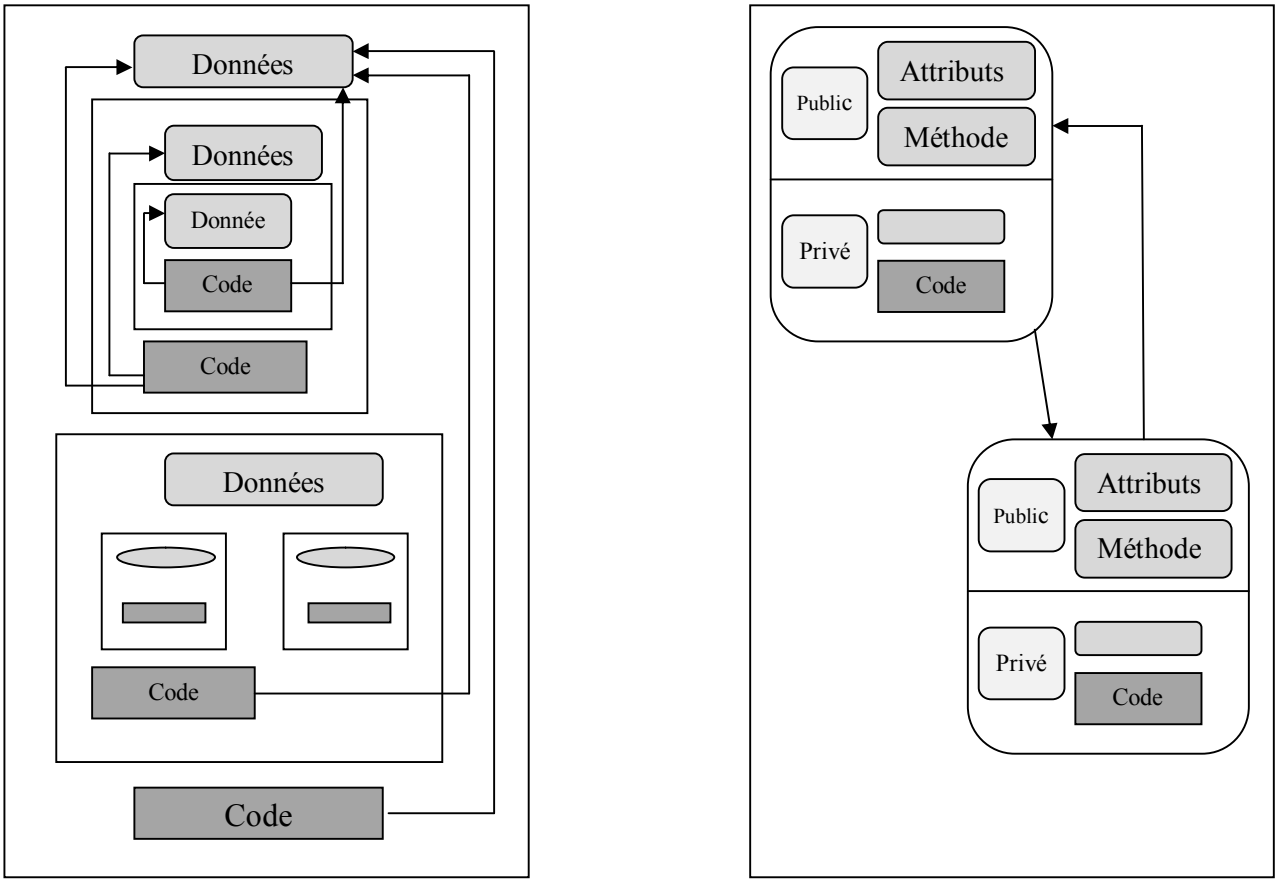
I.1 Introduction [1]

La programmation classique ou procédural telle que le débutant peut la connaitre à travers des langages de programmation comme Pascal, C etc.… traite les programmes comme un ensemble de données sur les quelles agissent des procédures. Les procédures sont les éléments actifs et importants, le donnés devenant des éléments passifs qui traversent l'arborescence de programmation procédurale en tant flot d'information.

Cette manière de concevoir les programmes reste proche des machines de Von Neuman et consiste en dernier ressort à traiter indépendamment les données et les algorithmes (traduit par des procédures) sans tenir compte des relations qui les lient.

En introduisant la notion de modularité dans la programmation structurée descendante, l'approche diffère légèrement de l'approche habituelle de la programmation algorithmique classique. Nous avons défini des machines abstraites qui ont une autonomie relative et qui possèdent leurs propres structures de données, la conception d'un programme relative dés lors essentiellement de la description des interactions que ces machines ont entre elles [1].

La programmation orientée objet relève d'une conception ascendante définie comme des « **messages** » échangés par des entités de base appelées objets.



Programmation procédure **Programmation objet**

Figure I.1 Comparaison des deux topologies de programmation

Les langages objets sont fondés sur la connaissance d'une seule catégorie d'entité informatique : l'objet.

Dans un objet, traditionnellement ce sont les données qui deviennent prépondérantes. On se pose d'abord la question : « de quoi parle-t-on ? » et non pas la question « que veut-on faire ? », comme en programmation algorithmique. C'est en ce sens que les machines abstraites de la programmation structurée modulaire peuvent être considérées comme pré-objets.

En fait la notion TAD (type abstrait de données) est utilisée dans cet ouvrage comme spécification d'un objet, en ce sens nous nous préoccupons essentiellement des services offerts par un objet indépendamment de sa structure interne.

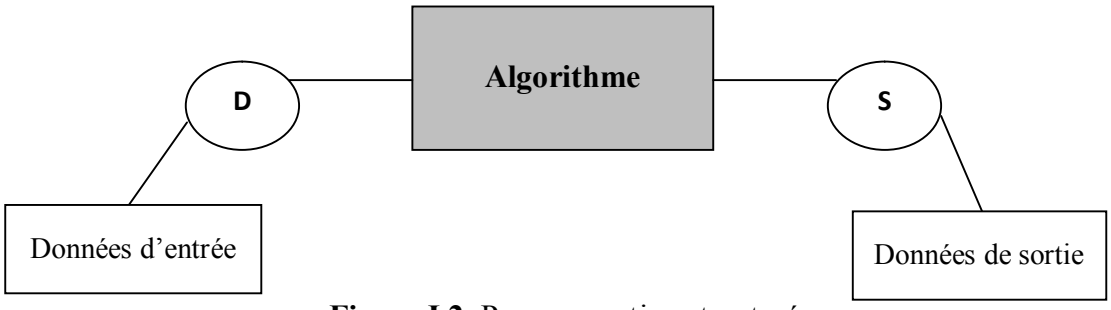


Figure I.2 Programmation structurée

I.2 Eléments de l'approche orientée objets

I.2.1 Objets

I.2.1.1 Nature d'un objet

Pour un être humain, un objet est :

- ✓ une chose visible ou tangible ;
- ✓ une chose qui peut être comprise intellectuellement;
- ✓ quelque chose vers qui une action est dirigée.

Un objet modélise une partie de la réalité qui existe dans le temps et dans l'espace. De plus, un objet peut être un mécanisme grâce auquel d'autres objets peuvent interagir. De manière plus formelle, en analyse orientée **objets,** un objet est défini comme suit [2] :

Un item individuel identifiable, réel ou abstrait, ayant un rôle bien défini dans le domaine du problème en traitement.

Dans certaines applications, les objets peuvent avoir des frontières bien définies et faciles à identifier. Par contre, dans d'autres applications, cette frontière est plus floue. Ceci nous amène à l'énoncé suivant ; décrivant encore mieux un objet du point de vue informatique [2] :

Un objet possède un état, un comportement et une identité; la structure et le comportement d'objets semblables sont définis par leur classe commune; le terme instanciation et objet sont synonymes.

Ainsi : **Objet = Etat + Comportement + Identité**

I.2.1.2 Etat d'un objet

Un objet possède généralement des **attributs** (un attribut est une information qui décrit l'objet) qui ont une valeur donnée à un moment donné dans le temps.

Ainsi [2] :

L'état d'un objet comprend toutes les propriétés d'un objet (qui sont habituellement statiques) de même que la valeur présente (habituellement dynamique) de chacune de ces propriété

En général, il est conseillé d**'encapsuler** l'état d'un objet plutôt que de l'exposer aux accès directs d'objets clients. En Java et en C++, cela signifie que l'état d'un objet devrait être gardé privé.

I.2.1.3 Comportement d'un objet

Les objets n'existent pas isolément mais interagissent plutôt avec d'autres objets en démontrent un comportement donné. Cette constatation conduit à la définition suivante [2] :

Le comportement d'un objet décrit comment celui-ci change d'état à la réception de messages d'autres objets et comment il transmet lui-même des messages aux autres objets.

En d'autres mots, le **comportement** d'un objet décrit son **activité externe visibl**e. Lorsqu'un **objet** effectue une **action** sur un autre objet, on dit qu'il lui transmet un message. En Java ou en C++, le passage de message signifie qu'un objet appelle une fonction membre (en programmation orientée objets, une "fonction membre" est aussi appelée "méthode") d'un autre. Lorsqu'un objet reçoit un message, celui-ci peut le faire changer d'état. On peut donc ajouter dans ce contexte que: l'état d'un objet représente l'effet cumulatif de son comportement.

En général, les messages sont passés via divers types de fonctions membres [2] :

- les **modificateur**s **:** fonctions membres qui peuvent modifier l'état de l'objet

pour lequel elles sont appelées ;

- les **sélecteurs :** fonctions membres qui lisent l'état d'un objet sans le modifier ;
- les **itérateurs :** mécanisme qui permet de visiter toutes les parties d'un objet dans un ordre défini ;

Il y a évidemment deux autres types de fonctions membres connues :

- les **constructeur**s **:** créent et initialisent un objet ;
- les **destructeur**s **:** libèrent l'espace occupé par un objet (et son état) et détruisent ensuite l'objet lui-même.

En général, les **fonctions membres** d'un objet forment son **protocole** d'interaction avec le monde extérieur.

I.2.1.4 Identité d'un objet

L'identité d'un objet permet de faire distinction entre les objets de même type. On définit l'identité d'un objet comme [2] :

L'identité d'un objet est cette propriété qui le distingue de tous les autres objets.

Il ne faut pas ici confondre le nom de l'objet avec l'objet lui-même. Le nom de l'objet n'est qu'un identificateur qui permet d'accéder à ce dernier.

I.2.1.5 Relations entre objets

La relation entre chaque couple d'objets renferme l'hypothèse que chacun connaît l'autre, y compris les opérations qui peuvent être effectuées et le comportement qui en résulte. Deux sortes de hiérarchies d'objets sont d'un intérêt particulier dans l'approche orientée objets : les relations de liens (ou utilisation) et les relations d'agrégation.

A- Relation d'utilisation

Un **lien** est une **connexion physique** ou **conceptuelle** entre des objets. Un objet coopère avec d'autres objets grâce aux liens qu'il possède avec ceux-ci. Un lien est généralement unidirectionnel (mais pas obligatoirement). Un message est

habituellement initié par un client et est dirigé selon un lien vers le serveur. De manière plus complète, un participant dans un lien peut jouer trois rôles:

- **acteur (ou client) :** objet pouvant agir sur d'autres objets mais sur qui les autres objets ne peuvent agir (il initie une interaction) ;
- **serveur :** objet qui ne peut jamais agir sur les autres objets mais sur qui les autres objets peuvent agir (il est la cible de messages) ;
- **agent :** objet qui peut agir sur les autres objets et sur qui les autres objets peuvent agir (il combine les caractéristiques d'un client et d'un serveur).

B- Relation d'agrégation

Alors que les liens dénotent une relation client/fournisseur, la contenance (ou agrégation) correspond à une hiérarchie ensemble/composant. Les agrégats dénotent une façon différente d'associer les objets. Par exemple, un objet peut en contenir un autre qui fait partie de son état. L'objet contenant peut informer le monde extérieur de ce qu'il contient et l'objet contenu peut éventuellement informer le monde extérieur qu'il appartient à un contenant ; si son interface le lui permet et si cette information est contenue dans son propre état (C'est d'ailleurs cet aspect qui différencie l'agrégat du simple lien). L'agrégation ne signifie cependant pas exclusivement l'inclusion physique mais peut être conceptuelle.

I.2.2 Classes

I.2.2.1 Nature d'une classe

Un objet est un individu appartenant à une **classe.** La création d'un objet à partir de sa classe est appelée instanciation (on dit que l'objet est une instance d'une classe...). Une **classe** est définit comme étant [2] :

Un ensemble d'objets partageant une structure et un comportement communs.

Une classe décrit donc les caractéristiques générales d'un ensemble d'objets. Une classe comprend généralement:

- une **interface** qui permet l'interaction des instances de cette classe avec les autres objets du problème (la vision externe que la classe donne d'elle-même),

elle décrit le domaine de définition et les propriétés des instances de cette classe ;

- une **implantation** réalisant l'interface (le comportement interne de la classe), elle contient le corps des opérations et les données nécessaires à leur fonctionnement.

L'interface d'une classe est divisée en trois parties [3,2] :

- **publique (public) :** une déclaration accessible à tous les clients ;
- **protégée (protected) :** une déclaration qui n'est accessible qu'à la classe, à ses amies et à ses sous-classes ;
- **privée (private) :** une déclaration qui n'est accessible qu'à la classe et à ses amies.

I.2.2.2 Relations entre classes

Les langages de programmation orientée objets supportent plusieurs types de relations entre classes à savoir : l'association, l'héritage, l'inclusion et l'utilisation [3,2].

A- Association

L'**association** signifie que des classes sont en relation mais que l'on n'a pas encore choisi leur niveau plus spécifique de relation ou encore que l'on ne désire que montrer leur association sans plus de détails.

B- Héritage

L'**héritage** permet de spécialiser une classe en dérivant une autre classe dont les propriétés sont plus spécifiques que celles de la classe dont elle dérive et qui ajoute aussi certaines fonctionnalités à la classe mère. Certaines propriétés peuvent demeurer inchangées et sont donc partagées par les deux niveaux de la hiérarchie. On dit que la classe de base est une **superclasse** et que la classe dérivée est une **sous-classe**. Certains langages comme le C++ admettent l'**héritage multiple** (c'est-à-dire qu'une classe peut hériter de deux superclasses distinctes). L'héritage multiple peut causer de sérieux problèmes que le Java a préféré éviter en permettant seulement l'héritage

simple et en offrant la possibilité de créer des interfaces (le mot réservé interface en Java).

C- Inclusion

L'**inclusion** (aussi appelée **agrégation** ou **composition)** est une relation entre les classes qui est parallèle à la relation d'agrégats entre les objets. Elle permet l'expression des relations de type : « maître et esclave », « une partie de », « composé de » etc. Dans l'agrégation l'une des classes est plus importante que l'autre.

D- Utilisation

L'**utilisation** est une relation qui survient quand une classe possède une (ou des) fonction membre dont l'un des arguments est une référence à un objet d'une autre classe.

I.2.3 Rôle des classes et des objets

Classes et objets sont des concepts séparés et pourtant intimement liés. Plus précisément, chaque objet est l'instance d'une certaine classe, et chaque classe a zéro ou plusieurs instances. Pour presque toutes les applications, les classes sont statiques ; par conséquent, leur existence, leur sémantique et leurs relations sont fixées préalablement à l'exécution d'un programme. De même, la classe de la plupart des objets est statique, ce qui signifie que, dès qu'un objet est créé, sa classe est fixée. Tout au contraire, les objets sont couramment crées et détruits à un rythme élevé durant la vie d'une application [2].

Au cours de l'analyse et au début de la phase de conception d'un projet, le développeur doit accomplir deux tâches principales :

- identifier les **classes** et les **objets** qui forment le vocabulaire du domaine du problème;
- inventer les **structures** par lesquelles des ensembles d'objets travaillent en coopération pour atteindre les comportements nécessaires à la solution du problème (mécanismes de mise en œuvre).

I.2.4 Classification

La **classification** est un moyen d**'ordonner les connaissance**s. En conception orientée objets, la reconnaissance de la similitude entre les choses permet d'exposer le caractère commun à l'intérieur des abstractions et les mécanismes et conduit à des architectures plus simples. Il n'y a malheureusement pas de recette pour identifier les classes et les objets.

L'identification des classes et des objets est la partie la plus difficile de l'analyse et de la conception orientée objets. L'expérience montre que cette classification peut être trouvée grâce à la découverte et l'invention. La découverte permet de reconnaître les abstractions et mécanismes qui forment le vocabulaire du problème. Par l'invention, on peut ensuite concevoir des abstractions plus générales et des mécanismes nouveaux qui permettent aux objets de collaborer. Le principal problème de la classification est qu'il y a autant de classifications que de raisons pour établir une classification! La classification dépend souvent de la façon dont on aborde un problème [2].

Quoiqu'il en soit, la conception d'une classification intelligente est un travail intellectuel difficile et la solution est atteinte de manière **incrémentale** et **itérativ**e. Ce développement incrémental de la classification a un impact direct sur la construction des classes et des hiérarchies dans la conception de systèmes complexes. En pratique, il est donc courant de choisir une certaine structure de classe assez tôt dans la phase de conception pour ensuite la réviser. Ce n'est que rendu à un certain moment dans la conception, et plus spécialement lorsque des clients ont utilisé la classe, qu'il est possible d'évaluer la qualité de la classification. Suite à cette évaluation, il est alors courant de décider de créer de nouvelles classes à partir de classes existantes, de séparer une grosse classe en plus petites classes (factoriser les classes), ou de créer une classe à partir de plus petites classes (composer une classe) [2] .

I.3 Modélisation orientée objets

L'orientée objets est une technique de modélisation des systèmes. Ainsi, si une modélisation orientée objets porte ce nom avec succès, c'est qu'elle doit effectivement modéliser n'importe quel type de système sous forme d'objets. La modélisation par objets utilise les **classes** et les **objets** comme **blocs** de base.

I.3.1 Eléments de la modélisation par objets

La modélisation par objets comporte quatre éléments principaux : l'abstraction, l'encapsulation, la modularité et la hiérarchie. Sans cette ossature conceptuelle, le programme n'est pas orienté objets même si le langage de programmation est orienté objets.

I.3.1.1 Abstraction

Une abstraction est une représentation des caractéristiques essentielles d'un objet qui permettent de le distinguer de tous les autres. Une abstraction s'intéresse à l'apparence extérieure d'un objet et permet de séparer le comportement essentiel de l'objet de son implantation. C'est ce qu'on appelle la "barrière de l'abstraction"[3,2].

Il existe plusieurs types d'abstractions:

- **abstraction descriptive (d'identité) :** un objet qui représente un modèle utile d'une composante du domaine du problème et de sa solution ;
- **abstraction d'action :** un objet qui offre un ensemble d'opérations générales dont chacune effectue le même type de fonction ;
- **abstraction de machine virtuelle :** un objet qui regroupe des opérations qui sont toutes utilisées par un niveau de contrôle supérieur, ou les opérations qui utilisent toutes un ensemble d'opérations d'un niveau plus élémentaire ;
- **abstraction fortuite (de coïncidence) :** un objet qui contient un ensemble d'opérations qui n'ont aucun lien entre elles.

Ainsi : **L'abstraction se concentre sur les caractéristiques essentielles d'un objet selon le point de vue de l'observateur** [2]**.**

I.3.1.2 Encapsulation

Une abstraction devrait d'abord être conçue indépendamment de son implantation. L'implantation doit généralement demeurer secrète et ne doit que refléter l'abstraction désirée. En résumé, aucune partie d'un système complexe ne devrait dépendre des détails internes d'un autre. L'**abstraction** et l'**encapsulation** sont deux concepts complémentaires. L'**abstraction** concerne la **description du comportement extérieur** d'un objet tandis que l'**encapsulation** vise à **implanter** (mettre en œuvre) cette description ou ce comportement.

L'encapsulation est le procédé de séparation des éléments d'une abstraction qui constituent sa structure et son comportement. Elle permet de diviser l'interface contractuelle de la mise en œuvre d'un objet. L'encapsulation occulte les détails (secrets) de mise en œuvre d'un objet [2].

I.3.1.3 Modularité

Dans des projets logiciels d'envergure, l'utilisation de **modules** est essentielle pour gérer la complexité.

La modularité est la capacité qu'a un système d'être décomposé en un ensemble de modules cohérents et faiblement couplés [3,2,4]**.**

Dans des langages comme le C++ et C#, les classes et les objets forment la structure logique d'un système, les modules contiennent les abstractions décrites par ces classes et forment l'architecture physique du système. Pour des systèmes incluant des milliers de classes, les modules sont un moyen de traiter la complexité.

La modularité regroupe les entités en unités discrètes.

La création de modules vise à **regrouper des classes pouvant être compilées séparément mais qui sont reliées à des classes contenues dans d'autres modules.** Les liens entre les modules sont les hypothèses que les modules font les uns sur les autres. Un module, comme une classe, comprend une interface et une implantation. **Modularité et encapsulation sont donc des concepts très voisins.**

Il faut remarquer que la division d'un système en modules est aussi difficile que de décider de sa division en abstractions. Une bonne division est très avantageuse. D'ailleurs, le but général de la modularisation vise à réduire le coût du logiciel en permettant de concevoir et de réviser des modules de manière indépendante. La structure de chaque module doit être assez simple pour être comprise facilement. Il faut également qu'il soit possible de modifier un module sans connaître ni affecter la structure des autres modules.

I.3.1.4 Hiérarchie

L'**abstraction** est une bonne chose mais il est souvent difficile de maîtriser toutes les abstractions à cause de leur nombre. L'**encapsulation** permet de gérer partiellement la complexité. La **modularité** aide aussi en regroupant les abstractions qui sont reliées. Mais cela n'est pas suffisant. En effet, les abstractions forment aussi une **hiérarchi**e [2].

On définit une hiérarchie comme étant un classement ou ordonnancement des abstractions.

Il existe deux types importants de hiérarchies :

- les **hiérarchies d'existence ou de classe** ("est un") ;
- les **hiérarchies d'appartenance ou d'objets** ("partie de").

Un premier exemple de hiérarchie est **l'héritag**e. Ce dernier dénote une relation "est un" d'une abstraction. Une **abstraction** au bas d'une hiérarchie spécialise une abstraction plus générale. Un second exemple de hiérarchie est **l'appartenance** ("partie de") pour laquelle une abstraction **fait partie** d'une autre abstraction. Alors que les hiérarchies "est un" désignent des relations de généralisation/spécialisation, les hiérarchies "partie de" décrivent des relations d'agrégation.

I.3.2 Phases de développement orienté objets d'un système

Les différentes phases du développement d'un système, d'un point de vue orientée objets sont : l'analyse orientée objets, la conception orientée objets et la programmation orientée objets.

I.3.2.1 Analyse orientée objets

L'**analyse orientée objets** (AOO ou OOA : Object Oriented Analysis) à pour but la compréhension du système que l'on doit développer et l'élaboration d'un modèle logique du système. Ce modèle est basé sur des objets naturels issus du domaine d'application. Ces objets contiennent des données et ont leurs propres comportements à partir desquels on peut exprimer le comportement du système entier. Ainsi l'AOO est définie comme suit :

L'AOO est une méthode d'analyse qui examine les besoins d'après la perspective des classes et objets trouvés dans le vocabulaire du domaine du problème (domaine d'application).

I.3.2.2 Conception orientée objets

La **conception orientée objets** (COO ou OOD : Object Oriented Design) signifie que le modèle d'analyse est conçu, elle met l'accent sur la structuration appropriée et efficace d'un système complexe. Ainsi :

La COO est une méthode de conception incorporant le processus de décomposition orientée objets et une notation permettant de dépeindre à la fois les modèles logiques/physiques et statiques/dynamiques du système à concevoir [2]**.**

I.3.2.3 Programmation orientée objets

La **programmation orientée objets** (POO ou OOP : Object Oriented Programming) utilise les objets et non les algorithmes comme blocs fondamentaux, chaque objet est une instance d'une certaine classe et les classes sont reliées l'une à l'autre par des relations d'héritage. Si l'un de ces éléments fait défaut, ce ne sera pas

un programme orienté objets. Plus précisément, programmer sans héritage n'est pas orienté objets. Ainsi :

La POO est une méthode d'implantation par laquelle les programmes sont organisés en un ensemble d'objets coopératifs, chaque objet représentant une instance d'une classe, chaque classe faisant partie d'une hiérarchie de classes unies par des relations d'héritage [4]**.**

A la lumière de cette définition, certains langages de programmation sont orientés objets et d'autres ne le sont pas. Un langage est orienté objets si et seulement si il répond aux conditions suivantes [3,2,4]:

- il supporte des objets qui sont des abstractions de données avec une interface d'opérations nommées et un état interne caché ;
- les objets ont un type associé (la classe) ;
- les types (les classes) peuvent hériter des attributs venant de super-types (les super-classes).

I.3.3 Processus de développement orienté objets

Dans ce processus, **l'analyse** et la **conception** sont étroitement liés et la frontière les séparant est floue. Le processus de développement s'intéresse aux activités suivantes :

- identifier les classes et les objets à un **niveau donné d'abstractio**n **;**
- identifier la sémantique des classes et des objets ;
- identifier les relations entre les classes et les objets ;
- spécifier l'interface et l'implantation des classes et des objets.

I.3.3.1 Identification des classes et des objets

L'identification des classes et des objets permet d'établir les bornes du problème et de décomposer celui-ci en objets. Durant la phase d**'analys**e, cette étape permet de mettre à jour les abstractions qui forment le **vocabulaire du problèm**e. Durant la phase de **conception,** cette étape permet de créer de nouvelles abstractions et même de concevoir des abstractions de bas niveau qui permettent de créer des

abstractions de niveau supérieur. En cours d'implantation, cette étape permet de découvrir des points communs entre les abstractions, ce qui contribue à simplifier l'architecture du système. Le résultat de cette étape est un **dictionnaire des données** qui est mis à jour graduellement en cours de développement. Au début, une liste des classes et des objets importants est suffisante. Dès ce moment, il convient d'utiliser des noms reflétant leur sens (sémantique).

I.3.3.2 Identification de la sémantique des classes et des objets

La sémantique d'une classe comprend son rôle, ses responsabilités de même que ses opérations. Le but de cette étape est d'établir le comportement et les attributs des abstractions identifiées à l'étape précédente. Les abstractions précédemment définies sont raffinées pour inclure une distribution mesurable de leurs responsabilités.

Durant la phase **d'analyse**, cette étape vise à allouer les responsabilités en fonction des différents comportements du système. A la phase de **conception,** cette étape permet de définir une séparation nette entre les différentes parties de la solution.

Au début du projet, les rôles peuvent être décrits en langage familier. Plus tard, ils doivent faire l'objet de spécifications précises des **protocoles** (ensemble des opérations qu'un client peut effectuer sur une abstraction) complets de chaque abstraction et de la description précise de la **signature (**l'ensemble des paramètres formels de même que le type de valeur de retour des méthodes de chaque opération) [3, 2,4].

En plus, il est aussi pertinent de définir des **diagrammes d'objet**s, des **diagrammes d'interaction** qui permettent d'établir la **sémantique des scénarios** créés. Ces diagrammes permettent de saisir de manière formelle les planches **(storyboard**s) des scénarios et témoignent de la répartition explicite des responsabilités entre les différents objets.

I.3.3.3 Identification des relations entre les classes et les objets

Durant la phase d'**analyse**, cette phase sert à identifier les associations entre les classes et entre les objets, incluant certaines relations d'**héritage** et d'agrégation.

L'existence d'une association met en évidence une dépendance sémantique entre deux abstractions de même que leur capacité à naviguer de l'une à l'autre (est-ce qu'une abstraction est seulement visible d'une autre ou se voient-elles mutuellement, etc.).

Durant la phase de **conception,** cette phase sert à spécifier les collaborations qui forment les mécanismes de l'architecture, de même que les regroupements de haut niveau de classes en catégories et des modules en sous-systèmes.

Au fur et à mesure de l'avancement de l'implantation, les relations sont raffinées jusqu'à transformer les associations simples en relations plus spécifiques comme l'instanciation (inclusion) et l'utilisation.

I.3.3.4 Implantation des classes et des objets

Durant l'analyse, le but d'**implanter** les classes et les objets est de raffiner les abstractions. Durant le design, cette phase permet de donner une représentation tangible des abstractions et de permettre le raffinement successif des versions exécutables du processus. Les produits de cette phase sont les décisions sur la façon dont les abstractions seront **implantées physiquement**. Au début, du pseudo-code est suffisant. Plus l'implantation avance, plus ce pseudo-code se transforme graduellement en code réel.

I.3.4 Méthodes orientées objets

Pour développer des systèmes logiciels orientés objets de qualité, il faut utiliser une méthode de développement. Il n'existe pas de méthode universelle ni pour l'analyse ni pour la conception. Beaucoup de méthodes traitent ces deux phases du cycle de vie d'un logiciel, d'autres ne traitent que l'analyse ou la conception. De nouvelles méthodes sont introduites chaque année pour compléter ou palier les défauts des anciennes méthodes.

Alors que les concepts orientés objets existaient depuis les années soixante, les méthodes orientée objets ne sont apparues que lors des trois dernières décennies. Ceci est du essentiellement à [3,2] :

- ✓ les concepts orientés objets demandaient du temps pour mûrir dans nos esprits;

- ✓ il était assez difficile de penser « orienté objets » tant que les langages industriellement répandus n'étaient pas orientés objets ;
- ✓ ce n'est que lors des trois dernières décennies que sont apparus les systèmes complexes de grande dimension ce qui a conduit à élaborer de nouvelles méthodes s'adaptant à ces besoins.

I.3.4.1 Méthode de conception OOD

En 1983, Grady Booch proposait une méthode de conception dite orientée objets. La deuxième version de cette méthode a été présentée en 1990 [3,2]. Cette méthode adoptait une démarche en 4 étapes pour concevoir un système :

— Identifier les classes et les objets à un niveau d'abstraction donnée.

— Identifier la sémantique de ces classes et de ces objets. Le développeur doit détailler la représentation interne des classes de manière à comprendre leur fonctionnement et leur rôle précis. Ceci permet de déterminer les interfaces de chaque classe.

— Identifier les relations entre les classes et les objets.

— Implémenter ces classes et ces objets.

Booch précise clairement qu'il ne définit pas une méthode d'analyse, mais uniquement une méthode de conception qui peut être utilisée en aval d'une méthode d'analyse.

I.3.4.2 Méthode d'analyse OOA

La technique d'analyse par objets proposée par Coad et Yourdon est une tentative d'incorporation de meilleures idées proposées. OOA utilise de manière explicite l'héritage pour la mise en commun des attributs et des services. La démarche pour obtenir un modèle OOA se compose de cinq activités principales [3]:

— Trouver les classes et les objets.

— Identifier les structures.

— Identifier les sujets.

— Définir les attributs.

— Définir les services.

I.3.4.3 Méthode d'analyse et de conception OMT

La méthode OMT (Object Modeling Technique), proposée par Jumes Rumbaugh et Mechael Blaha, s'applique à tous les processus de développement d'un logiciel, de l'analyse à l'implantation [3]. Cette méthode utilise trois vues différentes, chacune capturant les aspects importants du logiciel, ces trois vues sont :

— Le modèle objet qui représente l'aspect statique d'un logiciel (définitions des classes, relations d'héritages, d'agrégation,…).

— Le modèle dynamique qui présente le comportement du logiciel au cours du temps.

— Le modèle fonctionnel qui prend en compte l'aspect fonction de transformation du logiciel.

Chacun de ces modèles contient des références aux entités des autres modèles, ils ne sont donc pas complètement indépendants. La méthodologie proposée est indépendante des langages de programmation et utilise une notation graphique uniforme pour toutes les phases. Les trois modèles séparent un système en un ensemble de vues qui sont manipulées et qui évoluent tout au long du cycle de développement (analyse, conception et implémentation pour OMT). Le but de la construction du modèle objet OMT est de fournir le cadre de travail essentiel dans lequel les modèles dynamiques et fonctionnels vont se placer. Les objets sont considérés comme des composants de base qui doivent capturer les éléments de réalité que l'analyste considère comme importants pour une application [3].

Le modèle dynamique décrit les aspects temporels, les séquences d'opérations et les évènements qui entraînent des changements d'état au sein d'une classe. Le rôle du modèle dynamique est donc de présenter les différents aspects de contrôle du système. Ceci se traduit au niveau de la notation graphique par des diagrammes d'état et des séquences d'événement.

Le modèle fonctionnel décrit les transformations apportées par le système sur les fonctions réalisées et ceci sans se préoccuper de la manière dont cela est réalisé ni quand cela intervient. Le modèle fonctionnel est représenté avec des diagrammes de flots de données montrant les dépendances entre les données en entrée et celles en sortie des processus de traitement chargés de réaliser les fonctions précises du logiciel.

La méthode OMT semble relativement complète pour aborder une large catégorie de problèmes.

I.3.4.4 Méthode Objectory d'Ivar Jacobson

La méthode Objectory (ou OOSE : Object Oriented Software Engineering) est une autre méthode qui aborde aussi bien l'analyse que la conception des systèmes de taille importante. Elle couvre tout le cycle de développement d'un logiciel et propose un processus de développement qui produit cinq modèles [3]:

— modèle des besoins ;
— modèle d'analyse ;
— modèle de conception ;
— modèle d'implémentation ;
— modèle de test.

I.4 Conclusion

L'approche orientée objets s'est révélé applicable à une large variété de domaines d'applications. Elle constitue peut être la seule méthode pouvant de nos jours faire face à la complexité inhérente aux très gros systèmes. L'approche orientée objets présente de nombreux avantages, ainsi que quelques risques, l'expérience montre que les avantages l'emportent de beaucoup sur les risques.

Les avantages évoqués dans les paragraphes précédents s'avèrent très important pour le développement d'applications ou systèmes réseaux électriques. Il n'existe évidemment pas de recette miracle pour la conception de tels systèmes. Cependant les mécanismes de base intervenant au cœur même du système sont connus et spécifient clairement comment doivent se dérouler les choses. En revanche, certaines décisions

dépendent du type d'application réseaux électriques elle même (analyse de répartition de charges, analyse dynamique ou analyse d'estimation d'états, etc.).

Le début de l'application de la TOO aux réseaux électriques remonte au début des années 1990. La majorité des travaux traitent l'application de l'écoulement de puissances [1,5]. On rencontre également l'application d'interface graphique utilisateur, la simulation dynamique et la restauration des réseaux électriques. Récemment, quelques travaux ont abordé le calcul générique des systèmes linéaires pour les réseaux électriques [7].

Depuis l'année 2000, les auteurs défendent l'idée de séparation entre la modélisation des éléments physiques du réseau électrique et la modélisation de ses applications. En résumé la majorité des travaux actuels optent pour le développement de trois architectures : modélisation des réseaux électriques (éléments physiques), modélisations des applications (fonctions de calculs) et modélisation des facilitées mathématiques (moyens de calcul).