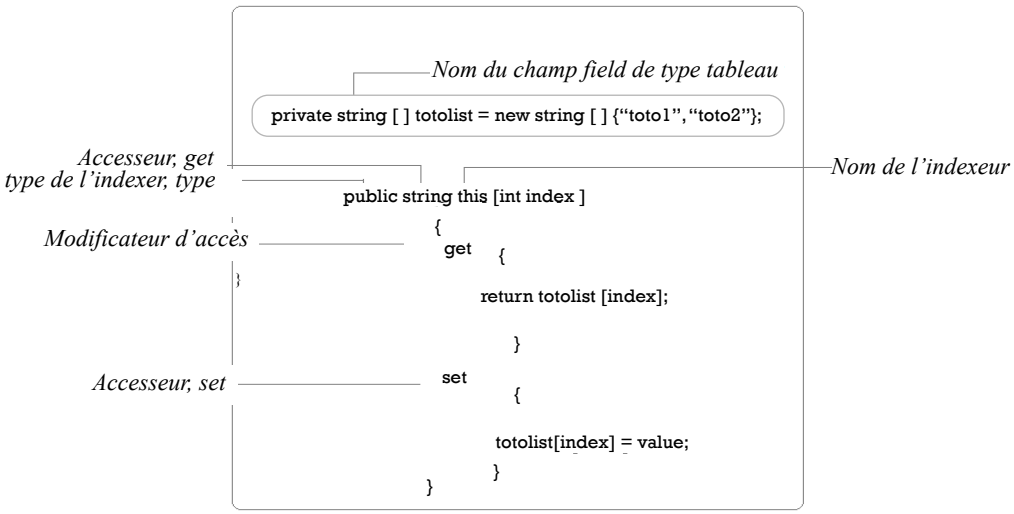

FIG. 3.11 – *La structure des indexers*

La création d'un objet de type *IIndexer* est représentée par la méthode *createIndexer()*, de l'interface *ICSharpFactory*

`IIndexeur createIndexer(final String type, final IField field)`

Cette méthode prend en paramètre, une chaîne de caractères, *type* correspondant au type de l'indexeur et un paramètre de type *IField*, représentant le champ sur lequel l'indexeur va opérer ;

– les accesseurs *get* et *set* sont représentés respectivement, par les interfaces *IGetterAccesseur* et *ISetterAccesseur*. La création des objets de type *ISetterAccesseur* et *IGetterAccesseur* est représentée par les méthodes *createSetterAccesseur* et *createGetterAccesseur* de l'interface *ICSharpFactory*

`ISetterAccesseur createSetterAccesseur( )`

`IGetterAccesseur createGetterAccesseur( )`

Ces méthodes ne prennent aucun paramètre. Un objet de type *ISetterAccesseur* ou *IGetterAccesseur* peut être ajouté aux propriétés et aux indexeurs ;

– l'interface *IBlockAccesseur* permet la réutilisation des méthodes partagées par

les indexeurs, les propriétés et les accesseurs *get* et *set*. Parmi ces méthodes on retrouve : (1) la méthode *addActor()*, qui prend en paramètre un objet de type *IConstituentOfMethod* et permet d'ajouter des objets aux accesseurs *get* et *set* ; (2) la méthode *setGetterAccesseur*, qui prend en paramètre un objet de type *IGetterAccesseur*, permet d'ajouter un accesseur de type *get* à une propriété ou à un indexeur ; (3) la méthode *setSetterAccesseur()*, qui prend en paramètre un objet de type *ISetterAccesseur* permet d'ajouter un accesseur de type *set* à une propriété, ou à un indexeur.

– l'interface *ICSharpIterator* représente un type de structure itérative, *foreach* propre au langage C♯ et qui sert à parcourir les éléments d'un tableau. La création des objets de type *ICSharpIterator* est représentée par la méthode *createChsarpIterator()*, de l'interface *ICSharpFactory*

```
IChsarpIterator createChsarpIterator(final IParameter parameter,
final IField Tabular)
```

Cette méthode prend en paramètre, un objet de type *IParameter*, qui représente le champ d'un tableau et un objet de type *IField* qui représente un tableau.

Dans la mesure où le méta-modèle MIPC♯ est dédié aux patrons de conception orientés objet, les constituants définis, dans ce méta-modèle, sont limités aux constituants nécessaires à l'implémentation des patrons de conception du GOF.

CHAPITRE 4

UTILISATION DU MÉTA-MODÈLE MIP

Après avoir étendu le méta-modèle PADL avec le méta-modèle MIP dans le but de permettre la modélisation des patrons de conception au niveau structurel et comportemental et étendu MIP en MIP$\mathcal{C}\sharp$ afin d'intégrer les constituants du langage $\mathcal{C}\sharp$, nous les avons mis en pratique en les utilisant pour la modélisation de 12 motifs de conception parmi les 23 du GOF (quatre patrons de chaque catégorie, tels que définit dans le GOF). Dans la mesure où chaque catégorie diffère de l'autre au niveau des rôles de patrons qu'elle inclut et leurs domaines d'application, nous avons ainsi vérifié le potentiel de description des méta-modèles MIP et MIP$\mathcal{C}\sharp$ pour chaque catégorie de patron.

Nous allons présenter dans cette section : (1) les étapes suivies pour la description d'un patron de conception avec le méta-modèle MIP, (les mêmes que celles du méta-modèle MIP$\mathcal{C}\sharp$) ; (2) un exemple d'utilisation, où nous allons modéliser le patron *Observer* du GOF avec MIP. Les modèles des onze autres patrons décrits avec le méta-modèle MIP et MIP$\mathcal{C}\sharp$ sont disponibles en l'annexe de ce mémoire, `http://ptidej.dyndns.org/Members/tagmouty/`.

4.1 Étape d'utilisation du méta-modèle MIP

Les étapes d'utilisation du méta-modèle MIP, figure 4.1 ne diffèrent pas énormément de celles du méta-modèle PADL, figure 2.2. En effet, la différence entre les deux utilisations réside principalement au niveau de l'étape 1 (point 1. Figure 2.2) qui consiste en le raffinement du méta-modèle PADL afin de lui rajouter des constituants structurels et comportementaux nécessaires à la modélisation d'un patron de conception. Dans la mesure où nous avons étendu le méta-modèle PADL, en lui rajoutant la plupart de ces constituants, l'utilisateur final n'a plus besoin de raffiner le modèle pour l'utiliser. Aussi en retirant cette étape, l'utilisateur peut passer di-

rectement à la modélisation de la solution. Il y a également une différence au niveau du système de génération de code, qui représentait la dernière étape d'utilisation du méta-modèle PADL (point 5. Figure 2.2) et qui n'offrait que la génération en Java. En effet, dans le modèle MIP, l'utilisateur a la possibilité de générer le code correspondant à son modèle dans deux langages différents : Java et $\mathcal{C}\sharp$. Le reste des distinctions entre les deux méthodes d'utilisation sont des différences d'appellation.

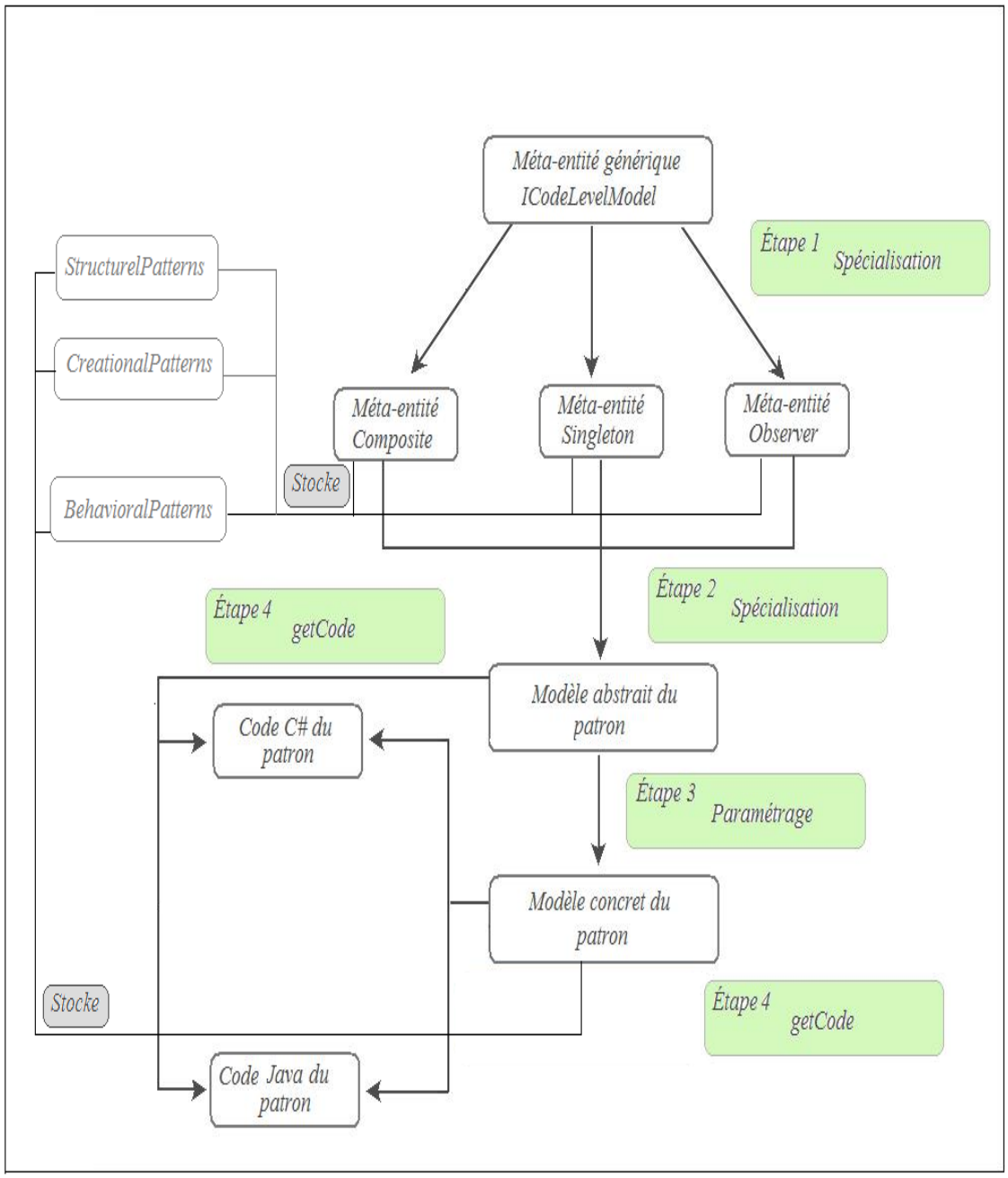

FIG. 4.1 – *Étapes d'utilisation des méta-modèles MIP et MIPC♯*

La figure 4.1 résume les étapes d'utilisation du méta-modèle MIP pour décrire un patron de conception.

1. La première étape de modélisation consiste en la définition de la méta-entité spécialisée qui correspond au patron portant le même nom (par exemple la méta-entité *Observer* pour le patron *Observer*) et qui est une extension de la méta-entité générique *IModel*.

   La méta-entité spécialisée va définir tous les constituants du patron ainsi que les règles régissant leurs interactions en suivant la sémantique du méta-modèle MIP.

2. La deuxième étape consiste en l'instanciation de la méta-entité spécialisée en un modèle abstrait du patron. Le modèle abstrait va contenir toute la logique du patron, qui se résume dans sa structure, les classes participantes et les collaborations entre elles, tels que définis dans le GOF. La définition des constituants du patron et leurs interactions dans le modèle abstrait se fait dans le but de reproduire la solution du patron, sans que cette dernière soit directement appliquée à un contexte particulier.

   Chaque modèle abstrait est directement relié à trois systèmes de génération de code : *SimpleJavaGenerator*, *CsharpGenerator* et *JavatoCsharpGenerator*. Tous les modèles abstraits peuvent être stockés dans des entrepôts de patrons, figure 4.1, schématisés par des classes en fonction du type de patron qu'il représente. Nous avons, ainsi trois types d'entrepôts (1) *BehavioralPattern*, pour les patrons comportementaux ; (2) *CreationalPatterns* pour les patrons créationnels et (3) *StructurelPattern*, pour les patrons structurels. Ces entrepôts correspondent à la classe *PatternRepository* dans le méta-modèle PADL.

3. La troisième étape comprend la définition du modèle concret du patron. Le modèle concret est obtenu par le paramétrage des constituants du modèle abstrait, définis à l'étape 2, telles que les méthodes, dans le but de l'appliquer à un contexte donné. Les modèles concrets peuvent être stockés dans les

même entrepôts de patron que les modèles abstraits. L'utilisateur aura ainsi la possibilité de naviguer dans ces entrepôts à la recherche d'une solution qui lui convient.

4. La dernière étape d'utilisation est la génération de code.

L'utilisateur a également la possibilité de passer directement à la génération de code à partir de l'étape 3 ; c'est-à-dire à partir du modèle abstrait du patron. Dans ce cas-ci, le code généré correspond au code prototypal représentant le patron.

4.2 Modélisation du patron Observer

Dans le but de présenter une mise en pratique du méta-modèle MIP, nous l'utiliserons pour modéliser le patron de conception *Observer* en suivant les étapes présentées à la partie 1 de cette section.

Observer fait partie du catalogue de GOF [GHJV94], (cf. page 139) et a pour but de définir les dépendances entre les objets, de telle manière que quand l'objet observable, le sujet, change d'état tous ses dépendants, les observateurs, sont informés et mis à jour automatiquement. Le patron *Observer* est ainsi constitué de quatre entités, figure 4.2.

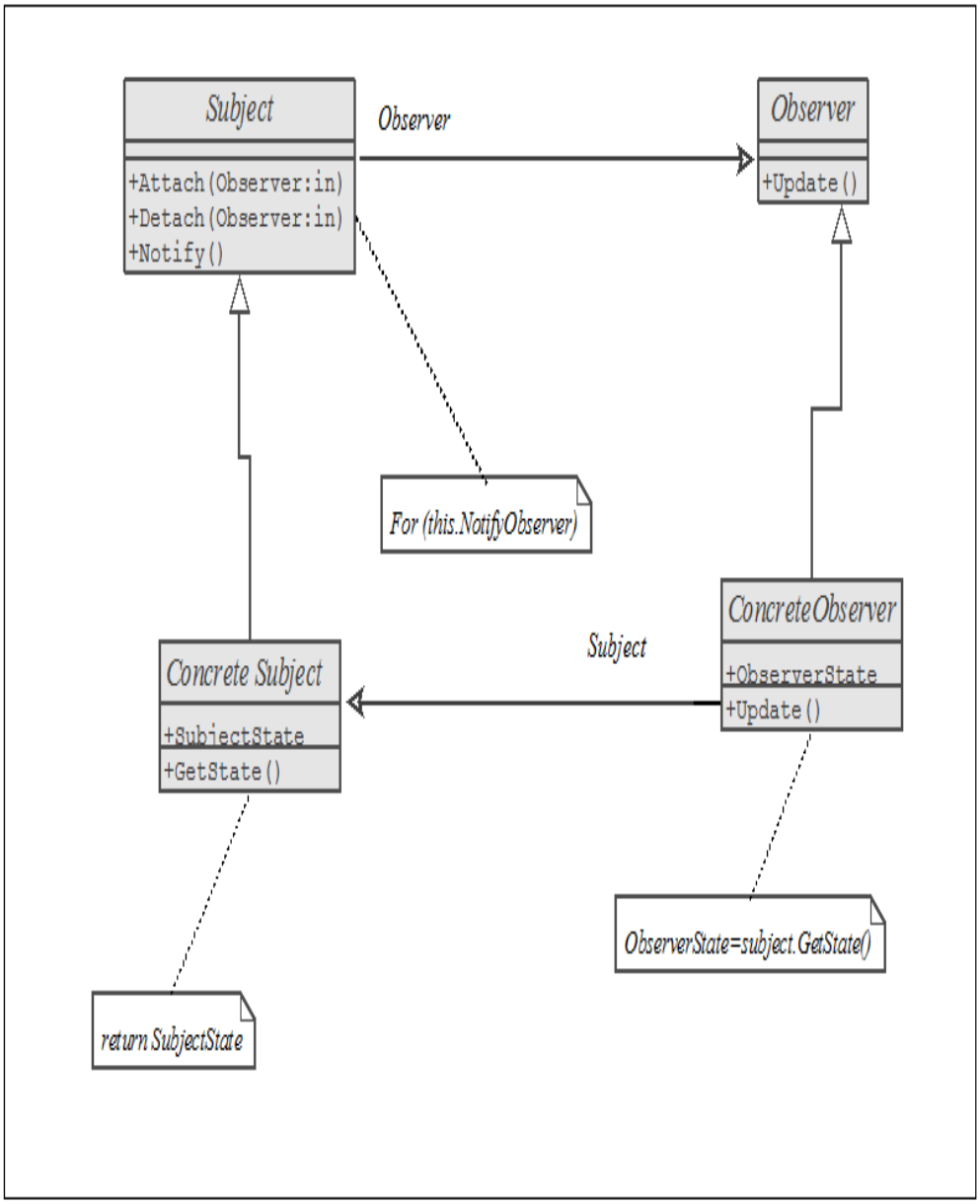

FIG. 4.2 – *Schéma UML du patron Observer*

Les entités du patron *Observer* sont :

1. *Subject* : joue le rôle d'interface pour attacher et détacher les objets *Observer*.
2. *ConcreteSubject* : stocke son état pour le *ConcreteObserver* et envoie une notification à ses observateurs quand son état change.
3. *Observer* : joue le rôle d'interface de mise à jour pour les objets qui doivent être notifiés au changement du sujet, *Subject*.
4. *ConcreteObserver* : maintient une référence à un objet *ConcreteSubject* et implémente l'interface *Observer* pour mettre à jour son état, afin que ce dernier soit compatible avec le sujet, *Subject*.

4.2.1 Étape 1

La première étape consiste à définir la méta-entité *Oberver* qui est une spécialisation de la méta-entité générique *Model*. La définition de la méta-entité *Observer* se fait en suivant la sémantique du méta-modèle MIP ; c'est-à-dire en énumérant ses différents constituants dans le but de reproduire la structure et le comportement du patron *Observer*.

La structure ciblée est celle définie par le digramme de classe du GOF et reprise dans le cadre de la figure 4.3, qui représente la description de la méta-entité *Observer* avec le méta-modèle MIP :

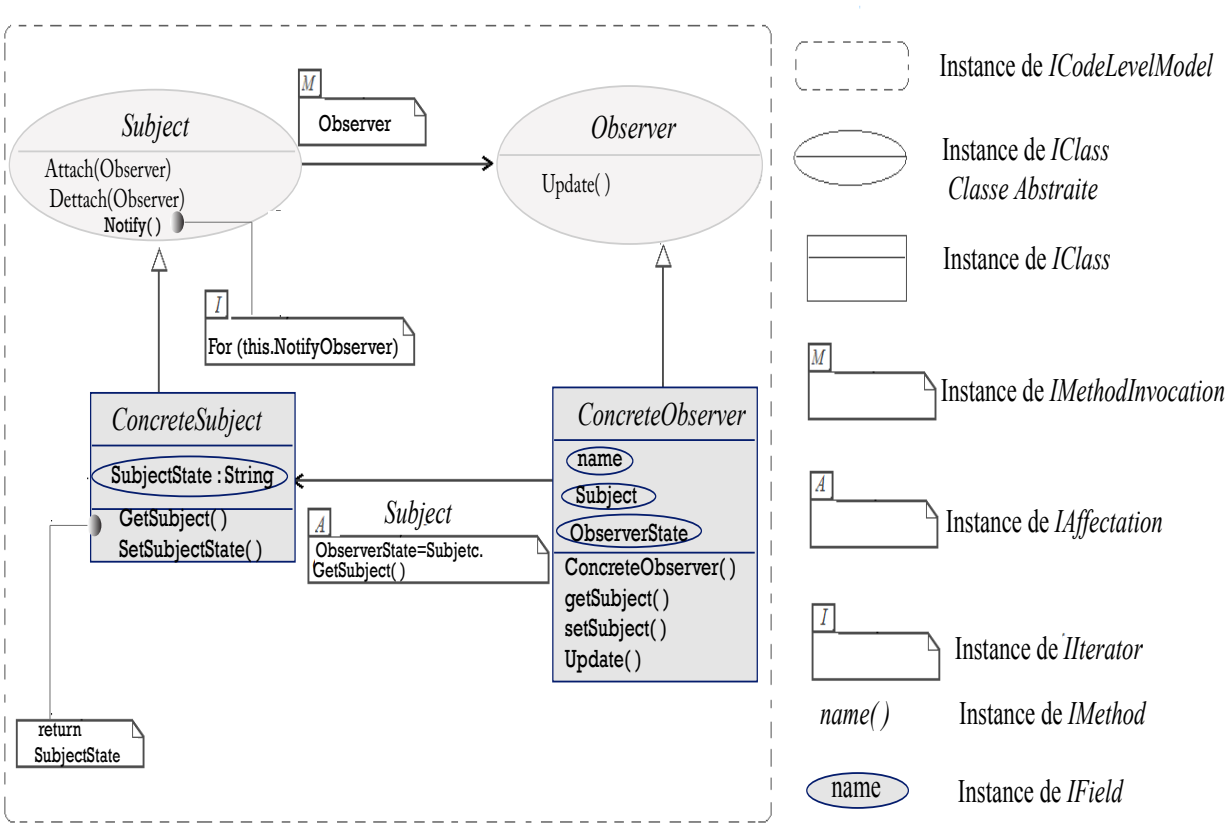

FIG. 4.3 – *Modélisation du patron Observer avec le méta-modèle MIP*

- la méta-entité *Observer* comprend les rubriques : *structures*, *participants* et *collaboration* tels que définis dans le GOF et représentant, en fait, la solution du patron ;
- le comportement ciblé par la description de la méta-entité *Observer* se résume aux méthodes permettant l'adaptation d'une instance de cette méta-entité à un contexte donné. Par exemple, la définition des méthodes *GetSubject* et *SetSubject* pour la classe *ConcreteObserver* ;
- la classe *Observer* est en relation avec la classe *Subject* ce qui permet à cette dernière de lui notifier un changement. Cette relation est représentée par une invocation de méthode dans le modèle du patron et prend ainsi l'instance de la classe *IMethodInvocation* dans le méta-modèle MIP, figure 3.5 ;
- la classe *ConcreteObserver* est en relation avec la classe *ConcreteSubject*. Cette relation permet à la classe *ConcreteObserver* d'obtenir l'état observé de la classe *ConcreteSubject*. La relation entre les deux classes est représentée par une affectation de variable dans le modèle et utilise une instance de la classe *IAffectation* dans MIP ;
- la méthode *Notify()* de la classe *Subject* comprend une structure itérative qui permet de vérifier à quel moment la classe *Observer* doit être notifiée d'un changement du sujet observé. La structure itérative prend alors l'instance de la classe *IIterator* dans le méta-modèle MIP.

4.2.2 Étape 2

Le modèle abstrait de la méta-entité *Observer* est obtenu par l'instanciation de la classe Java, *Observer* qui déclare la méta-entité *Observer*. Le tableau ci-dessous représente une déclaration partielle de la classe Java, *Observer*. Une version complète du modèle abstrait du patron *Observer* est présentée à l'annexe de ce mémoire, `http://ptidej.dyndns.org/Members/tagmouty/`.

Construction de la méta-entité Observer	*Commentaire*
ICodeLevelModel Observer = Factory.getInstance().createCodeLevelModel("Observer"); try {	La méta-entité *Observer*
IClass observer = Factory.getInstance().createClass("Observer");	Construction de la classe *Observer*
Observer.addActor(observer1);	Ajout de la classe *observer* à la méta-entité *Observer*

IMethod update = Factory.getInstance().createMethod("Update");
update.setAbstract(true);
observer.addActor(update);
	Création de la méthode abstraite *Update* qui fait partie de la classe *observer*

IMethodInvocation invoc =
Factory.getInstance().createMethodInvocation(2, 1, 1, subject);
invoc.addCallingField(observer);
invoc.setCalledMethod(update);
	Création d'une invocation de la méthode *Update*, afin de transmettre l'objet *observer* entre la classe *Observer* et *Subject* dans le cadre de la structure itérative *Iterator*

IIterator iterative = StatementFactory.getStateInstance().createIteratorS(NotifyB);	*Création de la structure itérative qui teste si les objets dépendant de l'observable doivent être notifiés*
iterative.addActor(invoc);	Ajout de l'invocation de méthode *Invoc* à la structure itérative *Iterative*
....}	

FIG. 4.4 – *La classe Java Observer*

4.2.3 Étape 3

Le modèle concret du patron est obtenu par paramétrage des constituants du modèle abstrait, afin de l'appliquer à un contexte particulier.

La figure 4.5 représente une spécialisation du patron *Observer* pour gérer un entrepôt de produits :

- la classe *Entrepôt* représente le Sujet, *Subject* à observer et la classe *Investisseur* représente l'observateur, *Observer*. La classe *Entrepôt* comprend les méthodes *AttachInves()* et *DettachInves()* qui permettent respectivement d'ajouter et de supprimer des investisseurs dans une liste qu'elles maintiennent. À chaque fois que l'entrepôt change d'état en modifiant, par exemple, un prix ou une marque de produit, il avise ses investisseurs de ce changement. Cet avis est fait par une invocation de méthodes entre les deux classes ;
- le déclenchement de l'invocation de méthode est décidé par une structure itérative appartenant à la méthode *Notify()* ;
- les classes *ARM* et *IBM* représentent des entrepôts et concrétisent la classe abstraite, *Entrepôt*. Ces classes maintiennent des relations d'affectation avec les classes *Investisseur1* et *Investisseur2*. Ces affectations permettent de mettre à jour l'état des investisseurs pour qu'il soit compatible avec l'état des entrepôts.

4.2.4 Étape 4

La génération se fait par l'appel de la méthode *getCode()* sur la classe Java *Observer*, représentant le modèle concret du patron. La génération de code peut se faire dans deux langages, dépendamment du type de générateur choisi. Dans l'exemple ci-dessous, nous avons choisi le générateur *CSharpGenerator*.

```
CsharpGenerator generator = new CsharpGenerator();
System.out.println(generator.getCode());
```

La méthode *getCode()* fournit l'équivalent en code de chaque entité du modèle visité par le générateur.

Le tableau ci-dessous représente partiellement le modèle concret correspondant à la figure 4.5 et le code $C\sharp$ lui correspondant, généré automatiquement. Une version complète du modèle concret du patron *Observer* est présentée en l'annexe de ce mémoire, `http://ptidej.dyndns.org/Members/tagmouty/`.

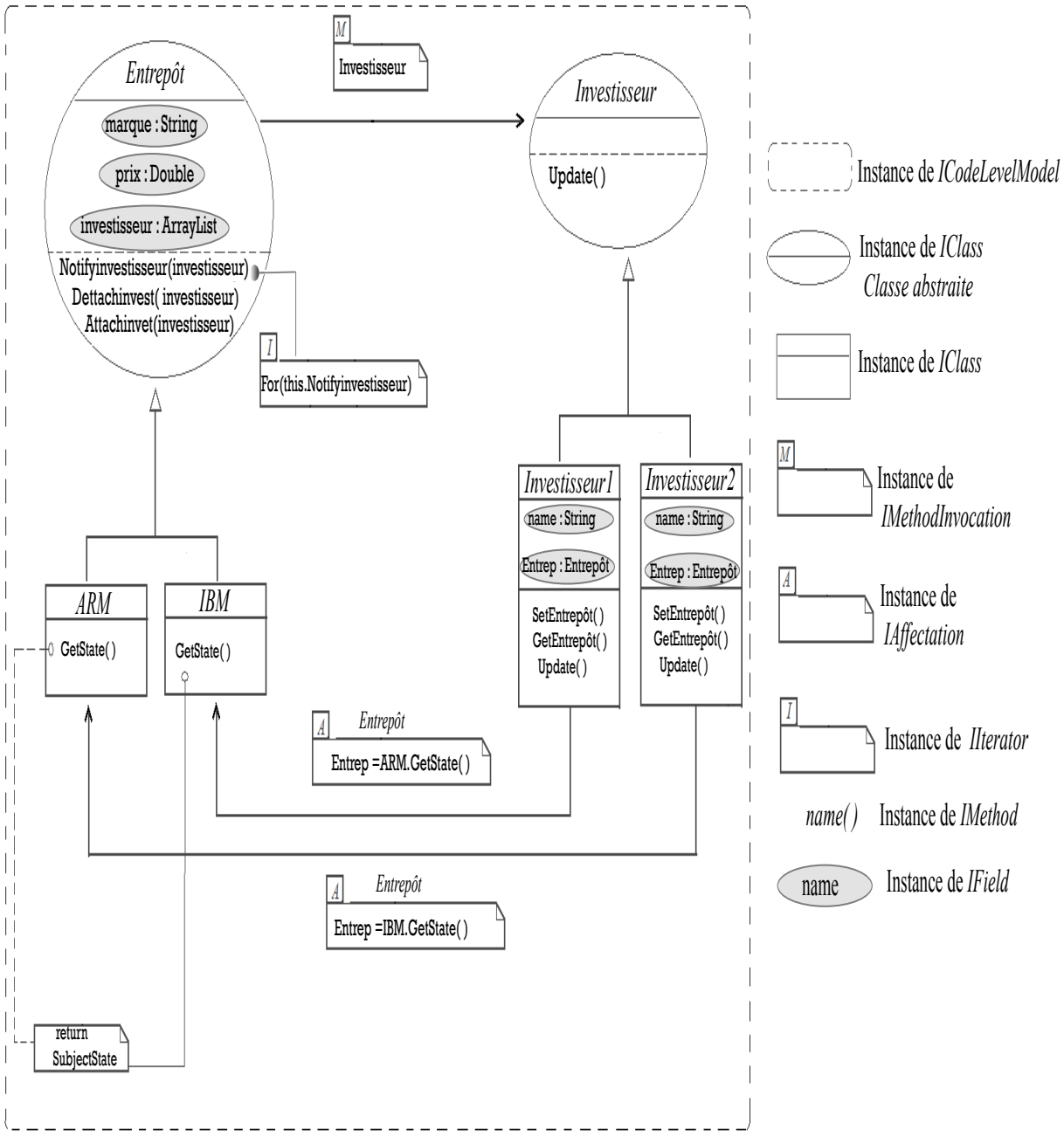

FIG. 4.5 – *Le modèle concret du patron Observer*

Création du modèle concret du patron Observer	*Commentaire*	*Code CSharp généré automatiquement*
IClass Entrepôt = Factory.getInstance().createClass("Entrepôt"); ConcreteModel.addActor(Entrepôt); Entrepôt.setAbstract(true);	- Crétation de la classe Entrepôt qui joue le rôle de la classe Subject dans le modèle du patron - Ajout de la classe Entrepôt au ConcreteModel qui est le modèle concret - Définir la classe Entrepôt comme une classe abstraite	public abstract class Entrepôt {
IField inves=Factory.getInstance().createField("investisseur", "Investisseur",2) Entrepôt.addActor(inves); inves.setPrivate(true);	- Création du champ investisseur - Ajout du champ investisseur à la classe Entrepôt - Définir le champ investisseur comme un champ privé	private Investisseur investisseur;
IMethod Notify = Factory.getInstance().createMethod("Notify"); Entrepôt.addActor(Notify);	- Création de la méthode Notify - Ajout de la méthode Notify à la classe Entrepôt	public void Notify() {

IBlock block = StatementFactory.getStateInstance().createBlock();
IIterator iterative=StatementFactory.getStateInstance().
createIteratorS(Notifyinvestisseur);
block.addActor(iterative);
Notify.addActor(block);
	- Création du block d'instruction qui va contenir la structure itérative
	- création de la structure itérative qui effectue un teste sur la méthode Notifyinvestisseur pour savoir à quel moment l'investisseur doit être notifé du changement de l'entrepôt
		while(this.Notifyinvestisseur( ))
		{

IMethodInvocation invoc =
Factory.getInstance().createMethodInvocation(2, 1, 1, Entrepôt);
invoc.addCallingField(inves);
invoc.setCalledMethod(update);
iterative.addActor(invoc);
	- Création de l'invocation de méthode Update sur le champ investisseur
	
	- Ajout de l'invocation de méthode à la structure itérative
		investisseur.Update();
		}
		}

IMethod Notifyinvestisseur = Foctory.getInstance().createMethod("Notifyinvestisseur"); Entrepôt.addActor(Notifyinvestisseur); Notifyinvestisseur.setReturnType("boolean");	- Création de la méthode Notifyinvestisseur sur lequelle le teste de la structure itérative sera effectué - Ajout de la méthode Notifyinvestisseur à la classe Entrepôt - Définir le type de la méthode Notifyinvestisseur comme une méthode boolean	public boolean Notifyinvestisseur( )

CHAPITRE 5

GÉNÉRATION AUTOMATIQUE DE CODE

Après avoir étendu le méta-modèle PADL en un méta-modèle plus riche, MIP et étendu ce dernier de nouveau en un méta-modèle MIP$\mathcal{C}\sharp$, nous avons utilisés ces derniers pour la modélisation de 12 motifs de conception du GOF. Chaque modèle construit, représente la solution proposée par un patron, afin de résoudre un problème de conception donné. Mais, au-delà de la description de cette solution, son implémentation est laissée aux utilisateurs. Or, l'implémentation d'un patron à partir de sa description n'est pas une tâche facile, surtout pour les concepteurs novices. Par contre, des utilisateurs plus expérimentés ne pourraient avoir aucune difficulté à traduire un modèle de patron en code. Mais cette opération demeure, tout de même répétitive puisqu'un changement de conception pourrait exiger une nouvelle implémentation du code.

Dans le but d'assurer l'implémentation des patrons de conception décrit avec le méta-modèle MIP et MIP$\mathcal{C}\sharp$, nous avons construit trois générateurs de code : *SimpleJavaGenerator*, *CSharpGenerator* et *JavatoCSharpGenerator*. Chaque générateur visite le modèle du patron pour générer le code correspondant à chacun de ses constituants dans les langages Java et $\mathcal{C}\sharp$.

Nous allons, ainsi présenter dans cette section : (1) l'intégration des générateurs de code dans notre environnement ; (2) le procédé de génération de code (3) les générateurs de code construits.

5.1 Intégration des générateurs de code dans notre environnement

La figure ci-dessous représente la structure des classes «générateurs» ainsi que les interactions entre elles :

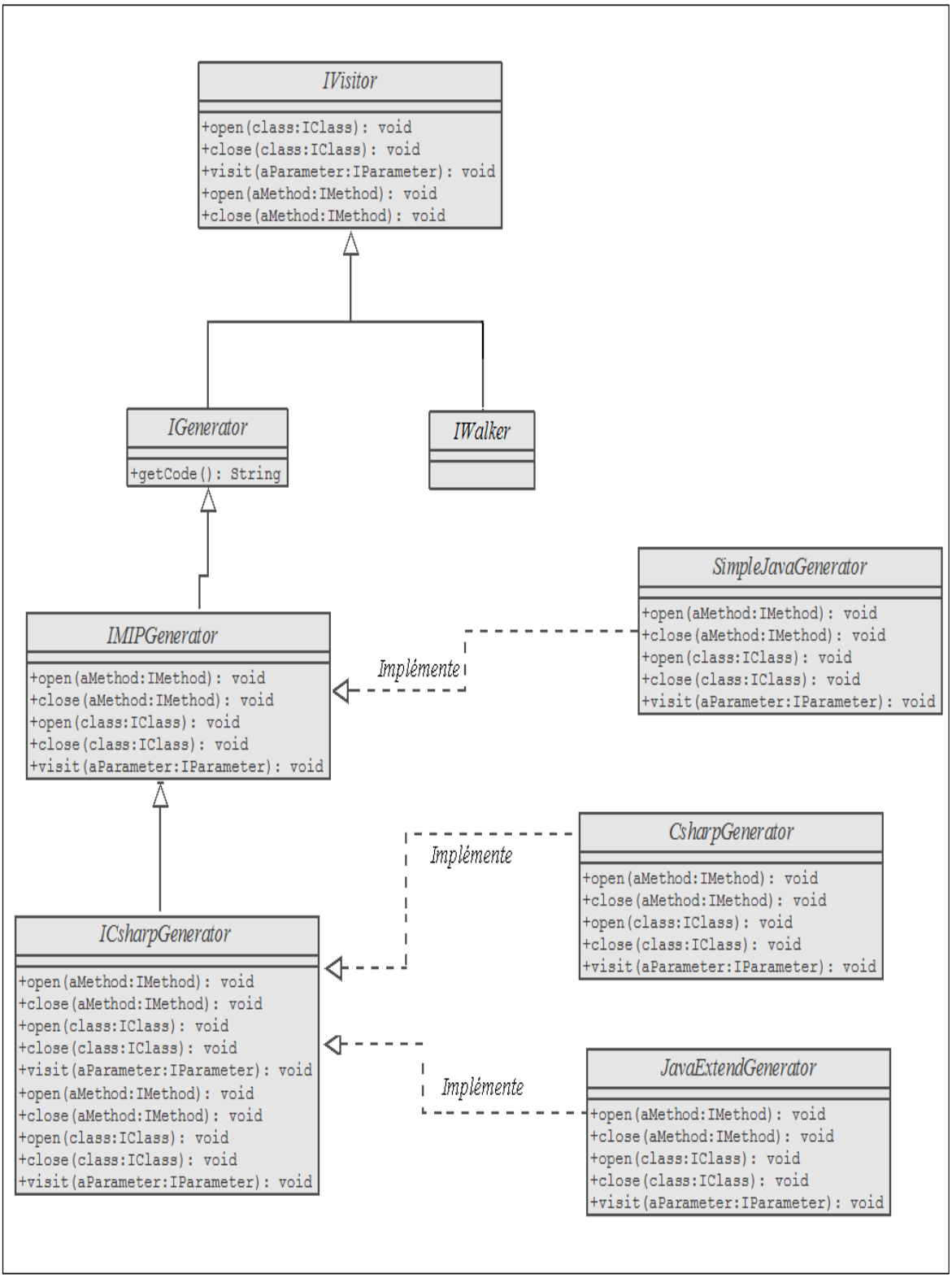

FIG. 5.1 – *Schéma UML représentant la structure des classes «générateurs»*

- les interfaces *IVisitor*, *IMIPGenerator* et *ICsharpGenerator* représentent les principales classes dans le système de génération, car elles définissent toutes les méthodes correspondant à l'ensemble des constituants d'un modèle de patron, tels que les classes, les constructeurs, les méthodes, etc. chacune de ces interfaces représente des constituants propres au projet dans lequel elles ont été définies, figure 1.1 ;
- l'interface *IVisitor* inclut tous les constituants propres à PADL et partagés par le méta-modèle MIP, telles que les classes, les méthodes, etc.
- l'interface *IMIPGenerator* inclut des méthodes de génération pour tous les constituants du méta-modèle MIP, telles que les structures itératives, les structures séléctives, etc. figure 3.5. *IMIPGenerator* hérite de *IVisitor* et englobe, ainsi toutes les méthodes définies dans l'interface *IVisitor* ;
- l'interface *ICsharpGenerator* définit tous les constituants propres au langage C♯ présentés dans le méta-modèle MIPC♯, figure 3.9. L'interface *ICsharpGenerator* hérite de la classe *IMIPGenerator* et englobe ainsi l'ensemble des méthodes définies dans les interfaces *IMIPGenerator* et *IVisitor* ;
- les méthodes définies dans les interfaces *IVisitor*, *IMIPGenerator* et *ICsharpGenerator* peuvent être divisées en trois catégories : (1) les méthodes *open()* et *close()* qui représentent tous les constituants d'un modèle qui peuvent contenir d'autres constituants, telles que les classes, les méthodes, les structures itératives, etc. (2) les méthodes *visit()* qui repésente tous les constituants d'un modèle qui ne peuvent pas en contenir d'autres, tels que les champs, les paramètres, les affectations, etc.
- les interfaces *IVisitor*, *IMIPGenerator* et *ICsharpGenerator* ne définissent que le nom des méthodes, leurs signatures et leur type de retour. En revanche l'implémentation concrète de chaque méthode est définie dans les classes qui les implémentent ; à savoir *CsharpGenerator*, *JavatoCsharpGenerator* et *SimplejavaGenerator* ;
- la classe *SimplejavaGenerator* implémente l'interface *IMIPGenerator*, c'est-à-dire qu'elle donne une implémentation concrète à toutes les méthodes définies

dans le cadre de cette interface ;

– Les classes *JavatoCSharpGenerator* et *CSharpGenerator* implémentent l'interface *ICSharpGenerator*. L'implémentation des méthodes diffère d'une classe à l'autre, dépendamment de la nature du code généré.

Chaque générateur prend en compte un ou plusieurs méta-modèles, c'est-à-dire qu'il donne une interprétation aux constituants du méta-modèle.

Générateur	*Le méta-modèle correspondant*
SimpleJavaGenerator	*MIP*
JavatoCsharpGenerator	*MIPC# et MIP*
CsharpGenerator	*MIPC# et MIP*

FIG. 5.2 – *La correspondance entre les méta-modèles et les générateurs de code*

La figure 5.2 représente les générateurs de code construits et les méta-modèles qu'ils prennent en compte.

5.2 Le procédé de la génération de code

La génération de code est réalisée en quatre étapes, figure 5.3 :

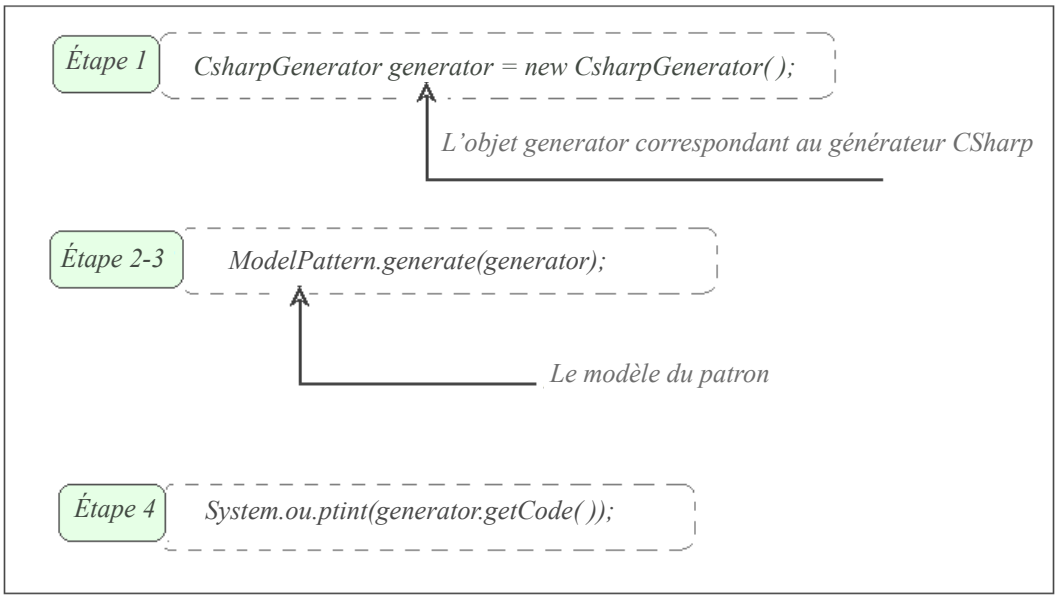

FIG. 5.3 – *La classe ICsharpGenerator*

- ***étape 1 :*** la création de l'objet *generator*, instance d'une classe correspondant à un générateur, par exemple, *CsharpGenerator* et qui représente un générateur *C♯* ;
- ***étape 2 :*** l'appel de la méthode *generate()* par l'objet *Model*, correspondant au modèle du patron ;
- ***étape 3 :*** l'objet *Model* qui délègue l'appel de la méthode *generate* à l'instance courante de la classe générateur, *generator* ;
- ***étape 4 :*** la classe *generator* qui génère le code source, en itérant sur chaque constituant dans le modèle de patron. Ce code peut être obtenu par la méthode *getCode()*.

5.3 Les générateurs de code

Chaque générateur visite le modèle du patron et itère sur chacun de ses constituants pour générer le code lui correspondant. Nous avons construit trois générateurs de code, indépendants mais reliés. Nous allons de ce qui suit examiner chacun de

ces générateurs.

5.3.1 Le générateur *Java*

Le premier générateur est un générateur de code Java. La construction de ce générateur nous a permis de vérifier la capacité de description du méta-modèle MIP, c'est-à-dire de vérifier si en nous basant sur le méta-modèle MIP, nous sommes capables de donner une interprétation à toutes les entités d'un patron de conception et de générer le code Java correspondant à chacune d'elles.

Dans la mesure où notre méta-modèle est basé sur Java, la construction du générateur Java n'a exigé aucune extension du méta-modèle MIP. Le générateur Java est représenté par la classe *SimpleJavaGenerator*, figure 5.6 et ne reconnaît, dans le modèle du patron visité, que les constituants propres au langage Java, figure 3.1.

La figure 5.4 représente une déclaration du modèle abstrait du patron *Singleton* et le code Java, correspondant à ses constituants, généré automatiquement. Le générateur Java prend en entrée un modèle de patron décrit par des constituants propres au langage Java et génère le code Java leur correspondant.

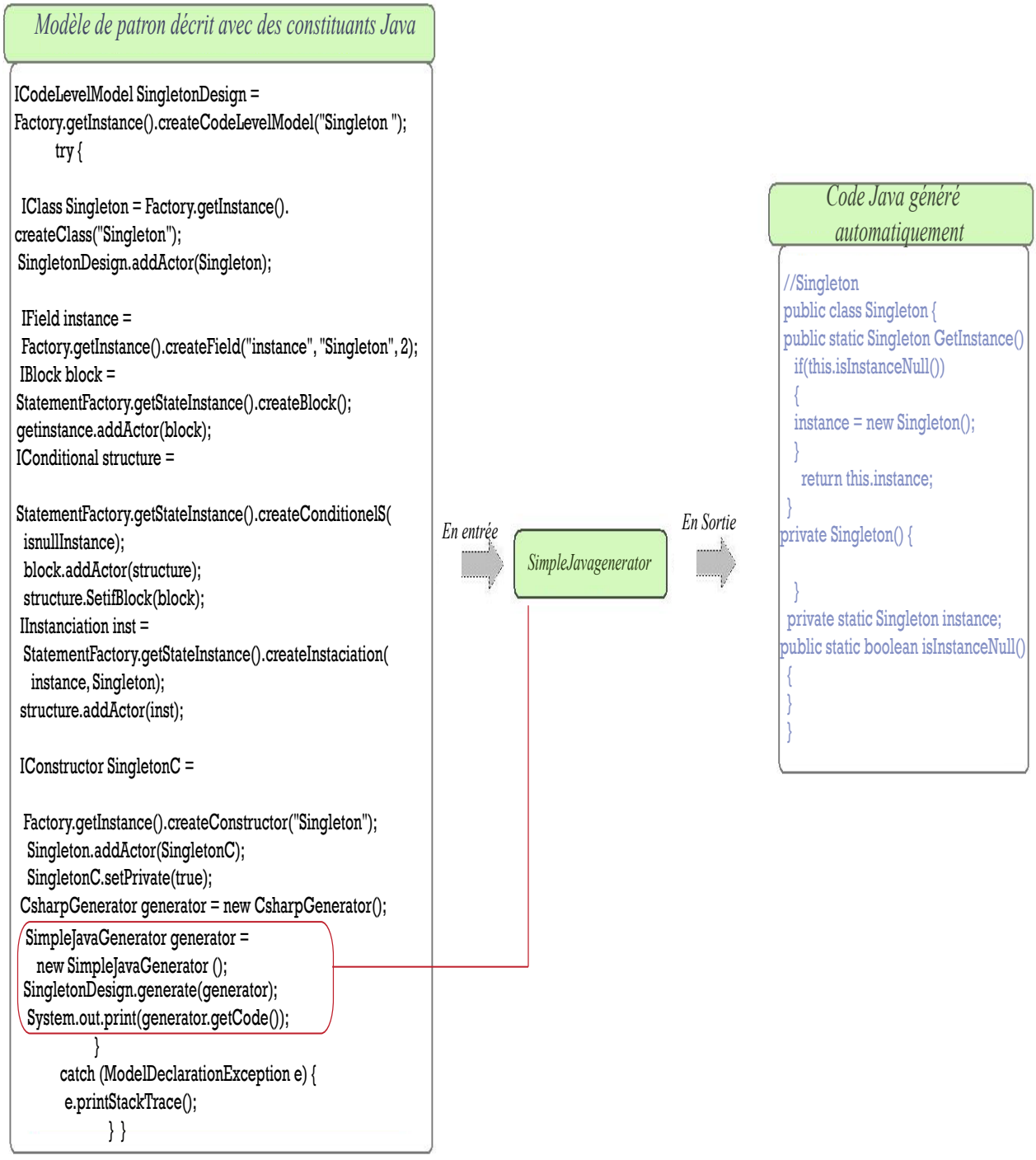

FIG. 5.4 – *Le générateur Java*

5.3.2 Le générateur C♯

Le deuxième générateur que nous avons construit est un générateur C♯. Dans la mesure où le méta-modèle MIP est basé sur le langage Java, il a fallu l'étendre en lui ajoutant une section C♯, contenant toutes les structures qui sont différentes entre les deux langages (Java et C♯).

Le générateur C♯ est représenté par la classe *CsharpGenerator*, figure 5.1 et reconnaît dans le modèle du patron visité les constituants C♯ et Java.

La figure 5.5, représente une déclaration du modèle abstrait du patron *Singleton* et le code C♯, correspondant à ses constituants, généré automatiquement. Le générateur C♯ prend en entrée un modèle avec des constituants C♯ pour générer du code C♯ en sortie.

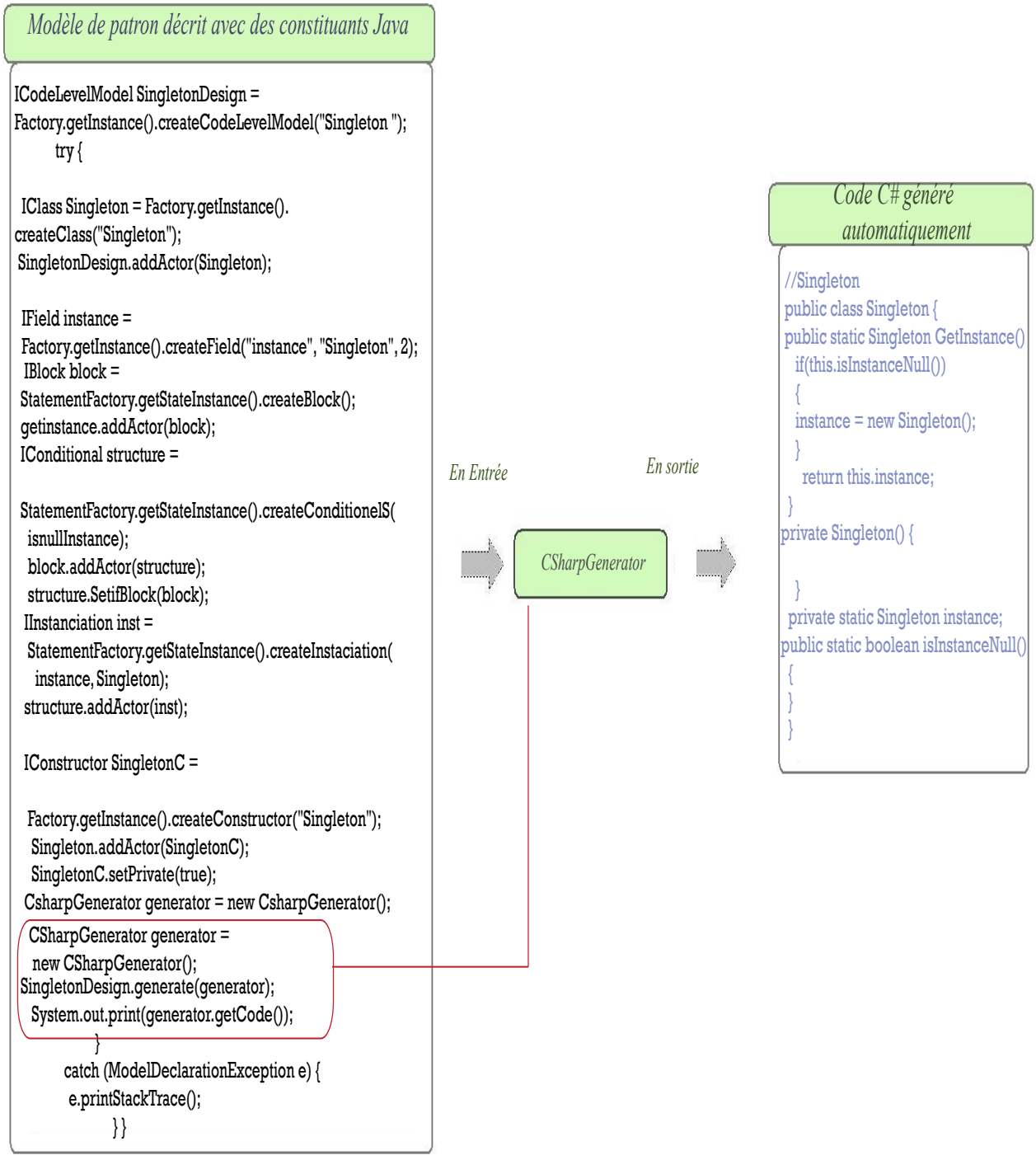

FIG. 5.5 – *Le générateur C♯*

En nous basant sur le générateur C♯ nous avons généré le code correspondant aux douze patrons de conception modélisés avec MIP et MIPC♯, à l'annexe de ce mémoire `http://ptidej.dyndns.org/Members/tagmouty/`.

5.3.3 Le générateur *JavatoCsharpGenerator*

Le générateur *JavatoCsharpGenerator*, est une extension des deux générateurs précédents. Il parvient à reconnaître dans le modèle du patron les constituants Java et C♯. Il est représenté par la classe *JavatoCSharpGenerator*, figure 5.6. La construction de ce générateur est nécessaire, car tous les générateurs doivent visiter le même modèle de patron et donner l'équivalent en Java et C♯ à tous ses constituants, que ce soit des constituants propres au langage Java ou ceux relatifs au langage C♯.

Ainsi, les trois générateurs ne permettent de générer que du code Java et C♯ dans le cadre du méta-modèle MIPC♯. La question que vous devez vous poser, alors est la suivante : «*Pourquoi utiliser trois générateurs, pour deux types de codes générés ?*». La raison à cela réside dans la différence entre les langages Java et C♯. En effet, les deux langages sont des langages orientés objet et leurs syntaxes sont très proches. Cependant, la grammaire du langage C♯ est plus riche que celle de Java. La grammaire C♯ comporte, en effet, toutes les entités Java, plus d'autres entités propres au langage C♯.

En ajoutant au méta-modèle MIP des entités du langage C♯, figure 3.9, un modèle de patron décrit avec MIPC♯ peut comporter des constituants Java et C♯. Le générateur C♯ peut ainsi interpréter l'ensemble de ses constituants. Cependant, le générateur Java peut faire face dans le modèle, à des constituants propres au langage C♯. Devant cette situation, nous avons deux solutions possibles : (1) le générateur Java ignore tous les constituants C♯ et ne donne l'équivalent en Java, qu'aux constituants Java. Cette solution est implémentée par le générateur *SimpleJavaGenerator* ; (2) le générateur *JavatoCsharpGenerator*, se résume à une mise en correspondance basée sur les rôles entre les composantes C♯ et Java. En d'autres termes, le générateur *JavatoCSharpGenerator* donne une équivalence en Java à

tous les constituants C♯ visités, selon leurs rôles dans le modèle du patron.

La figure 5.6 résume la mise en correspondance entre les principales structures C♯, ajoutées au méta-modèle MIP et les structures Java.

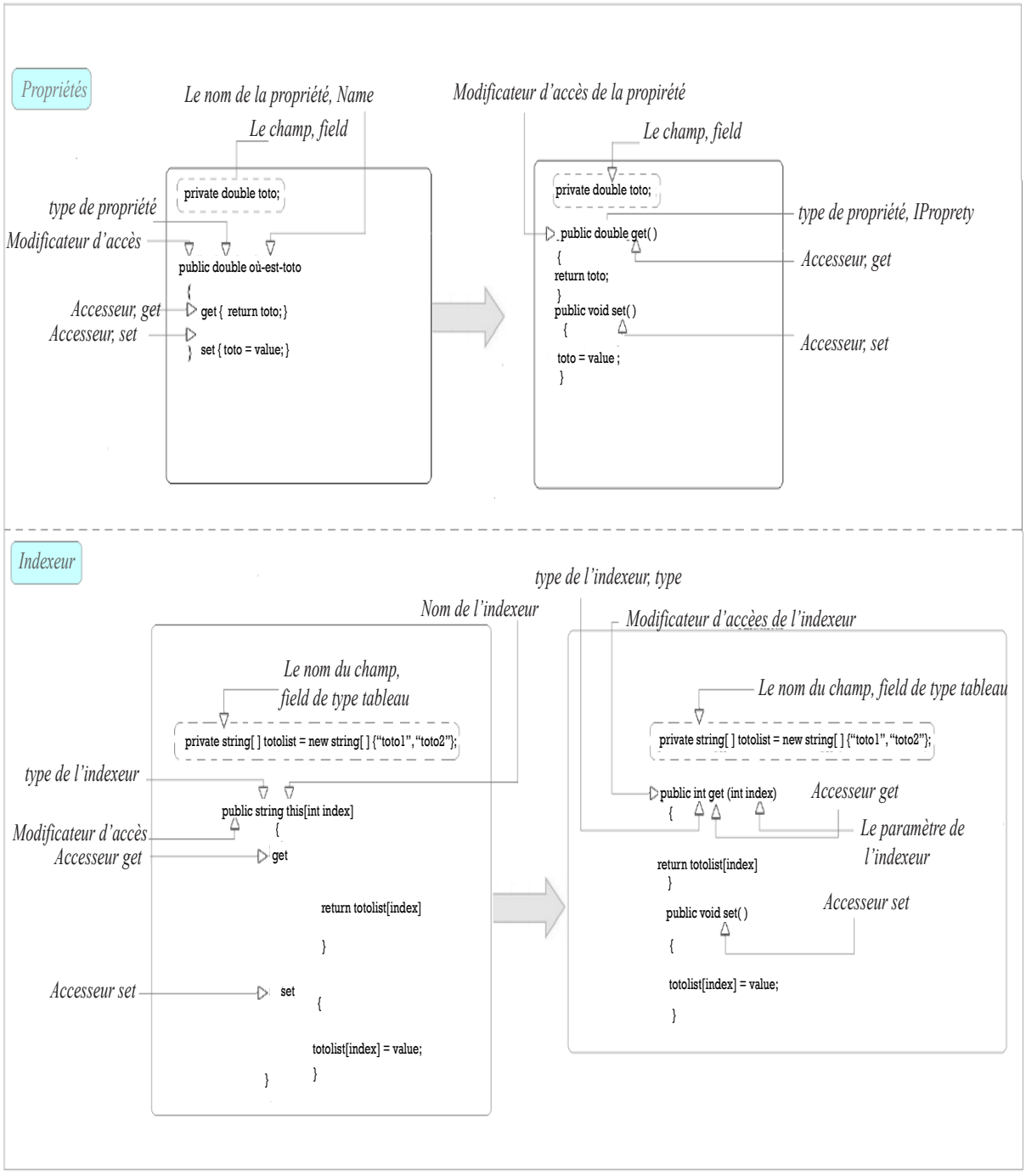

FIG. 5.6 – *génération des constituants C♯ en Java*

- dans la mesure où les propriétés et les indexeurs en C♯ sont des méthodes spéciales, *accesseur*, nous les traduisons en Java par des méthodes avec des noms particuliers ;
- les indexeurs C♯ sont traduit en Java par une méthode du même type que l'indexeur et a pour signature le paramètre de l'indexeur ;
- l'accesseur *get* est traduit par une méthode du même type que sa propriété ou son indexeur et qui retourne le champ sur lequel cette propriété ou cet indexeur opère ;
- l'accesseur *set* est traduit par une méthode qui ne retourne aucune valeur mais qui affecte une valeur *«value»* au champ de la propriété ou de l'indexeur. La valeur de *«value»* sera défini dans le programme.

La figure 5.7, représente la déclaration partielle du modèle abstrait du patron *Memento* à l'aide de constituants Java et C♯ ainsi que le code Java correspondant à ses constituants généré automatiquement.

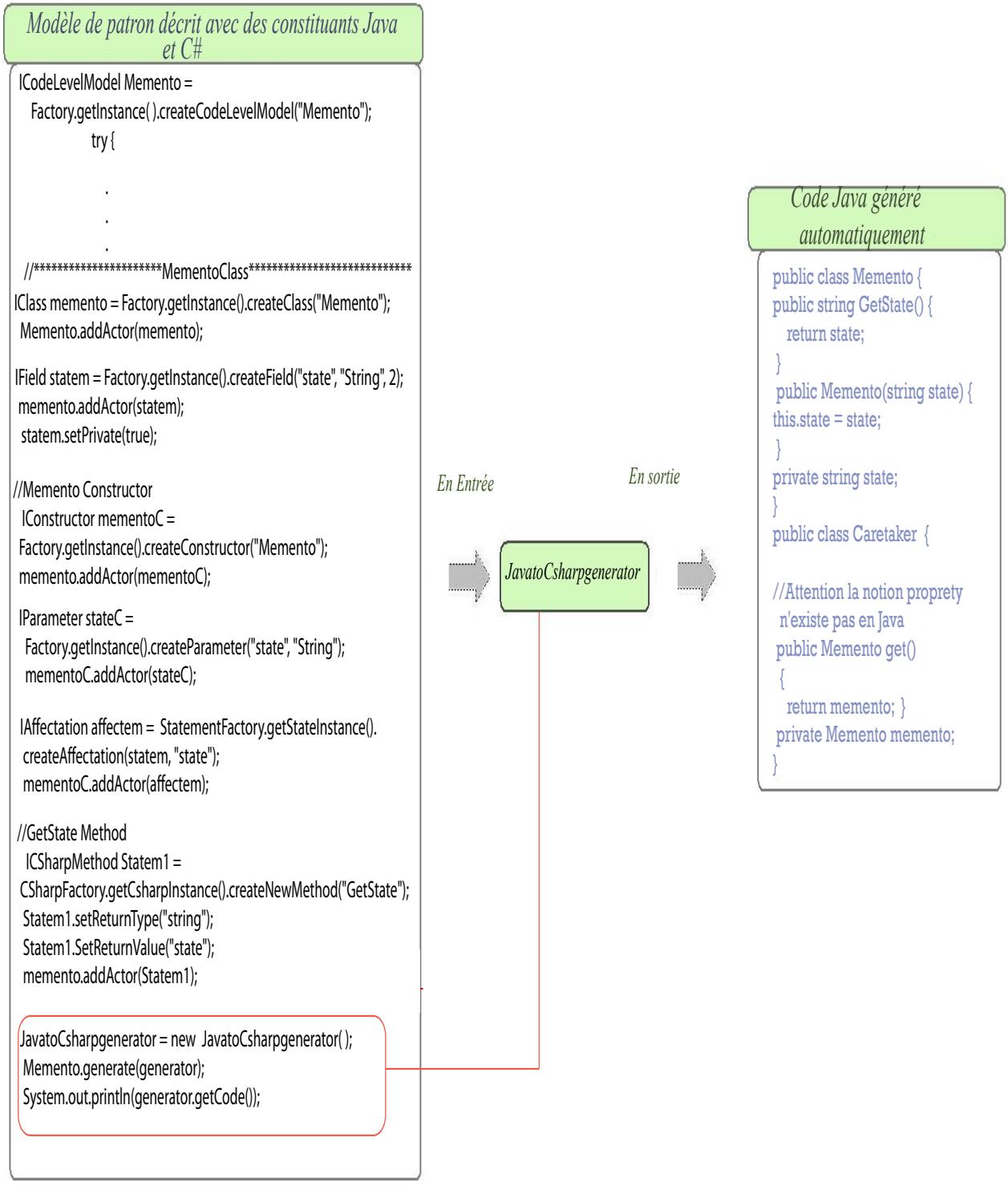

FIG. 5.7 – *Le générateur JavaToCsharpGenerator*

CHAPITRE 6

MÉTA-MODÈLE ESYS.NET

Dans la mesure où notre environnement est basé sur les méta-modèles et que Esys.net est une nouvelle bibliothèque développée en interne, dont les constituants et leurs interactions n'ont pas encore été organisés dans le cadre d'une structure, nous avons dans un premier temps (1) construit un méta-modèle Esys.net, intégrant ses principaux constituants et les règles régissant leurs interactions ; (2) proposé une transformation de modèle, basée sur les rôles, entre les composantes du méta-modèle Esys.net et celles du méta-modèle MIP, afin d'établir une mise en correspondance entre les deux méta-modèles et faciliter ainsi la description d'un patron de conception avec le méta-modèle Esys.net, de la même manière que nous le faisons avec le méta-modèle MIP. Pour établir nos choix de transformations entre les deux méta-modèles ; (3) modélisé quelques patrons de conception de ceux que nous avons modélisé auparavant avec le méta-modèle MIP, à l'aide des constituants du méta-modèle Esys.net ; (4) établi nos choix de transformation entre les méta-modèles Esys.net et MIP, ainsi que la construction du générateur Esys.net.

Nous allons donc consacrer cette section à (1) la présentation des principaux constituants de Esys.net ; (2) la présentation du méta-modèle Esys.net ; (3) la description de deux patrons de conception du GOF (*Observer* et *Singleton*) avec le méta-modèle Esys.net ; (4) nos choix de transformation entre les méta-modèles Esys.net et MIP ;

6.1 Les constituants de la bibliothèque Esys.net

Le projet Esys.net a poussé les limites d'un langage de description de matériel, *SystemC* [WWJ96] qui sert à la modélisation de systèmes au niveau comportemental, en créant un environnement plus polyvalent, Esys.net [LAN+04]. Cet environnement prendra en charge la modélisation et simulation de systèmes matériels afin

de faciliter à l'utilisateur le passage d'une description d'un modèle abstrait à la réalisation du prototype correspondant.

Les événements sont très importants dans Esys.net, car comme pour le langage *SystemC*, Esys.net est un système dont le fonctionnement est basé sur une série d'événement, DEES (*event-driven simulation*). Dans Esys.net le noyau de simulation est un groupe de processus qui doit s'exécuter, à une instance de temps donné et l'événement est une cause de déclenchement associé à une instance de temps.

Esys.net se base sur le principe de subdivision pour résoudre un problème, «*Diviser pour Régner*». Cette technique connue en programmation consiste à diviser un problème de grande taille en plusieurs sous-problèmes. Dans Esys.net, le premier concept dans la subdivision est le module.

6.1.1 Les modules

Un module peut être défini comme une boite noire qui encapsule une fonctionnalité. La description d'un système matériel avec Esys.net suit une hiérarchie basée sur les modules, où chaque module peut en contenir d'autres. En définissant des instances de modules dans d'autres on obtient un arbre hiérarchique permettant de décomposer un problème complexe en une hiérarchie de module simple à gérer et à comprendre, figure 6.1.

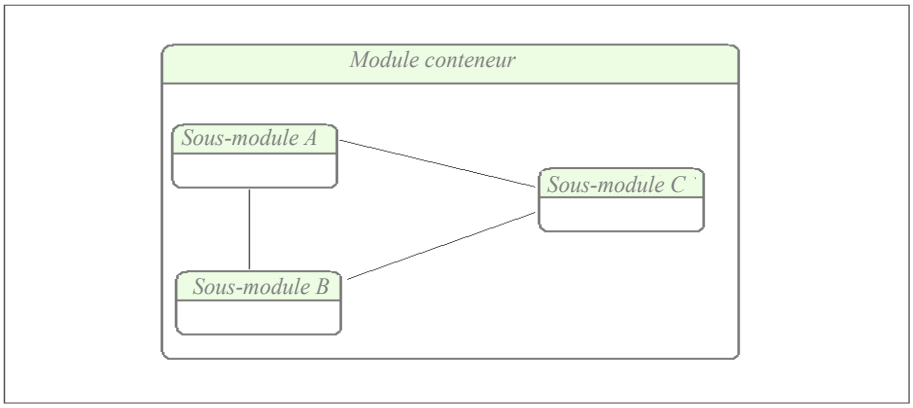

FIG. 6.1 – *La hiérarchie des modules*

L'utilisation d'un module se fait en deux parties imbriquées :

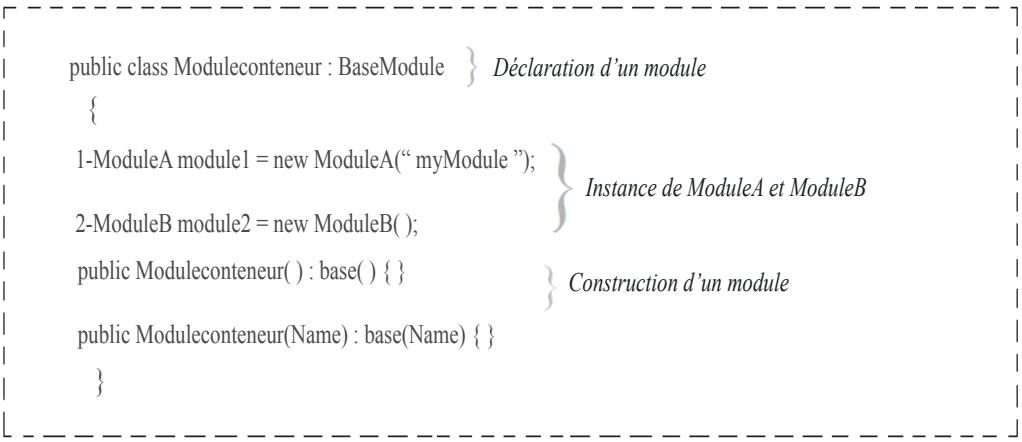

FIG. 6.2 – *Instanciation d'un module*

1. La déclaration du module : consiste à définir sa composition. Dans la mesure où Esys.net se base sur le concept orienté objet [LAN+04], tout module déclaré dans un système doit hériter de *BaseModule* qui représente le module supérieur et doit implémenter nécessairement un constructeur ;
2. L'instanciation d'un module dans le contexte d'un système : avant d'utiliser un module, il faut l'instancier en déclarant une variable de même type. L'instanciation d'un module peut se faire de deux manières : (i) le code sur la ligne 1, (figure 6.2) initialise une variable, (*module1*) en créant un objet de type *ModuleA* et lui assigne un nom d'identification, (*myModule*) ; (ii) le code sur la ligne 2 initialise une variable, *(module2)* en créant un objet de type *ModuleB* et son nom d'identification généré automatiquement sera *module2*.

Comme pour les langages orientés objet, le module est une classe avec une fonctionnalité et l'utilisateur peut y accéder par le biais de son interface.

6.1.2 Les interfaces

L'interface d'un module dans Esys.net représente le même concept que les interfaces en programmation orientée objet. L'utilisateur final n'interagit qu'avec l'interface du module et peut tout ignorer de son fonctionnement interne. Ainsi, l'utilisation des modules se fait en connectant un module à l'autre sans se soucier du fonctionnement interne de chacun.

Tout élément permettant d'interagir avec un module est considéré comme l'interface du module. En Esys.net l'élément basic pour créer une interface de module est le port.

6.1.3 Les ports

Dans un module les ports représentent les points d'entrée ou de sortie des données et sont équivalents aux broches d'un circuit permettant la circulation des informations à l'intérieur et à l'extérieur du module. Ainsi, c'est à travers les ports que les modules peuvent interagir avec leur environnement. Dans un module les ports sont des variables d'instances qui permettent de définir l'état de l'objet auquel chaque variable est associée.

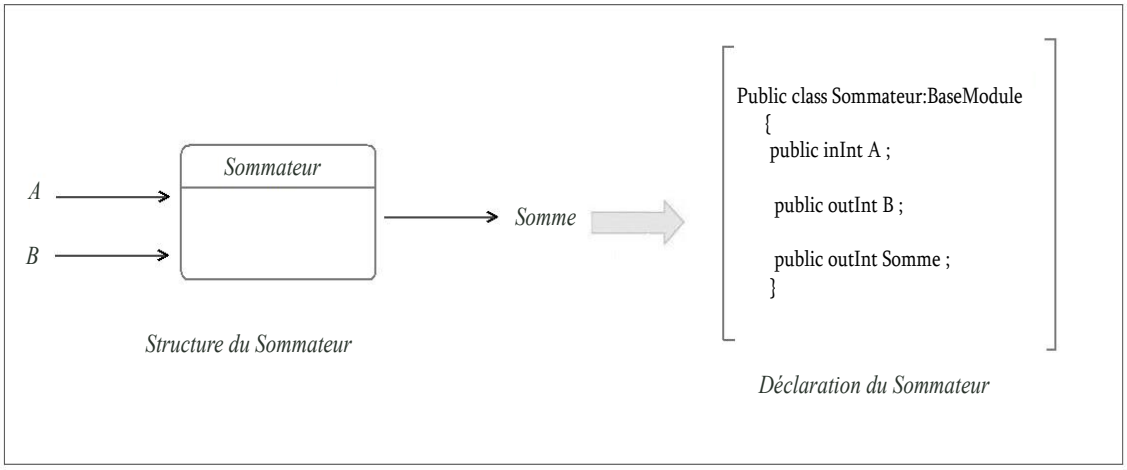

FIG. 6.3 – *Les ports d'un module*

Le module *Sommateur*, figure 6.3 reçoit des données de type boolean, via ses ports d'entrée A et B du même type et produit leur somme via son port de sortie, *Somme*. Ainsi, les ports peuvent être d'entrée et de sortie et chaque port dispose d'un nom d'identification et d'un type.

Le module *Sommateur* a une capacité de calcul qui se résume à faire la somme de deux nombres reçus en entée, A et B. Cette capacité est représentée par le processus du module.

6.1.4 Les processus

Suivant la sémantique de Esys.net tout système est partagé en module et tout module devant agir dans le système doit avoir une capacité de calcul qui est représentée par son processus. Le processus représente l'unité logique pour le traitement de données et est créé en ajoutant une balise [MethodProcess] à une méthode privée qui ne retourne aucune valeur et ne comporte aucun paramètre, figure 6.4

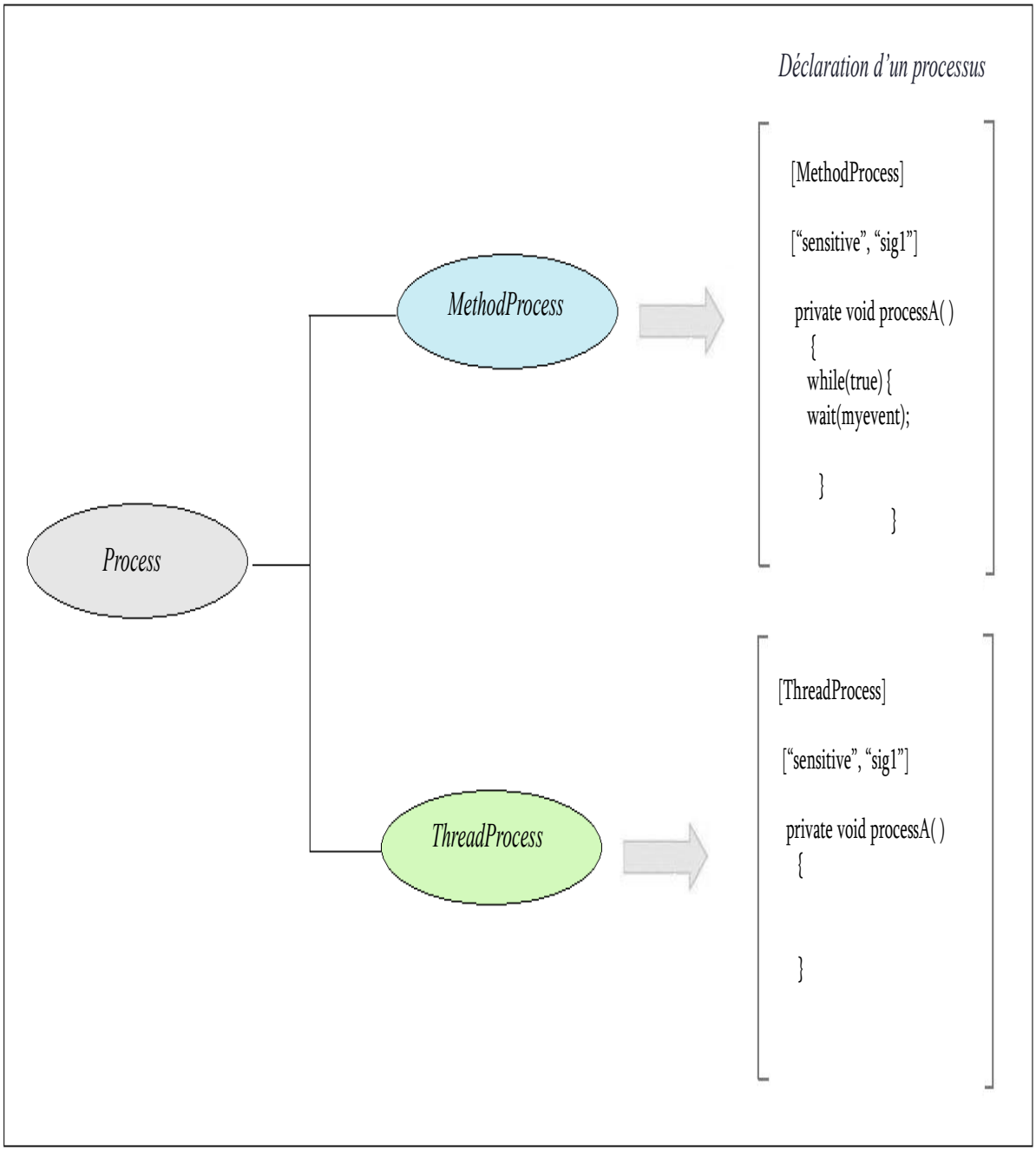

FIG. 6.4 – *Les processus d'un module*

Le processus est ainsi une méthode qui accède à toutes les variables, instances et méthodes d'un module. Dans la mesure où les ports d'un module sont des variables d'instances, le processus peut y accéder et c'est par ce mécanisme qu'un processus peut écrire sur les ports d'entée d'un module et lire sur ses ports de sortie.

Dans Esys.net, on retrouve deux types de processus, figure 6.4 :

- *ThreadProcess* agit comme un processus normal et se résume à un ensemble d'instructions à exécuter. *ThreadProcess* est créé en ajoutant une balise [ThreadProcess] et une liste de sensibilité [EventList] à une méthode privée qui ne retourne aucune valeur et ne porte aucune signature. Le *ThreadProcess* s'exécute jusqu'à l'appel de la méthode *wait()* dont l'exécution est contrôlée par un événement. Au moment de la suspension, l'état du processus est implicitement sauvé. Si un *ThreadProcess* doit être exécuté plus d'une fois, le processus doit implémenter une boucle, le plus souvent une boucle infinie, (*while*, figure 6.4) ;
- *MethodProcess* est quant à lui similaire à un processus, sauf que là où chaque processus possède sa propre mémoire virtuelle, les processus *MethodProcess* appartenant au même processus père se partage sa même mémoire virtuelle. Il est également impossible de suspendre l'exécution d'un *MethodProcess*, car il ne contient pas de méthodes *wait()* ni *while*, (figure 6.4).

Suivant la sémantique de Esys.net, toute action devant agir doit être reliée à un événement, dès que l'événement se déclenche l'action qui lui est sensible se déclenche à son tour.

6.1.5 Les événements

Chaque processus est sensible à un événement. Le lien entre les deux est défini dans une liste qui balise le processus appelée liste de sensibilité, (figure 6.4).

Comme pour les modules, l'utilisation d'un événement se fait en deux parties imbriquées, la déclaration de l'événement et son instanciation. Tous les événements sont une instance de la classe de base *Event*.

```
Event myEvent = new Event(); //Création et instanciation d'un
événement
```

Suivant la sémantique de Esys.net, un événement est toujours détenu. Il peut ainsi : (i) être la propriété d'un module, d'un canal ou d'un signal ; (ii) être global à un module et devient alors la propriété de tous ses sous-modules.

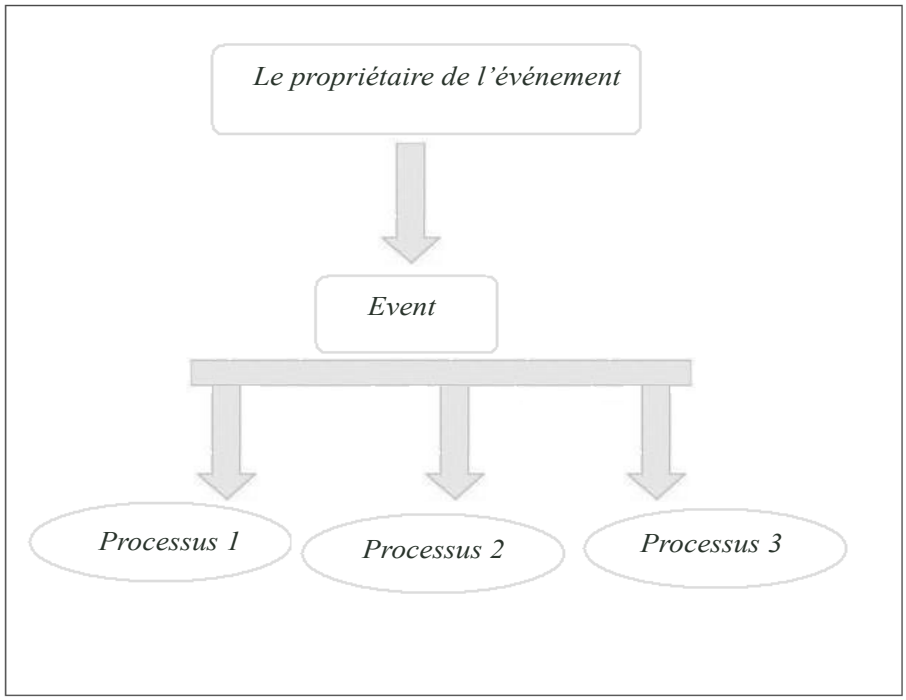

FIG. 6.5 – *L'organisation des événements*

Le propriétaire d'un événement a la responsabilité de sa notification lorsque la cause du déclenchement de l'événement survient, (figure 6.5). L'objet *Event*, à son tour, est chargé de tenir une liste de processus liés à l'événement. Ainsi quand l'événement est notifié, l'objet *Event* informe le noyau de la simulation des processus qui doivent être déclenchés, (figure 6.5). La cause de déclenchement d'un événement peut être l'arrivée d'un moment spécifique dans la simulation, le changement de la valeur d'un signal ou la survenance d'autres événements, etc.

```
myEvent.notify( ); // Notification immédiate

myEvent.notify(0); // Notification d'un delta-cycle retardé

myEvent.notify( 10); // Notification de 10 unités de temps
```

FIG. 6.6 – *Type de notification d'un événement*

L'événement peut être notifié de trois façons : (1) notification immédiate, si l'événement doit se déclencher lors de la phase d'évaluation du delta-cycle actuel, quand la méthode *notify()* n'a aucun argument, le déclenchement de l'événement est immédiat ; (2) la notification d'un delta-cycle retardé, signifie que l'événement doit être déclenché au prochain delta-cycle. Une méthode avec un argument *notify(0)* indique un delta-cycle retardé ; (3) avec un temps de notification, ce qui signifie que l'événement sera déclenché au temps indiqué à la méthode de notification. La méthode *notify(X)*, avec un argument différent de zéro, indique que l'événement sera déclenché à X unités de temps. Il existe aussi la possibilité d'annuler la notification d'un événement en utilisant la méthode *cancel()*, sauf pour les événements dont la notification est immédiate, car l'événement se déclenche immédiatement.

Après avoir présenté la notion de module, de port, de processus et d'événement nous pouvons maintenant diviser un problème complexe en sous-unités et décrire la capacité de calcul de ces unités. Cependant, nous ne pouvons pas encore décrire les interconnexions entre les différentes entités. Ainsi, afin de créer un modèle de simulation d'un système, nous devons définir la manière de transporter les données d'un module à un autre. Dans Esys.net le principal constituant permettant le transport des données est le signal.

6.1.6 Les signaux

Les signaux représentent l'unité basique permettant les interconnexions entre interfaces et modules et jouent le rôle de fil conducteur entre les modules, en assurant le transfert des informations. Comme pour les modules et les événements, l'utilisation d'un signal se fait en deux parties imbriquées, la déclaration du signal et son instanciation, (figure 6.7).

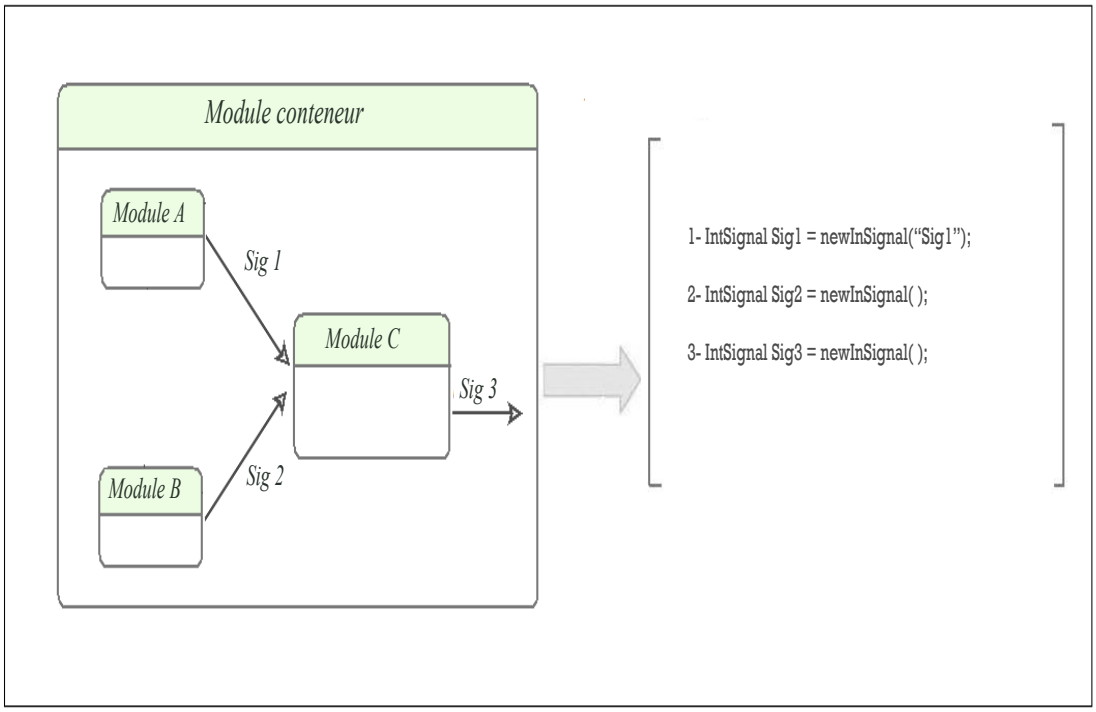

FIG. 6.7 – *Les signaux d'un module*

Un signal peut être de trois types : (1) *Inner*, c'est-à-dire qu'il est accessible à l'intérieur de son module conteneur et est considéré comme privé ; (2) *Outer*, c'est-à-dire qu'il est accessible à l'extérieur de son module conteneur et est considéré comme public ; (3) *Inner/Outer*, le signal dans ce cas est en entrée et sortie. Un signal peut être aussi en entrée pour un module et en sortie pour un autre, tels que *Sig1* et *Sig2*, (figure 6.7). Dans Esys.net, chaque signal possède deux événements spécifiques : (1) *sensitive* qui est déclenché lorsque le signal a une nouvelle valeur ;

(2) *transaction* qui est déclenché lorsque une valeur est écrite dans le signal.

6.1.7 Les horloges

Les horloges sont considérées comme des signaux à cause de leur fonctionnement, malgré que se sont, en fait, des canaux. Chaque horloge a un front montant et un front descendant et possède trois événements : *posedge*, *negedge* et *sensitive*. L'événement *posedge* est déclenché sur le front montant, *negedge* est déclenché sur le front descendant et *sensitive* lorsque l'horloge a une nouvelle valeur.

Avec les éléments vus jusqu'à maintenant, il est possible de créer un système assez complexe. Toutefois, comme la plupart des langages de description de système matériel, Esys.net inclut dans sa sémantique la notion de canaux.

6.1.8 Les canaux

Les canaux sont des éléments de modélisation pour des communications inter-modules complexes. C'est par le biais des canaux et des signaux que les modules peuvent communiquer entre eux. Chaque canal peut décrire une simple communication point-à point, mais aussi un réseau complexe.

Un canal est au même niveau d'abstraction qu'un module, mais peut aussi être considéré comme un signal avec un fonctionnement interne plus complexe. En effet, comme un signal un canal transporte des données entre les modules. Cependant, les canaux diffèrent des signaux car ils peuvent, tout comme les modules, contenir des sous-modules, des sous-canaux et des processus. La hiérarchie des canaux est souvent utile pour modéliser la communication entre modules dans un système.

Il y a deux parties impliquées dans l'utilisation d'un canal : la déclaration du canal et son instanciation dans le contexte d'un système, figure 6.8

```
1- public class Mychannel : BaseChannel : myInterface
{
  2- public Mychannel( ) { }

  3- public Mychannel(string name ) : base(name) { }

}
 4- Mychannel chanel1 = new Mychannel ("chanel1");
```

FIG. 6.8 – *Déclaration et Instanciation des canaux*

- les canaux héritent de la classe *BaseChannel*, qui hérite de la classe *BaseModule* et sont instanciés de la même manière que les modules, (*ligne 4*, figure 6.8) ;
- sur la *ligne 1*, nous avons déclaré un canal qui implémente l'interface utilisateur (*MyInterface*). Comme les canaux sont une forme de module, ils ont aussi une interface. Cependant, les canaux implémentent habituellement les interfaces logicielles de manière concrète ;
- la communication entre les modules et les canaux se fait par les appels de méthodes définies dans les interfaces ;
- un canal peut aussi implémenter plusieurs interfaces et une interface peut être implémentée de plusieurs manières par différents canaux.

6.2 Le méta-modèle Esys.net

Le méta-modèle Esys.net est défini d'une manière indépendante des méta-modèles PADL, MIP et MIP$\mathcal{C}\sharp$. L'utilisateur a la possibilité soit d'utiliser Esys.net de manière indépendante afin de modéliser des patrons de conception avec des constituants propres à Esys.net, soit de relier le méta-modèle Esys.net aux méta-modèles PADL, MIP ou MIP$\mathcal{C}\sharp$ afin d'ajouter des constituants orientés objet à ceux de Esys.net pour une modélisation de systèmes matériels/logiciels. Suivant la sémantique du langage Esys.net, ses différents constituants et les interactions entre

eux, le méta-modèle Esys.net, peut être résumé par la figure ci-dessous.

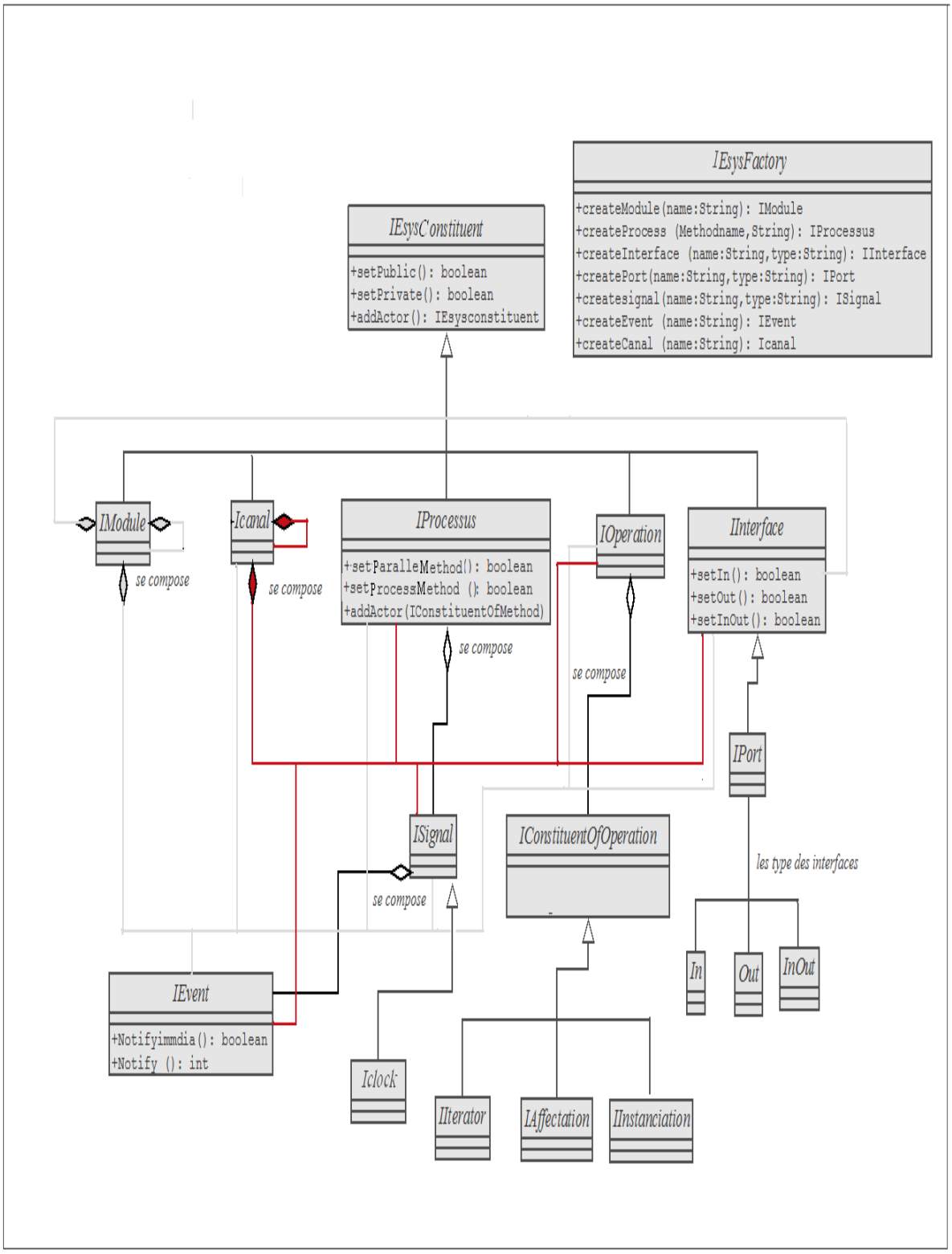

FIG. 6.9 – *Le méta-modèle Esys.net*

La structure des classes sur la figure 6.9 représente la hiérarchie des interfaces dans le méta-modèle Esys.net, car comme pour l'implémentation des méta-modèles PADL, MIP et MIP$\mathcal{C}\sharp$, l'implémentation de Esys.net se décompose en deux parties : les interfaces, structurées selon la sémantique de Esys.net et les classes d'implémentation leur correspondant, avec une hiérarchie possiblement différentes, selon nos besoins de développement.

En commençant par le niveau d'abstraction le plus haut on retrouve :

- l'interface *IEsysConstituent* qui représente la principale classe du méta-modèle Esys.net, car toutes les autres classes correspondant aux constituants de Esys.net sont ses sous-classes. L'interface *IEsysConstituent* définit toutes les méthodes partagées par l'ensemble des constituants du méta-modèle Esys.net. Parmi ces méthodes on retrouve *setPublic()*, *setPrivate()* qui définissent les modificateurs d'accès et *addActor(IEsysConstituent)* qui prend en paramètre un objet de type *IEsysContituent* et qui permet de créer des compositions entre les constituants du méta-modèle Esys.net en ajoutant les uns aux autres ;
- l'interface *IEsysFactory* qui englobe l'ensemble des méthodes permettant la création des constituants du méta-modèle Esys.net ;
- l'interface *IModule* représente, quant à elle, l'élément conteneur dans le méta-module Esys.net. Chaque module peut être composé d'autres modules, de processus, d'opérations et de canaux. Cette relation de composition est exprimée par la méthode *addActor()*, de l'interface *IEsysConstituent*. La création d'un objet de type *IModule* est exprimée par la méthode *createModule()*, de la classe *IEsysFactory*.

  `Public IModule createModule(final String name)`

  Cette méthode prend en paramètre une chaîne de caractères (*name*) représentant le nom du module ;
- l'interface *IProcessus* fait partie des constituants des modules (*IModule*) et représente les processus. L'interface *IProcessus* définit un ensemble de méthodes permettant la gestion des processus dans un système : (1) *setProcessMe-*

thod() permet de définir une méthode comme étant un *thread* ; (2) *setParalleMethod()* permet de définir une méthode comme étant un processus ; (3) la méthode *addActor(IConstituentOfMethod)* permet d'ajouter des constituants de méthodes orientées objet à un processus. Tout processus peut faire partie d'un canal ou d'un module et la création d'un objet de type *IProcessus* est représentée par la méthode *createProcess()* de la classe *IEsysFactory*.

`Public IProcess createProcess (final String Methodname)`

Cette méthode prend en paramètre une chaîne de caractères *Methodname* définissant le nom de la méthode représentant le processus ;

– l'interface *IInterface* représente les interfaces logicielles redéfinies dans Esys.-net. Chaque interface peut être de trois types et chacun de ses types est représenté par les méthodes : *setIn*() pour définir une interface en entrée, de type *In* ; (2) la méthode *setOut*() pour définir une interface en sortie, de type *Out* ; (3) *setInOut* pour définir une interface en entrée et sortie, de type *In/Out*. La création d'un objet de type *IInterface* est représentée par la méthode *createInterface()* de la classe *IEsysFactory*.

`Public IInterface createInterface (String name, String type)`

Cette méthode prend en paramètre une chaîne de caractères, *name* représentant le nom de l'interface et une chaîne de caractères, *type* représentant le type de l'interface ;

– les ports *IPort* sont considérés comme étant les interfaces des modules, ils sont définis comme un sous-type de l'interface, *IInterface*. Un port peut être en entrée, en sortie ou en entrée sortie, de la même manière que les interfaces, car l'interface *IPort* hérite de toutes les méthodes de *IInterface*. La création d'un objet de type *IPort* est exprimée par la méthode *createPort()* de l'interface *IEsysFactory*

`Public IPort createPort(final String name, final String type)`

Cette méthode prend en paramètre une chaîne de caractères, *name* représentant le nom du port et une chaîne de caractères, *type* représentant le type du port ;

– l'interface *ISignal* représente les signaux dans un système. Un signal peut être ajouté à un module, un canal ou une interface. Un signal peut aussi comporter un événement. Ces compositions sont exprimées par la méthode *addActor(IEsysConstituent)* de l'interface *IEsysConstituent*. La création d'un objet de type *ISignal* est exprimée par la méthode *createSignal()* de la classe *IEsysFactory*

```
Public ISignal createsignal(final String name, final String type)
```

Cette méthode prend un paramètre une chaîne de caractères, *name* représentant le nom du signal et une chaîne de caractères, *type* représentant le type de données que transporte le signal ;

– l'interface *IEvent* correspond aux événements. Un événement peut être la propriété d'un signal, d'un module ou d'un canal. L'interface *IEvent* offre quelques méthodes propres aux événements : (1) la méthode *notifyiImmdiately()* permet de déterminer une notification immédiate de l'événement ; (2) la méthode *notify(int)* prend en paramètre un entier et la valeur de ce dernier déterminera si l'événement aura une notification d'un delta-cycle ou une notification décalée d'un nombre d'unités de temps. La création d'un objet de type *IEvent* est exprimée par la méthode *createEvent()* de la classe *IEsysFactory*.

```
Public IEvent  createEvent (final String name)
```

Cette méthode prend en paramètre une chaîne de caractères, *name* représentant le nom de l'événement ;

– l'interface *ICanal* représente les canaux dans le méta-modèle Esys.net. Comme pour les modules, les canaux peuvent se composer de tous les constituants du méta-modèle Esys.net. La méthode *addInterface(IInterface)* permet de définir l'interface implémentée par le canal. La création d'un objet de type *ICanal* est exprimée par la méthode *createCanal()* de la classe *IEsysFactory*.

```
Public IEvent createEvent(final String name)
```

Cette méthode prend en paramètre une chaîne de caractères, *name* représentant le nom du canal ;

– l'interface *IOperation* correspond aux méthodes orientées objet et leur gestion est définie dans les processus, qui font appel à ces opérations selon le comportement du système modélisé. La classe *IOperation* est une sous classe de la classe *ICsharpMethod* définie dans le méta-modèle $\mathcal{C}\sharp$ (figure 3.9).

6.3 Description des patrons de conception avec le méta-modèle Esys.net

Après avoir construit le méta-modèle Esys.net, nous l'avons utilisé pour la modélisation des patrons de conception du GOF, afin de vérifier sa capacité de description. Cependant, dans la mesure où les constituants du méta-modèle Esys.net et leurs interactions sont différents de ceux des méta-modèles orientés objet, (PADL, MIP et MIP$\mathcal{C}\sharp$), la structure des modèles de patrons construits est différente. Par conséquent, la modélisation des patrons avec le méta-modèle Esys.net est orientée vers le comportement du patron et non sa structure.

6.3.1 Modélisation du patron Singleton

Nous avons choisi de modéliser le *Singleton* en premier, en raison de sa structure simple, car il se compose d'une seule classe, ce qui résume sa modélisation à la description de son comportement, (figure 3.2).

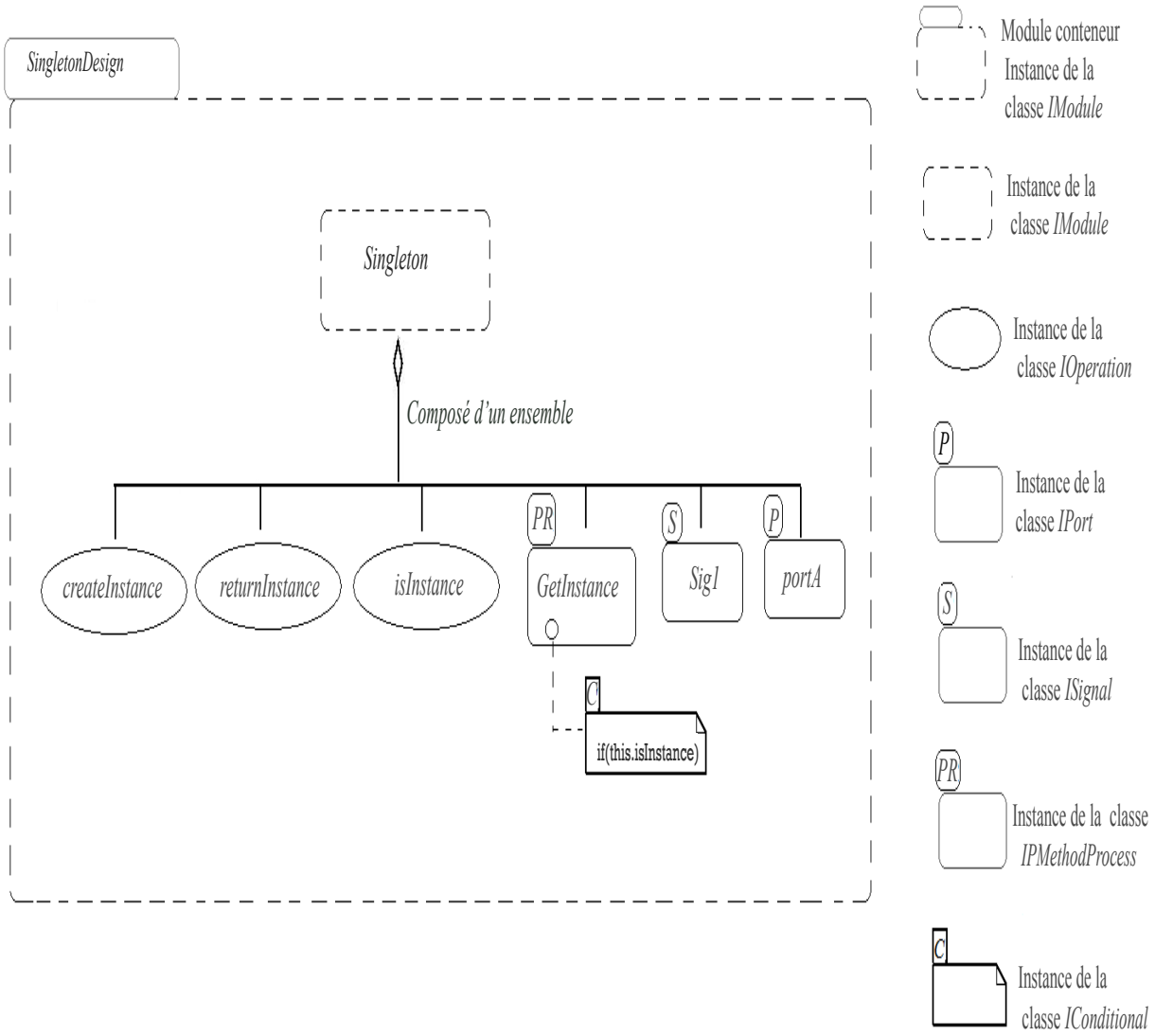

FIG. 6.10 – *Modélisation du patron Singleton avec le méta-modèle Esys.net*

La figure 6.10 représente la description du patron *Singleton* avec le méta-modèle Esys.net.

Nous y retrouvons les entités suivantes :

- le module *«SingletonDesign»* joue le rôle de module conteneur et représente la solution du patron *Singleton*. *«SingletonDesign»* est une instance de *IModule* dans le méta-modèle Esys.net, (figure 6.9) ;
- *«Singleton»* est un sous-module du module *«SingletonDesign»* et correspond à l'unique classe du patron *«Singleton»* (figure 3.2). La classe *«Singleton»* est une instance de *IModule* dans le méta-modèle Esys.net et définit le comportement du patron *Singleton* ;
- le sous-module *«Singleton»* est composé de trois opérations qui représentent des méthodes retournant une valeur et qui sont une instance de *IOperation* : (1) la première Opération, *isInstance()* est une opération retournant un boolean et nous sert à exprimer le fait qu'une instance de la classe *«Singleton»* existe ou non ; (2) la deuxième opération, *createInstance()* permet de créer une instance de la classe *«Singleton»* ; (3) la troisième opération *returnInstance()* permet de retourner l'instance de la classe *«Singleton»* ;
- le port d'entrée *portA* de type boolean fait partie des constituants du sous-module *«Singleton»* et joue le rôle de son interface. *portA* permet au module *«Singleton»* d'interagir avec son environnement, en lui transmettant des données en entrée. Dans le méta-modèle Esys.net, *portA* est une instance de la classe *IPort* ;
- le signal *Sig1*, de type boolean, fait partie des composantes du sous-module *«Singleton»* et permet de lui transmettre des données en entrée, via le port *portA*. Dans le méta-modèle Esys.net, *Sig1* est une instance de la classe *ISignal*.

  Avec les constituants déclarés jusqu'à maintenant, nous avons tous les apports nécessaires pour reproduire le comportement du patron *Singleton*. Ce comportement est reproduit par l'ordre d'appel de chacune des opérations. Selon la sémantique de Esys.net, tout module devant agir doit contenir un

processus, c'est donc dans un processus que nous allons organiser l'appel des méthodes dans le but de reproduire le comportement du patron *Singleton* ;

- le processus *«GetInstance»* est l'une des composantes du sous-module *«Singleton»*. Le processus *«GetInstance»* est sensible à l'événement *«Sensitive»* qui est la propriété du signal *Sig1*.
- à l'intérieur du processus *«GetInstance»*, on retrouve une structure conditionnelle, *«IConditional»* qui nous permet de vérifier si l'instance du module *«Singleton»* existe en testant la valeur de l'opération *«isInstance()»*, si ce n'est pas le cas, elle fait appel à l'opération *«createInstance()»* pour en créer une, sinon elle fait appel à la méthode *«returnInstance()»* qui retourne l'instance courante du module *«Singleton»* ;
- l'événement *«Sensitive»* est déclenché lorsque le signal *Sig1* reçoit une nouvelle valeur, ce qui entraîne à son tour le déclenchement du processus *«GetInstance»*.

Le module conteneur *«SingletonDesign»* est chargé de changer la valeur du signal *Sig1*, en déclenchant ainsi le processus *«GetInstance»* qui reproduira le comportement du patron *Singleton*.

Après la modélisation du patron *Singleton*, nous nous sommes intéressés au patron *Observer*, (figure 4.2), dont la structure des classes et les interactions entre elles sont plus importantes que celles du *Singleton*, afin de vérifier la capacité de modélisation de Esys.net au niveau des relations entre classes.

6.3.2 Modélisation du patron Observer

La sémantique du méta-modèle Esys.net ne permet pas de décrire comme c'est le cas pour les méta-modèles orientés objet, les relations structurelles entre les classes, tels que l'héritage ou les appels de méthodes. C'est la raison pour laquelle, dans la modélisation des patrons orientés objet avec le méta-modèle Esys.net, nous allons remplacer les relations entre les classes par d'autres types de relations en nous basant sur leurs rôles dans la solution du patron. La modélisation des classes composant le patron *Observer* avec le méta-modèle Esys.net se fera de manière

indépendante, c'est-à-dire que le comportement de chaque classe sera modélisé indépendamment dans un module et les relations entre ces classes seront gérées par les processus.

Nous commencerons, dans ce qui suit, par définir le scénario d'exécution du patron *Observer* dans la figure 6.11 et nous présenterons, par la suite, la structure de chaque module participant au scénario.

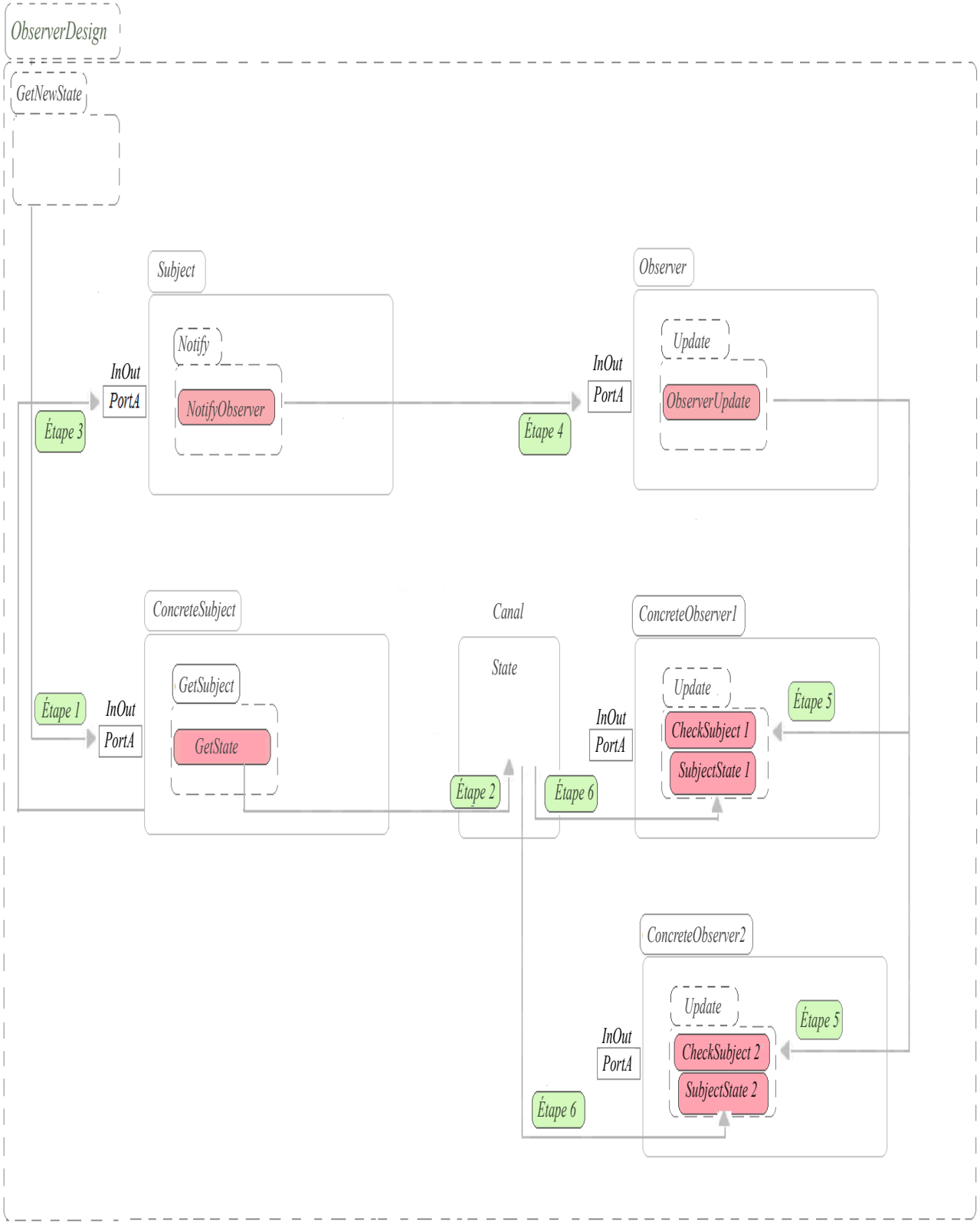

FIG. 6.11 – *Scénario d'exécution du patron Observer*

Dans le méta-modèle Esys.net le module représentant la classe *«ConcreteSubject»* comprend un processus *«GetSubject»*, qui retourne la valeur de l'état du sujet. Le module conteneur *«ObserverDesign»* se charge de déclencher l'exécution du processus, *«GetSubject»* qui sera suivi de l'exécution d'autres processus, en reproduisant ainsi le comportement du patron modélisé :

- étape 1 : le processus *«GetNewState»* du module conteneur, *«ObserverDesign»* déclenche le processus *«GetSubject»* du module *«ConcreteSubject»* en attribuant une nouvelle valeur au signal *«GetState»*, représentant l'état du sujet ;
- étape 2 : le processus *«GetSubject»* enregistre la valeur du signal, *«GetState»* sur le canal *«State»*, car contrairement au langage orienté objet dans Esys.net le retour d'une valeur doit se faire de manière concrète ;
- étape 3 : le processus *«GetSubject»* notifie sa classe parente, représentée par le module *«Subject»* de ce changement. Cette notification est faite en changeant la valeur du signal *«NotifyObserver»* qui va automatiquement déclencher le processus *«Notify»* du module *«Subject»* ;
- étape 4 : le processus *«Notify»* notifie à son tour le module *«Observer»*, en changeant la valeur du signal *«ObserverUpdate»* qui va déclencher le processus *«Update»* ;
- étape 5 : le processus *«Update»* représentant le module parent de *«ConcreteObserver1»* et *«ConcreteObserver2»* va notifier ces derniers en changeant la valeur des signaux *CheckState1* et *CheckState2* des processus *«Update1»* et *«Update2»* ;
- étape 6 : une fois que les processus *«Update»* se déclenchent, *«ConcreteObserver1»* et *«ConcreteObserver2»* se synchronisent avec le *«ConcreteSubject»* en enregistrant la valeur du canal *«State»* représentant l'état actuel du *«ConcreteSubject»* sur leurs signaux respectifs *«SubjectSate1»* et *«SubjectSate2»* .

Nous avons maintenu les modules parents *«Subject»* et *«Observer»* afin de reproduire le comportement du patron *Observer* en gardant ses quatre classes de base. Cependant, dans le scénario de la figure 6.11, nous pouvons éliminer les modules

«Subject» et *«Observer»*, car leurs tâches peuvent être assumées par les modules *«ConcreteSubject»* et *«ConcreteObserver»*.

Nous avons restreint les constituants de chaque module à ceux nécessaires à la reproduction du comportement du patron *Observer*. Les figures ci-dessous représentent la structure des modules participant au scénario d'exécution du patron *Observer*.

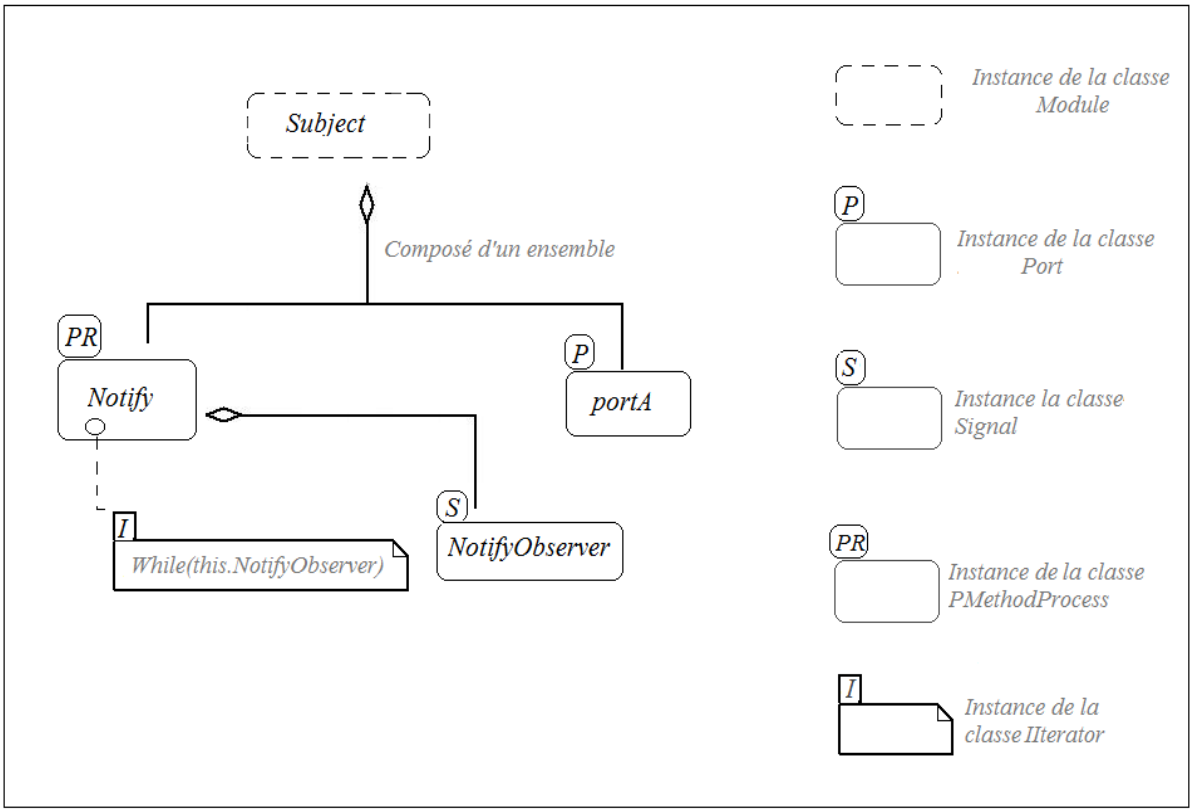

FIG. 6.12 – *La structure du sous-module Subject*

La figure 6.12 représente la structure du module *Subject*. Les entités de ce module sont les suivants :

– le module *«Subject»* comprend un processus *«Notify»* incluant une structure itérative chargée de vérifier si l'état du Sujet a changé. Cette information lui est transmise par le sujet, *«ConcreteSubject»* qui change la valeur du signal *«NotifyObserver»* ;

– le signal *«NotifyObserver»* possède l'événement *«Sensitive»* qui se déclenche dès que la valeur du signal change ; Le déclenchement de l'événement est une conséquence automatique du déclenchement du processus *«Notify»* qui lui est associé, figure 6.5 ;
– La nouvelle valeur du signal lui est alors affectée via le port d'entrée *«portA»* du module.

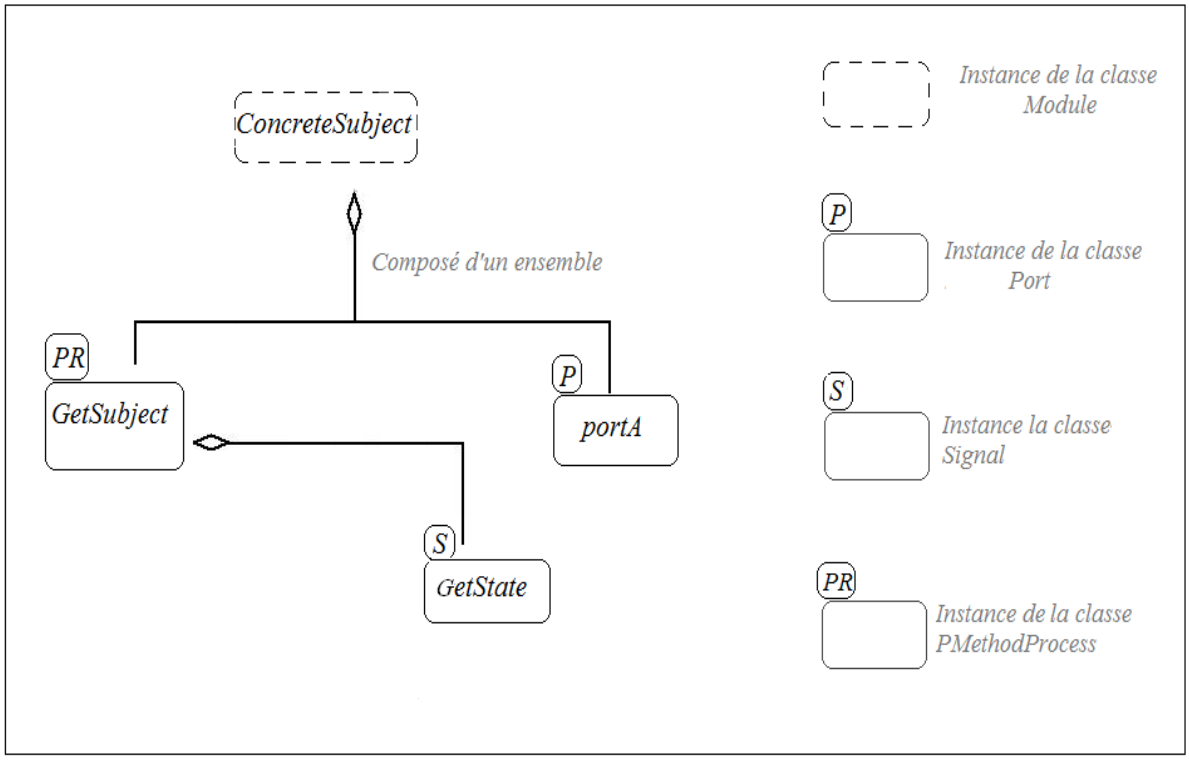

FIG. 6.13 – *La structure du sous-module ConctreteSubject*

La figure 6.13 représente la structure du module *ConctreteSubject*.

– le module *«ConctreteSubject»* comprend un processus *«GetSubject»* qui reçoit et transmet la valeur du signal *«GetState»* représentant l'état du sujet et ce via son port, *«portA»* qui est en entrée et sortie, figure 6.3 ;
– la valeur du signal *«GetState»* est reçu par le processus *«GetNewState»* du module conteneur *«ObserverDesign»* ;

– la valeur du signal *«GetState»* sera transmise au canal, *«State»* par le processus *«GetSubject»* via le port *«portA»*.

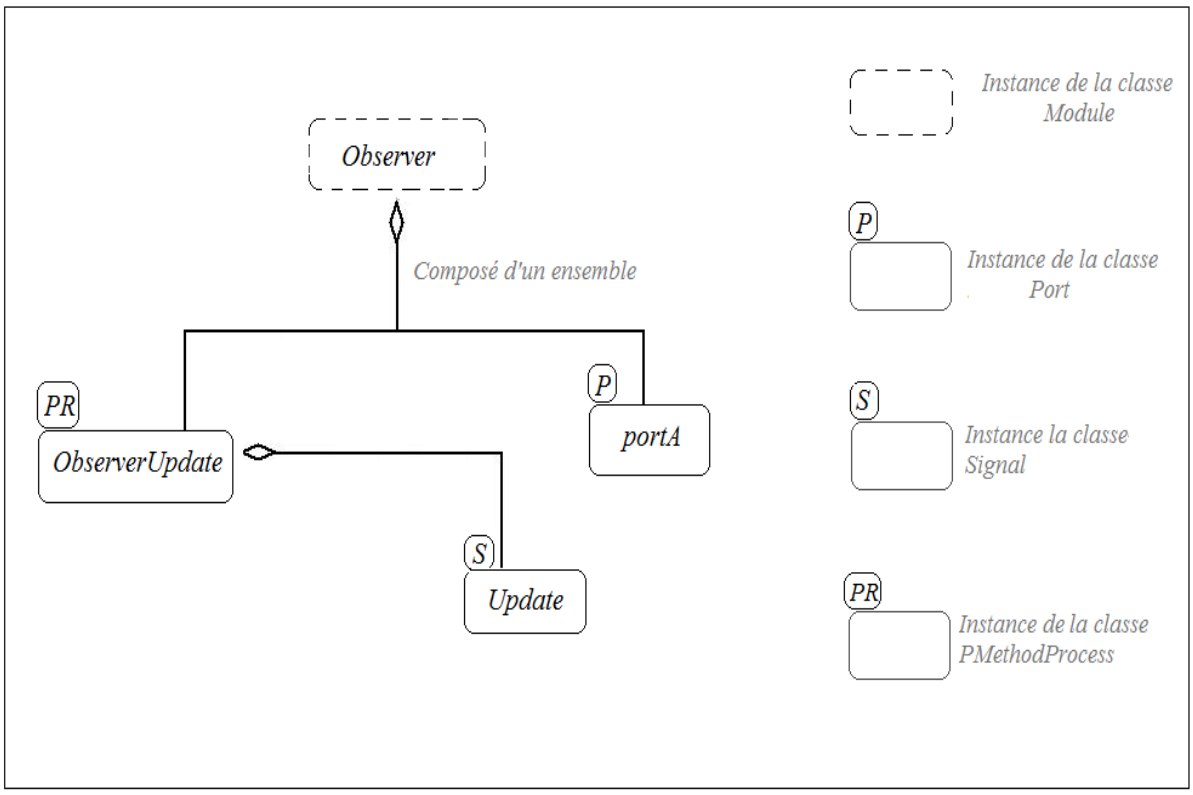

FIG. 6.14 – *La structure du sous-module Observer*

La figure 6.14 représente la structure du module *Observer*.

– le module *«Observer»* comprend un processus *«ObserverUpdate»* qui est déclenché par le processus *«Notify»* du module *«Subject»* ;
– le déclenchement du processus *«ObserverUpdate»* est réalisé en changeant la valeur du signal *«Update»* qui possède l'événement *«Sensitive»* ;
– la nouvelle valeur du signal *Update* est transmise via son port en entrée/sortie *«PortA»* ;
– le processus *«ObserverUpdate»* est également responsable de notifier le processus du module *«ConcreteObserver»*, afin de récupérer la nouvelle valeur du signal *«GetState»* du module *«ConcreteSubject»*.

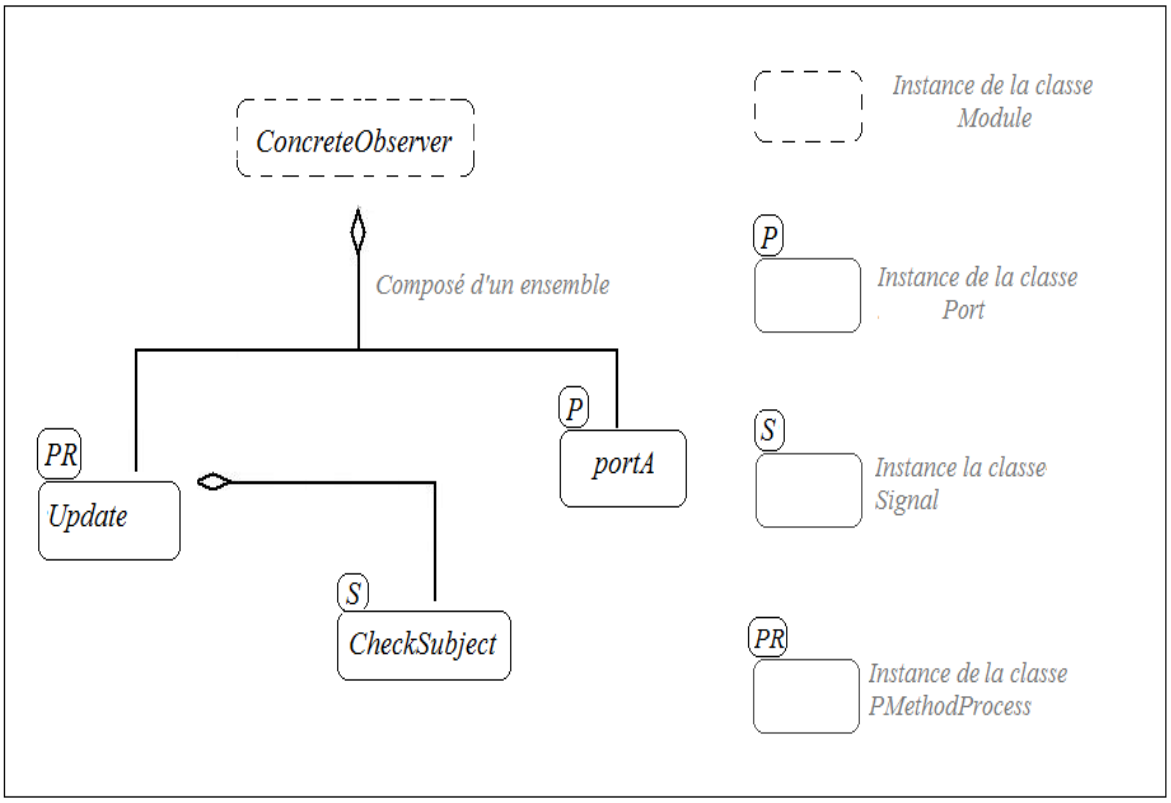

FIG. 6.15 – *La structure du sous-module ConcreteObserver*

La figure 6.15 représente la structure du module *ConcreteObserver*.

- les modules *«ConcreteObserver1»* et *«ConcreteObserver2»* comprennent les processus *«Update1»* et *«Update1»* responsablent de récupérer la valeur du canal *«State»*, afin de se synchroniser avec le module *«ConcreteSubject»* ;
- les processus *«Update1»* et *«Update2»* sont déclenchés par le changement de la valeur de leurs signaux respectifs *«CheckSubject1»* et *«CheckSubject2»*, qui possèdent des événements *«Sensitive»* ;

La figure ci-dessous rassemble le scénario d'exécution présentée dans la figure 6.11 et la structure des constituants participant à ce scénario.

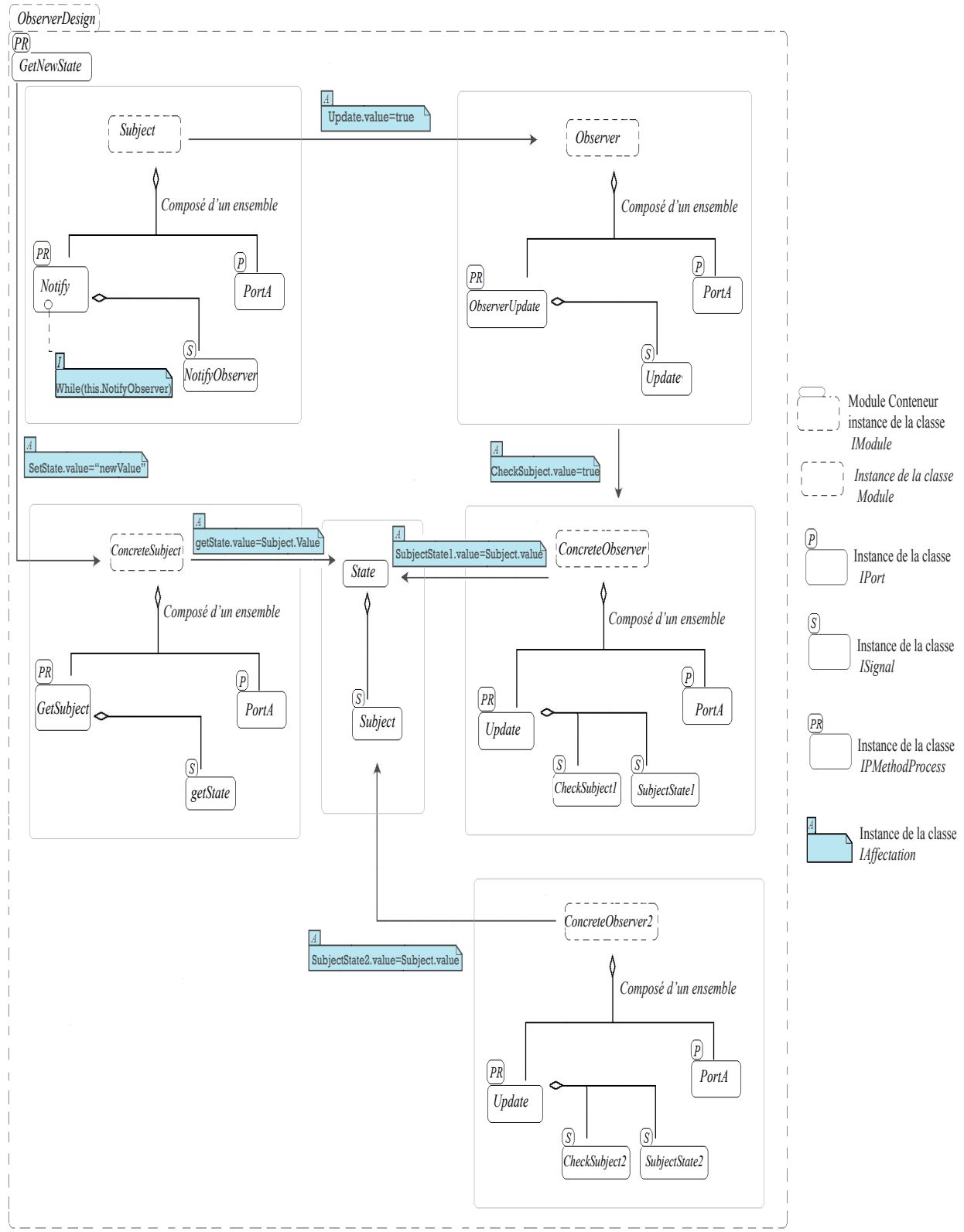

FIG. 6.16 – *Modélisation du patron Observer avec le méta-modèle Esys.net*

6.4 La transformation de modèle

Dans la mesure où la sémantique du méta-modèle Esys.net est différente de celle des langages orientés objets, sur laquelle est basée les méta-modèles PADL, MIP et MIP$\mathcal{C}\sharp$ ainsi que tous les patrons de conception que nous avons modélisé et implémenter, la transformation de modèle entre les modèles Esys.net et MIP ne pourrait pas trouver une mise en correspondance à tous les constituants des deux méta-modèles. Le tableau ci-dessous représente les principales propositions de transformation entre les méta-modèles Esys.net et MIP.

Les constituants du méta-modèle Esys.net	*Les constituants correspondants dans le méta-modèle MIP*
Un module	◦ *La solution du patron de manière globale* ◦ *Une classe*
Un Processus	◦ *Une méthode*
Un signal	◦ *Le champ d'une classe*
Un canal	◦ *Le champ d'une classe* ◦ *Une classe*
Une interface	◦ *Interface orienté objet*
Un port	◦*Un sous-type d'interface*

FIG. 6.17 – *Les choix de transformation entre les méta-modèles Esys.net et MIP*

1. La solution d'un patron de manière globale dans le méta-modèle Esys.net peut prendre l'instance d'un module, *IModule*1 dans le méta-modèle MIP. Notre choix est motivé par le fait qu'un module dans Esys.net ne représente qu'un conteneur englobant une fonctionnalité et un patron orienté objet est une structure englobant un comportement. De la même manière, toute classe avec un comportement dans le méta-modèle MIP correspond à un module dans le méta-modèle Esys.net.
2. Dans le méta-modèle MIP toute méthode portant une signature et retournant une valeur, correspond dans le méta-modèle Esys.net à une opération qui est un sous-type de la classe *IMethod*. Malgré que les opérations ne sont que des méthodes, nous avons changé leur nom afin de distinguer entre l'environnement MIP et celui de Esys.net.
3. Les processus dans Esys.net peuvent être considérés comme des méthodes, MIP de type privé, ne retournant aucune valeur et ne portant aucune signature. Cependant, dans la modélisation des patrons de conception avec le méta-modèle Esys.net, la gestion des appels d'opérations est organisée par les processus. On peut, ainsi considéré les processus dans Esys.net comme des méthodes orientées objet chargées d'un ordre d'appels de méthodes dépendamment du comportement du patron modélisé.
4. Dans le méta-modèle Esys.net, les ports représentent l'interface d'un module et permettent aux utilisateurs d'interagirent avec le module. Nous considérons ainsi, que les ports sont un sous-type d'interface du méta-modèle MIP.
5. Le signal dans Esys.net sert à transporter une donnée d'un module. Le signal peut être considéré comme un champ d'une classe ou un paramètre d'une méthode contenant une valeur et permettant aussi la transmission de données.
6. Dans le méta-modèle Esys.net, les canaux servent à transmettre les données entre les modules. Dans tous modèles de patron où il y a un changement de données entre module, nous utilisons les canaux afin de conserver une donnée transmise entre deux modules. Ainsi, les canaux peuvent être considérés

comme un champ d'une classe ou un paramètre d'une méthode, de la même manière que les signaux. Or, les canaux sont au même niveau que les modules, ils peuvent aussi contenir des sous-modules, des sous-canaux, des processus, etc. et peuvent ainsi être considérés comme des classes, selon leur rôle dans le modèle du patron.

CHAPITRE 7

VALIDATION

Après la construction du méta-modèle Esys.net et une transformation de modèle entre ce dernier et le méta-modèle MIP, nous avons pu décrire la solution d'un patron de conception tels que *Singleton* ou *Observer* avec les constituants du méta-modèle Esys.net. Afin d'assurer l'implémentation de cette solution, nous avons également construit un générateur de code Esys.net.

Dans le but d'appliquer le générateur Esys.net sur un cas d'étude réel, nous l'avons utilisé pour l'implémentation d'un modèle de régulateur de vitesse. Ce modèle a déjà été implémenté avec le langage Esys.net [Gir05] par Bruno Girodias. Nous allons donc, dans notre environnement, reproduire ce modèle du régulateur de vitesse en utilisant les constituants du méta-modèle Esys.net et générer le code Esys.net lui correspondant. Nous allons ainsi consacrer cette section à la présentation : (1) du système du régulateur de vitesse ; (2) de la modélisation de ce système avec le méta-modèle Esys.net ; (3) de la génération du code Esys.net correspondant au modèle construit à la deuxième partie.

7.1 Régulateur de vitesse

Le régulateur de vitesse est un système embarqué qui est composé d'éléments informatiques matériels et logiciels utilisés pour accomplir une tâche spécifique. Le régulateur de vitesse est utilisé la plupart du temps dans une voiture. Dans le modèle du régulateur de vitesse, figure 7.1, tous les modules sont contrôlés par des boutons permettant d'activer ou désactiver le régulateur en mettant ainsi une voiture «en marche» ou «en arrêt». Comme son nom l'indique, le régulateur de vitesse permet de contrôler la vitesse d'une voiture, cette vitesse peut être augmentée ou diminuée par l'accélérateur, modifiée par l'inclination de la route ou encore réglée en activant la commande de croisière. Dès que cette commande est activée, une

boucle calcule l'accélération/décélération exigée pour maintenir la vitesse désirée.

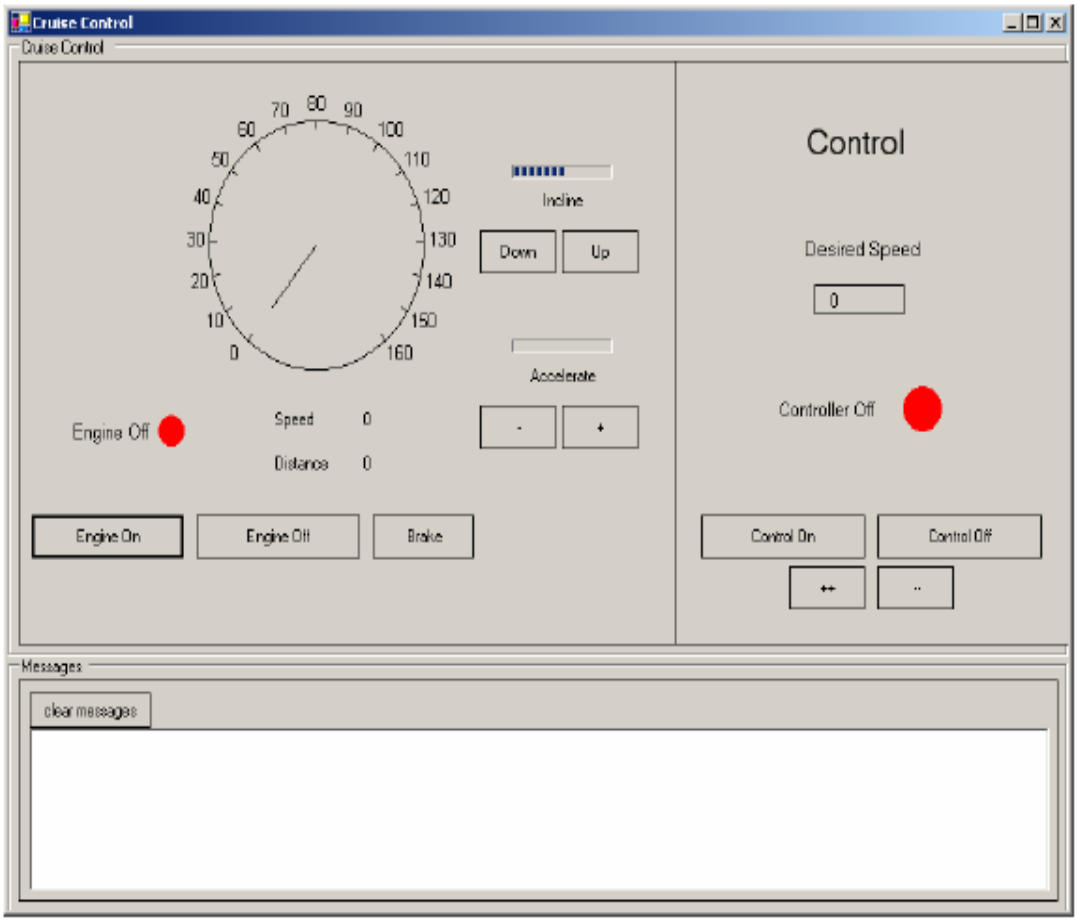

FIG. 7.1 – *Applications du régulateur de vitesse, figure*

Au moment où le régulateur de vitesse est activé, son accélération augmente et la vitesse de la voiture augmente aussi. Dès qu'il est désactivé son accélérateur est mis à zéro et la voiture ralentit. Le régulateur de vitesse est directement désactivé par le frein de la voiture, c'est-à-dire qu'au moment où le bouton du frein est pressé le régulateur de vitesse est directement désactivé, car l'acte de freiner est plus important que toutes les autres actions.

L'implémentation du régulateur de vitesse est réalisée à deux niveaux : (1) logiciel, où tous les aspects reliés au calcul de la vitesse ou de l'accélération sont

modélisés au niveau logiciel ; (2) matériel où l'utilisateur rentre directement en contact avec les constituants matériels en pressant par exemple le bouton d'accélération ou de freinage. Une interface est maintenue entre le matériel et le logiciel. Cette interface est simple puisque le matériel et le logiciel partagent le même environnement ainsi que plusieurs variables. La partie matérielle du régulateur de vitesse est modélisée avec Esys.net. Ce matériel, figure 7.1, est modélisé comme un clavier où chaque bouton est un événement. Un tampon est utilisé pour accumuler les différents événements. Tout bouton dans le système est relié à un module Esys.net. Chaque module est sensible à un événement particulier et à chaque coup d'horloge, l'ordonnanceur d'événement d'Esys.net traite un signal et le module qui lui est relié est ainsi déclenché.

Dans la mesure où les principaux constituants du méta-modèle Esys.net correspondent aux constituants matériels d'un système, nous nous sommes intéressés à la modélisation des constituants du régulateur de vitesse et leurs interactions.

7.2 Modélisation du régulateur de vitesse avec le méta-modèle Esys.net

La modélisation du régulateur de vitesse est réalisée en nous basant d'une part sur la structure du régulateur de vitesse, ses constituants et les interactions entre eux et d'autre part sur la sémantique du méta-modèle Esys.net. La modélisation du régulateur de vitesse se résume à la figure ci-dessous.

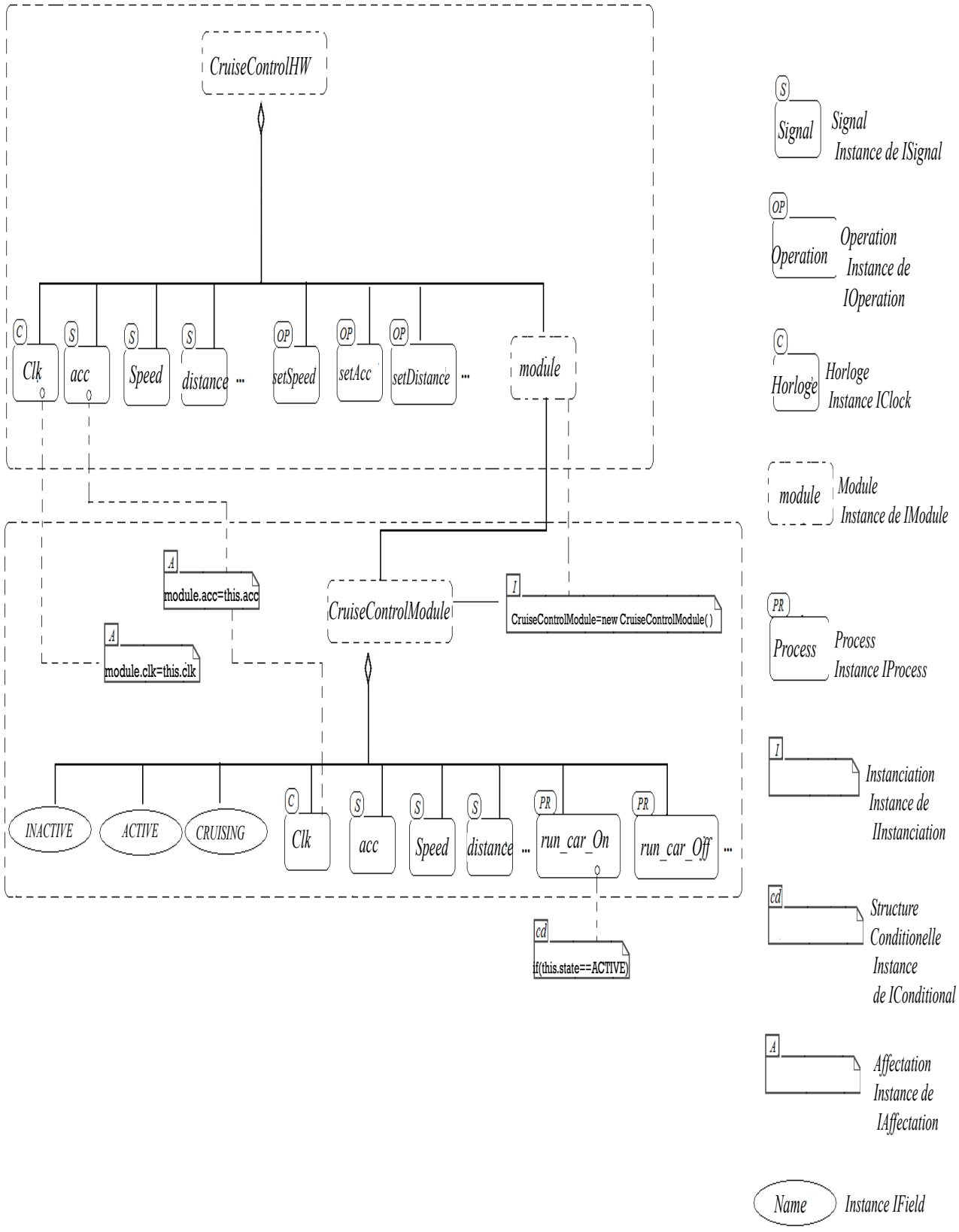

FIG. 7.2 – *Modélisation du régulateur de vitesse avec le méta-modèle Esys.net*

En commençant par le niveau d'abstraction le plus élevé, on retrouve le *CruiseControlHW* qui représente le module conteneur et qui constitue une instance de la classe *IModule* dans le méta-modèle Esys.net, figure 6.9. Le *CruiseControlHW* contient un autre module *CruiseControlModule*, qu'il crée en utilisant une instance de la classe *IInstanciation*. Le *CruiseControlHW* comporte un ensemble de constituants :

- l'horloge *Clk* qui constitue une instance de la classe *IClock* dans le méta-modèle Esys.net ;
- un ensemble de signaux de type différents (*Double*, *Int*, etc.) qui permettent au *CruiseControlHW* l'échange de données avec son sous-module *CruiseControlModule*. Cet échange se fait par une affectation de données entre les deux signaux appartenant aux deux modules. Cette affectation est une instance de la classe *IAffectation* ;
- un ensemble d'opérations correspondant aux méthodes orientées objet et retournant des valeurs ;
- le sous-module *CruiseControlModule* comporte également un ensemble de composantes :
  - les signaux, *ISignal* utilisés pour le transfert de données ;
  - les constantes (*INACTIVE*, *ACTIVE*, *CRUISING*) correspondant aux champs d'une classe et utilisées dans quelques structures conditionnelles ;
  - les processus, *run-Car-On* et *run-Car-Off* qui sont des instances de la classe *IProcess* et qui permettent aux modules d'agir au sein de l'application. certains processus contiennent des structures conditionnelles qui sont des instances de la classe *IConditionnel* ;
  - les événements (*transitive*, *sensitive*) instances de la classe *IEvent* sont rattachés aux signaux.

Nous avons restreint les composantes, présentées dans la figure 7.2 à celles nécessaires à la compréhension de ce chapitre.

7.3 La génération de code Esys.net

La figure ci-dessous représente l'intégration du générateur Esys.net à la structure des classes générateurs. La figure 7.3 reprend la structure de la figure 5.6 :

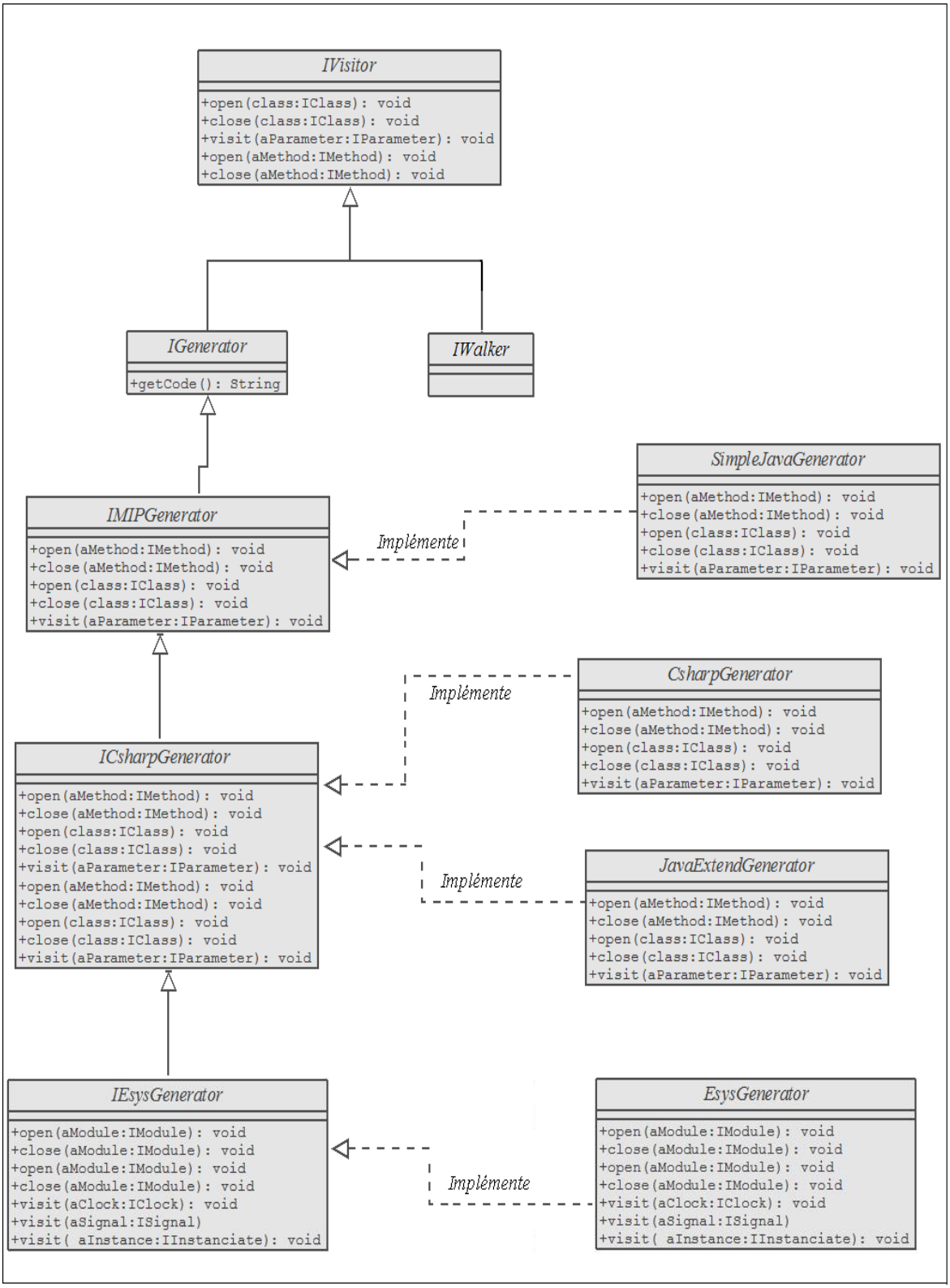

FIG. 7.3 – *Schéma UML représentant l'intégration du générateur Esys.net aux classes générateurs*

- l'interface *IEsysGenerator* inclut des méthodes de génération pour tous les constituants du méta-modèle Esys.net, tel que les signaux, les horloges, les processus, etc. L'interface *IEsysGenerator* hérite de *ICsharpGenerator* et englobe ainsi les méthodes définies dans les interfaces *IVisitor*, *IGenrator*, *IMIPGenerator*, *ICsharpGenerator*. Cet héritage est nécessaire pour l'utilisation des constituants de méthode tels que les affectations, les instanciations, les structures conditionnelles et itératives, etc. dans le corps des opérations ;
- comme pour les interfaces *IVisitor*, *IMIPGenerator* et *ICsharpGenerator*, l'interface *IEsysGenerator* ne définit que les noms des méthodes, leurs signatures et leur type de retour, mais l'implémentation concrète de chaque méthode est définie dans la classe qui l'implémente ;
- la classe *EsysGenerator* implémente l'interface *IEsysGenerator*, c'est-à-dire qu'elle donne une implémentation concrète à toutes les méthodes définies dans l'interface selon les besoins de développement et la nature du code généré.

Le tableau de la figure 7.3 représente une déclaration partielle du modèle de régulateur de vitesse et le code Esys.net lui correspondant, généré automatiquement. Le code généré correspond au code implémenté par le concepteur de l'application du régulateur de vitesse [Gir05]. Cependant, le générateur de code ne reconnait dans le modèle visité que les structures du méta-modèle Esys.net ce qui représente une limite au niveau de l'implémentation de tous les aspects de l'application. Une version complète du régulateur de vitesse est présentée en l'annexe de ce mémoire, `http://ptidej.dyndns.org/Members/tagmouty/`.

Création du modèle du régulateur de vitesse	*Commentaires*	*Code Esys.net généré automatiquement*
IModule module1 = esysfactory.getEsysInstance(). createModule("CruiseControlHW"); Conteneur.addActor(module1);	*- Création du module CruiseControlHW qui représente le module principal* *- Ajout du module CruiseControlHW au module Conteneur représente*	public class CruiseControlHW:SytemModel { private DoubleSignal acc=new DoubleSignal();
IClock clock=esysfactory.getEsysInstance(). createClock("clk", 10); module1.addActor(clock); clock.setPrivate(true);	*-Création de l'horloge clk avec un temps de déclenchementde 10 unités* *- Ajout de l'horloge au module conteneur CruiseControlHW* *- Définition de l'horloge avec des modificateurs d'accès privé*	private Clock clk = new Clock(10);
ISignal sig5=esysfactory.getEsysInstance(). createSignal("Car_On", "Int"); module1.addActor(sig5); sig5.setPrivate(true);	*- Création du signal acc de type Int* *- Ajout du signal au module conteneur CruiseControlHW* *- Définition du signal avec un modificateur d'accès privé*	private IntSignal Car_On = new IntSignal();
IAffecte affecte=esysfactory. getEsysInstance(). createAffectation("module",clock, "this.clk"); const1.addEsysActor(affecte);	*- Création d'une affectation de l'horloge clk de la valeur du signal du même nom appartenant au sous-module CruiseControlModule* *- Ajout de l'affectation au constructeur CruiseControlHW*	module.clk = this.clk;

Lors de la modélisation du système du régulateur de vitesse nous avons d'une part reproduit l'application du régulateur de vitesse dans le cadre d'un modèle représentant une spécialisation du méta-modèle Esys.net et d'autre part utiliser le générateur Esys.net afin de générer le même code que celui implémenter par les concepteurs de l'application du régulateur de vitesse.

7.4 Avantages d'utilisation du système

La description des systèmes logiciels et matériels complexe fait de leur programmation une grande difficulté, afin de faciliter la description d'un système, le niveau d'abstraction doit d'être suffisamment élevée. L'utilisation de notre système offre au concepteur la possibilité d'implémenter son système à un niveau d'abstraction plus élevé que s'il avait à implémenter son code à la main. Le but est qu'un concepteur puisse se concentrer sur le rassemblement de différents modules composants son système et leurs interactions et utiliser le générateur de code afin de générer le code correspondant à ses modules sans se soucier des détails d'implémentation, telle que les variables et les relations entre elles. De manière générale, l'utilisation de notre système présente plusieurs avantages :

- il permet une modélisation explicite des patrons de conception, où le mécanisme d'application du patron est indépendant du modèle du patron lui-même. Ce type de représentation donne à l'utilisateur la possibilité de manipuler le modèle du patron explicitement ;
- il permet également la spécialisation d'un modèle de patron construit, afin de l'adapter à un contexte particulier ;
- il inclut trois méta-modèles (MIP, MIP$\mathcal{C}\sharp$ et Esys.net) qui peuvent être utilisés ensemble ou de manière indépendante. Ainsi la modélisation des patrons peut inclure à la fois des constituants Java, $\mathcal{C}\sharp$ et Esys.net dépendamment des besoins de modélisation ;
- il inclut également des générateurs de code qui permettent de générer trois types de code (Java, $\mathcal{C}\sharp$ et Esys.net) et qui peuvent générer le code corres-

pondant aux constituants d'un ou plusieurs méta-modèles utilisés.

CHAPITRE 8

CONCLUSION ET TRAVAUX FUTURS

8.1 Conclusion

Dans le but d'apporter notre contribution à un effort d'exploitation plus large des patrons de conception, dans d'autres domaines de l'informatique que le génie logiciel, nous avons travaillé en collaboration avec des membres de l'équipe LASSO qui travaillent en particulier sur la conception de systèmes logiciels et matériels, dans l'un des volets du projet Esys.Net. Ce projet a pour objectif la création d'un environnement de modélisation et de simulation basé sur le langage de programmation orientée objet $\mathcal{C}\sharp$ et exploitant la puissance de la technologie .net. Esys.net. Cet environnement prendra en charge la modélisation et simulation des systèmes matériels/logiciels, afin de faciliter à l'utilisateur le passage d'une description d'un modèle abstrait à la réalisation du prototype correspondant. Le volet du projet Esys.Net dans lequel nous intervenons, a pour objectif de faire correspondre des patrons de conception orientés objet à des architectures matérielles. Dans le cadre de cette collaboration, nous avons offert à cette représentation matérielle/logicielle un environnement de modélisation afin de prendre en charge la représentation des patrons et, d'autre part, d'assurer leurs implémentations en générant automatiquement du code Java, $\mathcal{C}\sharp$ et Esys.net à partir de chaque modèle de patron.

Afin d'atteindre notre objectif : (1) nous avons étendu le méta-modèle PADL dédié à la modélisation des patrons en un méta-modèle plus riche, MIP, afin de modéliser les différents aspects des patrons de conception orientés objet ; (2) Nous avons ensuite étendu le méta-modèle MIP en un méta-modèle plus riche MIP$\mathcal{C}\sharp$ incluant les constituants du langage $\mathcal{C}\sharp$; (3) nous avons construit 12 modèles de patrons du catalogue du GOF, basés sur les méta-modèles MIP et MIP$\mathcal{C}\sharp$. Afin d'assurer l'implémentation des patrons modélisés, dans la deuxième partie (4) nous avons implémenté trois générateurs de code indépendants, mais reliés : (i) *Simple-*

JavaGenerator donne l'équivalent en code Java aux constituants propres au langage Java, visités dans le modèle du patron ; (ii) *CsharpGenerator* qui donne l'équivalent en code $\mathcal{C}\sharp$ aux constituants propres au langage $\mathcal{C}\sharp$, visités dans le modèle du patron (iii) *JavatoCsharpGenerator* qui est une fusion des deux générateurs précédents est capable de reconnaitre des constituants $\mathcal{C}\sharp$ et leur donné une équivalence en Java. Afin d'intégrer les constituants du langage Esys.net à notre environnement, (5) nous les avons structurés dans le cadre d'un méta-modèle Esys.net .Après la construction du méta-modèle Esys.net (6) nous avons proposé une transformation de modèle basée sur les rôles entre les constituants des méta-modèles Esys.net et MIP, afin de permettre la construction d'un générateur de code tel que les précédents pour généré du code Esys.net. Après la construction du méta-modèle Esys.net et du générateur Esys.net (7) nous les avons mis en pratique sur un réel cas d'étude qui se résume à une application d'un régulateur de vitesse. Enfin, (8) nous avons reproduit l'application du régulateur de vitesse en utilisant les constituants du méta-modèle Esys.net et généré le code Esys.net lui correspondant.

Dans la mesure où nous avons pu assurer la modélisation et implémentation des patrons de conception orientés objet de manière générale, intégrer les constituants du langage Esys.net dans notre environnement et construit le générateur de code Esys.net, dans le but de permettre la modélisation et implémentation des patrons de conception résultant de la correspondance matérielle logicielle, nous avons atteint les objectifs que ne nous sommes assignés aux début de notre travail.

8.2 Travaux futurs

Nous avons défini nos travaux futurs pour chacune de nos contributions :

1. Nous avons étendu le méta-modèle PADL avec le méta-modèle MIP et étendu ce dernier avec le méta-modèle MIP$\mathcal{C}\sharp$: Le méta-modèle MIP a permis de donner une représentation de la solution d'un patron orienté objet au niveau structurel et comportemental. Cependant, le méta-modèle MIP ne permet pas la description de tous les aspects reliés à un patron, de la même manière

qu'on peut le faire avec un langage orienté objet. Nous pouvons par la suite enrichir les méta-modèles MIP et MIP$\mathcal{C}\sharp$ afin de permettre la description de tous les aspects d'un patron de conception dans une application mettant en pratique plusieurs patrons de conception.

2. Construction de 12 modèles de patrons du catalogue du GOF basés sur les méta-modèles MIP et MIP$\mathcal{C}\sharp$: dans un travail futur nous pourrons modéliser d'autres patrons de conception que ceux définit dans le catalogue du GOF et ce afin de vérifier davantage la capacité de description du méta-modèle MIP et MIP$\mathcal{C}\sharp$.
3. En ce qui concerne les générateurs de code construits, leur évolution suivra l'évolution des méta-modèles leur correspondant, car les générateurs donnent l'équivalent en code aux constituants des méta-modèles. Dans la mesure où les codes générés ne sont pas compilables, mais peuvent être intégrés à une application en développement, l'évolution des générateurs de code générera du code prêt à la compilation.
4. La construction du méta-modèle Esys.net a permis de structurer ses constituants. La bibliothèque Esys.net est une nouvelle bibliothèque développée en interne de l'équipe LASSO et est toujours en évolution. Dans la mesure où Esys.net n'a pas atteint sa dernière version, le méta-modèle Esys.net pourra toujours être amélioré dans le cadre de travaux futurs.
5. Nous avons établi des correspondances entre le méta-modèle MIP et Esys.net en nous basant sur leurs rôles, mais nos choix de transformation ne sont qu'une proposition et ne sont pas les seuls choix possibles. Il peut y avoir d'autres choix, selon la nature de l'application en développement.
6. L'évolution de Esys.net exigera également d'autres choix de transformation de nouveaux constituants. En effet, Esys.net est toujours en développement et n'a toujours pas atteint sa dernière version.

BIBLIOGRAPHIE

[AACG02] Hervé Albin-Amiot, Pierre Cointe, and Yann-Gaël Guéhéneuc. un méta-modèle pour coupler application et détection des design patterns. In *ICPC '06 : Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 181–190. LMO Langage et modèle à objets, 2002.

[BFVY96] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Syst.J.*, 35(2) :151–171, 1996.

[BM05] Ghizlane El Boussaidi and Hafedh Mili. Les patrons de conception : Représentation et mise en oeuvre. *Langages et Modèles à Objets. Conference No11, Berne , SUISSE*, 2005.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[CM08] Luc Charest and Aboulhamid El Mostapha. Matchingdesignpatterns-conceptswith hardware concepts. 2008.

[Ede99] Amnon H. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, University of Tel Aviv, 1999.

[GAJM00] Sunyé Gerson, Le Guennec Alain, and Jézéquel Jean-Marc. Design patterns application in uml. pages 44–62. 2000.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.

[Gir05] Bruno Girodias. Une plateforme pour le raffinement des services d'os pour les systèmes embarqués, 2005.

[HJY] Amnon H, Eden Joseph, and Gil Amiram Yehudai. Automating the application of design patterns.

[JSQK07] Weixing Ji, Feng Shi, Baojun Qiao, and Muhammad Kamran. The design of a novel object-oriented processor : Oomips. In *ASAP*, pages 198–203, 2007.

[Lab98] Jean J. Labrosse. *Microc/OS-II*. R & D Books, 1998.

[LAN+04] J. Lapalme, E. M. Aboulhamid, G. Nicolescu, L. Charest, F. R. Boyer, J. P. David, and G. Bois. Esys.net : a new solution for embedded systems modeling and simulation. *SIGPLAN Not.*, 39(7), 2004.

[Mar04] Raphaël Marvie. Vers des patrons de métamodélisation. structuration de métamodèles par séparation des préoccupations. *Technique et Science Informatiques*, 23(10) :1355–1382, 2004.

[Mic] Microsoft. Ecma and iso/iec csharp and common language infrastructure standards.

[OQC] JGeorg Odenthal and Klaus Quibeldey-Cirkel. *Using patterns for design and documentation*. Springer Berlin / Heidelberg.

[Pas02] Rapicault Pascal. *Modèles et techniques pour spécifier, développer et utiliser un framework : une approche par méta-modélisation*. PhD thesis, Université de Nice, 2002.

[Pre94] Wolfgang Pree. Meta patterns - a means for capturing the essentials of reusable object-oriented design. In *ECOOP '94 : Proceedings of the 8th European Conference on Object-Oriented Programming*, pages 150–162, London, UK, 1994. Springer-Verlag.

[RBF00] Pascal Rapicault and Mireille Blay-Fornarino. Instanciation et vérification de patterns de conception : un méta protocole. In *LMO*, pages 43–58, 2000.

[RiC01] Rajeev R. Raje and ivakumar Chinnasamy. elelepus - a language for specification of software design patterns. In *SAC '01 : Proceedings of the 2001 ACM symposium on Applied computing*, pages 600–604, New York, NY, USA, 2001. ACM.

[R.J] R.Johnson. How to develop frameworks.

[SSM01] Luc Séméria, Koichi Sato, and Giovanni De Micheli. Synthesis of hardware models in c with pointers and complex data structures. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(6) :743–756, 2001.

[WS] Jimmy Wales and Larry Sanger. Wikipédia.

[WWJ96] Muller Wolfgang, Rosenstiel Wolfgang, and Ruf Jurgen. *SystemC*. 1996.