INP Grenoble
ENSIMAG
Deuxième année

Analyse, Conception et Validation de Logiciels

C. Oriat
2006–2007

Introduction

Les projets logiciels sont célèbres pour leurs dépassements de budget, leurs cahiers des charges non respectés. De façon générale, le développement de logiciel est une activité complexe, qui est loin de se réduire à la programmation. Le développement de logiciels, en particulier lorsque les programmes ont une certaine taille, nécessite d'adopter une méthode de développement, qui permet d'assister une ou plusieurs étapes du cycle de vie de logiciel. Parmi les méthodes de développement, les approches objet, issues de la programmation objet, sont basées sur une modélisation du domaine d'application. Cette modélisation a pour but de faciliter la communication avec les utilisateurs, de réduire la complexité en proposant des vues à différents niveaux d'abstraction, de guider la construction du logiciel et de documenter les différents choix de conception. Le langage UML (Unified Modeling Language) est un langage de modélisation, qui permet d'élaborer des modèles objets indépendamment du langage de programmation, à l'aide de différents diagrammes.

Ce cours commence par une introduction au génie logiciel et une présentation du langage UML. Ensuite sont abordés les problèmes liés à l'analyse, la modélisation, la conception et la validation de logiciels. Ce cours est illustré par plusieurs exemples et études de cas.

Chapitre I

Génie logiciel

1. Problèmes posés par le développement de logiciels

En 1979, une étude du gouvernement américain, basée sur neuf projets logiciels, fait apparaître les symptômes suivants : beaucoup de logiciels ne sont pas livrés, pas utilisés ou abandonnés. Plus précisément, le coût se répartit de la façon suivante :

Logiciels	Coût
payés, jamais livrés	3.2 M $
livrés, jamais utilisés	2.0 M $
abandonnés ou recommencés	1.3 M $
utilisés après modification	0.2 M $
utilisés en l'état	0.1 M $

D'après cette étude, seul 5% du coût, correspondant à des logiciels utilisés soit en l'état, soit après modification, est donc « acceptable ».

On a parlé de la « crise du logiciel ».

Selon une autre étude, réalisée aux Etats-Unis en 1995, le coût se répartit de la façon suivante :

Logiciels	Coût
abandonnés ou jamais utilisés	31%
coûts largement dépassés	53%
réalisés suivant les plans	16%

La « crise du logiciel » est donc loin d'être terminée.

Nature des problèmes

Si on examine plus en détail les difficultés rencontrées dans le développement de logiciels, on peut identifier les problèmes suivants :

Besoins des utilisateurs Il est difficile d'établir et stabiliser les besoins des utilisateurs. Cela provient de plusieurs facteurs :

- Il y a des difficultés de communication entre les informaticiens, qui connaissent mal le domaine d'application, et les utilisateurs, qui ne connaissent pas les capacités et limitations des systèmes informatiques.
- Il est difficile de savoir quand une spécification est « complète ».
- L'environnement instable : lorsque le développement est long, les machines et autres dispositifs physiques peuvent changer au cours du développement, ce qui nécessite des adaptations du logiciel.

« Malléabilité » du logiciel Un petit programme bien conçu est facile à modifier. Il est beaucoup plus difficile de modifier un gros programme. D'autre part, les modifications ont tendance à dégrader la structure du logiciel. Plus un logiciel est modifié, plus il devient donc difficile à modifier. On aboutit ainsi souvent à des logiciels qu'on n'ose plus modifier, une modification (qui peut simplement être la correction d'une erreur) risquant d'introduire d'autres erreurs.

Documentation Les logiciels sont difficiles à « visualiser » : il est difficile de comprendre l'architecture ou les choix de conception d'un logiciel à partir du seul texte source du programme. Un logiciel devrait donc toujours être accompagné d'une documentation qui explique ces choix. En pratique, cette documentation est souvent insuffisante.

Complexité des logiciels La complexité des logiciels provient de plusieurs facteurs :

- les algorithmes implantés peuvent être intrinsèquement complexes ;
- le domaine d'application peut nécessiter d'élaborer des théories spécifiques. Par exemple, pour développer un logiciel de contrôle aérien, il est nécessaire de modéliser comment le contrôle aérien est organisé.
- les logiciels sont de grande taille. Un logiciel de plusieurs millions de lignes de code ne peut pas être maîtrisé par un seul programmeur.

Évolution de l'informatique au cours du temps L'informatique a beaucoup évolué au cours du temps, et évolue encore. Cette évolution se fait dans plusieurs directions :

- Il y a un élargissement des domaines d'application. Jusque vers 1960, les logiciels sont de type scientifique. Ensuite sont apparues les applications de gestion, de traitement symbolique. Aujourd'hui, l'informatique est partout, et dans des domaines critiques (centrales nucléaires, avions, voitures, marchés boursiers).
- L'évolution technologique des ordinateurs s'accompagne d'une chute du prix des machines et d'une croissance de leur puissance. Ces deux facteurs impliquent une croissance des exigences des utilisateurs, de la complexité des logiciels et donc du coût des logiciels. Cela implique qu'il y a peu d'espoir de résoudre la crise du logiciel : les techniques de génie logiciel ont toujours du retard sur la technologie.

2. Génie logiciel

Définition

On peut définir le génie logiciel de la façon suivante :

> *« Le génie logiciel est l'ensemble des activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi. »*

Autrement dit, c'est l'art de produire de bons logiciels, au meilleur rapport qualité prix.

On peut remarquer que cette définition fait référence d'une part au processus de développement des logiciels (les « procédures »), et d'autre part à la maintenance des logiciels (le « suivi »).

Critères de qualité du logiciel

L'objectif étant de produire de « bons logiciels », il faut expliciter les critères de qualité d'un logiciel. Ces critères peuvent être rangés en deux catégories : les critères externes et internes.

Critères externes

Les critères externes expriment ce qu'est un bon logiciel du point de vue des utilisateurs. Un logiciel de qualité doit être :

- fiable (conforme aux spécifications),
- robuste (ne plante pas),
- efficace (espace mémoire, temps d'exécution),
- convivial (facile et agréable à utiliser),
- documenté (accompagné d'un manuel utilisateur, d'un tutoriel ou d'un cours).

Critères internes

Les critères de qualité internes expriment ce qu'est un bon logiciel du point de vue du développeur. Ces critères sont essentiellement liés à la maintenance d'un logiciel : un bon logiciel doit être facile à maintenir, et pour cela doit être :

- documenté (document de conception),
- lisible (écrit proprement, en respectant les conventions de programmation),
- portable sur d'autres plates-formes (la durée de vie d'un logiciel est, la plupart du temps, largement supérieure à la durée de vie d'une machine),
- extensible (ajout possible de nouvelles fonctionnalités),
- réutilisable (des parties peuvent être réutilisées pour développer d'autres logiciels similaires).

Catégories de logiciels

Logiciels amateurs Il s'agit de logiciels développés par des « amateurs » (par exemple par des gens passionnés ou des étudiants).

Logiciels « jetables » ou « consommables » Il s'agit de logiciels comme les traitements de texte ou les tableurs. Ces logiciels ne coûtent pas très cher, et peuvent être remplacés facilement au sein d'une entreprise. Ils sont souvent largement diffusés.

Logiciels essentiels au fonctionnement d'une entreprise

Logiciels critiques Il s'agit de logiciels dont l'exécution est vitale, pour lesquels une erreur peut coûter très cher ou coûter des vies humaines. Exemple : transport, aviation, médecine.

L'objectif de qualité d'un logiciel est différent suivant la catégorie de logiciel. En particulier, les logiciels essentiels au fonctionnement d'une entreprise, et plus encore les logiciels critiques, doivent avoir un haut niveau de qualité.

3. Les étapes du cycle de vie du logiciel

Le développement de logiciel impose d'effectuer un certain nombre d'étapes.

a) Analyse et définition des besoins

L'étape d'analyse et définition des besoins consiste à déterminer les attentes des futurs utilisateurs, par exemple avec un cahier des charges. Il faut décrire à la fois le système et l'environnement dans lequel le système sera exécuté. Cette étape comprend également une étude de faisabilité de la part des experts. C'est sur la base de cette étape que le contrat est signé.

b) Analyse et conception

L'étape d'analyse et conception consiste à analyser, spécifier et effectuer les choix de conception du système. Cette étape comporte plusieurs sous-étapes.

Spécification du système

La spécification du système est une description des fonctionnalités. Il s'agit de décrire ce que le système doit faire, sans préciser comment ces fonctionnalités seront implémentées.

Conception de l'architecture

La conception de l'architecture consiste à décrire la structure générale du système.

Conception détaillée

La conception détaillée du système consiste en une description des composants, des algorithmes et des structures de données.

c) Mise en œuvre

L'étape de mise en œuvre consiste à programmer le logiciel, en suivant les choix effectués lors de l'analyse et la conception.

d) Validation

La validation consiste à s'assurer que le programme est de qualité. Il existe plusieurs techniques de validation :

- analyse statique (typage, conventions de programmation, détection d'erreurs pouvant survenir à l'exécution) ;
- preuve formelle (coûteuse, peu utilisée) ;
- revue de code (efficace) ;
- tests. Les tests constituent la principale méthode de validation. On distingue les tests unitaires, les tests d'intégration, les tests système et les tests d'acceptation.

e) Évolution et maintenance

L'étape d'évolution et de maintenance consiste à effectuer des modifications du logiciel après sa livraison. On distingue plusieurs types de maintenance :

- maintenance corrective (correction de défauts),
- maintenance évolutive (ajout de nouvelles fonctionnalités),
- maintenance adaptative (portage sur une nouvelle plate-forme).

Répartition de l'activité

Si on exclut l'analyse et la définition des besoins, la maintenance représente la plus grande part du coût du logiciel.

Étape	Pourcentage
Développement initial	40%
Maintenance	60%

Hors maintenance et analyse des besoins, l'activité se répartit comme suit :

Étape	Pourcentage
Analyse et conception	40%
Mise en œuvre	20%
Validation	40%

On peut donc remarquer que globalement, la programmation ne représente qu'environ 8% de l'effort dans le développement de logiciel. D'autre part, la validation représente environ deux fois plus de travail que la programmation.

4. Modélisation du processus de développement

Un processus de développement permet de décrire l'enchaînement des différentes étapes du développement. L'objectif est de proposer un processus qui permet de contrôler le développement, afin que le logiciel :

- soit livré dans les délais ;
- respecte le budget ;
- soit de qualité.

Les modèles du cycle de vie du logiciel sont des « plans de travail » qui permettent de planifier le développement. Plus le logiciel à développer est complexe (taille, algorithmes) et critique, plus il est important de bien contrôler le processus de développement et plus les documents qui accompagnent le logiciel doivent être précis et détaillés.

a) Modèle du cycle de vie en cascade

Dans le modèle en cascade (cf. figure I. 1), on effectue les différentes étapes du logiciel de façon séquentielle. Les interactions ont lieu uniquement entre étapes successives : on s'autorise des retours en arrière uniquement sur l'étape précédente. Par exemple, un test ne doit pas remettre en cause la conception architecturale.

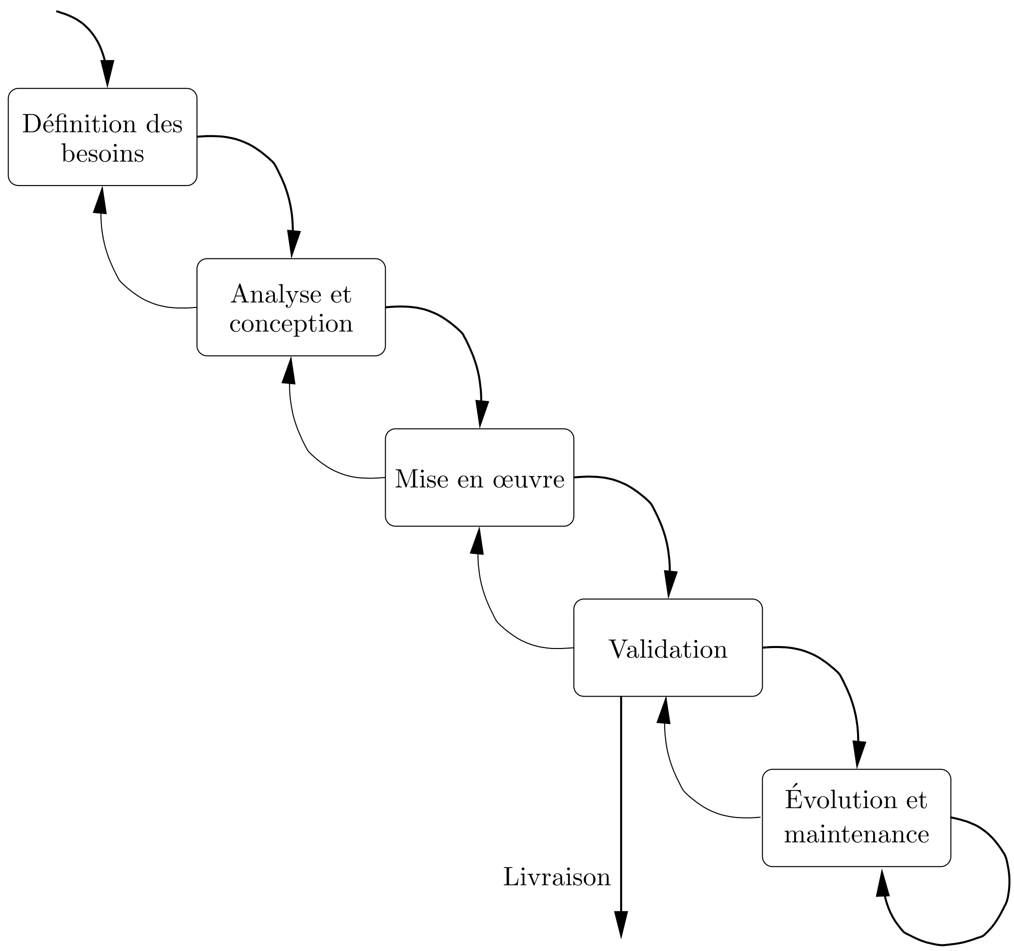


Fig. I. 1 – Modèle du cycle de vie en cascade

Modèle incontrôlable

Lorsque qu'un test remet en cause la conception détaillée ou architecturale ou, pire, les spécifications ou la définition des besoins, le modèle devient incontrôlable (cf. figure I. 2).

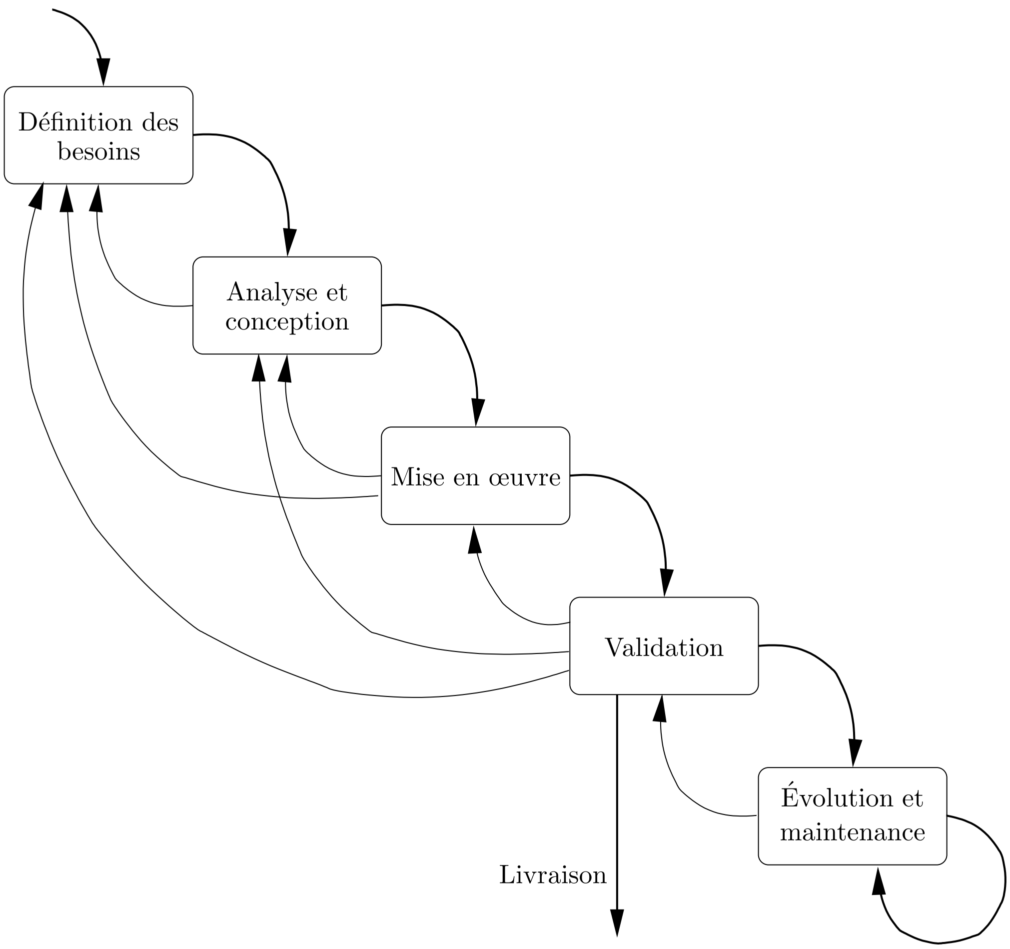


Fig. I. 2 – Modèle incontrôlable

b) Modèle du cycle de vie en V

Le modèle en V du cycle de vie du logiciel (cf. figure I. 3) précise la conception des tests : les tests système sont préparés à partir de la spécification ; les tests d'intégration sont préparés à partir de la conception architecturale ; les tests unitaires sont préparés à partir de la conception détaillée des composants.

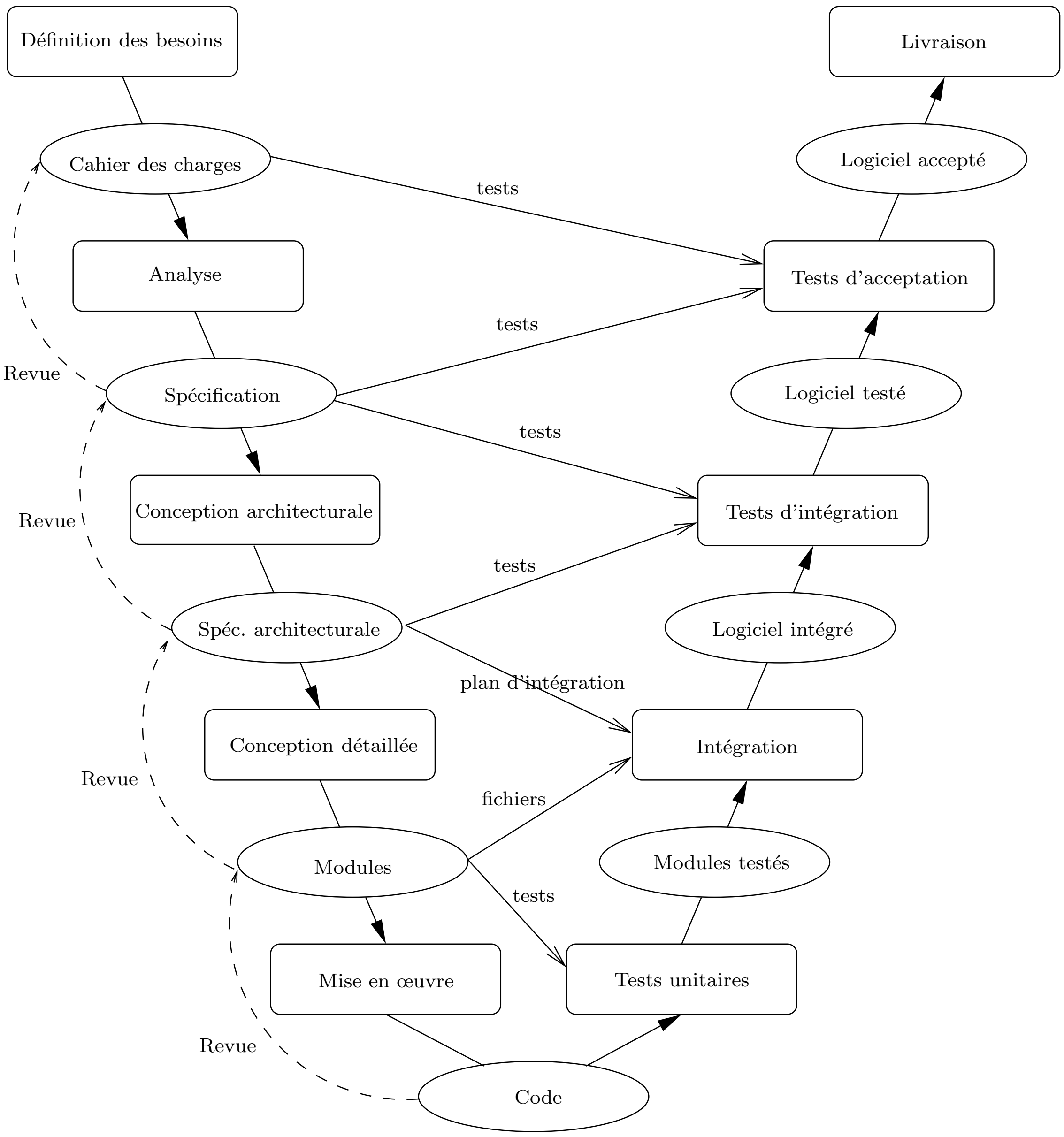


FIG. I. 3 – Modèle du cycle de vie en V

Ce modèle fait également apparaître les documents qui sont produits à chaque étape, et les « revues » qui permettent de valider les différents produits.

c) Modèle du cycle de vie incrémental

Le modèle incrémental (cf. figure I. 4) est un modèle itératif, qui consiste à sélectionner successivement plusieurs *incréments*. Un incrément du logiciel est un sous-ensemble du logiciel complet, qui consiste en un petit nombre de fonctionnalités.

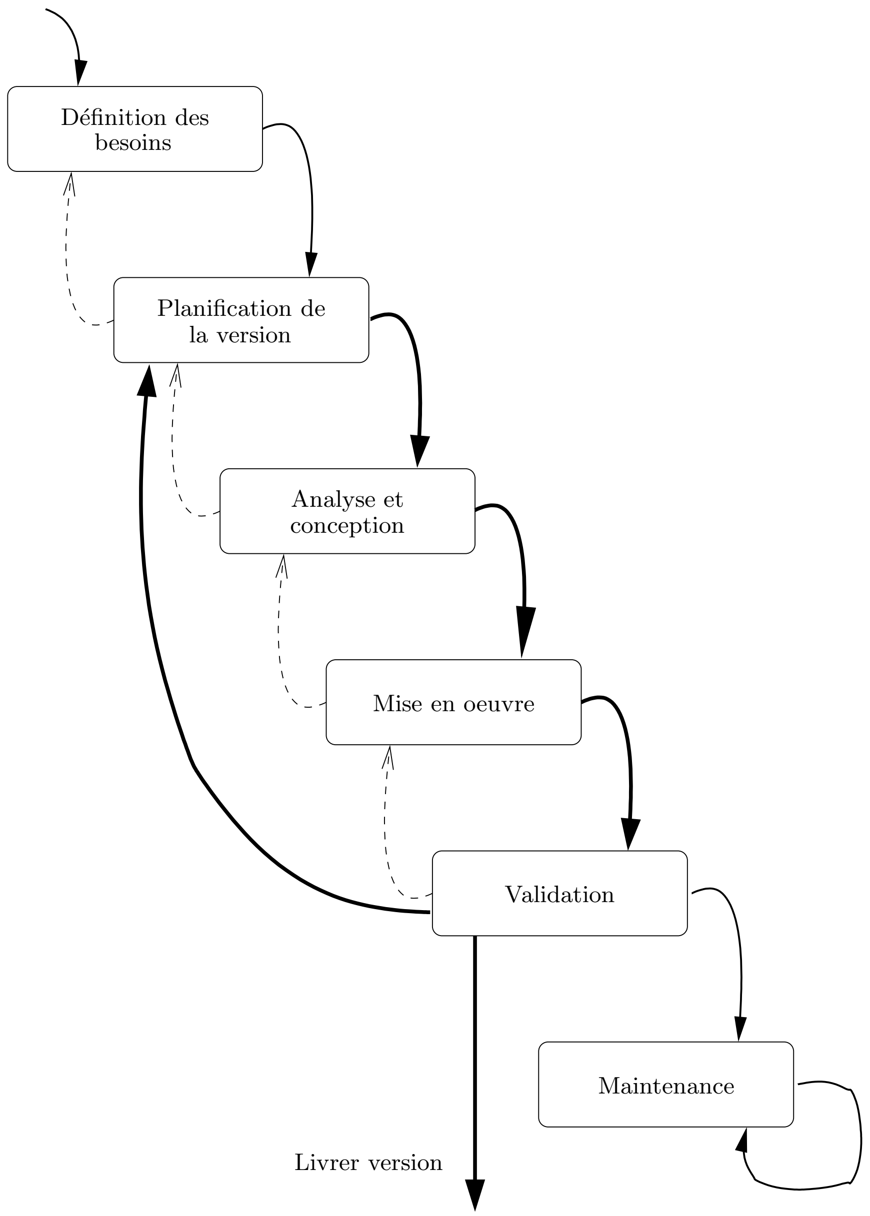


Fig. I. 4 – Modèle du cycle de vie incrémental

Pour chaque incrément, on réalise l'analyse, la conception, l'implémentation et la validation, puis on livre la version du logiciel. On recommence ainsi jusqu'à obtenir un logiciel complet. On procède donc par versions successives, chaque version donnant lieu à une livraison.

Les intérêts du modèle incrémental sont les suivants :

- ce modèle permet de révéler certains problèmes de façon précoce ;

- on a rapidement un produit que l'on peut montrer au client ;
- le client peut effectuer des retours sur la version livrée ;
- ce modèle fournit plus de points de mesure pour apprécier l'avancement du projet.

Un danger du modèle incrémental consiste à faire certains choix uniquement en fonction de l'incrément courant, et à ne pas anticiper les incréments suivants.

d) Modèle du cycle de vie en spirale

Le modèle du cycle de vie en spiral est un modèle itératif, où la planification de la version se fait selon une analyse de risques. L'idée est de s'attaquer aux risques les plus importants assez tôt, afin que ceux-ci diminuent rapidement.

Risques liés au développement de logiciels

De façon générale, les risques liés au développement de logiciels peuvent être répartis en quatre catégories :

- les risques commerciaux (placement du produit sur le marché, concurrence) ;
- les risques financiers (capacités financières suffisantes pour réaliser le produit) ;
- les risques techniques (la technologie employée est-elle éprouvée ?) ;
- les risques de développement (l'équipe est-elle suffisamment expérimentée ?).

Supposons qu'une équipe doive développer un logiciel comportant une interface graphique. Un exemple de risque pourrait être un manque de compétence de l'équipe de développement dans l'écriture d'interfaces. Dans ce cas, il faut tenir compte d'un temps de formation, et d'un retard possible dans le développement de l'interface. Dans processus de développement en spiral, on s'attaque au développement de l'interface assez tôt, afin de détecter le plus rapidement possible les problèmes éventuels, et d'avoir le temps d'y remédier.

Dans un cycle de vie en cascade, les activités qui comportent le plus de risques ont lieu à la fin, lors de la mise en œuvre et de la validation (cf. figure I. 5).

En s'attaquant dès le début aux activités les plus risquées, le modèle en spirale permet d'accélérer la réduction du risque au cours du développement du logiciel : les risques sont importants lors des premières itérations, et diminuent lors des dernières itérations (cf. figure I. 6).

5. Méthodes d'analyse et de conception

a) Méthode de développement

Une méthode définit une démarche en vue de produire des résultats. Par exemple, les cuisiniers utilisent des recettes de cuisine, les pilotes déroulent des check-lists avant de décoller, les architectes font des plans avant de superviser des chantiers. Une méthode permet d'assister une ou plusieurs étapes du cycle de vie du logiciel. On distingue les méthodes fonctionnelles, basées sur les fonctionnalités du logiciel, et les méthodes objet, basée sur différents modèles (statiques, dynamiques et fonctionnels). Les méthodes ont évolué suivant les langages et les techniques : les méthodes fonctionnelles ont pour origine la programmation structurée ; les méthodes objet ont pour origine les langages à objets.

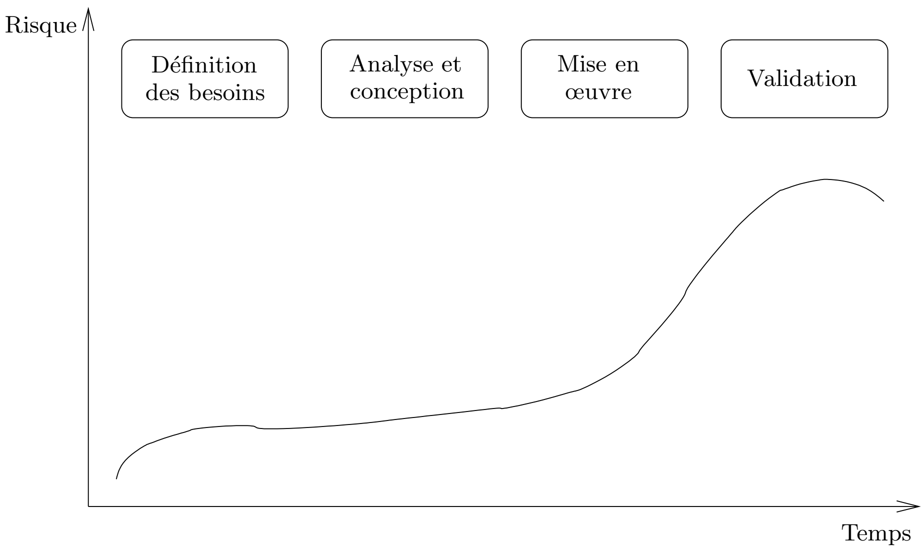


FIG. I. 5 – Évolution des risques dans le modèle en cascade

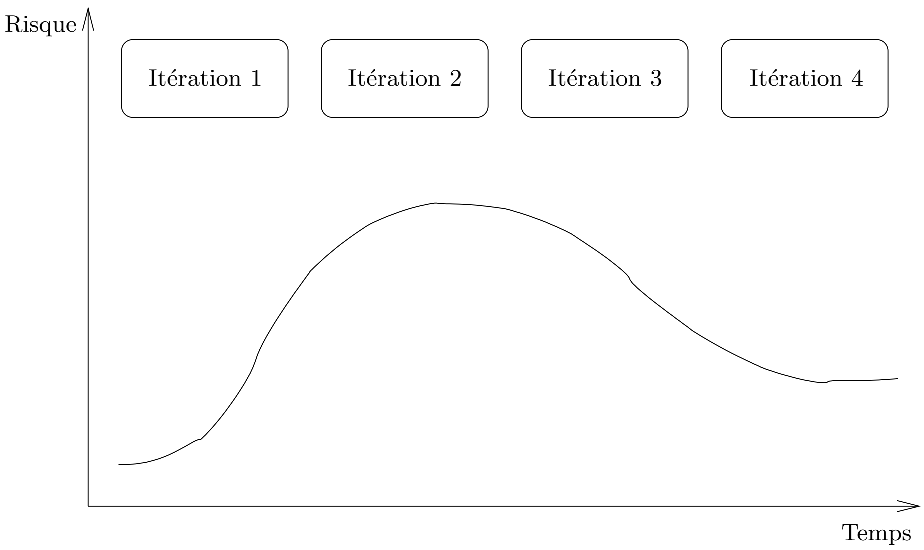


FIG. I. 6 – Réduction des risques grâce au modèle en spirale

b) Méthodes fonctionnelles

Les méthodes fonctionnelles ont pour origine la programmation structurée. Cette approche consiste à décomposer une fonctionnalité (ou fonction) du logiciel en plusieurs sous fonctions plus simples (cf. figure I. 7). Il s'agit d'une conception « top-down », basée sur le principe « diviser pour mieux régner ». L'architecture du système est le reflet de cette décomposition fonctionnelle. La programmation peut ensuite être réalisée soit à partir des fonctions de haut niveau (développement « top-down »), soit à partir des fonctions de bas niveau (développement « bottom-up »).

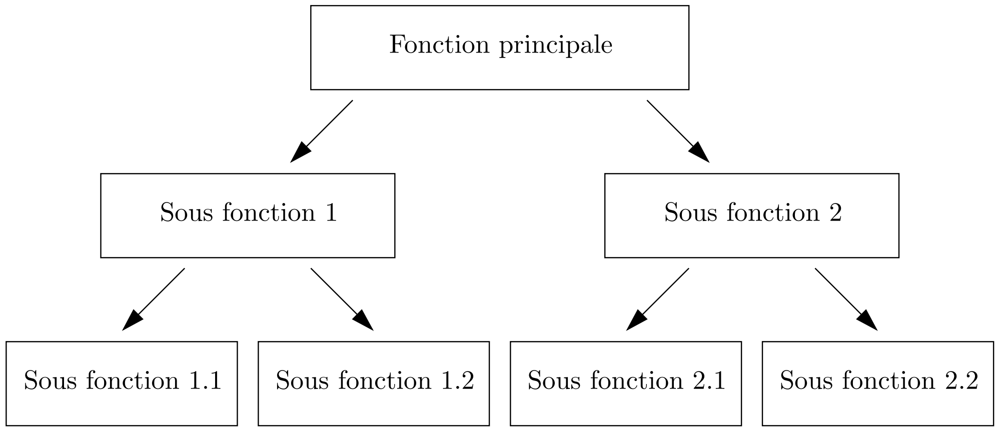


Fig. I. 7 – Décomposition fonctionnelle

Cette méthodes a les inconvénients suivants :

- L'architecture étant basée sur la décomposition fonctionnelle, une évolution fonctionnelle peut remettre en cause l'architecture. Cette méthode supporte donc mal l'évolution des besoins.
- Cette méthode ne favorise pas la réutilisation de composants, car les composants de bas niveau sont souvent *ad hoc* et donc peu réutilisables.

c) Méthodes objet

Les approches objet sont basées sur une modélisation du domaine d'application. Les « objets » sont une abstraction des entités du monde réel. De façon générale, la modélisation permet de réduire la complexité et de communiquer avec les utilisateurs. Plus précisément un modèle :

- aide à visualiser un système tel qu'il est ou tel qu'on le souhaite ;
- permet de spécifier la structure ou le comportement d'un système ;
- fournit un guide pour la construction du système ;
- documente les décisions prises lors de la construction du système.

Ces modèles peuvent être comparés avec les plans d'un architecte : suivant la complexité du système on a besoin de plans plus ou moins précis. Pour construire une niche, on n'a pas besoin de plans, pour construire un chalet il faut un plan, pour construire un immeuble, on a besoin d'un ensemble de vues (plans au sol, perspectives, maquettes).

Dans les méthodes objet, on distingue trois aspects :

- un aspect statique, où on identifie les objets, leurs propriétés et leurs relations ;
- un aspect dynamique, où on décrit le comportements des objets, en particuliers leurs états possibles et les événements qui déclenchent les changements d'état ;
- un aspect fonctionnel, qui, à haut niveau, décrit les fonctionnalités du logiciel, ou, à plus bas niveau, décrit les fonctions réalisées par les objets par l'intermédiaire des méthodes.

Intérêts des approches objet

Les intérêts des approches objet sont les suivants.

- Les approches objet sont souvent qualifiées de « naturelles » car elles sont basées sur le domaine d'application. Cela facilite en particulier la communication avec les utilisateurs.
- Ces approches supportent mieux l'évolution des besoins que les approches fonctionnelles car
  - la modélisation est plus stable,
  - les évolutions fonctionnelles ne remettent par l'architecture du système en cause.
- Les approches objet facilitent la réutilisation des composants (qui sont moins spécifiques que lorsqu'on réalise une décomposition fonctionnelle).

6. Méthodes adaptatives

a) Opposition entre méthodes prédictives et méthodes adaptatives

On distingue les méthodes prédictives et les méthodes adaptatives.

Les méthodes *prédictives*, qui correspondent à un cycle de vie du logiciel en cascade ou en V, sont basées sur une planification très précise et très détaillée, qui a pour but de réduire les incertitudes liées au développement du logiciel. Cette planification rigoureuse ne permet pas d'évolutions dans les besoins des utilisateurs, qui doivent donc être figés dès l'étape de définition des besoins.

Les méthodes *adaptatives* ou *agiles*, qui correspondent à un cycle de vie itératif, considèrent que les changements (des besoins des utilisateurs, mais également de l'architecture, de la conception, de la technologie) sont inévitables et doivent être pris en compte par la méthode de développement. Ces méthodes privilégient la livraison de fonctionnalités utiles au client à la production de documentation intermédiaire sans intérêt pour le client.

Parmi les méthodes agiles, on peut citer le processus unifié et la programmation extrême.

b) Processus unifié

Le *processus unifié* (« Unified Process ») est un processus itératif et incrémental, basé sur les besoins des utilisateurs (« piloté par les cas d'utilisation ») et centré sur l'architecture du logiciel. Le « Rational Unified Process » ou « RUP » est une spécialisation de ce processus qui a été développé par Rational.

Ce processus est basé sur un cycle de vie du logiciel en quatre phases, effectuées séquentiellement :

- Étude d'opportunité : consiste à se doter d'une vision générale du produit, et à déterminer si la construction de ce produit est économiquement justifiée (études de marché, analyse de la concurrence).

- Élaboration : consiste à définir, réaliser et valider les choix d'architecture. Cette phase commence par une analyse et une modélisation du domaine.
- Construction : consiste à développer successivement plusieurs incréments du logiciel pouvant être livrés.
- Transition : consiste à transférer le logiciel vers les utilisateurs (cette phase comporte en particulier l'installation et la formation).

Chaque phase est réalisée par une suite d'itérations (cf. figure I. 8), chaque itération comportant plusieurs étapes du cycle de vie : définition des besoins, analyse, conception, mise en œuvre et validation.

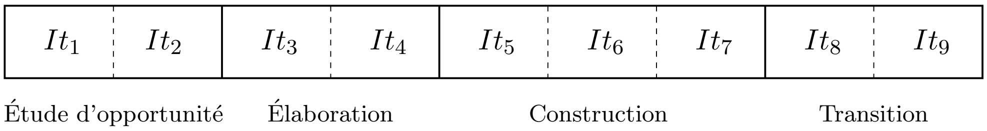


FIG. I. 8 – Phases et itérations du processus unifié

La figure I. 9 illustre les différentes phases et itérations de chaque phase. Elle montre également la quantité de travail à effectuer pour chaque phase et itération dans chaque activité du cycle de vie.

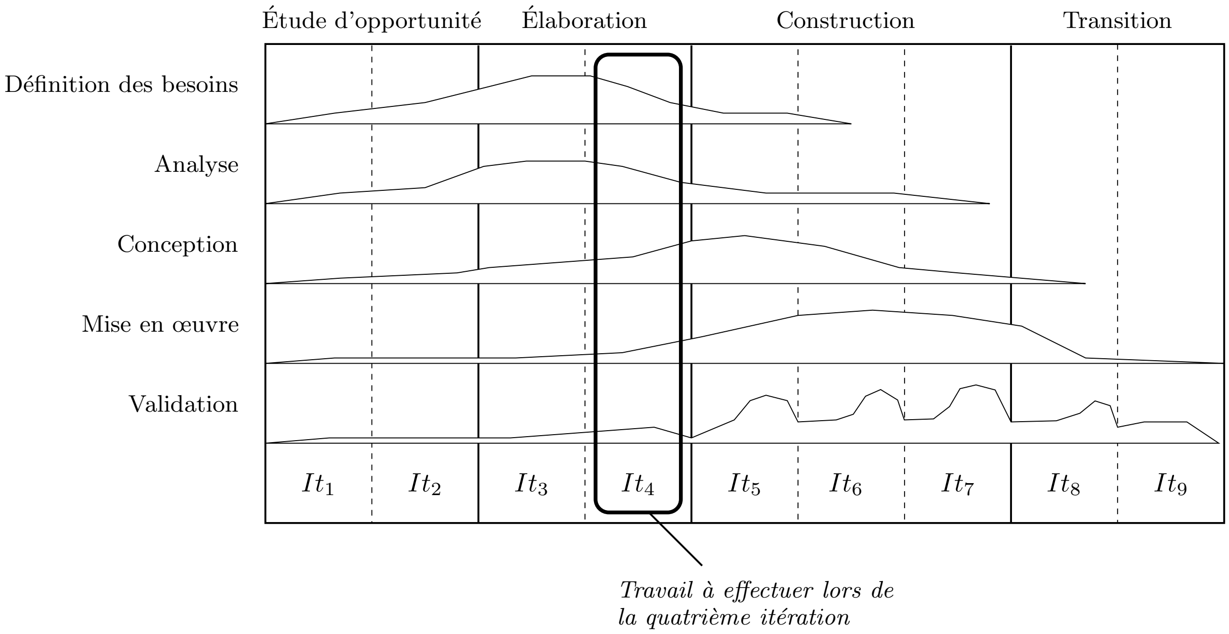


FIG. I. 9 – Quantité de travail à effectuer pour chaque activité du cycle de vie

c) Programmation extrême

Certaines méthodes nécessitent d'appliquer un grand nombre de règles, de produire beaucoup de documents (document de spécification, de conception, d'architecture, d'implémentation, de validation ... etc.) : elles sont « lourdes » et difficiles à appliquer ; le moindre changement a des répercutions importantes non seulement dans le code, mais aussi dans de nombreux documents.

La programmation extrême (« eXtreme Programming », ou « XP ») est une méthode de développement « légère », qui a pour but de produire des logiciels de qualité avec un minimum de règles et de documents à produire. Cette méthode est basée sur quatre règles essentielles :

- communication (entre développeurs, et avec les utilisateurs),
- simplicité (conception et programmation simples et claires),
- retour (prise en compte des remarques des utilisateurs),
- courage (prise en compte que le développement de logiciel est une activité complexe).

Pratiques

La programmation extrême insiste sur un certain nombre de *pratiques* qui doivent être appliquées.

Travail en équipe, avec un représentant des utilisateurs Le développement du logiciel se fait en équipe, qui inclut un représentant des utilisateurs (le « client sur site »). Le client définit les besoins, décide des priorités et conduit le projet. Il peut y avoir un chef de projet qui s'occupe de coordonner le travail.

Planning Le planning consiste à décider des fonctionnalités qui feront partie de la prochaine version livrée.

Petites livraisons À chaque itération correspond une livraison, qui correspond à une amélioration visible du logiciel. Le logiciel est également livré aux utilisateurs finaux régulièrement.

Conception simple et restructuration du code (« code refactoring ») La conception, effectuée tout au long du processus, doit rester simple. Le code est réécrit, restructuré, factorisé, afin d'améliorer sa qualité.

Programmation par paires et propriété collective du code Les programmeurs développent à deux, d'une part pour écrire un meilleur code, d'autre part pour communiquer les connaissances dans l'équipe (les paires doivent tourner). Le code appartient à toute l'équipe, ce qui signifie que toute paire de développeurs peut modifier et améliorer une portion quelconque du code. Cette modification est validée par des tests (en particulier des tests unitaires). L'équipe suit des conventions de programmation communes (le choix de conventions particulières n'étant pas crucial).

Développement piloté par les tests Les modules développés sont testés (tests unitaires). Les tests sont conservés, automatisés et ré-exécutés régulièrement (cf. JUnit). De plus, chaque livraison est accompagnée de tests d'acceptation.

Intégration continue Le code est intégré de façon continue, afin de réduire et détecter au plus tôt les problèmes d'intégration.

Métaphore L'équipe développe une vision commune du logiciel (la « métaphore »), qui permet à chaque développeur de comprendre le fonctionnement global du logiciel et ainsi de pouvoir travailler sur une partie quelconque du logiciel.

Rythme supportable Les développeurs doivent pouvoir tenir un rythme soutenu sur une longue période. Pour cela, ils doivent éviter de faire des heures supplémentaires, qui à moyen terme, réduisent leur productivité.

La programmation extrême n'impose pas la production de documentation. Le moyen privilégié de faire passer de l'information entre les membres de l'équipe est la communication (à travers la programmation par paires, les réunions, le client sur site). En programmation extrême, la clarté du code prime sur la documentation (par exemple, il n'est même pas conseillé de produire de la documentation à l'aide d'outils comme javadoc). Néanmoins, il est toujours possible de rédiger de la documentation si le besoin est réel, en particulier pour la communication externe (manuels utilisateurs, cours . . . etc.).

La programmation extrême a probablement des limites dans les projets de grande taille, pour lesquels les programmeurs ne peuvent maîtriser de grandes portions de code, même si celui-ci est particulièrement clair et lisible. Pour de tels projets, il n'est pas raisonnable de réaliser une intégration continue, ni de ré-exécuter les tests (l'intégration et l'exécution des tests peuvent en effet prendre plusieurs jours). De tels projets peuvent également être développés par plusieurs équipes sur différents sites, ce qui peut rendre la communication difficile.

Chapitre II

UML

1. Introduction

UML (Unified Modeling Language) est un langage, plus précisément une notation graphique, de modélisation à objets. UML permet de visualiser, spécifier, construire et documenter les différentes parties d'un système logiciel. Il s'agit d'un langage graphique, basé sur neuf diagrammes. UML n'est pas une méthode, mais peut être employé dans tout le cycle de développement, indépendamment de la méthode.

Initialement, les buts des concepteurs d'UML étaient les suivants :

- représenter des systèmes entiers (pas uniquement logiciels) par des concepts objets ;
- lier explicitement des concepts et le code qui les implantent ;
- pouvoir modéliser des systèmes à différents niveaux de granularité, (pour permettre d'appréhender des systèmes complexes) ;
- créer un langage de modélisation utilisable à la fois par les humains et les machines.

a) Historique

Programmation objet

Les approches objets ont pour origine la programmation objet.

1967	Simula introduit la notion de classe pour implémenter des types abstraits
1976	SmallTalk introduit les principaux concepts de programmation objet (encapsulation, agrégation, héritage)
1983	C++
1986	Eiffel
1995	Java, Ada95
2000	C#

Méthodes de développement

Les méthodes objet sont apparues après les premiers langages objet, à la fin des années 1980, après les méthodes fonctionnelles et les méthodes « systémiques », qui permettent de modéliser à la fois les données et les traitements.

1970	Premières méthodes fonctionnelles
1980	Approches systémiques : modélisation des données et des traitements (Merise)
1990–1995	Apparition d'une cinquantaine de méthodes objet (Booch, OMT, OOA, OOD, HOOD, OOSE)

UML

Suite à l'apparition d'une grande quantité de méthodes objet au début des années 1990, Booch (auteur de la méthode Booch), Rumbaugh (auteur de la méthode OMT) et Jackobson (auteur de la méthode OOSE, et des cas d'utilisation) commencent à travailler sur la « méthode unifiée » (Unified Method).

En 1996 est créé un consortium de partenaires pour travailler sur la définition d'UML dans L'OMG. Parmi les participants, on trouve Rationale, IBM, HP, Microsoft, Oracle. . ..

1995	Début du travail de Booch, Rumbaugh et Jackobson sur la méthode unifiée
1996	Création d'un consortium de partenaires pour travailler sur la définition d'UML dans l'OMG.
1997	UML 1.1 (normalisé par l'OMG)
1998	UML 1.2
1999	UML 1.3
2001	UML 1.4
2003	UML 1.5
2005	UML 2.0

OMG (Object Management Group)

L'OMG est une organisation internationale, comportant plus de 800 membres (informaticiens et utilisateurs), créée en 1989 pour promouvoir la théorie et la pratique de la technologie objet dans le développement de logiciel.

b) Concepts objet

L'idée principale de l'approche objet est de centraliser les données et les traitements associés dans une même unité, appelée objet.

Objet

Un objet est caractérisé par :

- une identité (ou un nom) ;

- un état, défini par un ensemble de valeurs d'attributs ;
- un comportement, défini par un ensemble de méthodes.

Classe

Une classe est une abstraction qui représente un ensemble d'objets de même nature (c'est-à-dire ayant les mêmes attributs et méthodes). On dit qu'un objet est une « instance » de sa classe.

Encapsulation

L'encapsulation est la possibilité de masquer certains détails de l'implantation. Ceci peut être réalisé en particulier par des constituants *privés* des objets.

Agrégation et composition

L'agrégation et la composition permettent de définir des objets composites, fabriqués à partir d'objets plus simples.

Extension

L'extension est la possibilité de définir une nouvelle classe, appelée classe *dérivée*, à partir d'une classe existante. On peut ajouter des attributs ou des méthodes. L'extension permet de définir des hiérarchies de classes.

Héritage

Une classe dérivée hérite des attributs et méthodes de la classe mère. L'héritage évite la duplication de constituants (attributs, méthodes) et encourage la réutilisation.

Concepts de programmation objet

Les concepts d'objet, de classe, de composition, d'extension... que nous venons de présenter sont communs aux approches objet et à la programmation objet.

Par contre, les notions de *redéfinition* et de *liaison dynamique* sont propres à la programmation objet.

Redéfinition de méthodes héritées

Dans une classe dérivée, certaines méthodes héritées peuvent être redéfinies (ou spécialisées).

Liaison dynamique

L'exécution d'une méthode dépend du type dynamique (type à l'exécution) d'un objet. L'intérêt de ce mécanisme est de pouvoir définir de façon uniforme des opérations qui s'appliquent à des objets de types différents, mais dérivés d'une classe commune.

Exemple des points en Java

Dans ce paragraphe, on définit une classe Point2D qui permet de définir des points en deux dimensions. Cette classe comporte une méthode distanceOrigine qui calcule la distance à l'origine d'un Point2D.

```
/**
 * Classe des points en deux dimensions.
 */
class Point2D {
    int x, y ; // abscisse et ordonnée
    /**
     * Constructeur.
     */
    Point2D(int x, int y) {
        this.x = x ;
        this.y = y ;
    }
    /**
     * Distance à l'origine d'un Point2D.
     */
    double distanceOrigine() {
        return Math.sqrt(x * x + y * y) ;
    }
}
```

Les attributs x et y sont des attributs d'instance, la méthode distanceOrigine est une méthode d'instance car elle fait référence à des attributs d'instance.

La classe Dispersion permet de calculer la dispersion d'un nuage de points autour de l'origine.

```
/**
 * Classe qui définit la dispersion d'un nuage de points.
 */
class Dispersion {
    // Il s'agit d'une classe utilitaire.
    private Dispersion() { }
    /**
     * Calcul de la dispersion autour de l'origine d'un tableau de points.
     */
    static double calc(Point2D[] tab) {
        double res = 0 ;
        for (int i = 0 ; i < tab.length ; i++) {
            res += tab[i].distanceOrigine() ;
        }
        return res ;
    }
}
```

La classe Dispersion n'est pas destinée à être « instanciée » : on ne souhaite pas créer d'instances de cette classe. Il s'agit d'une classe utilitaire, qui sert à définir des méthodes de classes. La méthode calc est une méthode de classe (introduite par le mot réservé *static* en Java).

On peut écrire un programme qui calcule la dispersion d'un tableau de deux points :

```
Point2D p1 = new Point2D(3, 4) ;
Point2D p2 = new Point2D(1, 0) ;
Point2D[] tab = {p1, p2} ;
double dispersion = Dispersion.calc(tab) ;
System.out.println("dispersion = " + dispersion) ;
```

Supposons maintenant que nous ayons besoin de manipuler des *points colorés*. On peut définir une classe Point2DColores par extension à partir de la classe Point2D. On choisit ici de coder la couleur d'un point par un entier.

```
/**
 * Classe des points colorés à deux dimensions.
 */
class Point2DColore extends Point2D {
    int couleur ; // Couleur d'un point
    /**
     * Constructeur.
     */
    Point2DColore(int x, int y, int couleur) {
        super(x, y) ; // Appel du constructeur de Point2D.
        this.couleur = couleur ;
    }
}
```

La classe Point2DColore *hérite* de Point2D, en particulier des attributs x et y et de la méthode distanceOrigine. Cela signifie qu'un Point2DColore a une abscisse et une ordonnée, et qu'on peut en calculer la distance à l'origine.

On peut par exemple écrire le programme suivant :

```
Point2DColore c = new Point2DColore(3, 4, 10) ;
System.out.println("abscisse = " + c.x) ;
System.out.println("ordonnee = " + c.y) ;
System.out.println("distance a l'origine : " + c.distanceOrigine()) ;
```

On voit qu'on peut donc utiliser les attributs x et y et la méthode distanceOrigine sur des objets de type Point2DColore.

De plus, le type Point2DColore est un *sous-type* du type Point2D, ce qui signifie qu'on peut utiliser un objet de type Point2DColore à la place d'un objet de type Point2D (c'est la propriété de *sous-typage*, appelée également propriété de *substitution*).

Par exemple, on peut passer en paramètre un tableau d'objets de type Point2DColore à la méthode Dispersion.calc.

```
Point2DColore c1 = new Point2DColore(3, 4, 10) ;
Point2DColore c2 = new Point2DColore(1, 0, 20) ;
Point2DColore[] tabC = {c1, c2} ;
double dispC = Dispersion.calc(tabC) ;
```

Supposons maintenant que l'on veuille manipuler des points à trois dimensions. On définit pour cela une classe Point3D qui étend la classe Point2D, dans laquelle on ajoute un attribut z pour la hauteur. La méthode distanceOrigine définie dans Point2D ne fournit pas un calcul correct pour les points à trois dimensions, puisqu'il faut tenir compte de la hauteur. On *redéfinit* (ou *spécialise*) donc cette méthode.

```
/**
 * Classe des points à trois dimensions.
 */
class Point3D extends Point2D {
    int z ; // Hauteur
    /**
     * Constructeur.
     */
    Point3D(int x, int y, int z) {
        super(x, y) ; // Appel du constructeur de Point2D
        this.z = z ;
    }
    /**
     * Distance à l'origine. (La méthode de Point2D est redéfinie)
     */
    double distanceOrigine() {
        return Math.sqrt(x * x + y * y + z * z) ;
    }
}
```

Supposons que l'on écrive le programme suivant :

```
Point2D p = new Point3D(0, 3, 4) ;
System.out.println("distance a l'origine : " + p.distanceOrigine()) ;
```

Dans ce programme, p a pour type statique (ou type à la compilation) Point2D. Par la propriété de sous typage, on peut affecter à p un objet d'un sous-type de point2D, donc un point à trois dimensions. Lorsqu'on exécute le programme, p a pour valeur un objet de type Point3D, donc pour type *dynamique* (ou type à l'exécution) Point3D.

Lors de l'appel p.distanceOrigine(), la méthode distanceOrigine choisie dépend du type *dynamique* de p. C'est donc la méthode de Point3D qui est appelée, et le programme affiche

```
distance a l'origine :  5.0
```

Ce choix de la méthode *suivant le type dynamique* de l'objet qui invoque la méthode (p dans l'exemple) s'appelle *la liaison dynamique*.

Ce mécanisme permet la réutilisation de code. On peut par exemple appeler la méthode Dispersion.calc sur un tableau de points à trois dimensions.

```
Point3D q1 = new Point3D(0, 3, 4) ;
Point3D q2 = new Point3D(0, 0, 1) ;
Point3D[] tab2 = q1, q2 ;
double disp2 = Dispersion.calc(tab2) ;
System.out.println("dispersion = " + disp2) ;
```

Dans cet exemple, lors de l'appel Dispersion.calc(tab2), le programme effectue l'opération

```
res += tab[i].distanceOrigine() ;
```

sur tous les éléments du tableau.

Lorsque cette instruction est exécutée, tab[i] a pour type statique Point2D et pour type dynamique Point3D. Par conséquent, c'est la méthode distanceOrigine de Point3D qui est appelée.

On remarque donc qu'on a pu réutiliser le code de Dispersion.calc pour des points à trois dimensions, alors qu'il avait été conçu au départ pour des points à deux dimensions.

c) Les différents diagrammes UML

La notation UML définit treize diagrammes, qui permettent de modéliser différents aspects d'un système. On se concentre ici sur les neuf diagrammes de UML 1.5.

Besoins des utilisateurs

Les besoins des utilisateurs peuvent être décrits à l'aide de *diagrammes de cas d'utilisation*. Un diagramme de cas d'utilisation définit les différents acteurs (qui interagissent avec le système) et les différents cas d'utilisations (ou fonctionnalités) du système.

Aspects statiques

Les aspects statiques d'un système sont décrits à l'aide de deux sortes de diagrammes :

- Les diagrammes de classes définissent les différentes classes, ainsi que les relations entre ces classes.
- Les diagrammes d'objets définissent différentes instances de ces classes, ainsi que les liens entre ces instances.

Aspects dynamiques

Les aspects dynamiques d'un système sont décrits par quatre sortes de diagrammes :

- Les diagrammes de séquence permettent de représenter des interactions d'un point de vue chronologique. Il permettent d'une part de représenter les interactions entre le système et les acteurs, d'autre part de représenter les interactions entre objets à l'intérieur du système.

- Les diagrammes de collaboration permettent de représenter les interactions entre objets d'un point de vue spacial.
- Les diagrammes d'états-transitions sont des automates hiérarchiques qui permettent de décrire des comportements : comportement d'un acteur, d'un système ou d'un objet d'une certaine classe.
- Les diagrammes d'activités permettent de modéliser le comportement d'une méthode en représentant un enchaînement d'activités.

Aspects physiques

Les aspects physiques d'un système peuvent être décrits à l'aide de deux sortes de diagrammes :

- Les diagrammes de composants permettent de décrire les composants (ou modules) d'un système (fichiers sources, fichiers objets, librairies, exécutables...).
- Les diagrammes de déploiement décrivent la disposition physique des matériels, et la répartition des composants sur ces matériels.

2. Diagrammes des cas d'utilisation

Les cas d'utilisation permettent de décrire le système du point de vue de l'utilisateur. Ils permettent de définir les limites du système et les relations entre le système et son environnement.

Un cas d'utilisation est une manière spécifique d'utiliser le système. Un cas d'utilisation correspond à une fonctionnalité du système.

Un acteur représente une personne ou une chose qui interagit avec le système. Plus précisément, un acteur est un rôle joué par une personne ou une chose qui interagit avec le système. Une même personne physique peut donc correspondre à plusieurs acteurs si celle-ci peut jouer plusieurs rôles.

Relation entre un acteur et un cas d'utilisation

Un acteur qui interagit avec le système pour utiliser une fonctionnalité est représenté par une relation entre cet acteur et le cas d'utilisation qui représente cette fonctionnalité. On dit également que l'acteur *déclenche* le cas d'utilisation auquel il est lié.

La figure II. 1 montre la représentation d'un acteur, d'un cas d'utilisation et d'une relation entre l'acteur et le cas d'utilisation.

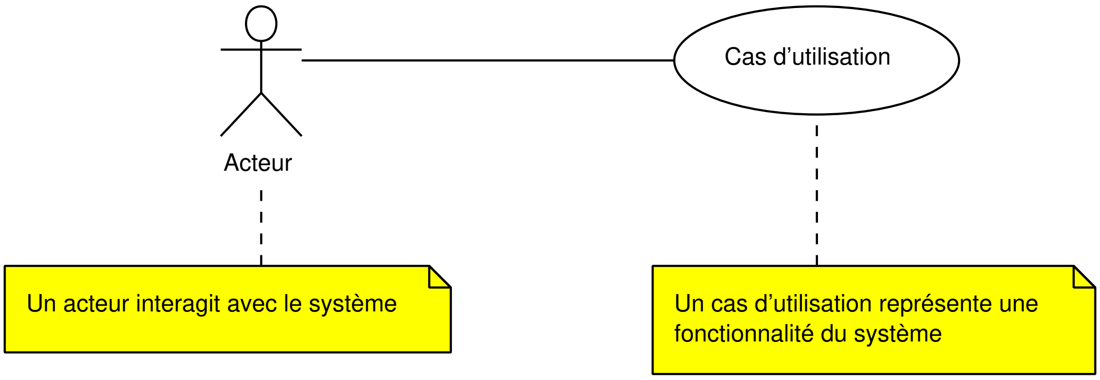


Fig. II. 1 – Relation entre un acteur et un cas d'utilisation

Généralisation entre deux acteurs

Un acteur, appelé *acteur parent*, *généralise* un acteur, appelé *acteur enfant*, lorsque tout ce qui peut être réalisé par l'acteur parent peut être réalisé par l'acteur enfant. On dit également que l'acteur enfant spécialise l'acteur parent.

La figure II. 2 montre la représentation d'une généralisation entre deux acteurs.

Inclusion entre deux cas d'utilisation

Un cas d'utilisation, appelé *cas source*, *inclut* un cas d'utilisation, appelé *cas destination*, si les comportements décrits par le cas source contiennent les comportements décrits par le cas

Fig. II. 2 – Généralisation entre deux acteurs

destination (cf. figure II. 3). Les inclusions entre cas d'utilisation sont fréquemment utilisés, en particulier pour factoriser une fonctionnalité partagée par plusieurs cas d'utilisation.

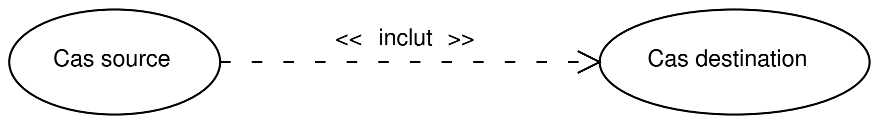


Fig. II. 3 – Inclusion entre deux cas d'utilisation

Extension d'un cas d'utilisation

Un cas d'utilisation, appelé *cas source*, *étend* un cas d'utilisation, appelé *cas destination*, si les comportement décrits par le cas source étendent les comportement décrits par le cas destination (cf. figure II. 4). L'extension peut être soumise à une condition.

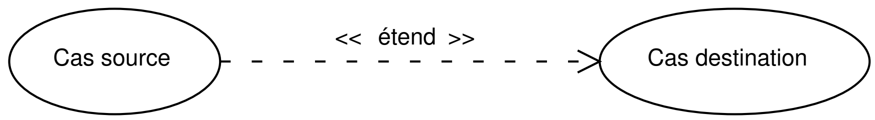


Fig. II. 4 – Extension entre deux cas d'utilisation

Généralisation entre deux cas d'utilisation

Un cas d'utilisation, appelé *cas d'utilisation parent*, *généralise* un cas d'utilisation, appelé *cas d'utilisation enfant*, lorsque les comportements décrits par le cas d'utilisation parent spécialisent les comportements décrits par le cas d'utilisation enfant (cf. figure II. 5). Une généralisation indique que le cas d'utilisation enfant est une variation du cas d'utilisation parent. Les généralisations entre cas d'utilisation sont assez rarement utilisées.

Limites du système

Les limites du système peuvent être précisées à l'aide d'un rectangle (cf. figure II. 6). Les acteurs, en général, ne font pas partie du système (ils interagissent avec le système).

Exemple 1. Système de virements bancaires

La figure II. 7 représente un diagramme de cas d'utilisation d'un système permettant d'effectuer des virements bancaires.

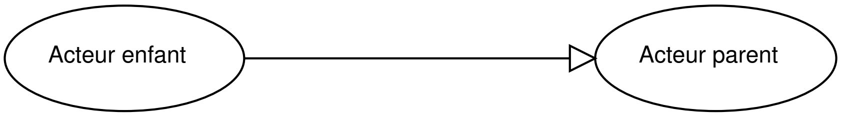


FIG. II. 5 – Généralisation entre deux cas d'utilisation

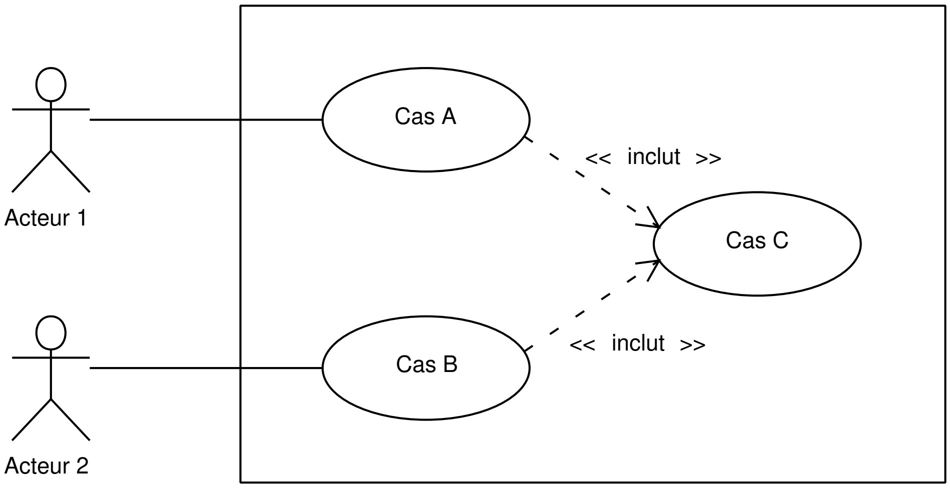


FIG. II. 6 – Limites du système

On a deux sortes d'acteurs : les clients locaux et les clients distants. Un client local peut effectuer un virement (cas d'utilisation *Virement*, qui inclut le cas d'utilisation *Identification*). Un client distant peut effectuer un virement par minitel (cas d'utilisation *Virement par minitel*, qui étend le cas d'utilisation *Virement*).

Exemple 2. Distributeur de billets

La figure II. 8 est un diagramme de cas d'utilisation pour un distributeur de billets de banque.

On a deux acteurs : le client, qui peut soit consulter le solde de son compte, soit retirer de l'argent au distributeur ; le technicien, qui peut allumer ou éteindre le distributeur et ravitailler le distributeur.

Une *note* (ou *commentaire*) indique que le technicien doit éteindre le distributeur avant de le ravitailler.

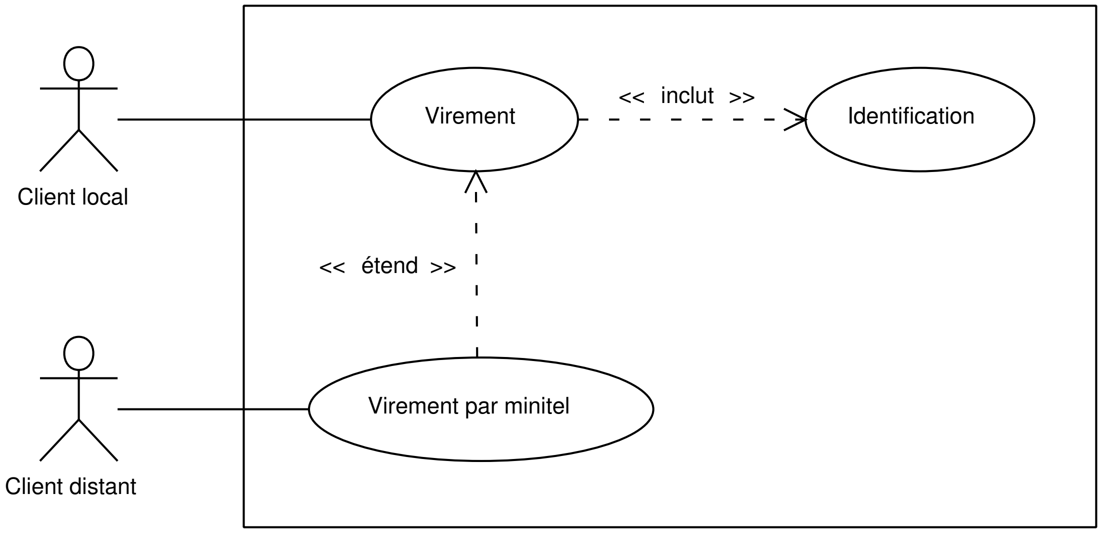


Fig. II. 7 – Diagramme de cas d'utilisation pour un système de virements bancaires

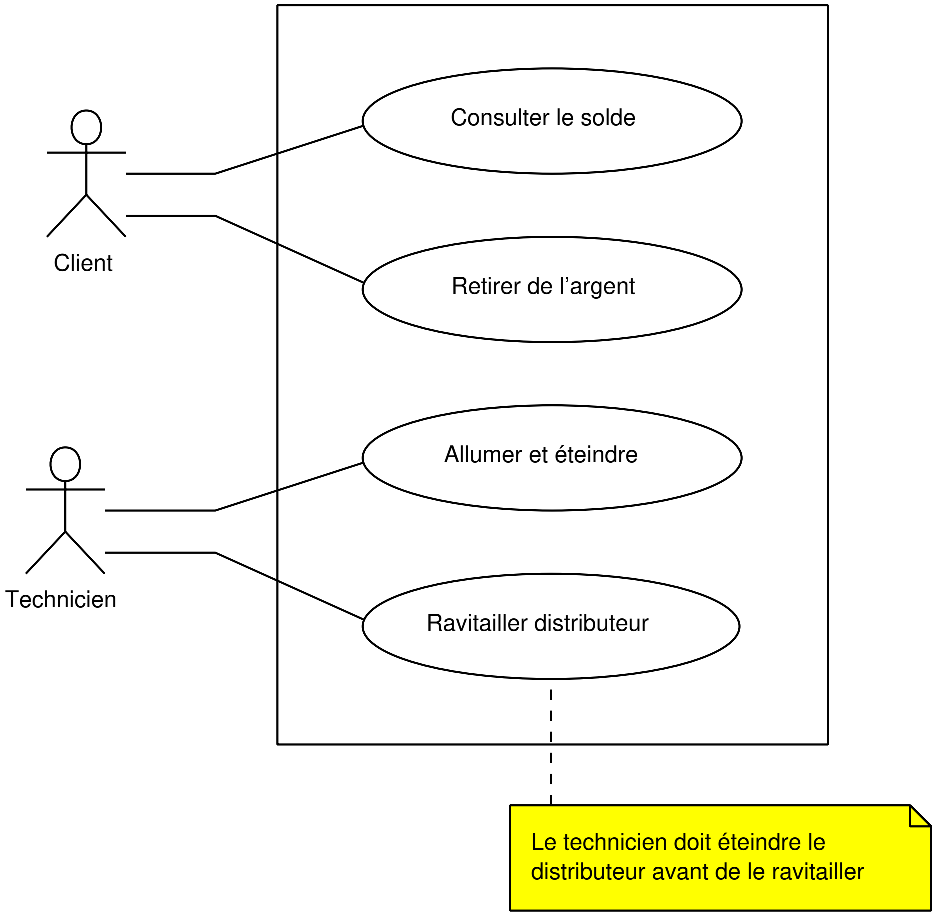


Fig. II. 8 – Diagramme de cas d'utilisation pour un distributeur de billets

3. Diagrammes de classes et d'objets

a) Classe, objet et association

Classe

Une classe est une abstraction qui représente un ensemble d'objets de même nature. On dit qu'un objet est une « instance » de sa classe. Une classe est « instanciée » lorsqu'elle contient au moins un objet.

Une classe comporte un nom, des attributs et des opérations. La figure II. 9 représente deux classes : la classe Compte (pour des comptes bancaires) et la classe Personne. La classe Compte comporte deux attributs : solde, qui représente le montant disponible sur le compte, et min, qui représente le montant minimal que le compte peut contenir.

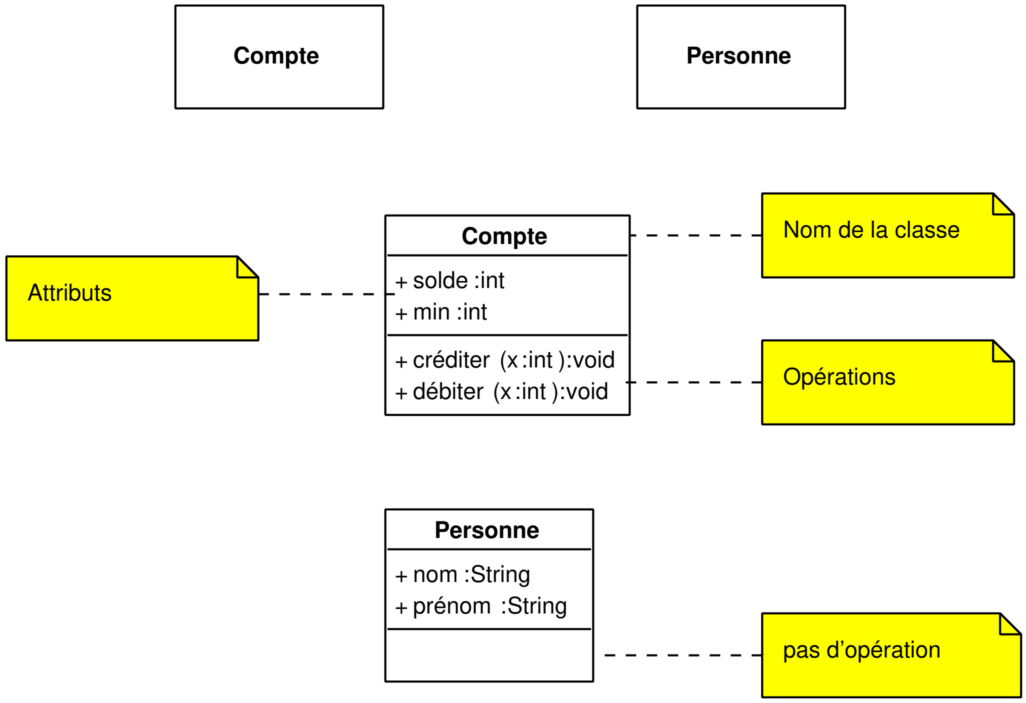


FIG. II. 9 – Représentation des classes Compte et Personne

Un attribut est une propriété d'un objet de la classe. Il comporte un nom et éventuellement un type et une valeur initiale. Les types possibles ne sont pas spécifiés en UML.

On peut remarquer que la notation UML n'oblige pas le concepteur à indiquer tous les attributs et méthodes d'une classe. De même, il n'est pas obligatoire de toujours préciser le type des attributs et le profil des opérations.

Un attribut *dérivé* est un attribut dont la valeur peut être calculée à partir des attributs non dérivés.

Dans l'exemple représenté figure II. 10, l'âge d'une personne peut être déduit de l'année courante et de sa date de naissance. De même, la surface d'un rectangle peut être déduite de sa largeur et de sa longueur.

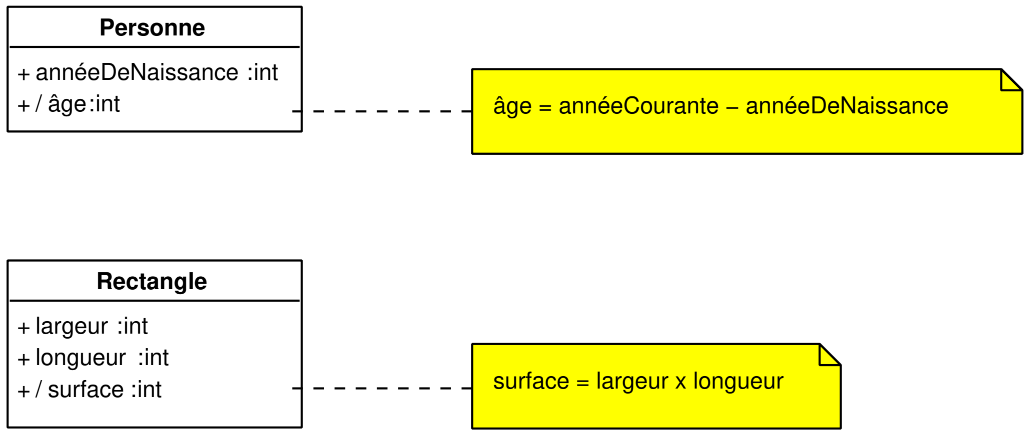


FIG. II. 10 – Exemples d'attributs dérivés

Un attribut *de classe* est un attribut qui est partagé par toutes les instances de la classe.

Une opération est un service offert par les objets de la classe. En UML, on distingue *opération*, qui spécifie un service, et *méthode* qui implante l'opération. Les opérations peuvent comporter des paramètres (qui peuvent être typés et comporter une valeur initiale) et un résultat.

La syntaxe d'une opération est la suivante :

$$\text{Opération } (mode_1\ arg_1 : Type_1 = val_1, \ldots\ mode_n\ arg_n : Type_n = val_n) : \textit{TypeRésultat}$$

où :

- le mode des paramètre peut être *in* (paramètre d'entrée), *out* (paramètre de sortie) ou inout (paramètre d'entrée sortie). Le mode par défaut est *in*.
- les paramètres, le type, et la valeur initiale sont optionnels.

Objet

Une classe est une abstraction qui représente un ensemble d'*objets*. Chaque objet est une « instance » d'une classe.

La figure II. 11 représente deux instances C1 et C2 de la classe Compte, deux instances Pierre et Paul de la classe Personne, ainsi que des instances anonymes de ces deux classes.

Niveaux de visibilité

Il y a quatre niveaux de visibilité en UML pour les attributs et les opérations :

- le niveau public (+), qui indique qu'un élément est visible pour tous les clients de la classe ;
- le niveau protégé (#), qui indique qu'un élément est accessible uniquement pour les sous-classes de la classe où il apparaît ;

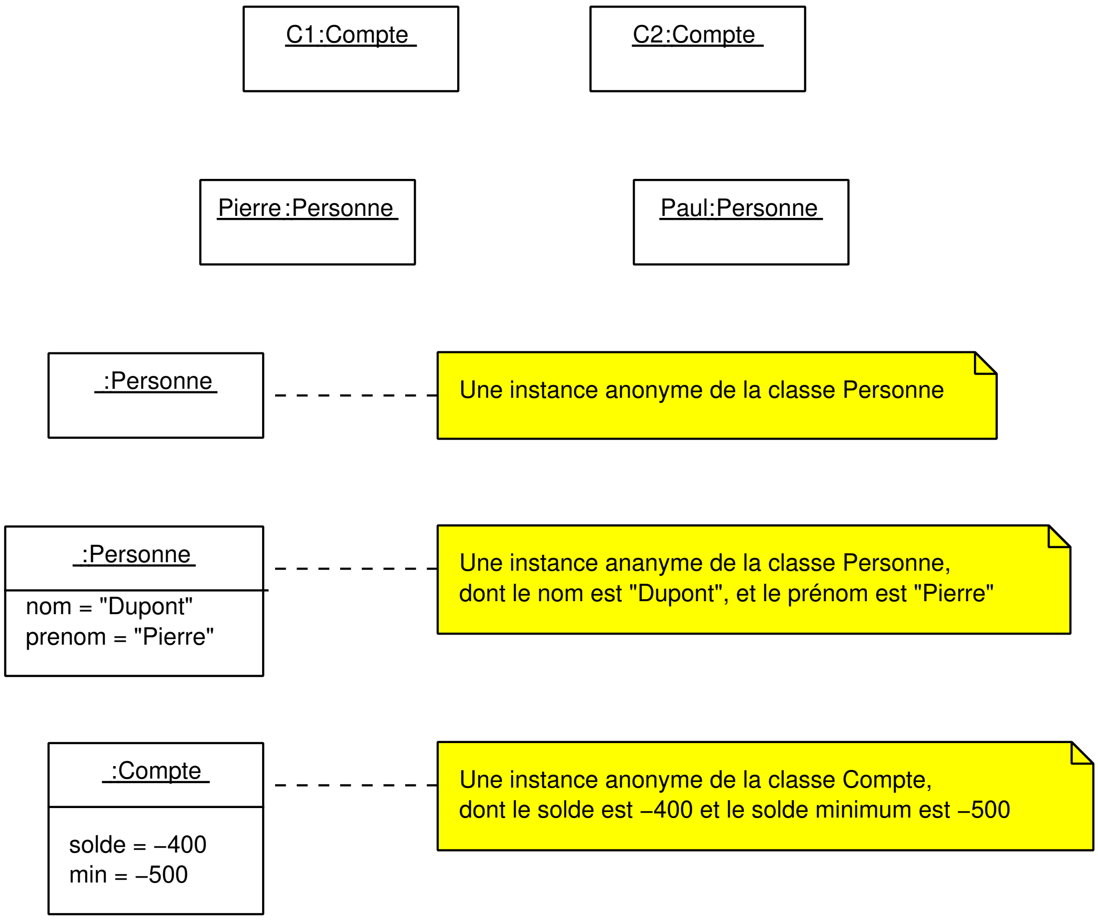


FIG. II. 11 – Exemples d'instances des classes Compte et Personne

- le niveau paquetage (~), qui indique qu'un élément est visible uniquement pour les classes définies dans le même paquetage ;
- le niveau privé (-), qui indique que seule la classe où est défini cet élément peut y accéder.

La figure II. 12 représente une classe pour les nombres complexes, qui comporte quatre opérations publiques (addition, soustraction, multiplication et division). Cette classe ne précise pas l'implantation d'un nombre complexe, ni les paramètres et le résultat des opérations.

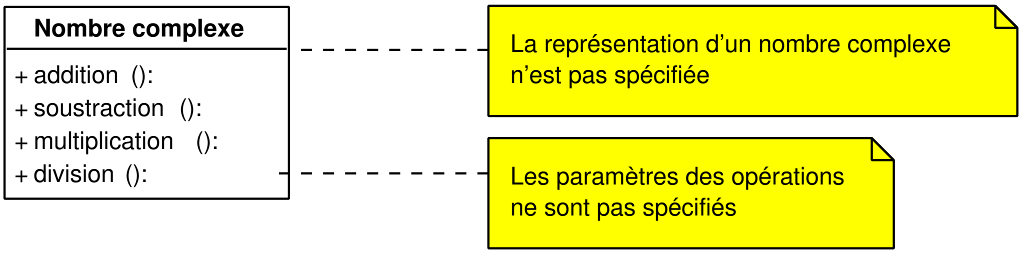


Fig. II. 12 – Classe des nombres complexes ; implantation non précisé

La figure II. 13 implante les nombres complexes à l'aide du module et de l'argument, qui sont des attributs privés de la classe.

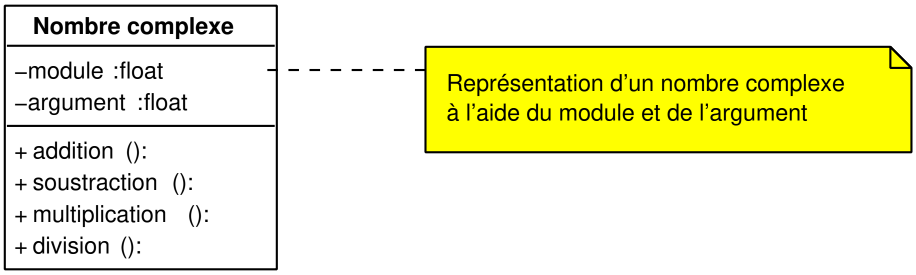


Fig. II. 13 – Classe des nombres complexes ; implantation avec module et argument

La figure II. 14 implante les nombres complexes à l'aide de la partie réelle et de la partie imaginaire, qui sont des attributs privés de la classe.

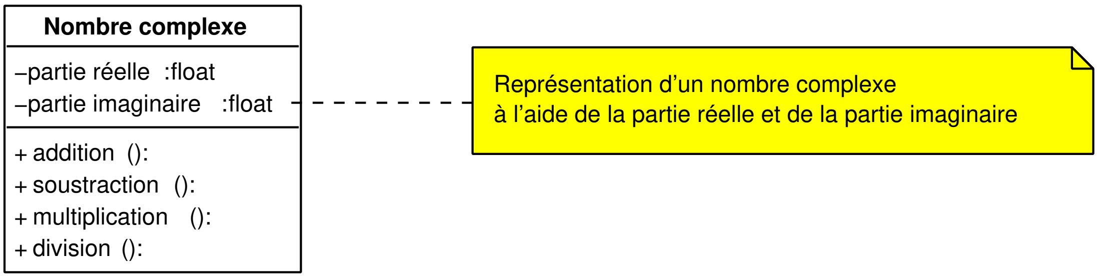


Fig. II. 14 – Classe des nombres complexes ; implantation avec parties réelle et imaginaire

Classe utilitaire

Une classe utilitaire permet de regrouper un ensemble de valeurs (valeurs d'attributs) et d'opérations. Une classe utilitaire ne peut pas être instanciée, et ne comporte que des attributs et des opérations *de classe*.

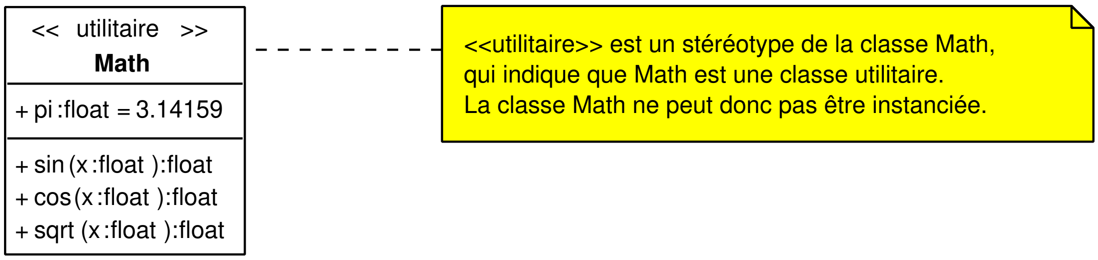


FIG. II. 15 – Classe utilitaire Math

Association

Une association est, de façon générale, une relation n-aire entre n classes d'objets. Les relations les plus utilisées sont les relations binaires, entre deux classes.

La figure II. 16 représente une relation R entre les classes A et B.

Remarque

Une association entre les classes A et B est une relation binaire, au sens mathématique, autrement dit un sous-ensemble de $A \times B$. Cela signifie qu'il existe au plus un lien entre un objet de la classe A et un objet de la classe B.

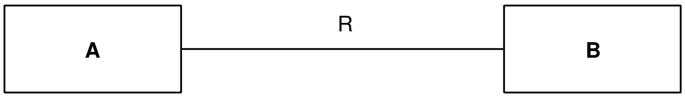


FIG. II. 16 – Relation R entre les classes A et B

Exemple

On peut spécifier une relation entre la classe Compte et la classe Personne (*diagramme de classes* figure II. 17).

« `*` » et « `1..2` » sont des *multiplicités*, dont la signification est la suivante :

- « `*` » signifie qu'une personne peut ouvrir un nombre quelconque de comptes ;
- « `1..2` » signifie qu'un compte est ouvert pour une ou deux personnes.

La figure II. 18 est un *diagramme d'objets* associé au diagramme de classes représenté figure II. 17.

Ce diagramme d'objets spécifie que :

- Pierre et Marie ont un compte commun (C1) ;

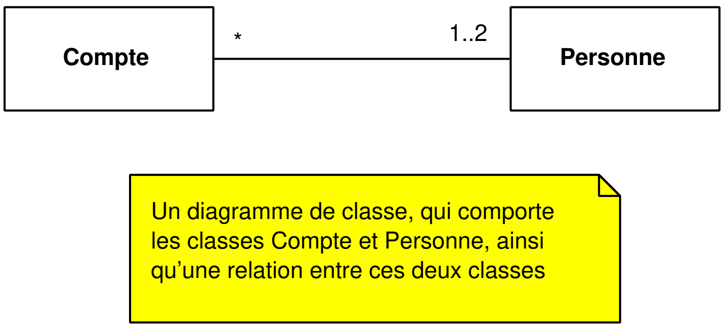


FIG. II. 17 – Diagramme de classes

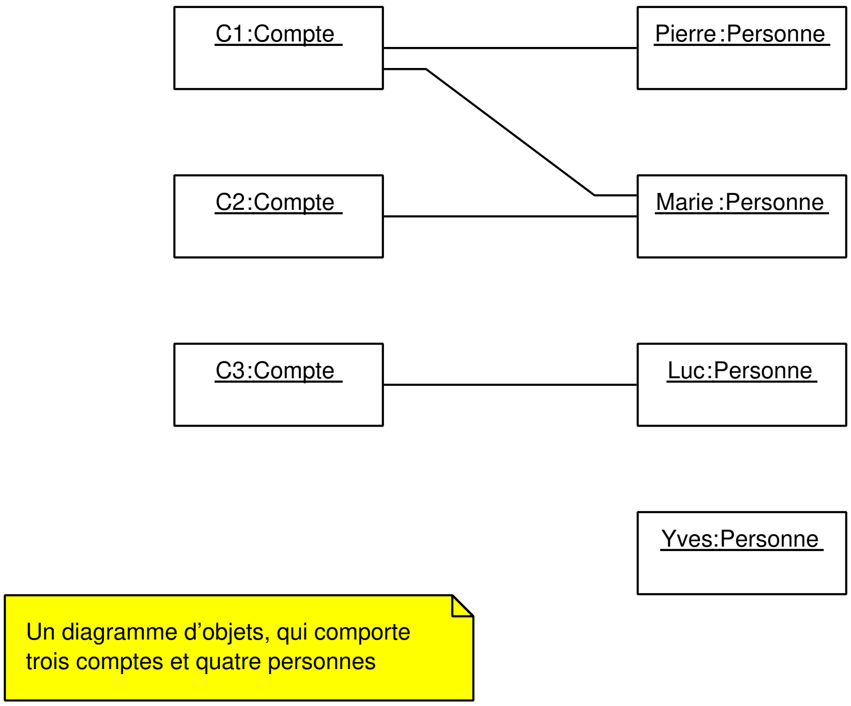


FIG. II. 18 – Un diagramme d'objets correspondant au diagramme de classes

- Marie a deux comptes (C1 et C2) ;
- Luc a un compte (C3) ;
- Yves n'a pas de compte.

Navigabilité

La relation entre les classes Personne et Compte que nous avons définie permet :

- d'une part, à partir d'une personne, de déterminer l'ensemble des comptes qu'elle possède ;
- d'autre part, à partir d'un compte, de retrouver son ou ses propriétaires.

On dit que la relation est « navigable dans les deux sens ». Pour spécifier qu'un seul sens de navigation est autorisé, on peut orienter la relation. Par exemple, dans le diagramme de classes représenté figure II. 19, on peut à partir d'un objet de la classe A, accéder aux objets de la classe B qui sont en relation avec celui-ci, mais pas l'inverse.

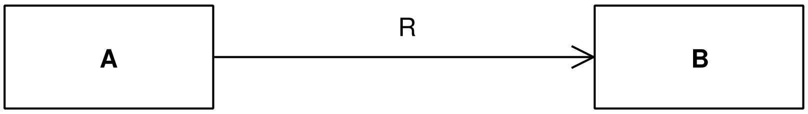


FIG. II. 19 – Sens de navigation de la relation : de A vers B

Rôles

Les extrémités d'une association sont appelées des *rôles* et peuvent être nommées. Par exemple, dans le diagramme de classes représenté figure II. 20, on a une relation « travaille pour » entre des personnes et des entreprises. Dans cette relation, une personne joue le rôle d'un employé, et une entreprise joue le rôle d'un employeur.

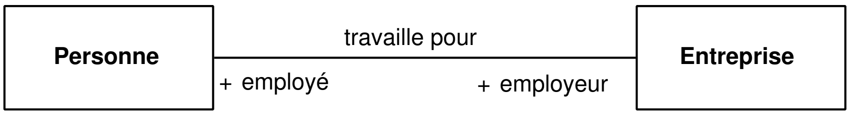


FIG. II. 20 – Rôles *employé* et *employeur* associés à la relation *travaille pour*

Multiplicités

Une multiplicité est un sous-ensemble de $\mathbb{N}$. Les notations utilisées en UML pour décrire des multiplicités sont les suivantes :

Syntaxe	Sémantique
`*`	$\mathbb{N}$
m `..` n	$\{x \in \mathbb{N}\ ;\ m \leq x \leq n\}$
m `..` `*`	$\{x \in \mathbb{N}\ ;\ m \leq x\}$
m, n, p	$\{m, n, p\}$
M_1, M_2	$M_1 \cup M_2$

Soit une relation R entre deux classes A et B, avec les multiplicités M_A associée à A et M_B associée à B.

Les multiplicités M_A et M_B imposent les contraintes suivantes :

- pour chaque objet b de la classe B, le nombre d'objets de la classe A liés à b appartient à M_A, autrement dit :
$$\forall b \in B,\ \mathrm{Card}\{(a,b) \in R\ ;\ a \in A\} \in M_A\ ;$$
- pour chaque objet a de la classe A, le nombre d'objets de la classe B liés à a appartient à M_B, autrement dit :
$$\forall a \in A,\ \mathrm{Card}\{(a,b) \in R\ ;\ b \in B\} \in M_B.$$

Exemple

Le diagramme de classes représenté figure II. 21 exprime qu'une personne travaille pour au plus une entreprise.

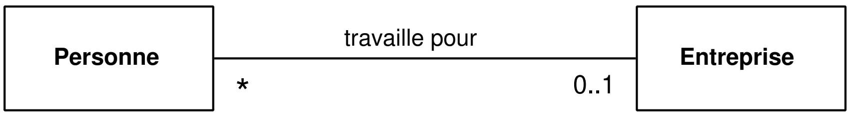


Fig. II. 21 – Diagramme de classes (relation « travaille pour »)

Le diagramme d'objets représenté figure II. 22 est correct car il respecte cette contrainte.

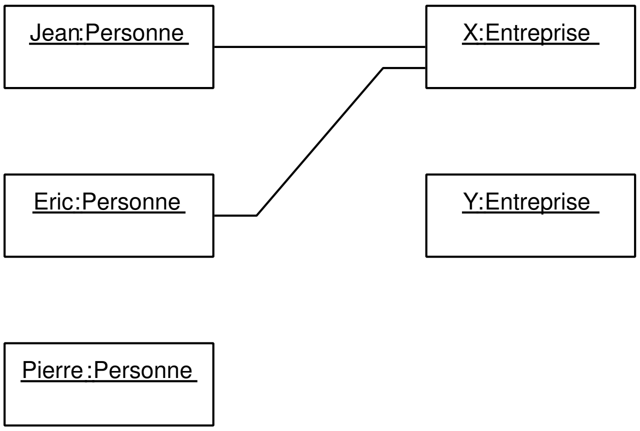


Fig. II. 22 – Diagramme d'objet correct par rapport au diagramme de classes figure II. 21

Le diagramme d'objets représenté figure II. 23 n'est pas correct par rapport au diagramme de classes figure II. 21 car Eric ne peut pas travailler pour deux entreprises.

Cas particuliers d'associations

La figure II. 24 représente certains cas particuliers d'associations :

- application de la classe X vers la classe Y : à chaque objet de la classe X est associé un unique objet de la classe Y.

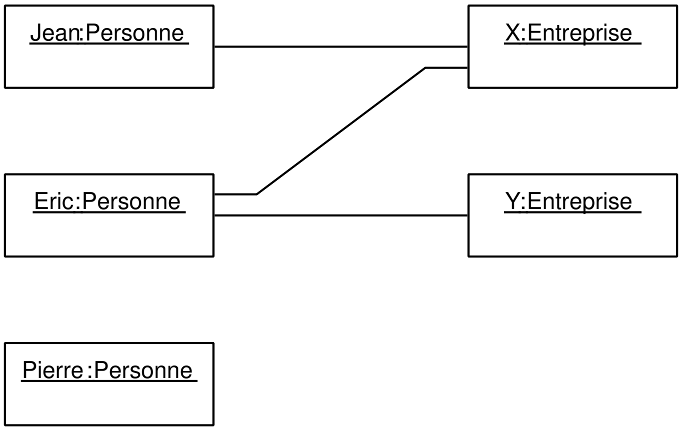


FIG. II. 23 – Diagramme d'objet incorrect par rapport au diagramme de classes figure II. 21

- fonction partielle de la classe X vers la classe Y : à chaque objet de la classe X est associé au plus un objet de la classe Y.
- isomorphisme entre les classes X et Y : chaque objet de la classe X est en relation avec un unique objet de la classe Y, et chaque objet de la classe Y est en relation avec un unique objet de la classe X. Par conséquent, les ensembles d'objets correspondants aux classes X et Y sont isomorphes.

b) Classe-association

Une classe-association est une entité qui combine les propriétés d'une classe et les propriétés d'une association. Une classe-association R entre deux classes A et B permet d'associer à tout lien entre un objet de la classe A et un objet de la classe B un objet de la classe R.

Dans l'exemple présenté figure II. 25, Travail est une classe-association entre les classes Personne et Entreprise. Travail permet d'associer à tout lien entre une personne et une entreprise un objet de la classe Travail, qui comporte donc une fonction, un salaire, et peut utiliser l'opération feuille_paie (qui sert à produire des feuilles de paie).

Multiplicités

Comme les associations, les classes-associations peuvent spécifier des multiplicités à chacune de leurs extrémités.

Dans l'exemple figure II. 26, le nombre d'objets de la classe Y qui peuvent être associés à un objet de la classe X est spécifié par la multiplicité M_X ; le nombre d'objets de la classe X qui peuvent être associés à un objet de la classe Y est spécifié par la multiplicité M_Y.

De façon générale, la figure II. 27 montre comment ce schéma peut être simulé par deux relations. Intuitivement,

- chaque objet de la classe Z est associé à un objet de la classe X et un objet de la classe Y ;
- chaque objet de la classe X est associé à n objets de la classe Z, avec $n \in M_Y$;
- chaque objet de la classe Y est associé à n objets de la classe Z, avec $n \in M_X$.

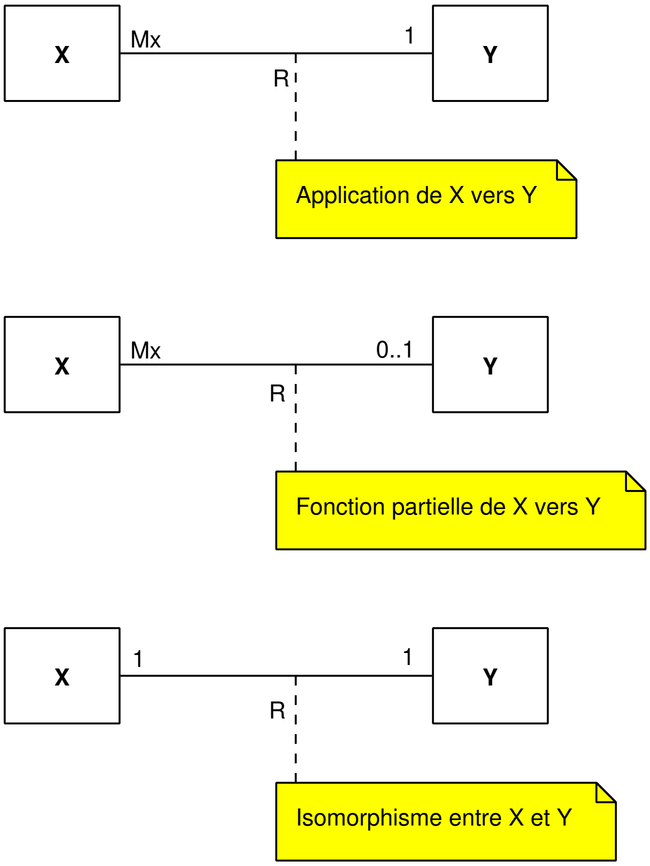


FIG. II. 24 – Cas particuliers d'associations binaires

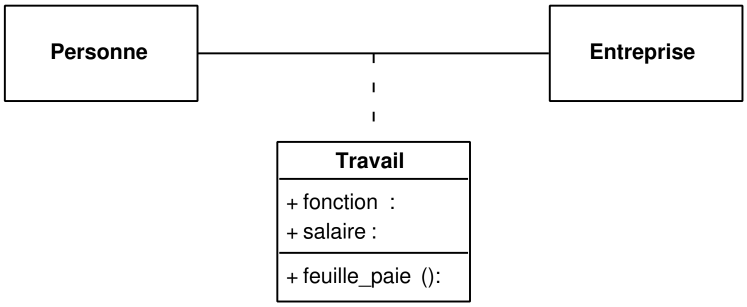


FIG. II. 25 – Exemple de classe-association entre les classes Personne et Travail

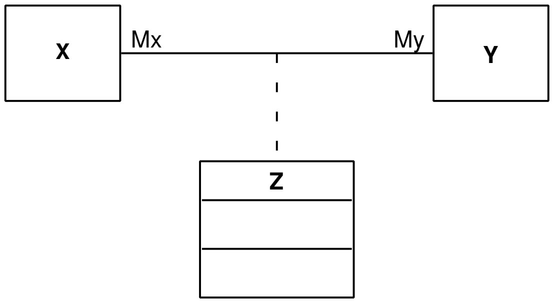


FIG. II. 26 – Multiplicités d'une classe-association

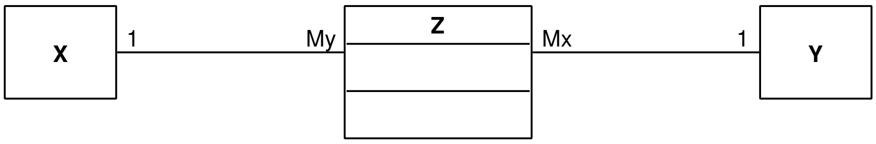


FIG. II. 27 – Simulation d'une classe-association

Exemple

Le diagramme de classes D_1 représenté figure II. 28 est simulé par le diagramme de classes D_2 représenté figure II. 29.

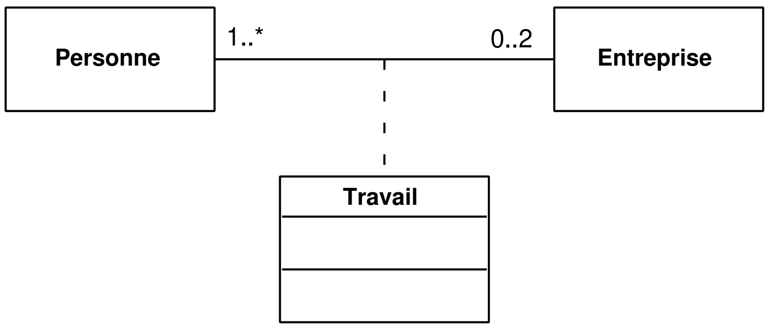


FIG. II. 28 – Diagramme de classes D_1

La figure II. 30 montre un diagramme d'objets correspondant au diagramme de classe D_1. La figure II. 31 montre un diagramme d'objets correspondant au diagramme de classe D_2.

c) Agrégation et composition

Agrégation

Une agrégation est une association binaire particulière qui modélise une relation entre un « tout » et une « partie », autrement dit, une relation d'appartenance.

La figure II. 32 montre une agrégation entre les classes A et B : un objet de la classe B, appelé *agrégat*, contient un certain nombre d'objets de la classe A, appelés *composants*.

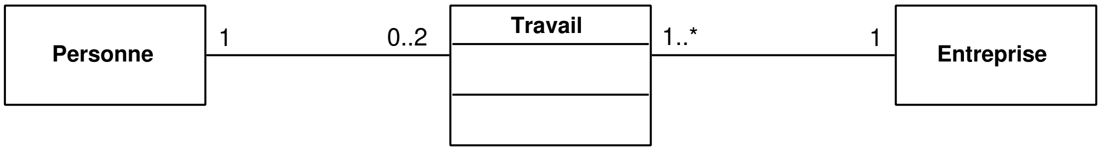


FIG. II. 29 – Diagramme de classes D_2

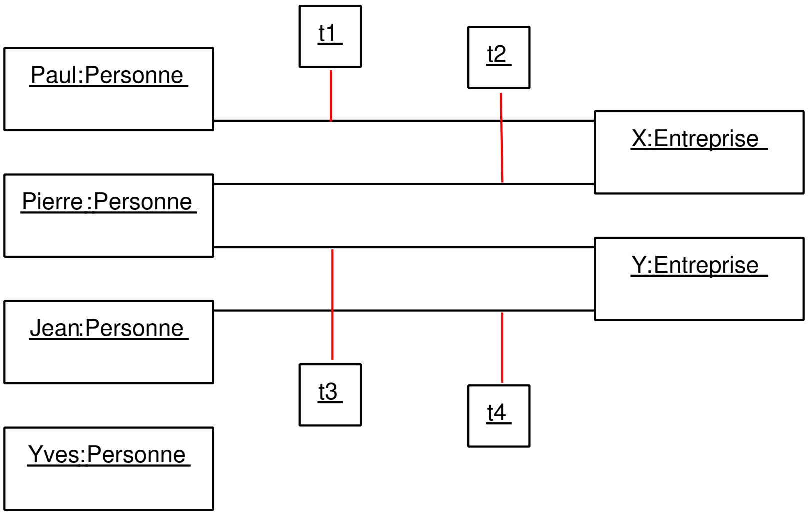


FIG. II. 30 – Diagramme d'objets correspondant au diagramme de classes D_1

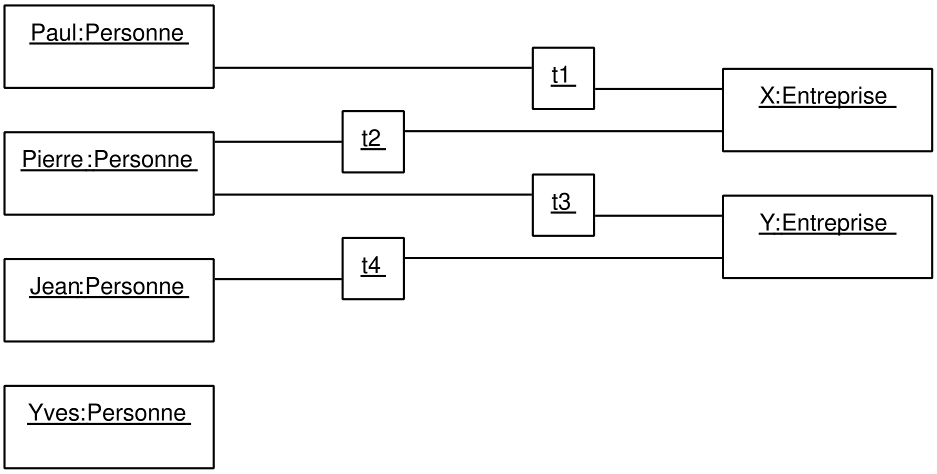


FIG. II. 31 – Diagramme d'objets correspondant au diagramme de classes D_2

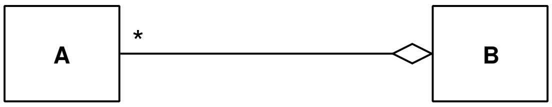


FIG. II. 32 – Relation d'agrégation entre les classes A et B

Exemple : segment

La figure II. 33 montre la modélisation de points et de segments : un segment est formé de l'agrégation de deux points.

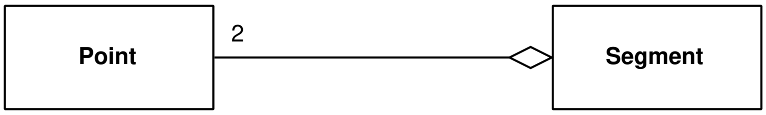


FIG. II. 33 – Modélisation d'un segment

La figure II. 34 est un diagramme d'objets correspondant au diagramme de classes précédent, qui modélise trois points A, B et C, ainsi que deux segments AB et BC. On peut remarquer que le point B appartient à la fois au segment AB et au segment BC.

Dans une agrégation, certaines parties peuvent être partagée : par défaut, il n'y a pas de contrainte de multiplicité sur l'agrégat. Dans l'exemple, un point peut être partagé entre plusieurs segments.

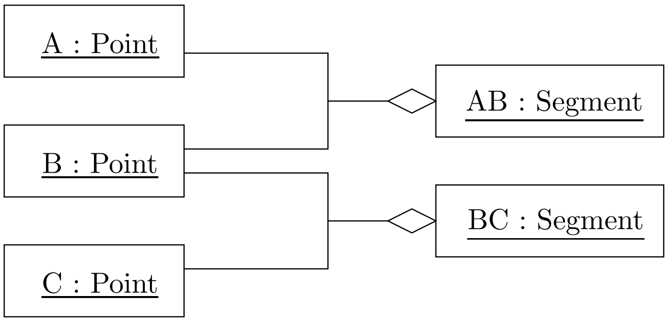


FIG. II. 34 – Diagramme d'objets modélisant les segments AB et BC

La relation d'agrégation interdit les cycles dans les diagrammes d'objets. On peut par exemple définir une classe A en relation d'agrégation avec elle-même. Par contre, on ne peut pas définir de cycles dans un diagramme d'objets (cf.figure II. 35).

Composition

Une composition est une agrégation particulière, pour laquelle une partie ne peut être partagée entre plusieurs agrégats. Autrement dit, la multiplicité du coté de l'agrégat est 0 ou 1.

Exemple

La figure II. 37 est un diagramme de classes qui modélise des voitures. Une voiture comporte quatre roues. Une roue ne peut pas appartenir à deux voitures.

La figure II. 38 est un diagramme d'objets correspondant au diagramme de classes précédent, qui montre une voiture composée de quatre roues.

On peut également représenter les composants d'un objet à l'intérieur de l'objet lui-même.

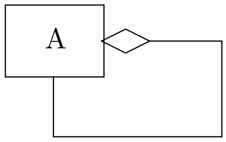


Diagramme correct

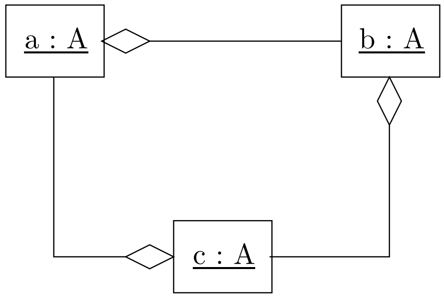


Diagramme incorrect : cycles interdits

FIG. II. 35 – Cycles et agrégations

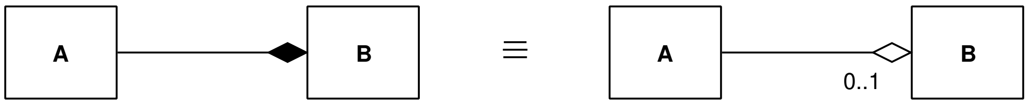


FIG. II. 36 – Relation de composition entre les classes A et B

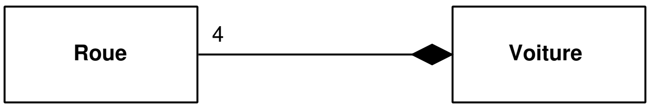


FIG. II. 37 – Diagramme de classes

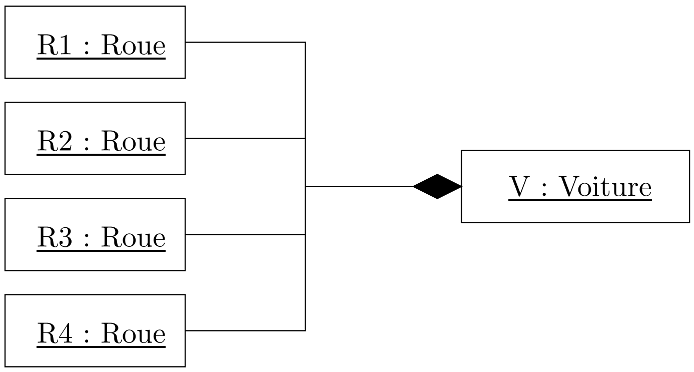


FIG. II. 38 – Un diagramme d'objets correspondant au diagramme de classes figure II. 37

Cela est possible uniquement pour les compositions car deux objets ne peuvent pas partager un même composant. La figure II. 39 montre un diagramme d'objets équivalent au diagramme d'objets de la figure II. 38.

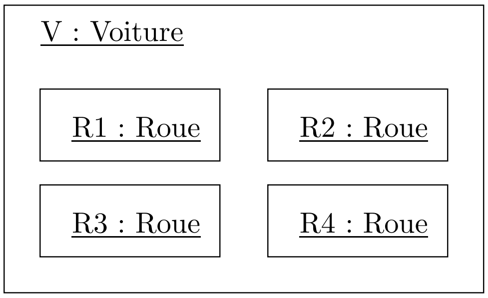


FIG. II. 39 – Représentation équivalente à la figure II. 38

d) Association n-aire

Une association n-aire est une relation entre n classes. Mathématiquement, une association n-aire entre les classes A_1, $A_2 \ldots A_n$ est un sous-ensemble R de $A_1 \times A_2 \cdots \times A_n$, donc un ensemble de n-uplets de la forme $(a_1, a_2, \ldots a_n)$ avec $\forall i \in \{1, 2, \ldots n\},\ a_i \in A_i$.

Notation

La figure II. 40 montre une association ternaire R entre les classes X, Y et Z.

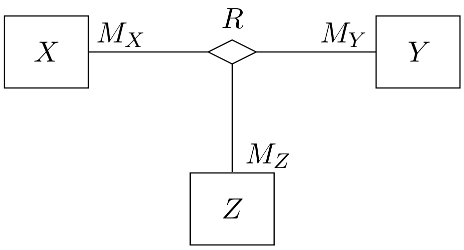


FIG. II. 40 – Association ternaire entre les classes X, Y et Z

Multiplicités

Les multiplicités M_X, M_Y et M_Z sont associées respectivement aux classes X, Y et Z. La multiplicité associée à une classe pose une contrainte sur le nombre de n-uplets de la relation lorsque les $n-1$ autres valeurs sont fixées. Ces multiplicités posent les contraintes suivantes :

- $\forall y_0 \in Y,\ \forall z_0 \in Z,\ \text{Card}\,\{(x, y_0, z_0) \in R\ ;\ x \in X\} \in M_X$;
- $\forall x_0 \in X,\ \forall z_0 \in Z,\ \text{Card}\,\{(x_0, y, z_0) \in R\ ;\ y \in Y\} \in M_Y$;
- $\forall x_0 \in X,\ \forall y_0 \in Y,\ \text{Card}\,\{(x_0, y_0, z) \in R\ ;\ z \in Z\} \in M_Z$;

Exemple

La figure II. 41 montre une relation ternaire entre les classes Cours, Enseignant et Classe, qui modélise les cours effectués par des enseignants dans différentes classes. De plus, cette association ternaire est une classe-association : à chaque triplet *(cours, enseignant, classe)* est associée une localisation, qui comporte une salle et un horaire.

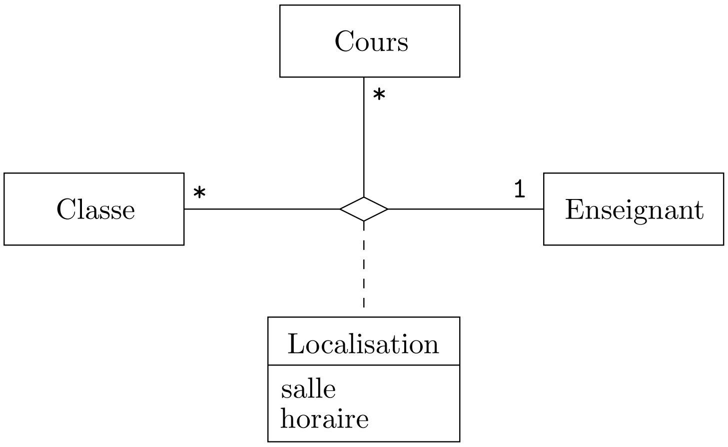


FIG. II. 41 – Relation ternaire entre les classes Cours, Enseignant et Classe

Les multiplicités posent les contraintes suivantes :

- un enseignant peut faire un même cours à plusieurs classes (multiplicité « * » associée à la classe Classe) ;
- dans une classe, un cours est effectué par un seul enseignant (multiplicité « 1 » associée à la classe Enseignant) ;
- un enseignant peut faire plusieurs cours dans une même classe (multiplicité « * » associée à la classe Cours). Un enseignant peut donc enseigner plusieurs matières.

e) Extension

Une classe Y *étend* ou *spécialise* une classe X si tout objet de la classe Y est, ou peut être considéré comme, un objet de la classe X. La notation UML est montrée figure II. 42.

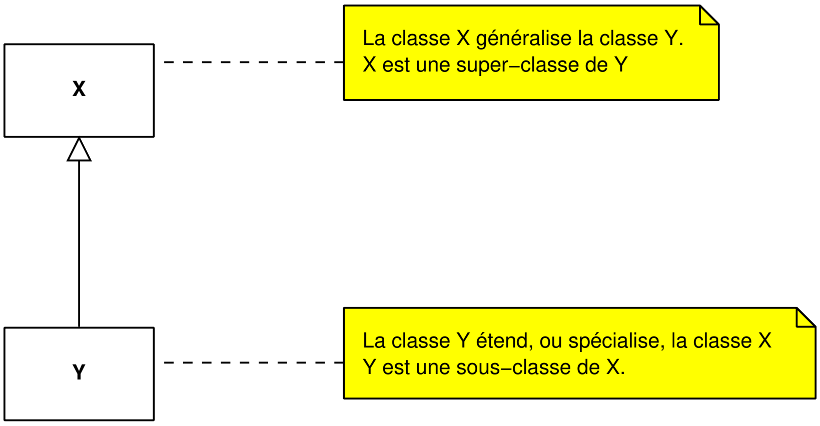


FIG. II. 42 – La classe Y étend la classe X

Lorsqu'on réalise une extension, on a deux propriétés qui sont satisfaites : la propriété d'héritage et la propriété de substitution.

Propriété d'héritage

Les attributs et opérations définis dans X, qui ne sont pas privés, et qui ne sont pas *redéfinis* dans Y, sont *hérités* dans Y.

Propriété de substitution

La propriété de substitution exprime que tout objet de la classe Y *est* ou *peut être considéré comme* un objet de la clases X. Chaque fois qu'on a besoin d'une instance de X, on peut utiliser à la place une instance de Y.

Exemple

La figure II. 43 montre une classe Point2D qui comporte deux attributs x et y de type entier, représentant respectivement l'abscisse et l'ordonnée d'un point en deux dimensions. La méthode distanceOrigine calcule la distance à l'origine d'un point en deux dimensions.

La classe Point2DColoré (classe des points colorés) étend la classe Point2D : elle comporte un attribut couleur, qui représente la couleur d'un point coloré et hérite des attributs x et y, ainsi que de la méthode distanceOrigine.

La classe Point3D (classe des points en dimension trois) étend la classe Point2D : elle comporte un attribut z, qui représente la hauteur d'un point en dimension trois. La méthode distanceOrigine est *redéfinie* dans la classe Point3D.

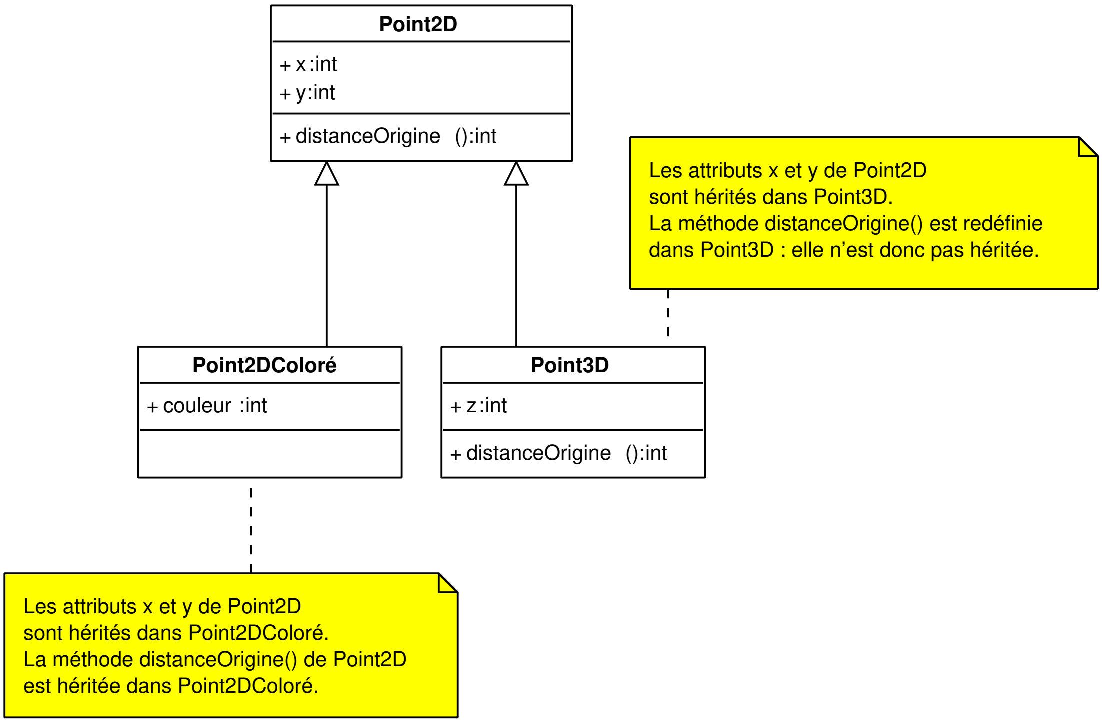


Fig. II. 43 – Diagramme de classes pour des points

Hiérarchies de classes

L'extension permet de gérer la complexité en organisant les classes sous forme de *hiérarchies de classes*.

La figure II. 44 montre une hiérarchie de classes pour différents types de véhicules : aériens, aquatiques et terrestres.

Par exemple, un hydravion est à la fois un avion et un bateau. La classe Hydravion étend donc les deux classes Avion et Bateau.

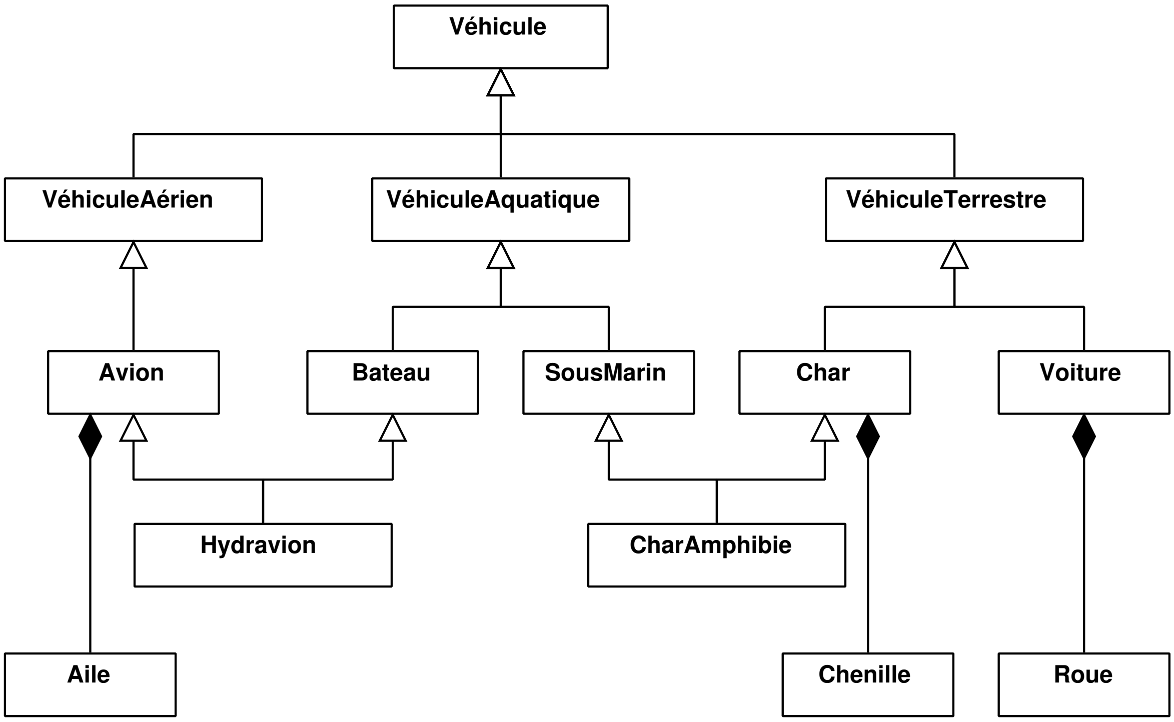


FIG. II. 44 – Hiérarchie de classes pour différents types de véhicules

Héritage de relation

Lorsqu'une classe A_1 hérite d'une classe A, elle hérite de toutes les relations de A. Dans l'exemple figure II. 45, la relation entre les classes A et B est héritée par A_1 : on a donc également une relation entre A_1 et B.

Dans la figure II. 44, la relation entre Char et Chenille est héritée par CharAmphibie. En effet, un char a des chenilles, un char amphibie est un char, donc un char amphibie a des chenilles. De même, un hydravion a des ailes.

f) Classe abstraite

Une classe abstraite est une classe qui ne peut être instanciée, et qui peut contenir des opérations abstraites. Une opération abstraite est une opération sans implémentation, c'est-

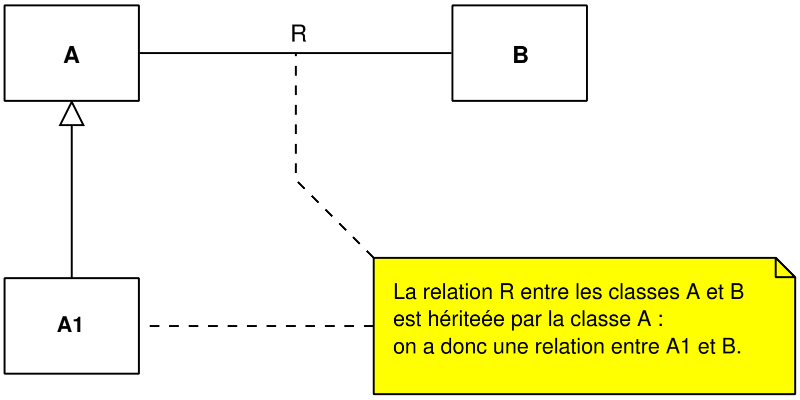


FIG. II. 45 – La relation R est hérité par A_1

à-dire à laquelle ne correspond aucun code. Une classe abstraite peut également contenir des opération concrètes (c'est-à-dire non abstraites).

Les classes abstraites servent notamment dans les hiérarchies de classes, où elles permettent de regrouper des attributs et opérations communes à plusieurs classes.

En programmation objet (par exemple en Java), une classe concrète doit implémenter les opérations abstraites dont elle hérite.

La figure II. 46 montre deux notations possibles pour la classe abstraite Véhicule : soit on met le nom de la classe en italiques, soit on utilise le stéréotype « abstrait ».

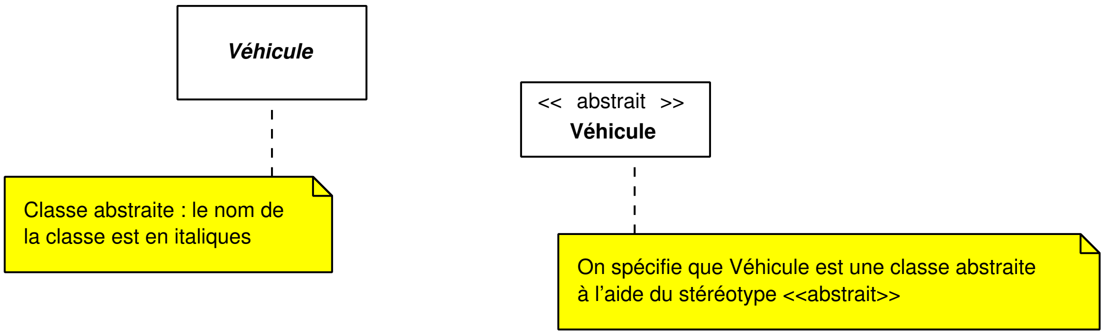


FIG. II. 46 – Notations pour la classe abstraite Véhicule

Exemple

La figure II. 47 montre une hiérarchie de classes qui modélise différentes œuvres, en particulier des livres et des films.

On a une classe abstraite Oeuvre qui représente les différentes œuvres possibles. Les livres et les films sont des œuvres ; les romans et les bandes dessinées sont des livres.

La classe Roman hérite des attributs titre et auteur de la classe Oeuvre. Livre est une classe concrète, ce qui autorise la création de livres qui ne sont ni des romans, ni des bandes dessinées.

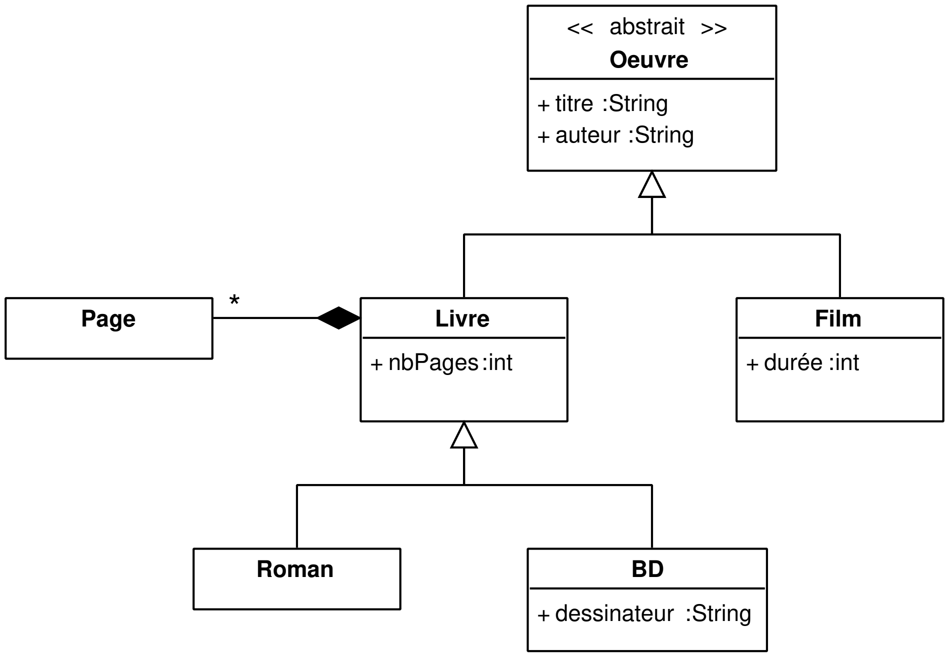


FIG. II. 47 – Hiérarchie de classes pour des œuvres

De plus, un livre est composé de pages, donc un roman et une bande dessinée sont également composés de pages.

g) Interface

Une interface spécifie un ensemble d'opérations qui constituent un service cohérent. Une interface contient uniquement des opérations abstraites, sans implémentation. Une interface est formellement équivalente à une classe abstraite qui ne contient que des opérations abstraites. Une classe *implémente* une interface lorsqu'elle fournit toutes les opérations de l'interface.

La figure II. 48 montre deux notations possibles pour l'interface Liste. Cette interface comporte une opération élément(n : int) : Object qui renvoie le n-ième élément de la liste. La classe Tableau implémente l'interface Liste, donc fournit une méthode élément(n : int) : Object.

Exemple : les collections Java

L'interface Collection, représentée figure II. 49, offre des services pour manipuler des collections d'objets, en particulier des ensembles (sous-interface Set) et des listes (sous-interface List). L'interface Collection utilise l'interface Iterator (il s'agit du type retourné par la méthode iterator()).

La classe HashSet implémente l'interface Set et la classe Vector implémente l'interface List.

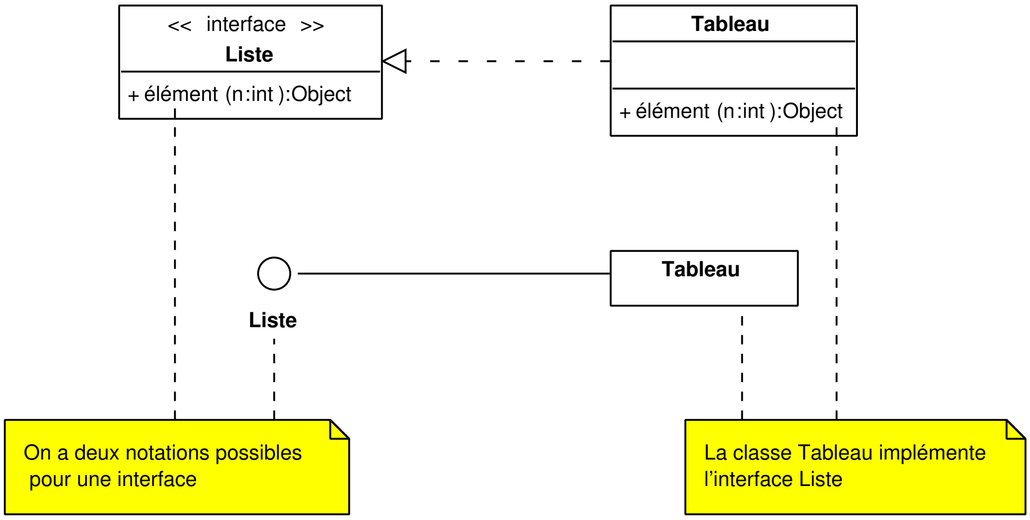


FIG. II. 48 – La classe Tableau implémente l'interface Liste

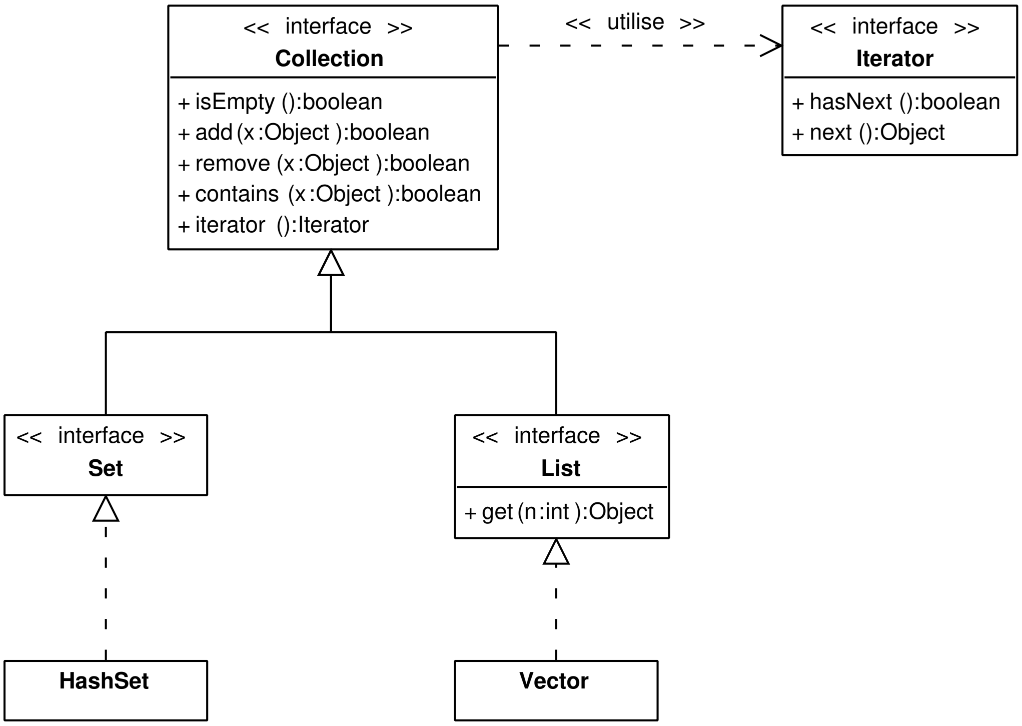


FIG. II. 49 – Les collections Java

4. Diagrammes de séquence

Les diagrammes de séquence permettent de représenter les interactions entre objets d'un point de vue chronologique.

Les diagrammes de séquence permettent de documenter les cas d'utilisation, en représentant les interactions entre les acteurs et le système.

La figure II. 50 montre un diagramme de séquence décrivant une communication téléphonique entre le système (le système de télécommunications) et deux personnes (l'appelant et l'appelé).

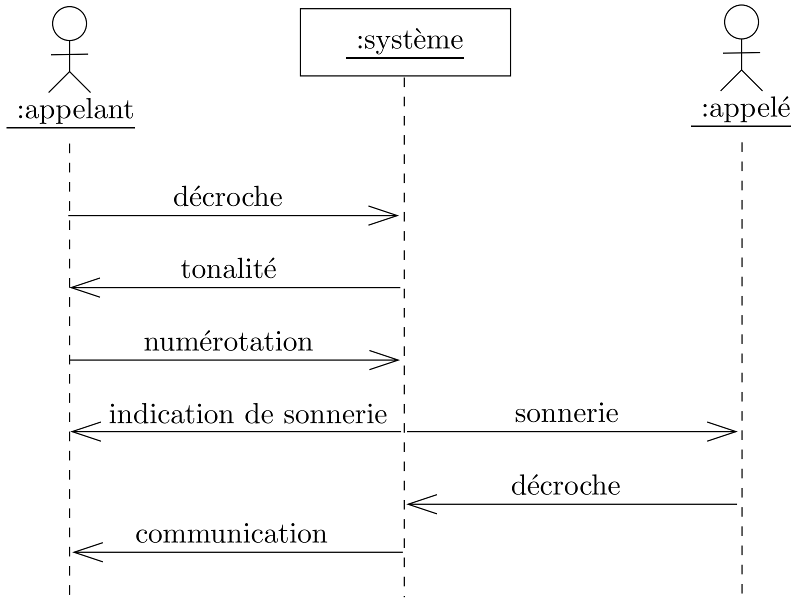


FIG. II. 50 – Diagramme de séquence représentant une communication téléphonique

Les diagrammes de séquence permettent également de représenter de façon précise les interactions entre objets qui composent le système. Dans ce cas, les messages correspondent à des appels de procédure ou à des signaux.

Les différentes sortes de message

On a trois sortes de messages :

1. Message *synchrone* : correspond à un appel de procédure ou flot de contrôle imbriqué. La séquence imbriquée est entièrement faite avant que l'appelant ne continue : l'appelant est bloqué jusqu'à ce que l'appelé termine.
2. Message *asynchrone* : correspond à un message « à plat », non imbriqué. L'appelant n'est pas bloqué jusqu'à ce que l'appelé termine.
3. Retour de procédure. Les messages qui indiquent des retours de procédure peuvent ne pas être dessinés (ils sont alors implicites).

La notation UML correspondant à chaque sorte de message est indiquée figure II. 51.

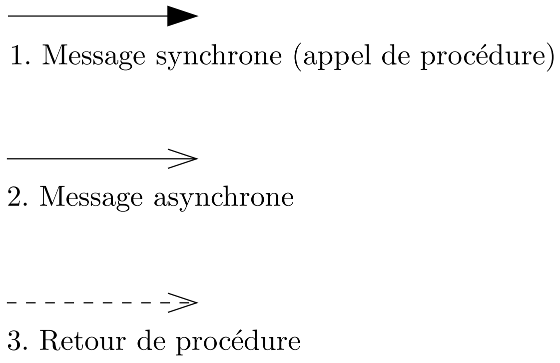


FIG. II. 51 – Les différentes sortes de message

Période d'activation d'un objet

La période d'activation d'un objet est la période de temps pendant laquelle un objet effectue une action, soit directement, soit par l'intermédiaire d'un autre objet qui lui sert de sous-traitant, par exemple lors d'un appel de procédure.

La période d'activation d'un objet est indiquée par un rectangle sur sa ligne de vie (cf. figure II. 52).

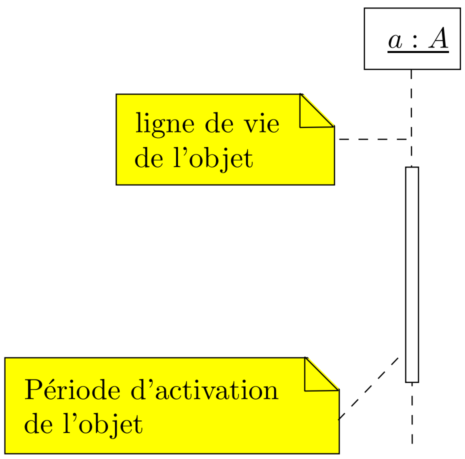


FIG. II. 52 – Période d'activation d'un objet

La figure II. 53 montre la période d'activation de différents objets lors de deux appels de procédure imbriqués.

Création et destruction d'objets

On a deux messages particuliers qui permettent de créer et de détruire des objets (cf. figure II. 54).

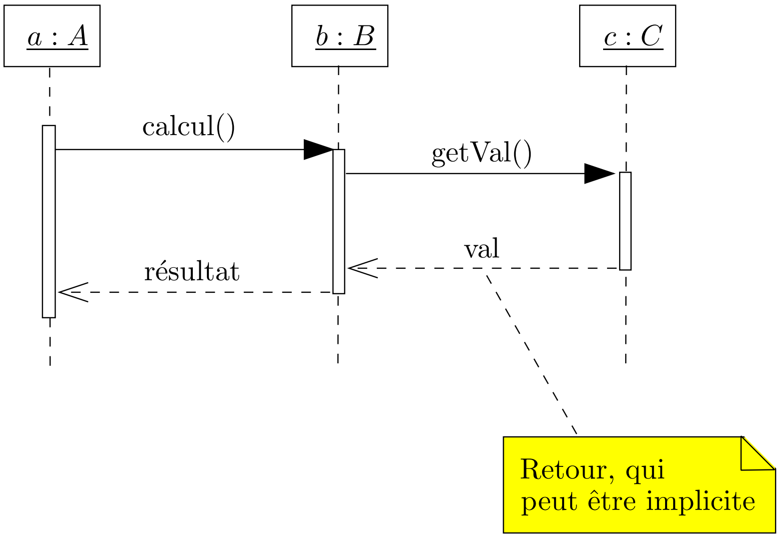


FIG. II. 53 – Périodes d'activation lors d'appels de procédure

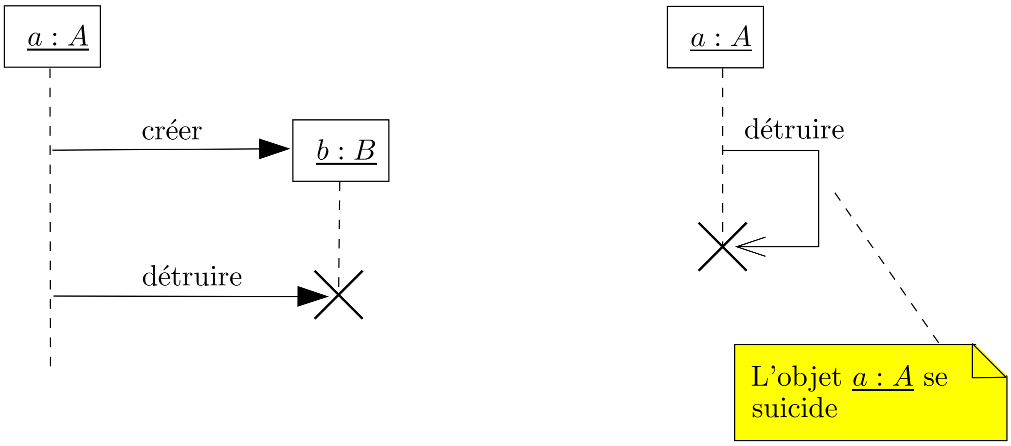


FIG. II. 54 – Création et destruction d'objets

Conditions

Les messages peuvent être conditionnels : le message est envoyé uniquement si la condition est satisfaite. Dans le diagramme de séquence figure II. 55, si la condition X est satisfaite, l'objet a envoie le message m_1 à l'objet b, sinon il envoie le message m_2 à l'objet c.

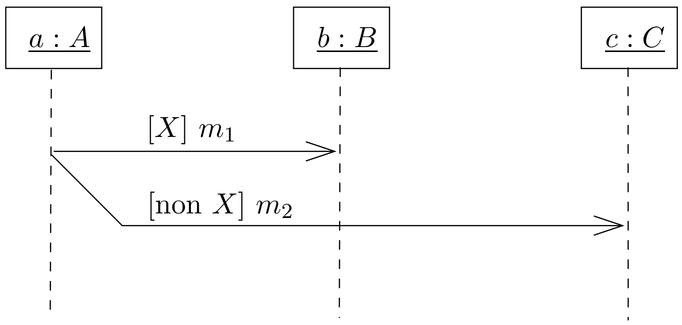


FIG. II. 55 – Messages conditionnels

La ligne de vie d'un objet peut être dédoublée pour indiquer des actions qui sont effectuées suite à un message conditionnel.

Dans le diagramme figure II. 56, si la condition X est satisfaite, alors l'objet a envoie le message m_1 à b, puis b envoie p_1 à c ; sinon a envoie le message m_2 à b, puis b envoie p_2 à c. Ensuite, dans les deux cas, a envoie le message m à b, puis b envoie p à c.

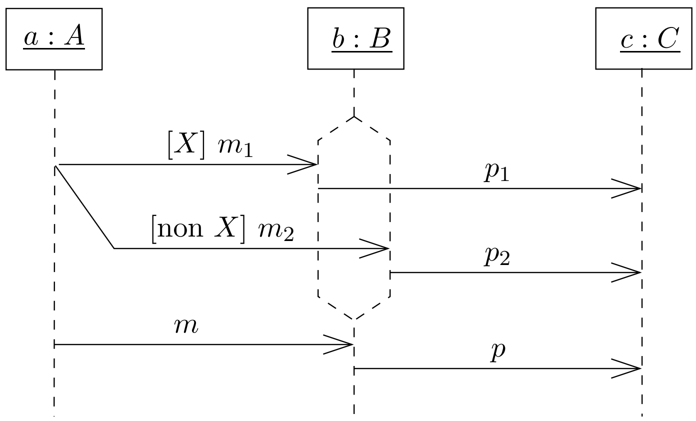


FIG. II. 56 – Dédoublement d'une ligne de vie

Contraintes temporelles

On peut nommer l'instant d'émission d'un message, ainsi que l'instant de réception. Cela permet de poser des contraintes de temps sur l'envoi et la réception de messages.

Par convention, lorsque l'instant d'émission d'un message est x, l'instant de réception est x'.

La figure II. 57 pose les deux contraintes suivantes :

- il s'écoule moins d'une seconde entre l'envoi des messages m_1 et m_2 ;

- il s'écoule moins de deux secondes entre l'envoi et la réception du message m_3.

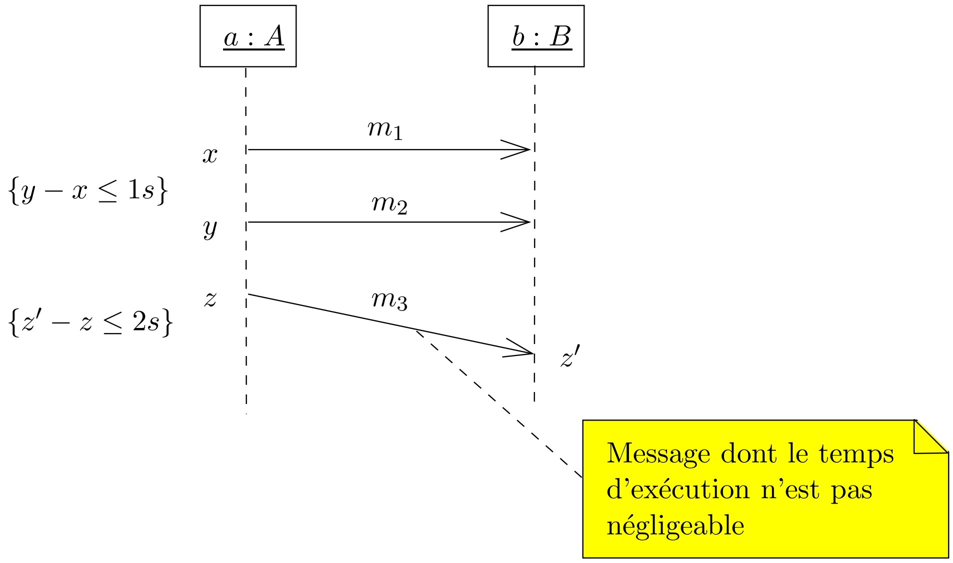


FIG. II. 57 – Contraintes temporelles sur l'envoi et la réception des messages

Objet actif

Un objet *actif* est un objet qui a son propre flot d'exécution. Les processus et les threads sont des exemples d'objets actifs. Les objets sources d'événements, comme les boutons dans les interfaces graphiques, sont des objets actifs. Les minuteries, comme les « timers » Java, sont des des objets actifs. Ces objets permettent d'exécuter une opération soit une seule fois, après un certain intervalle de temps, soit de façon répétée à intervalles de temps donnés.

Un objet actif peut activer, le temps d'une opération, un objet passif, qui peut alors activer d'autres objets passifs. Lorsque l'opération est terminée, l'objet passif redonne contrôle à l'objet qui l'a activé. Dans un environnement multi-tâches, plusieurs objets peuvent être actifs simultanément.

Un objet actif est noté en UML par un rectangle à bordure épaisse.

Dans l'exemple représenté figure II. 58, a et c sont des objets actifs, b est un objet passif. L'objet a active l'objet b par l'appel de procédure `erreur`, puis b envoie le message asynchrone `traitement` à c.

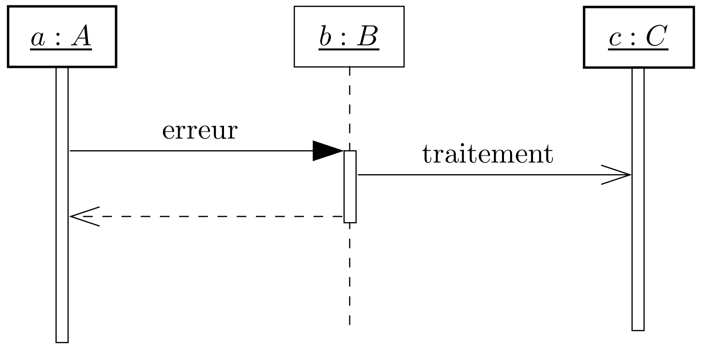


FIG. II. 58 – Les objets a et c sont actifs

5. Diagrammes de collaboration

Un diagramme de collaboration représente les interactions entre des objets (et éventuellement des acteurs) d'un point de vue spatial. Par opposition aux diagrammes de séquence, les liens entre les différents objets sont explicitement représentés. Pour mettre en évidence la dimension temporelle, les messages envoyés par les différents objets peuvent être numérotés.

La figure II. 59 montre un diagramme de collaboration qui représente une communication téléphonique.

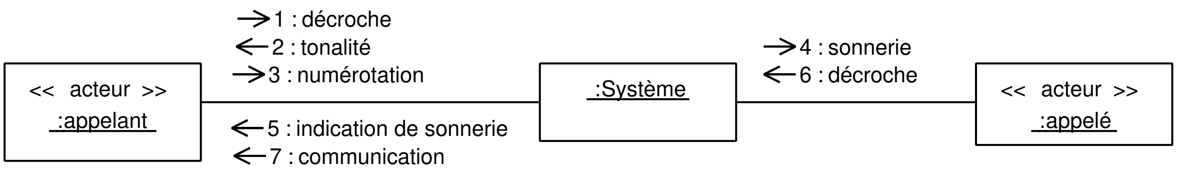


FIG. II. 59 – Diagramme de collaboration représentant une communication télephonique

Dans cet exemple, les messages sont asynchrones.

La figure II. 60 montre un diagramme de collaboration qui représente l'affichage d'une figure composée de segments.

Pour afficher une figure, on affiche l'ensemble des segments dont la figure est composée. Dans ce diagramme, :Segment est un multi-objet, qui représente l'ensemble des segments dont la figure est composée. Le message « `2 *:afficher` » signifie qu'on envoie le message afficher à chaque instance de la classe `Segment` en relation avec cette figure. Les messages sont des appels et des retours de procédure.

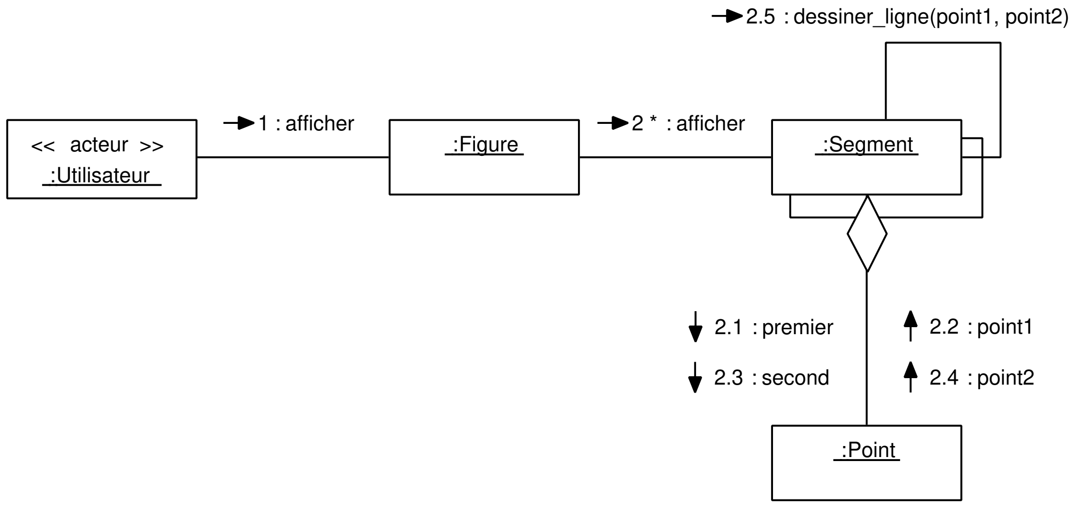


Fig. II. 60 – Diagramme de collaboration représentant l'affichage d'une figure

6. Diagrammes d'états-transitions

Les diagrammes d'états-transitions permettent principalement de décrire le comportement des objets d'une classe. Il peuvent également décrire les aspects dynamiques d'un cas d'utilisation, d'un acteur, d'un système ou d'un sous-système.

Les diagrammes d'états-transitions d'UML sont inspirés des « Statecharts » de David Harel. Il s'agit d'automates hiérarchiques, qui peuvent être mis en parallèle.

a) État

L'état d'un objet correspond à l'ensemble des valeurs de ses attributs et à l'ensemble des liens qu'il entretient avec d'autres objets.

L'état d'un automate peut être vu comme une abstraction représentant un ensemble d'états de l'objet.

Un état est une condition ou une situation, dans la vie d'un objet, qui dure un certain temps pendant lequel cet objet satisfait une condition, effectue une activité, ou attend un événement.

Si on reprend l'exemple des personnes employées dans des entreprises, on peut considérer les états suivants pour une personne : en activité, à la retraite et sans emploi (cf. figures II. 61 et II. 62).

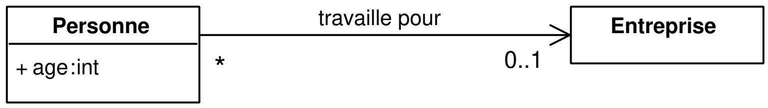


Fig. II. 61 – Diagramme de classes qui représente l'emploi des personnes

Ici, l'abstraction qui nous intéresse est l'emploi d'une personne. On pourrait s'intéresser à

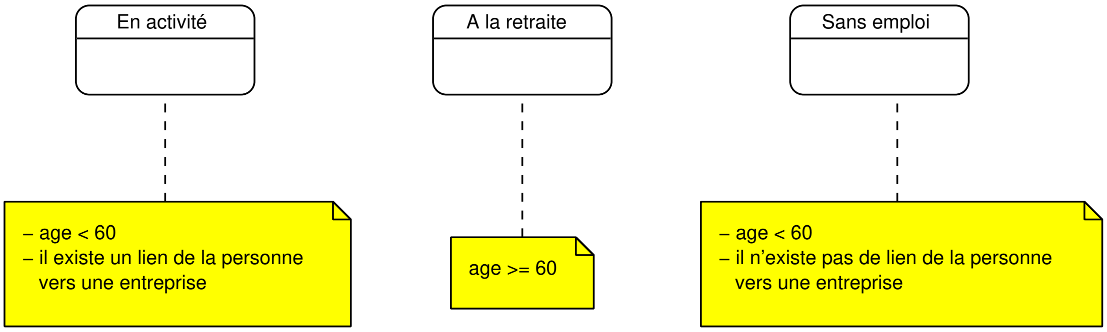


FIG. II. 62 – Trois états possibles pour une personne

d'autres abstractions.

Etats initial et final

Un état initial est un pseudo-état qui permet de montrer l'état dans lequel un objet se trouve au moment de sa création. Un état final est un pseudo-état qui permet de montrer la fin du comportement d'un objet, en particulier le moment de sa destruction.

La figure II. 63 montre la notation UML pour les pseudo-états initiaux et terminaux.

FIG. II. 63 – Pseudo-états initial et terminal

b) Transition

Les transitions permettent à un objet de changer d'état, en fonction des événements qu'il reçoit.

Une transition comporte un état source, un état destination, un événement, une condition (appelée garde) et une action.

Supposons que l'objet soit dans l'état A (figure II. 64). Si l'événement se produit et si la condition est vraie, alors l'objet effectue l'action et passe dans l'état B. Le passage d'un état à l'autre est considéré comme instantané (l'action doit donc également pouvoir être considérée comme instantanée).

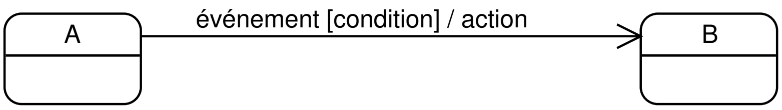


FIG. II. 64 – Transition entre les états A et B

Si aucune transition n'est étiquetée par l'événement reçu, alors rien ne se passe, et, en particulier, l'objet ne change pas d'état.

c) Événement

On a quatre sortes d'événements en UML.

- Un *événement d'appel* (« call event ») est un événement causé par l'appel d'une opération. Dans ce cas, l'événement est de la forme $op(x_1, x_2, \ldots x_n)$, où op est une opération de la classe.
- Un *événement modification* (« change event ») est un événement causé par le passage d'une condition de la valeur faux à la valeur vrai, suite à un changement de valeur d'un attribut ou d'un lien.
- Un *événement temporel* (« time event ») est un événement qui survient quand une temporisation arrive à expiration. Une temporisation peut être relative (délai), ou absolue (spécification de l'heure à laquelle une transition doit être effectuée).
- Un *événement signal* (« signal event ») est un stimulus asynchrone entre deux objets. Par exemple, un clic de souris est un signal.

Exemple 1

Dans la classe Personne, on considère les deux opérations embauche et perteDEmploi (cf. figure II. 65). La figure II. 66 est un diagramme d'états-transitions associé à la classe Personne.

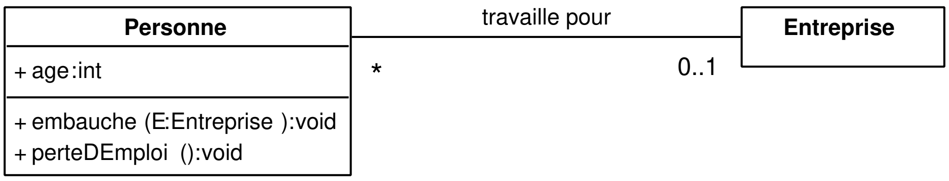


FIG. II. 65 – Diagramme de classes qui représente l'emploi des personnes

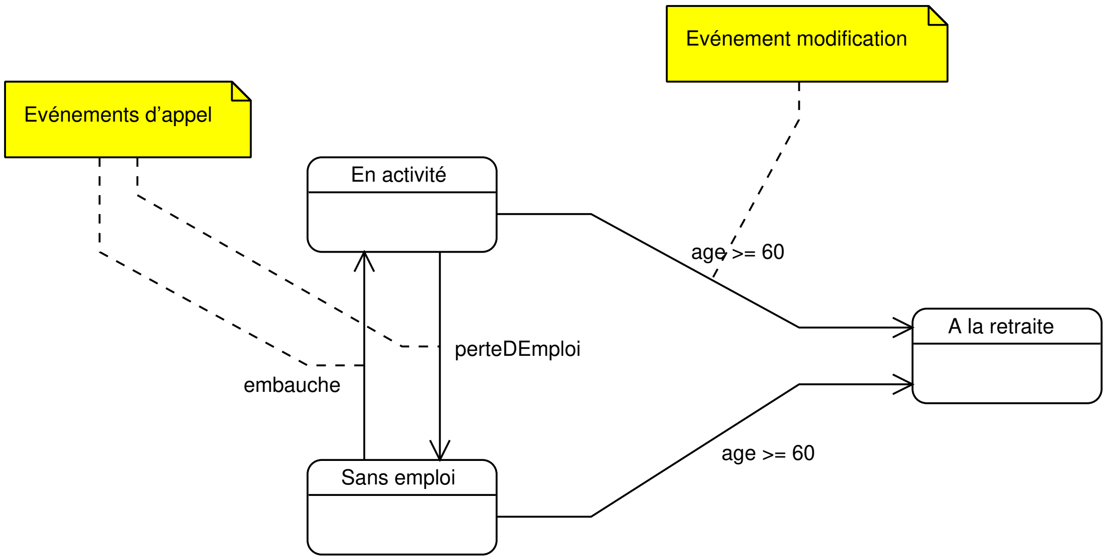


FIG. II. 66 – Diagramme d'états-transitions associé à la classe Personne

Exemple 2

On considère une machine qui comporte deux boutons : on a un bouton pour mettre la machine sous tension (signal : on) et un bouton pour mettre la machine hors tension (signal : off). Un voyant indique si la machine est sous tension ou hors tension. Après une minute sans utilisation, la machine se met automatiquement hors tension.

Les figures II. 67, II. 68 et II. 69 montrent un diagramme de classes de la machine et des diagrammes d'états-transitions associés aux classes Voyant et Machine.

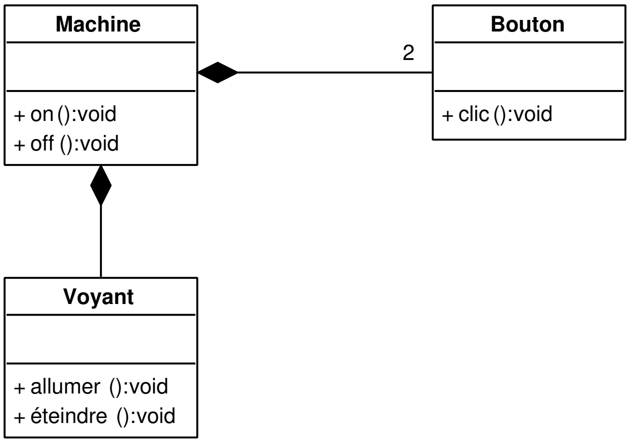


FIG. II. 67 – Diagramme de classes de la machine

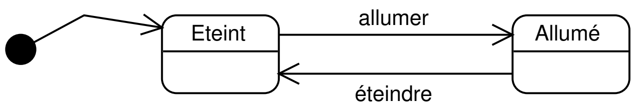


FIG. II. 68 – Diagramme d'états-transitions associé à la classe Voyant

Remarque

Les automates considérés en UML sont *a priori* non-déterministes. On peut donc avoir deux transitions étiquetées par le même événement qui partent du même état. Néanmoins, d'un point de vue méthodologique, il est souvent préférable d'utiliser des automates déterministes pour plus de clarté.

d) Garde

Une garde est une condition booléenne notée entre crochets. Une garde est évaluée lorsque l'événement se produit.

Supposons que plusieurs transitions partant du même état A soient déclenchées par le même événement (cf. figure II. 70). Pour que l'automate soit déterministe, il faut que les gardes c_1, c_2 et c_3 soient mutuellement exclusives.

Si aucune condition n'est vérifiée, alors rien ne se passe et l'objet ne change pas d'état.

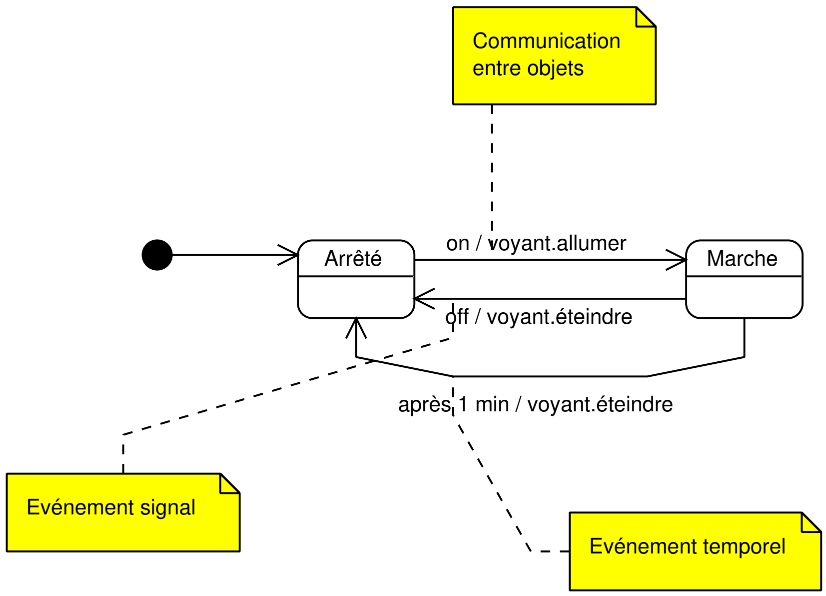


FIG. II. 69 – Diagramme d'états-transitions associé à la classe Machine

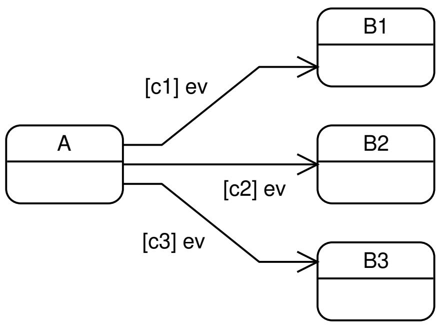


FIG. II. 70 – Transitions gardées, étiquetées par le même événement

Remarque

Il ne faut pas confondre un événement de changement et une garde. Un événement de changement est un événement qui déclenche la transition lorsque la condition passe à vrai ; une garde est une condition booléenne évaluée lorsque l'événement se produit.

e) Action et activité

Une action consiste en la génération d'un signal ou l'invocation d'une opération. Une action est considérée comme *instantanée* (c'est-à-dire dont le temps d'exécution est négligeable) et *atomique* (c'est-à-dire non interruptible).

Une activité correspond à une opération qui prend un temps non négligeable et peut être interrompue.

Les actions sont généralement associées aux transitions, mais on peut également les associer aux états. On peut en particulier :

- spécifier une action à effectuer lorsqu'on entre dans un état : *entry/* action ;
- spécifier une action à effectuer si un événement survient : on événement/ action (événement « interne ») ;
- spécifier une action à effectuer lorsqu'on sort d'un état : *exit/* action ;
- spécifier une *activité* effectuée lorsqu'on est dans l'état : *do/* activité. L'activité peut prendre un certain temps, et être interrompue.

Remarque

Le déclenchement d'un événement interne n'entraîne pas l'exécution des actions d'entrée et de sortie.

Dans l'exemple représenté figure II. 71, on suppose qu'on entre dans l'un des états E1 ou E2, qu'on reçoit une suite d'événements e, puis qu'on sort de l'état.

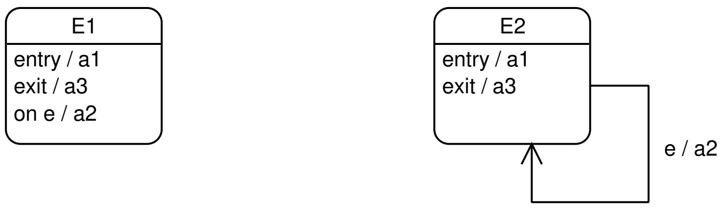


Fig. II. 71 – Action associée à un événement interne et à un événement d'une transition

Dans l'état E1, les actions a_1 et a_3 sont effectuées une seule fois, lors de l'entrée et de la sortie de l'état. Les séquences d'actions générées sont donc de la forme $a_1\, a_2^*\, a_3$. Dans l'état E2, les actions a_1 et a_3 sont effectuées à chaque réception de l'événement e. Les séquences d'actions générées sont donc de la forme $a_1\, (a_3\, a_2\, a_1)^*\, a_3$.

f) États composites

Un état *composite* est un état qui se décompose en plusieurs sous états. Les états composites permettent de structurer les automates pour les rendre plus lisibles.

Les états composites permettent en particulier de factoriser des transitions similaires qui partent de plusieurs états.

On reprend le diagramme d'états-transitions de la figure II. 66. On introduit l'état composite « Age inférieur à 60 » pour les états « En activité » et « Sans emploi ». On peut ainsi factoriser les deux transitions « age >= 60 » (cf. figure II. 72).

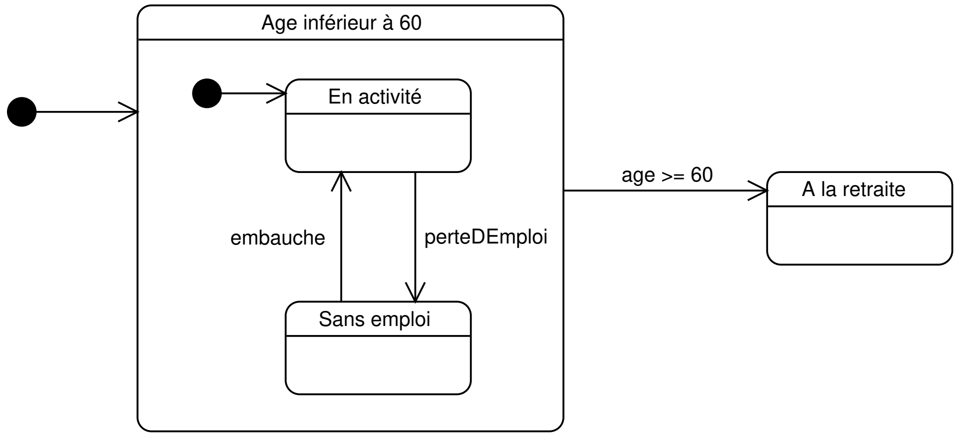


Fig. II. 72 – Exemple d'état composite

Il peut exister des transitions entre différents niveaux de l'automate, mais il est préférable de limiter les transitions entre différents niveaux.

Lorsque deux transitions peuvent être effectuées, l'une sur un sous-état et l'autre sur l'état englobant, c'est la transition sur le sous-état (la plus « spécialisée ») qui est effectuée.

Dans l'exemple figure II. 73, si l'objet est dans l'état X et si l'événement e survient, alors l'objet passe dans l'état Y (et non dans l'état Z ou T).

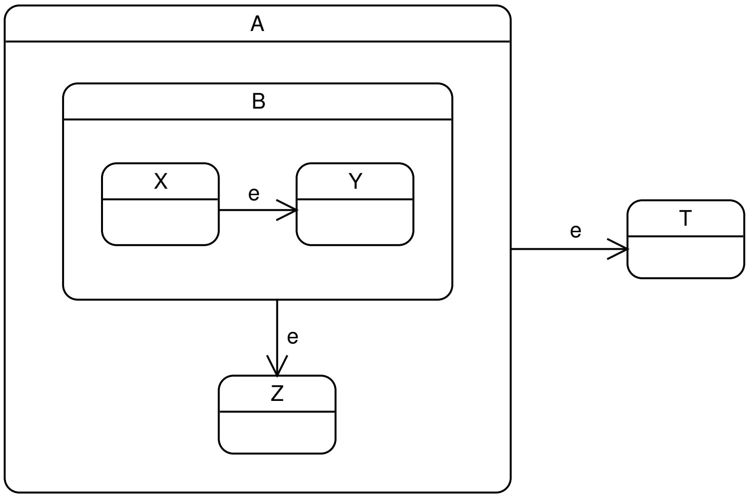


Fig. II. 73 – L'objet passe de l'état X à l'état Y

g) Indicateurs d'historique

L'indicateur d'historique, noté « $\textcircled{H}$ », est un pseudo état qui permet de mémoriser le dernier état visité d'un automate pour y retourner ultérieurement.

La figure II. 74 est un diagramme d'états-transitions d'une machine à laver. Si on souhaite ouvrir la porte lorsque la machine est en marche, il faut interrompre le programme et la vidanger. Une fois la porte refermée, la machine peut reprend son cycle dans l'état où elle s'était arrêtée lors de l'interruption.

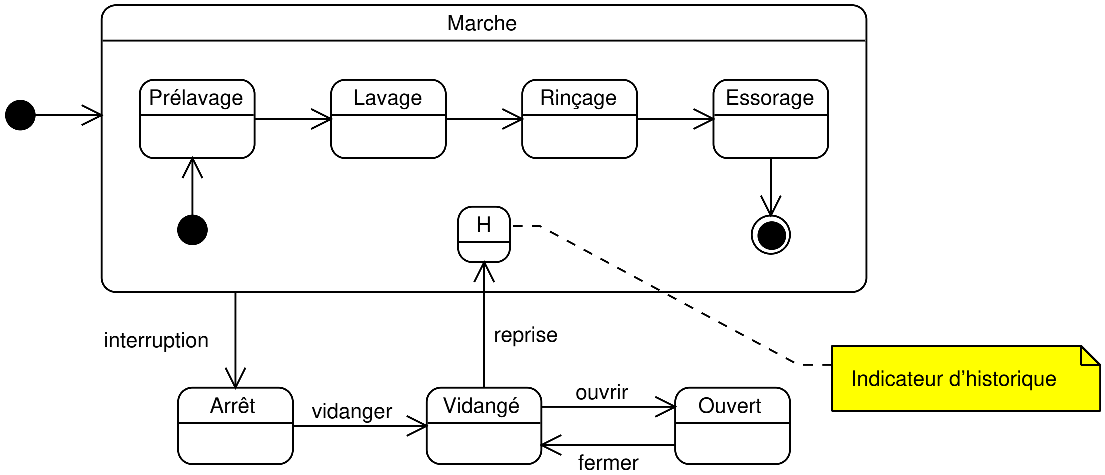


FIG. II. 74 – Diagramme d'états-transitions d'une machine à laver

La transition reprise a pour cible l'indicateur d'historique. Lorsque la transition reprise est effectuée, l'automate reprend son exécution dans l'état où il se trouvait lorsque la transition interruption s'est produite, c'est-à-dire l'un des états Prélavage, Lavage, Rinçage ou Essorage.

L'indicateur d'historique à un niveau quelconque, noté « $\textcircled{H^*}$ », est un pseudo état qui permet de mémoriser le dernier état visité, à un niveau d'imbrication quelconque, pour y retourner ultérieurement.

Par exemple, dans la figure II. 75, lorsque la transition reprise est effectuée, l'automate revient dans l'état où il était lors de la transition interruption, à un niveau quelconque, autrement dit, dans l'état A, B ou C.

h) Automates en parallèle

À l'intérieur d'un état, plusieurs automates peuvent s'exécuter en parallèle. Chaque sous-automate a un état initial et un certain nombre d'états terminaux.

L'activité d'un tel état se termine lorsque tous les sous-automates parviennent à un état final.

Lorsqu'un événement se produit, toutes les transitions qui peuvent être effectuée sont effectuées.

La figure II. 76 montre un exemple d'état qui contient deux sous-automates s'exécutant en

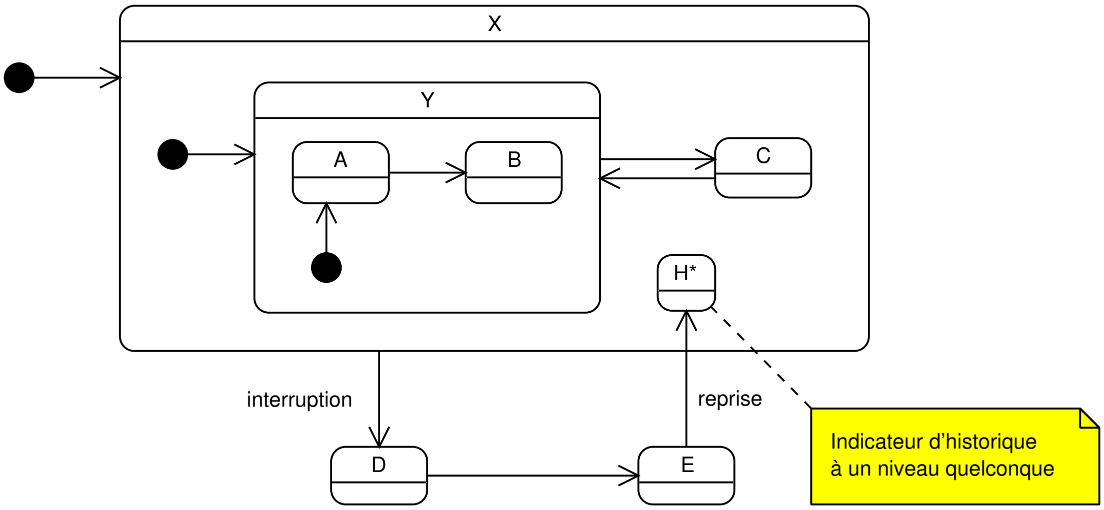


FIG. II. 75 – Indicateur d'historique à niveau quelconque

parallèle.

Si A_1 est dans l'état X et A_2 est dans l'état A, et si l'événement e_1 se produit, alors A_1 passe dans l'état Y et A_2 passe dans l'état B. Si A_1 est dans l'état Y et A_2 est dans l'état B, et si l'événement e_2 se produit, alors A_1 passe dans l'état X et A_2 reste dans l'état B.

L'automate de la figure II. 76 est équivalent à l'automate « aplati » représenté figure II. 77.

Un cas typique où on peut utiliser des automates en parallèle est lorqu'un objet est formé de la composition ou de l'agrégation d'autres objets, en particulier si le comportement de l'objet composite correspond à la mise en parallèle des comportements des composants.

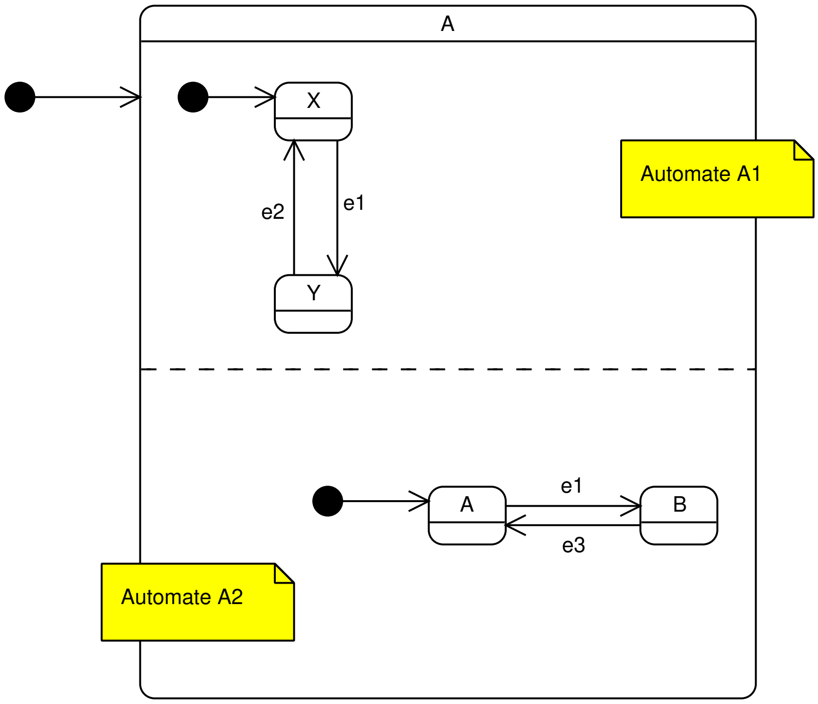


Fig. II. 76 – L'état A contient deux automates qui s'exécutent en parallèle

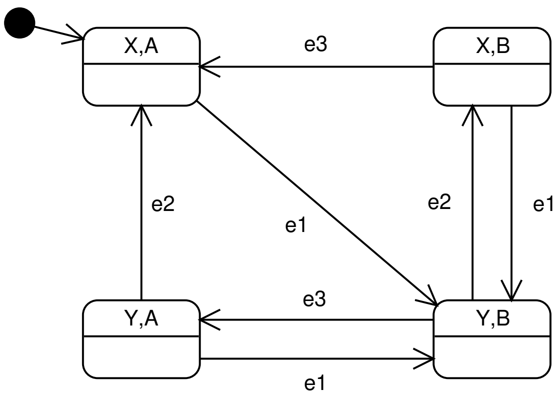


Fig. II. 77 – Automate « aplati » équivalent

7. Diagrammes d'activités

Les diagrammes d'activités permettent de modéliser le comportement d'une méthode ou le déroulement d'un cas d'utilisation, en représentant un enchaînement d'activités.

La figure II. 78 est un diagramme d'activités représentant la commande d'un produit.

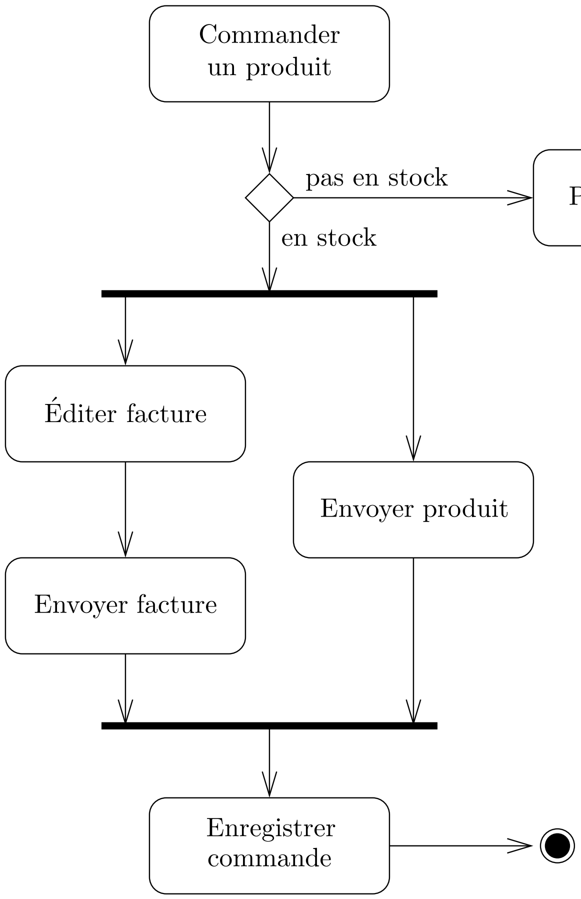


Fig. II. 78 – Diagramme d'activités pour une commande de produits

8. Diagrammes de composants

Un diagramme de composants permet de décrire l'architecture physique d'une application en termes de modules : fichiers sources, librairies exécutables... etc.

La figure II. 79 est un diagramme de composants d'une application construite à partir de deux fichiers Java et d'une librairie mathématique.

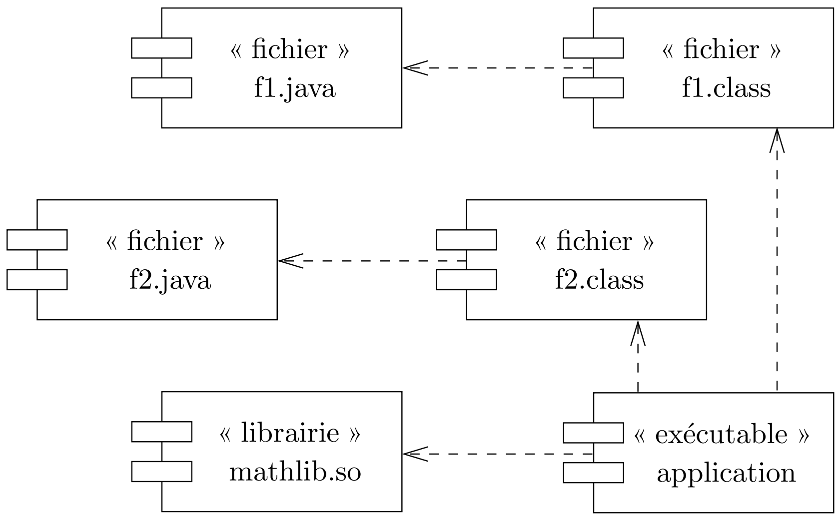


Fig. II. 79 – Diagramme de composants d'une application

La figure II. 80 est un diagramme de composants d'un serveur Oracle utilisant une base de données qui stocke un catalogue de produits.

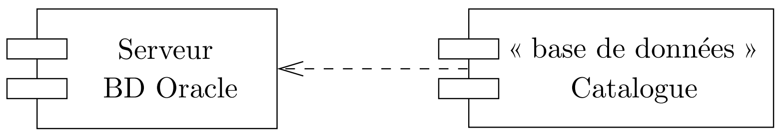


Fig. II. 80 – Diagramme de composants d'une application

9. Diagrammes de déploiement

Le diagramme de déploiement décrit la disposition physique des matériels et la répartition des composants sur ces matériels.

La figure II. 81 est un diagramme de déploiement d'une application utilisant une base de donnée distante.

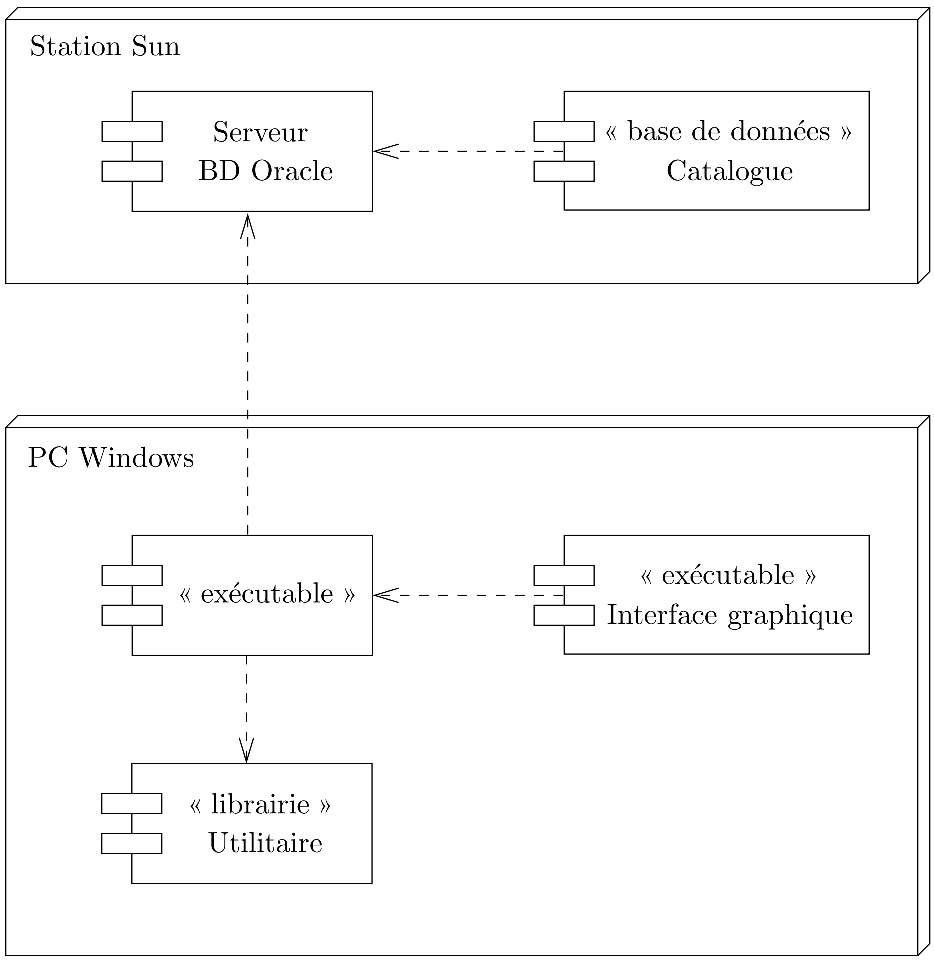


Fig. II. 81 – Diagramme de déploiement d'une application

Chapitre III

Analyse

Les objectifs de l'analyse sont les suivants :

- comprendre les besoins du client ;
- effectuer une étude de faisabilité ;
- obtenir une bonne compréhension du domaine ;
- éliminer le maximum d'ambiguïtés du cahier des charges ;
- obtenir une première ébauche de la structure du système.

Dans ce chapitre, on s'intéresse d'abord à l'expression des besoins, qui a pour but de comprendre et reformuler les besoins des utilisateurs, puis à la modélisation objet du domaine, qui permet de mettre en évidence les concepts significatifs du domaine d'application.

1. Expression des besoins

a) Gestion des besoins

La gestion des besoins consiste à comprendre, exprimer et mémoriser les besoins du système sous une forme compréhensible par le client et l'équipe de développement. Les besoins ne sont pas figés une fois pour toutes au début du projet, mais sont amenés à évoluer, en particulier si on utilise un modèle de cycle de vie itératif.

Il est important de bien contrôler la gestion et l'évolution des besoins, car un tiers des problèmes rencontrés lors du développement de logiciels viennent de cette étape. On rencontre en particulier les problèmes suivants :

- les utilisateurs peuvent fournir des informations erronées ;
- les besoins du système peuvent être incomplets ;
- les besoins des utilisateurs évoluent d'une façon incontrôlée.

Ces problèmes ont des répercussions importantes sur la qualité du logiciel, le budget, les délais de livraison. Pour les éviter, il est important d'impliquer les utilisateurs dans l'étape d'expression des besoins.

b) Types de besoins

On distingue en général les besoins fonctionnels, qui concernent les fonctionnalités du logiciel, et les besoins non fonctionnels.

Parmi les besoins non fonctionnels, on trouve :

- la fiabilité (robustesse, possibilité de récupération après une panne) ;
- la facilité d'utilisation (ergonomie, aide, documentation) ;
- efficacité (temps de réponse) ;
- portabilité ;
- maintenabilité (facilité à corriger des erreurs, à faire des améliorations ou des adaptations du logiciel) ;
- effort de validation (tests, couverture des tests).

c) Analyse des besoins

L'analyse des besoins consiste à comprendre les besoins du client. Ces besoins sont, en général, décrits par un cahier des charges, qui décrit le système et son environnement.

On détermine les besoins fonctionnels et non fonctionnels par une analyse du cahier des charges. Il s'agit d'un analyse de texte. On peut par exemple déduire les aspects fonctionnels des verbes utilisés dans le cahier des charges. Les substantifs permettront de déduire les objets du système (cf. modélisation objet). Il faut tenir compte du fait que la langue naturelle est souvent imprécise ou ambiguë : certains mots ne sont pas pertinents, plusieurs mots peuvent avoir la même signification, plusieurs concepts peuvent correspondre au même mot. De plus, le cahier des charges comprend souvent une grande part d'implicite : il suppose que le lecteur a une bonne connaissance du domaine, ce qui n'est pas toujours le cas.

Le but de cette analyse de texte est de préciser le cahier des charges et de lever certaines ambiguïtés. En particulier, il peut être utile de rédiger un glossaire qui définit les principaux termes utilisés.

L'analyse du cahier des charges est en générale insuffisante pour comprendre entièrement les besoins. Le cahier des charges est en effet non seulement imprécis et ambigu, mais également incomplet, voire contradictoire. La compréhension des besoins nécessite alors d'utiliser d'autres méthodes qui consistent à communiquer avec les utilisateurs. Cette communication peut prendre différentes formes :

- on peut organiser des entrevues avec les utilisateurs, les techniciens, les gestionnaires ;
- on peut demander aux utilisateurs de remplir des questionnaires, en particulier lorsqu'il est nécessaire d'obtenir des données précises auprès d'un grand nombre de personnes. Cette méthode présente un inconvénient : les personnes interrogées sont souvent peu motivées pour répondre.
- on peut observer les activités des utilisateurs sur site, afin de mieux comprendre leurs besoins. Inconvénient : certains utilisateurs peuvent modifier leur méthode de travail lorsqu'ils se savent observés.

À l'issue de l'analyse des besoins, les documents suivant peuvent être produits :

- un cahier des charges, qui comporte :
  - une description de l'environnement du système : les machines, le réseau, les périphériques (imprimantes, capteurs...), l'environnement physique... etc.
  - le rôle du système ;
- un glossaire, qui définit les principaux termes utilisés dans le cahier des charges ;
- un manuel utilisateur ;
- un document de spécification globale, précisant les besoins fonctionnels et non fonctionnels du système.

d) Expression des besoins fonctionnels

Plusieurs diagrammes UML peuvent être utilisés pour exprimer les besoins fonctionnels du système :

- cas d'utilisation ;
- diagrammes de séquence ;
- diagrammes d'états-transitions.

Rédaction de cas d'utilisation

La rédaction de cas d'utilisation permet de reformuler les besoins fonctionnels du système. Les cas d'utilisation ont été introduits en 1986 par Ivar Jacobson.

La rédaction de cas d'utilisation permet de :

- comprendre et clarifier le cahier des charges, à travers une reformulation ;
- structurer les besoins.

Un cas d'utilisation décrit ce que le système doit faire, du point de vue des utilisateurs, sans décrire comme cela sera implémenté. Il s'agit donc d'un document de spécification, et non de conception.

Un cas d'utilisation :

- décrit une manière spécifique d'utiliser le système ;
- regroupe une famille scénarios d'utilisation du système.

On utilise le langage naturel et on utilise la terminologie des utilisateurs. Les cas d'utilisations doivent être compréhensibles à la fois par les utilisateurs et les développeurs. Un cas d'utilisation regroupe une famille de scénarios d'utilisation suivant des critères fonctionnels. C'est une abstraction du dialogue entre les acteurs et le système. Les scénarios seront décrits ensuite par d'autres diagrammes (diagrammes d'interaction, diagrammes d'états-transitions).

Pour exprimer ces besoins fonctionnels, on commence par définir les acteurs.

Un acteur est quelqu'un ou quelque chose qui interagit avec le système (personne, environnement, autre système). Plus précisément, il s'agit du *rôle* joué par quelqu'un ou quelque chose qui interagit avec le système.

La détermination des acteurs permet de préciser les limites du système.

Lorsqu'il y a beaucoup d'acteurs, on les regroupe par catégories. On peut distinguer :

- les acteurs *principaux* : personnes qui utilisent les fonctions principales du système ;
- les acteurs *secondaires* : personnes qui effectuent des tâches administratives ou de maintenance ;
- le matériel externe : les dispositifs matériels périphériques ;
- les autres systèmes (avec lesquels le système interagit).

Chaque cas d'utilisation peut comprendre :

- une pré-condition (condition qui doit être satisfaite pour que la fonctionnalité puisse être utilisée) ;
- une description de la suite des interactions entre le système et les acteurs, en distinguant les différents scénarios possibles. On décrit en particulier la répétition des comportements et les échanges d'information.
- une post-condition (condition satisfaite une fois l'interaction terminée) ;
- des exceptions (cas exceptionnels, ou ce qui se passe si une pré-condition n'est pas satisfaite).

La description des cas d'utilisation est faite en langue naturelle. Par rapport au cahier des charges, cette description doit être plus précise, mieux structurée, et décrire précisément les interactions entre le système et l'utilisateur. Il s'agit donc d'une reformulation et d'une clarification du cahier des charges.

Pour être plus précis, cette description évitera par exemple l'utilisation de pronoms impersonnels (« on »), mais utilisera à la place « l'utilisateur » ou « le système ». Cette description évitera les synonymes, les termes flous ou non définis dans le glossaire.

Pour que la description soit mieux structurée que le cahier des charges, elle pourra comporter des énumérations à différents niveaux, par exemple :

1. Le système demande à l'utilisateur d'entrer son code secret.
   - 1.1. L'utilisateur entre son code secret et termine en appuyant sur « valider ».
   - 1.2. Le système vérifie que le code secret entré par l'utilisateur est correct.
     - 1.2.1. Si le code est correct...
     - 1.2.2. Si le code est incorrect...
2. ...

Les cas d'utilisation structurent les différentes fonctionnalités du logiciel. Cette structuration peut être utilisée dans l'ensemble du développement du logiciel, de la spécification aux tests. Ils permettent en particulier d'aider à la planification des versions successives, si on utilise un développement incrémental du logiciel. Par exemple, un incrément peut consister à réaliser (de la spécification aux tests) un ou plusieurs cas d'utilisation.

Diagrammes de séquence

Les diagrammes de séquence permettent d'illustrer les différents cas d'utilisations : chaque cas d'utilisation est accompagné de plusieurs diagrammes de séquence qui montrent différents scénarios de fonctionnement du système logiciel.

On utilise des « diagrammes de séquence système », c'est-à-dire des diagrammes de séquence qui représentent les interactions entre les acteurs et le système, sous forme d'actions et de réactions. Les actions internes au système ne sont, en général, pas représentées.

Le diagramme de séquence représenté figure III. 1 décrit un scénario d'utilisation du téléphone.

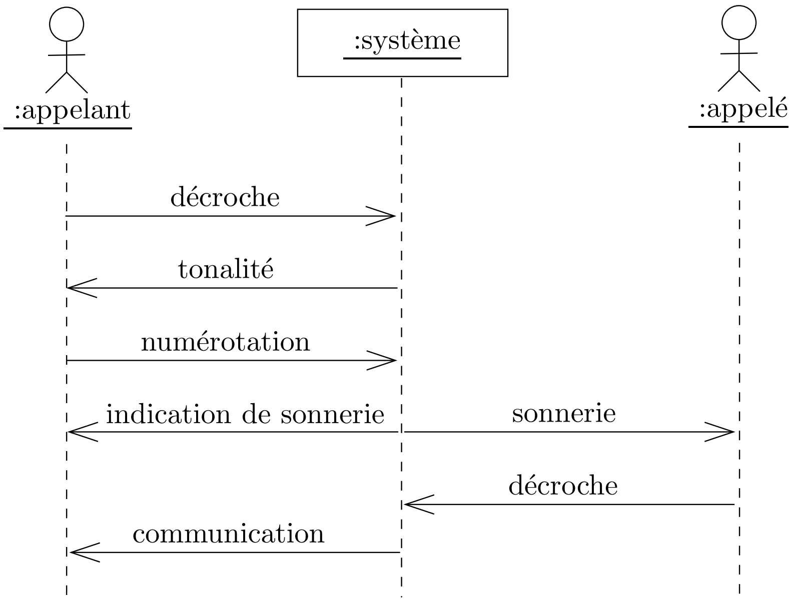


FIG. III. 1 – Diagramme de séquence représentant une communication téléphonique

Ce scénario peut se dérouler sous un certain nombre d'hypothèses, ou pré-conditions :

- le téléphone de l'appelé est correctement branché ;
- la numérotation correspond à un numéro de téléphone correct ;
- le téléphone de l'appelant est correctement branché ;
- l'appelé n'est pas déjà en communication ;
- l'appelé décroche le téléphone ;
- … etc.

Pour décrire de tels scénarios, il est nécessaire de définir les événements auxquels le système doit réagir et les événements que le système peut générer.

Les diagrammes de séquences doivent être compréhensibles par le client. Pour cela, comme pour les cas d'utilisation, on utilise la terminologie des utilisateurs.

Diagrammes d'états-transitions

Les diagrammes d'états-transitions permettent de spécifier le comportement général du système, en précisant :

- les états possibles du système ;
- tous les enchaînements possibles d'opérations.

On utilise souvent des « diagrammes d'états-transitions de protocole ». Il s'agit de diagrammes d'états-transitions dans lesquels on ne représente que les événements auxquels le système peut réagir, accompagnés de conditions. On ne représente pas les actions internes au système. Ces diagrammes permettent de définir l'ensemble des séquences d'événements intéressantes pour utiliser une fonctionnalité du système.

2. Modélisation objet du domaine

La modélisation objet du domaine consiste à élaborer un *modèle objet du domaine*. C'est un des modèles les plus importants à créer lors d'une analyse objet, qui constitue une source d'inspiration pour la conception du système.

a) Modèle du domaine

Un modèle du domaine est constitué d'un ou plusieurs diagrammes de classes qui représentent les classes significatives du domaine d'application. Ces classes, appelées « classes conceptuelles », représentent des objets du monde réel, qui existent indépendamment du système.

Les classes conceptuelles ne sont pas destinées à représenter des objets purement logiciels, comme des classes Java, des fenêtres, une interface graphique, des tables d'une base de données... etc.

Ces diagrammes de classes sont constitués de classes dans lesquels aucune opération n'est définie. Ils comportent plus précisément :

- les classes conceptuelles ;
- les attributs de ces classes ;
- les associations entre ces classes.

L'objectif est d'identifier les éléments du monde réel qui sont utiles pour le système, et de faire abstraction des détails inutiles.

b) Classes conceptuelles

Une classe conceptuelle comporte :

- un *nom* ;
- une *intention*, qui indique ce que représente cette classe, en incluant le rôle de ses attributs ;
- une *extension*, qui décrit l'ensemble des objets qui sont instances de la classe, à un instant donné.

Pour identifier les classes conceptuelles, une technique consiste à repérer les noms et les groupes nominaux dans le cahier des charges. Il faut ensuite effectuer une classification de ces noms : le cahier des charges peut comporter des synonymes (deux mots différents représentent le même concept) et des ambiguïtés (un même mot représente deux concepts distincts).

Il est souvent utile d'introduire des classes qui permettent de représenter des objets qui sont des descriptions ou des spécifications d'autres objets. Par exemple, si on manipule des produits, on utilise une classe Produit. Il peut être utile de stocker des informations sur ces produits, comme le prix, la référence... etc., qui sont communes à plusieurs produits similaires. On introduit pour cela une classe SpécificationProduit. On a une relation entre les produits et leur spécification : à chaque produit est associé sa spécification et à une spécification est associée l'ensemble des produits ayant cette spécification.

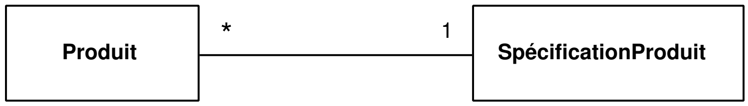


Fig. III. 2 – Produit et spécification de produit

c) Relations entre les classes conceptuelles

Lors de l'identification des relations entre classes conceptuelles, il faut éviter d'avoir un nombre trop important de relations. Pour cela, on se concentre sur les liens qui devront être stockés un certain temps dans le système. D'autre part, on évite de faire apparaître les relations *dérivées*, c'est-à-dire qui peuvent se déduire d'autres relations.

Parmi les associations, il faut identifier les agrégations et les compositions. On introduit ces relations lorsqu'on veut faire apparaître qu'un élément « appartient » à, ou « est contenu » dans un autre élément. On a une composition lorsque le composant ne peut pas être partagé entre deux parties.

Une fois qu'on a identifié un certain nombre de classes, il est utile de construire des généralisations. Pour cela, on identifie les éléments (attributs, associations) qui sont communs à plusieurs classes ; si ces éléments correspondent à une abstraction cohérente, on peut définir une classe qui comporte ces éléments et qui généralise les classes de départ.

d) Attributs

Dans le modèle du domaine, les attributs doivent être d'un type primitif (entier, réel, booléen, chaîne de caractères...). Lorsqu'on a besoin de stocker des valeurs de type non primitif, il est en général préférable d'introduire une classe et une association. Lors de l'implantation, par exemple en Java ou C++, ces associations seront codées par des attributs de type non primitif. On introduira en particulier des classes lorsque :

- la valeur à stocker est composée de plusieurs valeurs ;
- des opérations sont associées à cette valeur (par exemple une vérification que la valeur est correcte) ;
- il s'agit d'une abstraction représentant plusieurs autres types.

Par exemple, il peut être utile de stocker un montant sous la forme d'un objet d'une classe, en particulier si le paiement peut être effectué dans différentes monnaies, et que des conversions sont nécessaires.

Chapitre IV

Conception

Dans ce chapitre, on s'intéresse à la conception de logiciels, en particulier à la conception architecturale, qui consiste à déterminer la structure générale du système, et à la conception détaillée.

1. Architecture logicielle

L'architecture logicielle décrit la structure générale du logiciel en constituants de haut niveau, ainsi que l'interaction entre ces éléments.

a) Description d'une architecture

Le modèle des $4+1$ vues de Kruchten représenté figure IV. 1 permet de décrire l'architecture d'un système selon un ensemble de cinq vues.

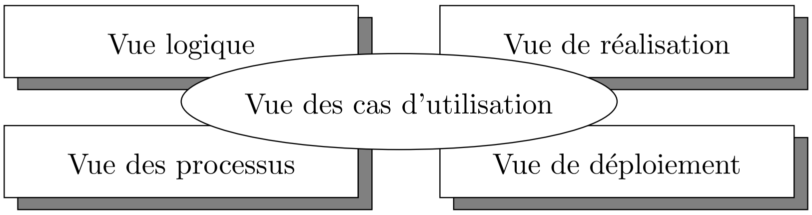


Fig. IV. 1 – Le modèle des $4+1$ vue de Philippe Kruchten

Vue logique

La vue logique de l'architecture décrit l'organisation du système en sous-systèmes, couches, paquetages, classes et interfaces. Un paquetage regroupe un ensemble de classes et d'interfaces ; un sous-système un sous-ensemble du système, fournissant un ensemble d'interfaces et d'opérations.

Vue de réalisation

La vue de réalisation concerne l'organisation des différents fichiers (exécutables, code source, documentation...) dans l'environnement de développement, ainsi que la gestion des versions et des configurations. Cette vue peut être en partie décrite à l'aide d'un diagramme de composants.

Vue des processus

La vue des processus représente la décomposition en différents flots d'exécution : processus, fils d'exécution (threads). Cette vue est importante dans les environnements multi-tâches.

Vue de déploiement

La vue de déploiement décrit les différentes ressources matérielles et l'implantation du logiciel sur ces ressources. Cette vue concerne les liens réseau entre les machines, les performances du système, la tolérance aux fautes et aux pannes. Cette vue peut être décrite à l'aide d'un diagramme de déploiement.

Vue des cas d'utilisation

Cette vue, décrite par un diagramme de cas d'utilisation, sert à motiver et justifier les différents choix architecturaux.

Paquetages UML

Les paquetages UML permettent de regrouper un ensemble d'éléments de modélisation, en particulier des classes et des interfaces.

On peut utiliser les paquetages pour décrire la vue logique de l'architecture.

La figure IV. 2 montre la notation UML pour un paquetage P.

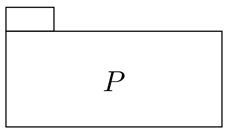


FIG. IV. 2 – Notation UML pour un paquetage P

b) Principes de conception architecturale

Pour réaliser une architecture logique, découpée en paquetages, on peut appliquer certains principes de conception architecturale.

Couplage

Le couplage est une mesure du degré selon lequel un paquetage est lié à d'autres paquetages. Des paquetages fortement couplés ne sont pas indiqués, car ils seront sensibles à beaucoup de modifications. De plus, ils sont plus difficiles à comprendre isolément, et difficiles à réutiliser.

Un principe de conception architecturale consiste donc à concevoir des paquetages faiblement couplés.

Cohésion

La cohésion mesure les liens, la cohérence entre les différents services proposés par le paquetage. Il est préférable de réaliser des paquetages ayant une forte cohésion, car ils sont plus faciles à comprendre et à réutiliser.

Variations

Au cours du développement d'un logiciel, certaines parties sont rapidement stables, et d'autres au contraire subissent de nombreuses modifications. Le principe de *protection des variations* consiste à éviter qu'un paquetage utilisé par de nombreux autres paquetages ne subisse trop de variations. En d'autres termes, un paquetage utilisé par de nombreux autres paquetages doit être rapidement stable.

Un exemple de « mauvais » paquetage

Le paquetage Java `java.util` est un exemple de « mauvais » paquetage. La documentation de ce paquetage indique :

> *Le paquetage java.util contient les collections, le modèle d'événements, des utilitaires de date et heure, d'internationalisation et des classes utilitaires diverses comme un « string tokenizer » [analyseur lexical], un générateur aléatoire et un tableau de bits.*

On remarque que la cohésion de ce paquetage n'est pas très forte : il propose un certain nombre d'utilitaires n'ayant aucun rapport entre eux.

Dans le reste de ce paragraphe, nous examinons deux exemples d'architecture logique : l'architecture en couches et l'architecture « modèle – vue – contrôleur ».

c) Architecture en couches

Le logiciel est organisé en couches. Une couche regroupe un ensemble de classes et propose un ensemble cohérent de services à travers une interface.

Les couches sont ordonnées : les couches de haut niveau peuvent accéder à des couches de plus bas niveau, mais pas l'inverse. Dans une architecture *étanche*, une couche de niveau n ne peut accéder qu'aux couches de niveau $n - 1$.

La figure IV. 3 montre l'architecture en couches d'un système comportant une interface graphique et communiquant avec une base de données.

Le rôle de chaque couche est le suivant :

- Couche A : présentation (interface graphique).
- Couche B : couche « application », réalisant la médiation entre l'interface graphique et les couches C et D.

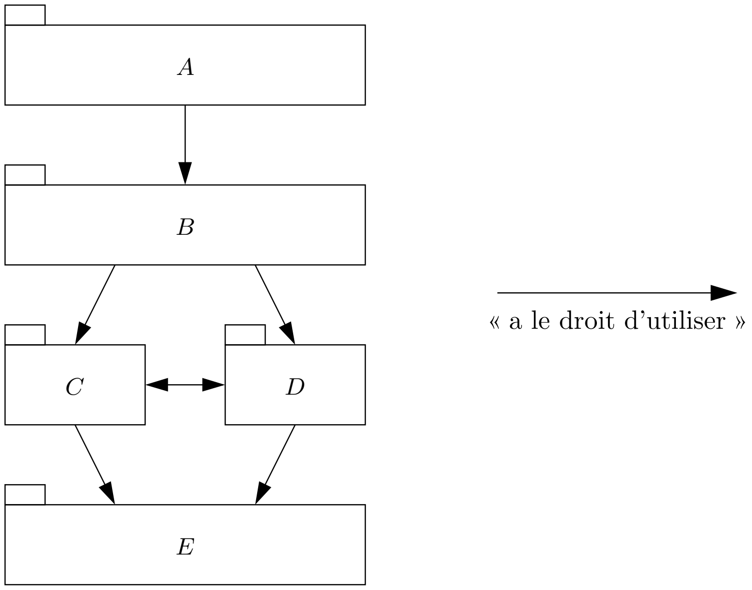


FIG. IV. 3 – Architecture en couches d'une application

- Couches C et D : deux couches « domaine », contenant les principales classes du domaine d'application, conçues afin d'être indépendantes de l'interface graphique et de l'infrastructure.
- Couche E : couche « infrastructure » comportant des services techniques de bas niveau (par exemple des tables SQL, des classes permettant de communiquer avec une base de données... etc.)

Avantages d'une architecture en couches

Les avantages d'une architecture en couche sont les suivants :

Maintenance
: Le système est plus facilement modifiable. Une modification d'une couche n'affecte pas les couches de niveau inférieur. Une modification d'une couche qui ne modifie pas l'interface publique n'affecte pas les couches de niveau supérieur ou égal.

Réutilisation
: Des éléments de chaque couche peuvent être réutilisés. Par exemple, les couches « domaine » peuvent être communes à plusieurs applications.

Portabilité
: On confine les éléments qui dépendent du système d'exploitation aux basses couches. On réécrit les couches basses pour chaque système d'exploitation, et les couches hautes sont portables.

Inconvénients d'une architecture en couches

Si on a un grand nombre de couches, et si en plus ces couches sont étanches, l'appel de fonctions de bas niveau est moins efficace, puisqu'il faut traverser toutes les couches pour parvenir à ces fonctions. Il faut donc trouver un compromis entre une bonne encapsulation et une bonne efficacité.

d) Architecture Modèle – Vue – Contrôleur

L'architecture « Modèle – Vue – Contrôleur », ou architecture MVC, est fréquemment utilisée pour les interfaces graphiques utilisateurs. Historiquement, cette architecture a été introduite dans le langage SmallTalk en 1980.

C'est un moyen de séparer le traitement des entrées, des sorties et des fonctionnalités principales du logiciel.

Le principe de cette architecture est de découper le logiciel (ou un morceau du logiciel) en trois parties :

- le « modèle » ;
- la « vue », qui correspond au traitement des sorties ;
- le « contrôleur », qui correspond au traitement des entrées.

Au départ, ce modèle a été développé pour correspondre au modèle classique :

$$\begin{array}{ccccc} \text{Entrées} & \rightarrow & \text{Traitement} & \rightarrow & \text{Sorties} \\ \text{Contrôleur} & \rightarrow & \text{Modèle} & \rightarrow & \text{Vue} \end{array}$$

Modèle

La partie « modèle » comporte les classes principales correspondant aux différentes fonctionnalités de l'applications : données et traitements. Cette partie, indépendante des parties « vue » et « contrôleur », effectue des actions en réponse aux demandes de l'utilisateur (par l'intermédiaire de la partie « contrôleur »), et informe la partie « vue » des changements d'états du modèle pour permettre sa mise à jour.

Vue

La partie « vue » comporte les classes relatives à l'interface graphique (ce que l'utilisateur voit). Cette partie est informée des changements d'états du modèle et est mise à jour lors de ces changements d'états.

Contrôleur

La partie « contrôleur » récupère les actions de l'utilisateur (clics sur les boutons et appuis sur les touches) et associe à ces événements des actions qui modifient le modèle.

Interactions

Les principales interactions entre ces trois parties sont les suivantes (cf. figure IV. 4) :

- le contrôleur, en fonction des événements qu'il reçoit de l'utilisateur, effectue des modifications du modèle ;
- ces modifications du modèle sont ensuite transmises à l'interface graphique.

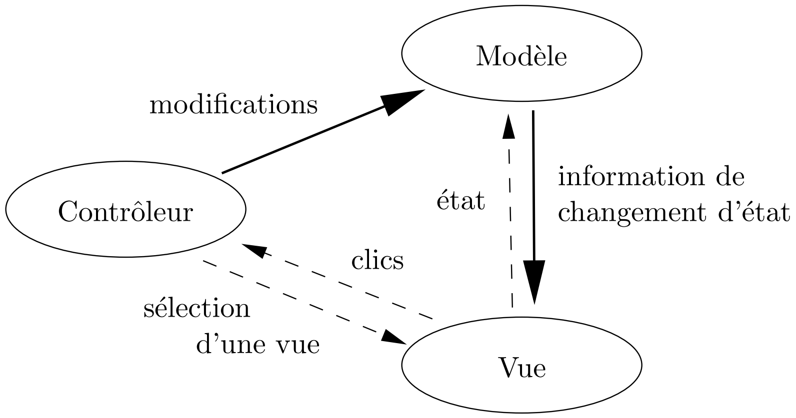


Fig. IV. 4 – Architecture MVC

Avantages d'une architecture Modèle – Vue – Contrôleur

Les avantages d'une architecture MVC sont les suivants :

Vues multiples
: On peut facilement gérer et afficher plusieurs vues du même modèle.

Portabilité
: Si on mélange le code de l'application avec le code de l'interface, le portage sur d'autres plates-formes est plus difficile. Le portage de l'interface sur d'autres plates-formes doit être possible sans modifier le code du noyau de l'application.

Évolution
: L'ajout de nouvelles fonctionnalités relatives à l'interface graphique, comme l'ajout de lignes dans un menu ou l'ajout de boutons, peut se faire plus facilement. Ces modifications sont également possibles lorsque le logiciel s'exécute : de nombreux logiciels permettent la personnalisation de leur interface à l'exécution.

2. Conception objet

Le but de cette étape est de réaliser la conception détaillée du logiciel. On se concentre dans ce paragraphe sur une conception « objet ». Un travail important de cette étape consiste à réaliser un diagramme de classes de conception. Il s'agit d'un diagramme de classes logicielles, inspiré du modèle objet du domaine. Par rapport au diagramme de classes du modèle objet, des classes, des attributs et des associations peuvent être ajoutés, modifiés, voire supprimés. Le diagramme de classes logicielles peut ensuite être traduit dans un langage de programmation objet.

a) Affectation des responsabilités

Une activité de cette étape consiste à affecter les responsabilités aux objets. Les responsabilités sont de deux types : connaissance et comportement.

Les connaissances peuvent être :

- la connaissance de données encapsulées ;

- la connaissance d'objets connexes ;
- la connaissance d'éléments qui peuvent être dérivés ou calculés.

Les comportements peuvent être :
- la réalisation d'une action, comme effectuer un calcul ou créer un objet ;
- le déclenchement d'une action d'un autre objet ;
- la coordination des activités d'autres objets.

Le but de ce travail est de permettre de préciser le contenu des classes, en détaillant les méthodes de chaque classe et en déterminant comment les objets interagissent pour réaliser certaines actions.

b) Principes de conception

Pour réaliser une bonne conception, on applique certains principes de conception.

Principe de l'expert

Une tâche est effectuée par un objet qui possède, ou a accès à, l'information nécessaire pour effectuer cette tâche.

Principe du créateur

On affecte à la classe B la responsabilité de créer des instances de la classe A s'il existe une relation entre A et B (typiquement, si B est une agrégation d'objets de A, ou si B contient des objets de A). L'idée est de trouver un créateur qui est connecté à l'objet créé.

Principe de faible couplage

Le couplage est une mesure du degré selon lequel un élément est relié à d'autres éléments. Une classe faiblement couplée s'appuie sur peu d'autres classes. Une classe fortement couplée a besoin de connaître un grand nombre d'autres classes. Les classes à couplage fort ne sont pas souhaitables car :
- elles sont plus difficiles à comprendre ;
- elles sont plus difficiles à réutiliser, car leur emploi demande la présence de nombreuses autres classes ;
- elles sont plus sensibles aux variations, en cas de modification d'une des classes auxquelles celle-ci est liée.

Le principe de faible couplage consiste minimiser les dépendances afin d'obtenir un faible couplage entre les classes.

Principe de forte cohésion

La cohésion est une mesure des liens entre les tâches effectuées par une classe. Une classe a une faible cohésion si elle effectue des tâches qui ont peu de liens entre elles, et dont certaines auraient du être affectées à d'autres classes. Il est préférable d'avoir des classes fortement cohésives car elles sont plus faciles à comprendre, à maintenir, à réutiliser. Elles sont moins affectées par une modification.

Principe du contrôleur

Le principe du contrôleur consiste à affecter la responsabilité du traitement des événements systèmes (c'est-à-dire des événements générés par un acteur externe) dans une ou plusieurs « classes de contrôle » (des « contrôleurs »). Par exemple, si on a une interface graphique, les événements reçus par l'interface sont délégués à un contrôleur. Cela permet de séparer la logique de l'application de l'interface : la logique de l'application ne doit pas être gérée par la couche interface.

c) Utilisation des diagrammes UML

On peut utiliser différents diagrammes UML pour effectuer la conception.

Diagramme des classes logicielles

Le diagrammes des classes logicielles est inspiré du diagramme de classes correspondant au modèle du domaine. Des classes peuvent être ajoutées, des liens ajoutés ou modifiés. On définit pour chaque classe les opérations qu'elle contient. On peut également donner des diagrammes d'objets montrant des configurations typiques pouvant être construites.

Diagrammes de séquence ou de collaboration

Les diagrammes de séquence ou de collaboration illustrent la façon dont les objets collaborent pour réaliser une fonctionnalité. Ces diagrammes permettent d'illustrer et de motiver les choix d'affectation de responsabilités aux objets.

Diagrammes d'états-transitions

Les diagrammes d'états-transitions permettent de spécifier le comportement des objets d'une classe. Ces diagrammes peuvent être considérés comme généralisant les diagrammes de séquence et de collaboration car ils doivent décrire tous les comportements, et non seulement quelques comportements possibles.

Diagrammes d'activités

On peut enfin utiliser des diagrammes d'activités pour décrire le comportement des opérations d'une classe.

d) Cohérence entre les différents diagrammes

Il est important de s'assurer de la cohérence entre les différents diagrammes élaborés au niveau de la conception. En effet, la mise en œuvre est *a priori* très proche de ces modèles. Cela suppose d'effectuer certaines vérifications.

Les vérifications contextuelles consistent à vérifier que les différents éléments de modélisation (objets, attributs, événements, opérations, associations, rôles. . . etc.) sont correctement déclarés et utilisés.

Exemple 1

Si on dispose de diagrammes d'objets, il faut vérifier que chaque diagramme d'objets est cohérent avec le diagramme de classes associé :

- correction des attributs de chaque objet ;
- correction des liens entre les objets ;
- respect des multiplicités.

Exemple 2

On considère une classe A, à laquelle est associé un automate comportant une transition étiquetée par « m_1 / $o.m_2$ » (cf. figure IV. 5).

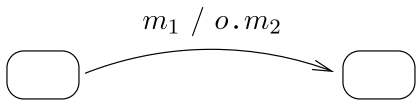


FIG. IV. 5 – Une transition de l'automate

Il faut vérifier que :

- m_1 est une méthode de la classe A ;
- o est un objet, d'une certaine classe B, accessible depuis un objet de la classe A (attribut de A de type B, ou lien entre les deux classes A et B, ayant pour nom de rôle o) ;
- m_2 est une méthode de la classe B.

Exemple 3

Dans un diagramme de séquence, on peut mettre en correspondance les séquences d'événements reçues par un objet avec une suite de transitions de l'automate associé à sa classe.

Chapitre V

Patrons de conception

Concevoir un logiciel est une activité difficile, écrire un programme facile à maintenir, extensible et réutilisable est encore plus difficile. Le développement orienté objet peut faciliter l'extension et la réutilisation, à condition que les développeurs soient expérimentés.

L'objectif des patrons de conception (« design patterns » en anglais) est de recueillir l'expérience et l'expertise des programmeurs, afin de la transmettre à d'autres programmeurs qui pourront la réutiliser.

Les patrons de conception ont été introduits en 1977 par Christopher Alexander, architecte en bâtiment. Alexander a proposé un catalogue de problèmes classiques en construction (bâtiments, villes), avec des solutions classiques. Les patrons de conception orientés objet reprennent cette idée de fournir un catalogue de solutions classiques à des problèmes de conception objet. Ils ont été introduits par Beck et Cunningham en 1987, et la thèse de Erich Gamma, soutenue en 1991 porte sur ce sujet.

Référence : *Design Patterns. Elements of Reusable Object Oriented Software.* Gamma, Helm, Johnson et Vlissides. Addison Wesley, 1995.

1. Notion de patron

Un patron de conception est la description d'un problème récurrent, dans un certain contexte, accompagné d'une description des différents éléments d'une solution à ce problème. Il s'agit de décrire une solution suffisamment générale et flexible à un problème qu'on rencontre souvent.

Un patron de conception comporte différents éléments :

- le nom, qui doit permettre de reconnaître le patron et indiquer son utilisation ;
- le problème, qui doit décrire l'objectif du patron ;
- le contexte, qui décrit les circonstances d'utilisation du patron ;
- la solution, qui décrit le schéma de conception résolvant le problème ;
- les conséquences, qui décrivent les avantages et inconvénients de la solution proposée.

On a différentes sortes de patrons de conception, qui peuvent être utilisés dans différentes étapes du cycle de vie.

a) Analyse et définition des besoins

Lors de cette étape, le principal problème est la communication entre les utilisateurs et les informaticiens d'une part et l'évaluation de la complexité d'autre part. Les patrons d'analyse servent à aider à résoudre ce genre de problèmes.

b) Analyse et conception

Lors de cette étape, les problèmes sont la définition d'une architecture adéquate, la résolution de sous-problèmes et la communication entre les développeurs. Les patrons architecturaux et les patrons de conception permettent d'aider à résoudre ce genre de problèmes.

Les patrons de conception peuvent être classifiés en différentes sortes :

1. les patrons « de création », qui concernent la création d'objets ;
2. les patrons « structurels », qui concernent la structure des objets et les relations entre ces objets ;
3. les patrons « comportementaux », qui sont relatifs au comportement des objets.

c) Mise en œuvre

Lors de cette étape, l'objectif est de produire un code correct et facile à maintenir. On utilise des patrons de programmation (ou *idiomes*), qui sont des solutions spécifiques à un langage de programmation.

Exemples d'idiomes :

- implémentation, en C, de tableaux de taille variable, réalloués dynamiquement si la taille dépasse les bornes ;
- implémentation de l'héritage multiple en Java.

2. Étude de quelques patrons de conception

Dans ce paragraphe, nous étudions en détail quelques patrons de conception.

a) Singleton

Le Singleton est un patron de conception de création.

But

L'objectif de ce patron est d'assurer qu'une classe a une instance unique, et d'en donner un accès « global ».

Motivation

Ce patron peut servir dans beaucoup de circonstances, en particulier lorsque la classe correspond à un objet unique dans le monde réel.

Exemples

- système de fichiers ;
- gestionnaire de fenêtres ;
- système de comptabilité pour une entreprise ;
- ... etc.

Code Java

```
class Singleton {

    /* Attribut privé contenant l'instance unique de Singleton. */
    final private static Singleton instanceUnique = new Singleton() ;

    /* Méthode qui renvoie l'instance unique de Singleton. */
    static Singleton instance() {
        return instanceUnique ;
    }

    /* On déclare le constructeur privé afin d'interdire
        l'instanciation de cette classe depuis une autre classe. */
    private Singleton() { }
}
```

b) Fabrique Abstraite

La Fabrique Abstraite est un patron de création.

But

L'objectif de ce patron est de créer une famille d'objets qui dépendent les uns des autres, sans que l'utilisateur de cette famille d'objets ne connaisse la classe exacte de chaque objet.

Principe

L'utilisateur a accès uniquement à différentes classes abstraites (une par produit), par exemple ProduitAbstraitX et ProduitAbstraitY et à une classe qui permet de créer des instances de ces produits : Fabrique1 ou Fabrique2 (cf. figure V. 1). Ces instances sont créées à l'aide des méthodes créerX et créerY.

Si l'utilisateur utilise la classe Fabrique1 pour créer les objets, alors il obtient des instances de ProduitX1 et ProduitY1. S'il utilise la classe Fabrique2, alors il obtient des instances de ProduitX2 et ProduitY2.

Avantages de ce patron

Les avantages de ce patron sont les suivants :

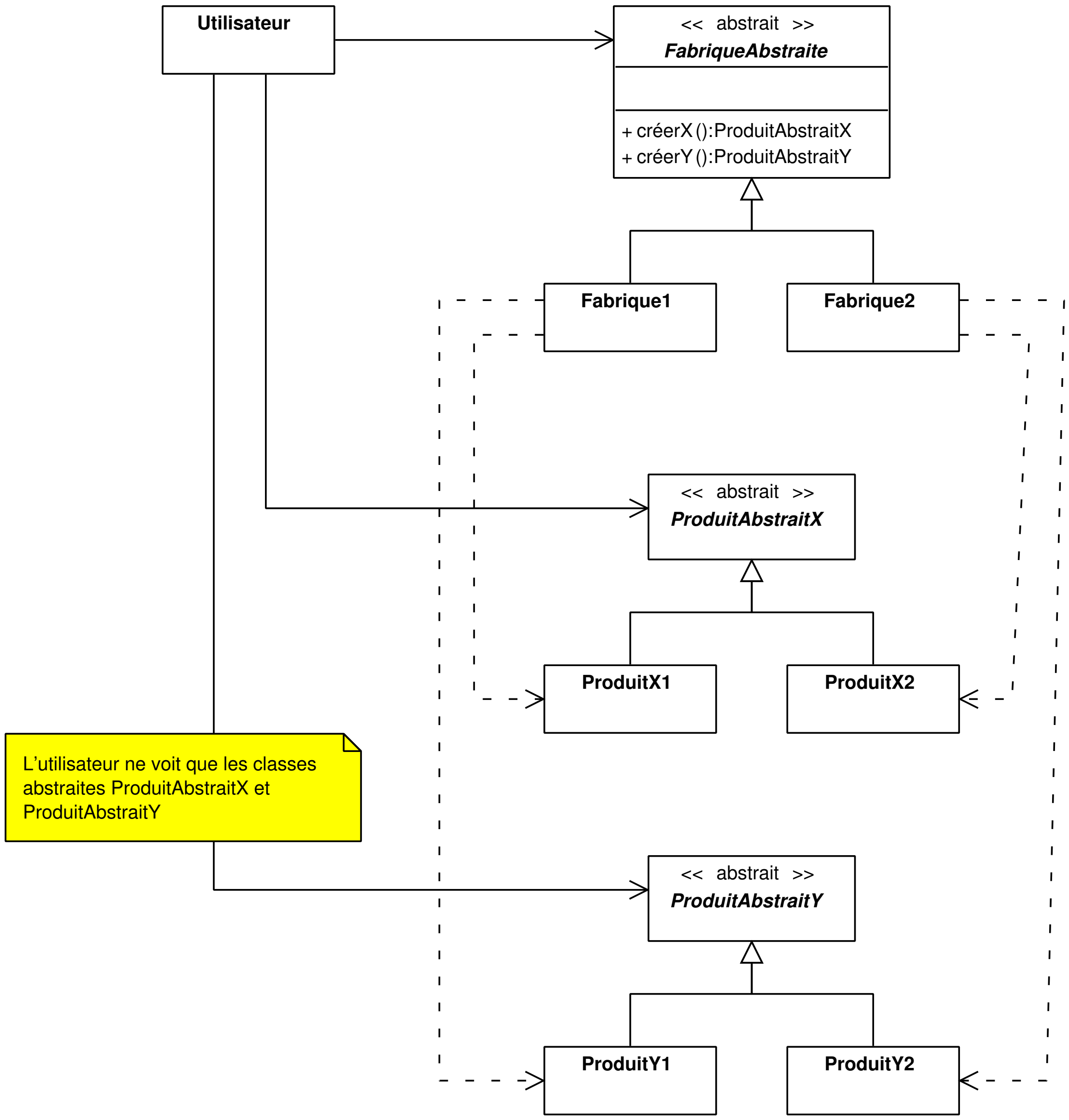


Fig. V. 1 – Diagramme de classes du patron Fabrique Abstraite

- Le patron facilite l'utilisation cohérente des différents produits : si le client utilise toujours la même classe, par exemple Fabrique1, pour créer des objets, il est sûr de toujours utiliser des produits de la même famille.
- L'utilisateur peut facilement changer de famille de produits, puisqu'il suffit de changer l'instantiation de la fabrique.
- On peut assez facilement ajouter une nouvelle famille de produits, en ajoutant une nouvelle sous-classe pour chaque produit abstrait.

Inconvénient de ce patron

Il peut être difficile d'ajouter de nouveaux produits puisqu'il faut modifier toutes les classes qui dérivent de FabriqueAbstraite.

Remarque

On peut implémenter les sous-classes de FabriqueAbstraite en utilisant le patron Singleton.

c) Objet composite

Objet composite est un patron de conception structurel.

But

L'objectif de ce patron est de créer des objets simples ou composés, avec des méthodes de traitement uniformes, pour lesquelles le client n'a pas à savoir s'il applique un certain traitement à un objet simple ou composé.

Solution

On utilise une classe abstraite Composite contenant une (ou plusieurs) méthodes abstraites de traitement (cf. figure V. 2).

La figure V. 3 montre un exemple d'application du patron objet composite pour modéliser des figures géométriques.

d) Adaptateur

L'adaptateur est un patron structurel. On utilise un adaptateur lorsque, pour implémenter une interface « cible », on souhaite réutiliser une classe « source » qui ne respecte pas exactement cette interface. La solution qui consiste à modifier la classe source n'est pas satisfaisante lorsque cette classe est réutilisée à d'autres endroits.

Solution par héritage

Une solution consiste à réaliser une sous-classe Adaptateur de Source qui implémente l'interface Cible (cf. figure V. 4).

Cette solution a les inconvénients suivants :

- on ne peut pas adapter des sous-classes de Source ;
- on peut redéfinir des méthodes de Source.

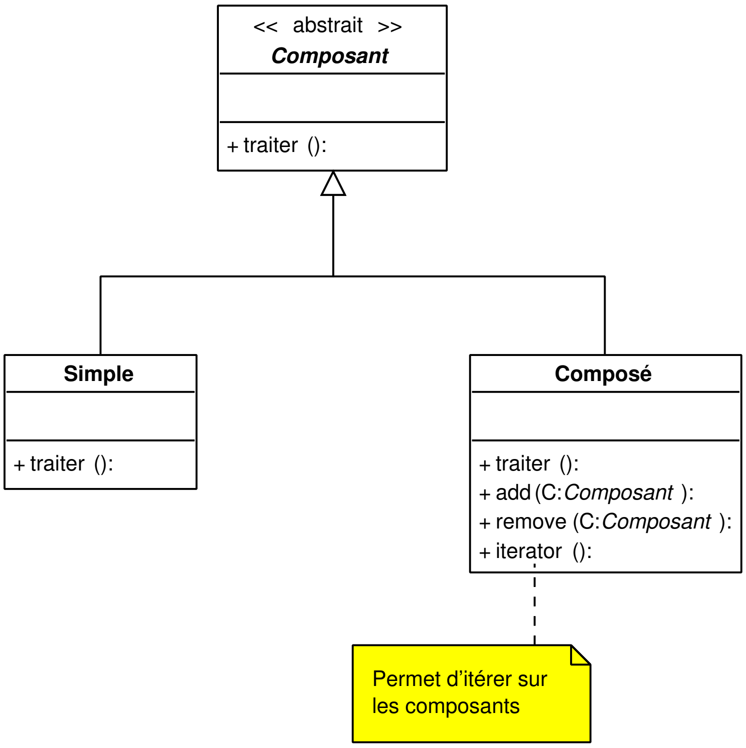


FIG. V. 2 – Diagramme de classes du patron Composite

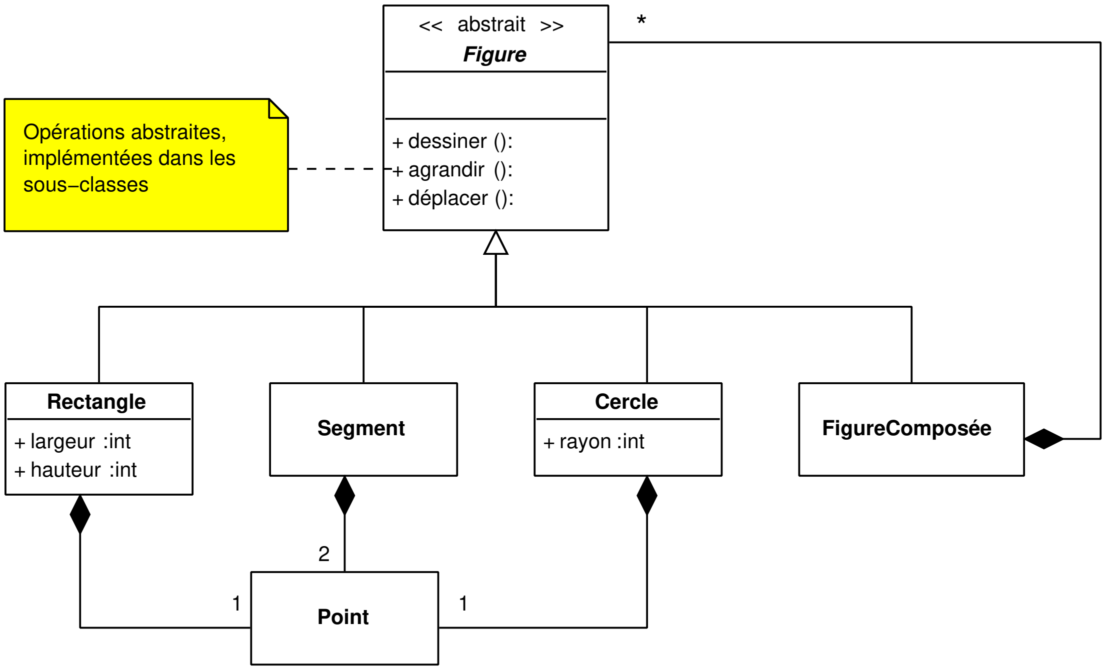


FIG. V. 3 – Figures géométriques

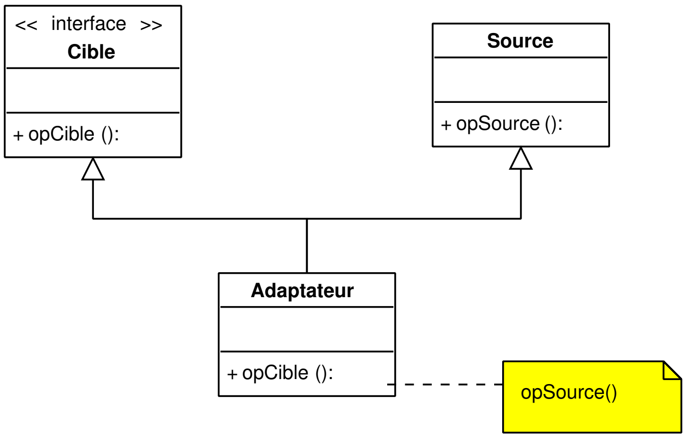


Fig. V. 4 – Patron Adaptateur, par héritage

Solution par délégation

Le principe de la délégation consiste à créer un lien entre l'adaptateur et la source et à ce que l'adaptateur délègue le travail à effectuer à la classe source. La source joue le rôle du délégué.

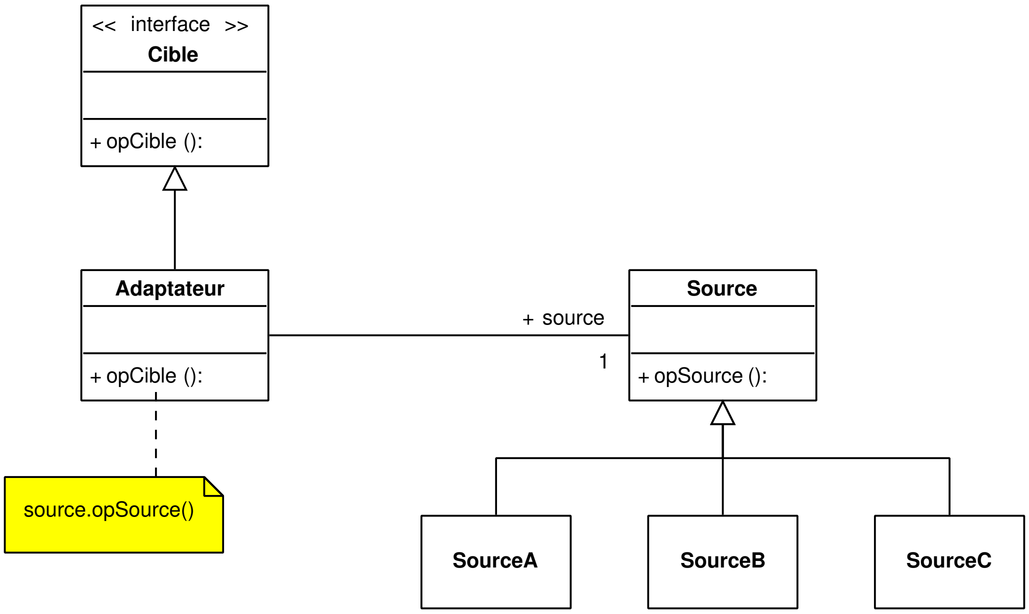


Fig. V. 5 – Patron Adaptateur, par délégation

Remarque : cette solution ressemble à l'implémentation de l'héritage multiple.

Cette solution a les avantages suivants :

- on peut adapter des sous-classes de Source : SourceA, SourceB, SourceC ;
- cette solution ne permet pas la redéfinition des méthodes de Source.

e) Patrons Stratégie, Commande et État

Les patrons de conception Stratégie, Commande et État sont des patrons comportementaux assez similaires qui consistent à :

- associer à certains traitements (ou méthodes) des objets ;
- définir une hiérarchie de classes pour effectuer ces traitements de façon uniforme ;
- découpler les données et les traitements : la hiérarchie de classes des traitements est indépendante de celle des objets qui vont utiliser ces opérations.

Patron Stratégie

L'objectif de ce patron est de définir une famille d'algorithmes encapsulés dans des objets, afin que ces algorithmes soient interchangeables dynamiquement.

Motivation

Pour résoudre un problème, il existe souvent plusieurs algorithmes ; dans certains cas il peut être utile de choisir à l'exécution quel algorithme utiliser, par exemple selon des critères de temps de calcul ou de place mémoire.

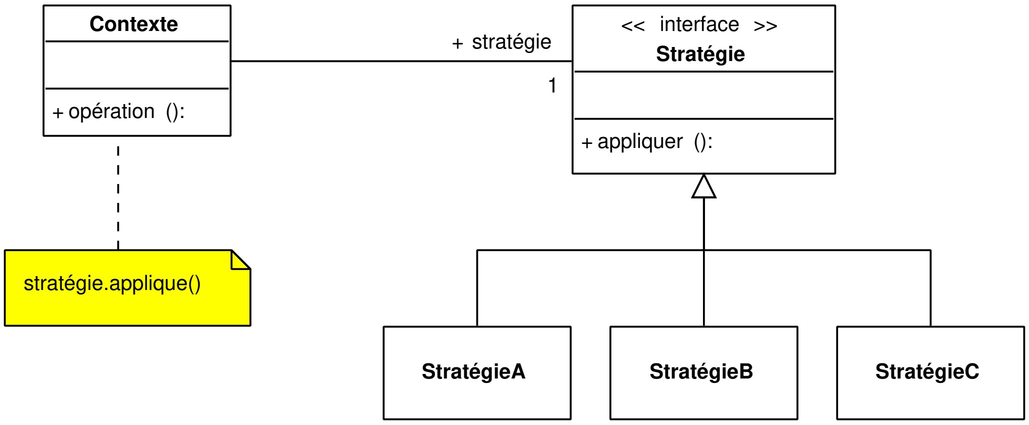


Fig. V. 6 – Patron Stratégie

Les avantages de ce patron sont les suivants :

- en découplant les algorithmes des données, la classe Contexte peut avoir des sous-classes indépendantes des stratégies ;
- on peut changer de stratégie dynamiquement.

Ce patron a les inconvénients suivants :

- on a un surcoût en place mémoire, car il faut créer des objets Stratégie ;
- on a un surcoût en temps d'exécution à cause de l'indirection `stratégie.applique()`.

Patron Commande

L'objectif de ce patron est d'encapsuler des « commandes » dans des objets.

Motivation

Dans des applications qui comportent une interface graphique, on peut associer des commandes à des boutons, ou à des lignes dans des menus ; revenir en arrière d'une ou plusieurs commandes (« annuler ») ou repartir en avant (« rétablir »). Pour cela, on doit associer à des actions des objets que l'on peut stocker, passer en paramètre... etc.

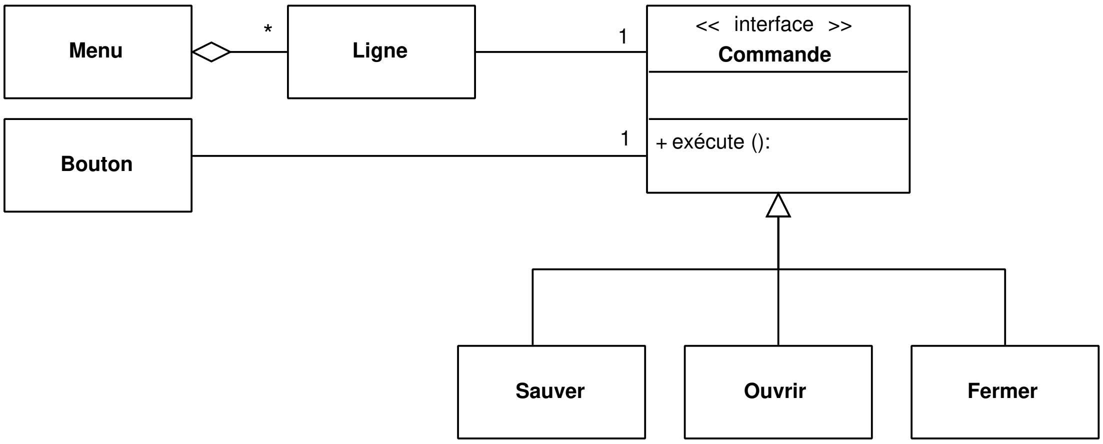


Fig. V. 7 – Patron Commande

Les avantages de ce patron sont les suivants :

- on peut modifier une association Ligne — Commande à l'exécution, afin de paramétrer le logiciel à l'exécution ;
- on peut effectuer la même action par différents moyens (en passant par un menu, ou en appuyant sur un bouton... etc.) ;
- on peut définir des « macro commandes », composées de séquences de commandes (cf. figure V. 8) ;
- on peut revenir en arrière (annuler, rétablir). Cela nécessite de stocker l'historique des commandes, et un état interne associé à chaque commande effectuée (cf. figure V. 9).

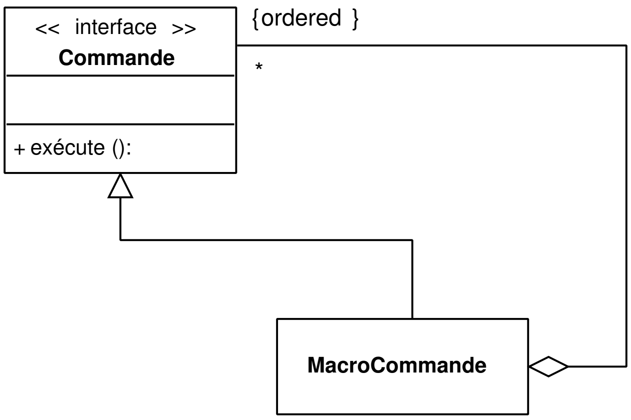


Fig. V. 8 – Macro commandes

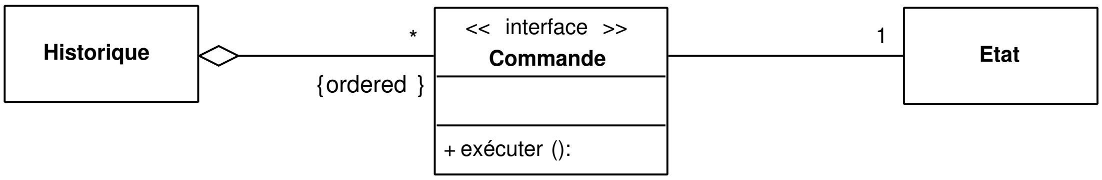


FIG. V. 9 – Historique de commandes

Patron État

Le patron État permet de réaliser des objets dont le comportement change lorsque leur état interne est modifié.

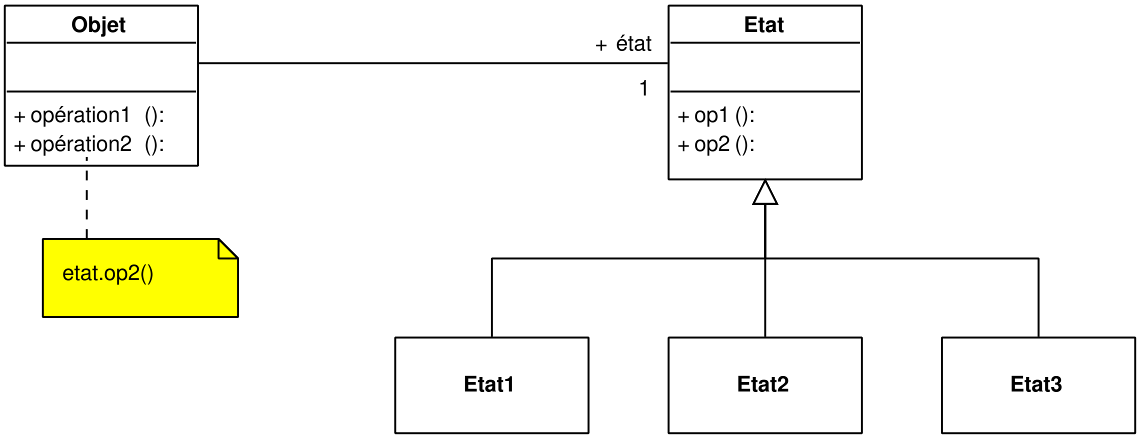


FIG. V. 10 – Patron État

Différences entre les patrons Stratégie, Commande et État

Les trois patrons ont une structure assez similaire, les différences entre ces patrons sont liées à ce que l'on cherche à implémenter.

- Pour le patron Stratégie, le but est d'implémenter une même opération de plusieurs façons différentes. On peut considérer qu'on a une seule spécification de l'opération.
- Pour le patron Commande, on peut vouloir associer une même commande à différents objets, et définir des séquences de commandes (macros) ou des historiques de commandes.
- Pour le patron État, plusieurs opérations peuvent être associées à un état.

f) Patron Observateur

L'Observateur est un patron comportemental, dont le but est de définir une dépendance entre un objet (appelé sujet) et un ensemble d'objets (appelés observateurs) de sorte que lorsque le sujet change d'état, tous les observateurs qui en dépendent soient informés et mis à jour automatiquement.

Par exemple, une application peut comporter plusieurs représentations graphiques pour un même objet (cf. figure V. 11).

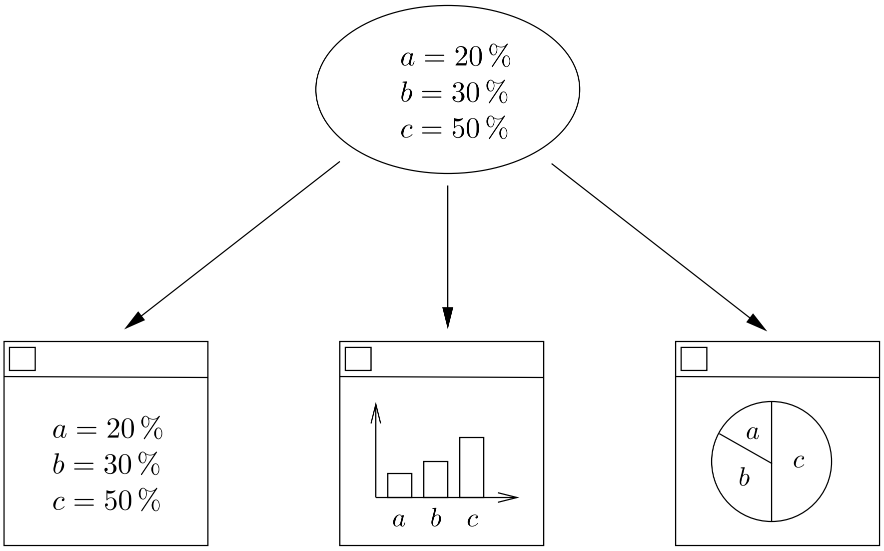


Fig. V. 11 – Plusieurs représentation graphiques pour un même objet

Le sujet est en relation avec un ensemble d'observateurs. Lorsque le sujet change d'état, tous les observateurs sont informés.

On définit une classe abstraite Sujet, dont héritent toutes les classes pouvant servir de sujet, autrement dit, pouvant être mises en relation avec un ensemble d'observateurs.

On définit une interface Observateur qui contient une méthode miseAJour.

Un sujet est en relation avec un ensemble d'observateurs. On dispose de méthodes pour attacher ou détacher un observateur à un sujet, autrement dit, ajouter ou enlever un observateur de l'ensemble des observateurs de ce sujet.

La méthode informe appelle miseAJour sur chaque observateur.

L'interface Observateur est implémentée par un certain nombre d'observateurs concrets, qui contiennent le code correspondant à l'affichage et la mise à jour de cet affichage.

Remarques sur la navigabilité des relations :

- Sujet doit connaître Observateur pour l'informer (mais Observateur n'a pas besoin de pouvoir accéder à Sujet) ;
- ObservateurConcret doit connaître SujetConcret pour effectuer la mise à jour de l'affichage : il doit en effet connaître l'état du sujet (mais SujetConcret n'a pas besoin de connaître ObservateurConcret car Sujet connaît Observateur).

Rapport avec l'architecture Modèle – Vue – Contrôleur : le patron Observateur est un moyen d'implémenter (en partie) une architecture MVC. Les classes qui héritent de Sujet sont des classes du modèle, et les classes qui implémentent Observateur sont des classes de la vue.

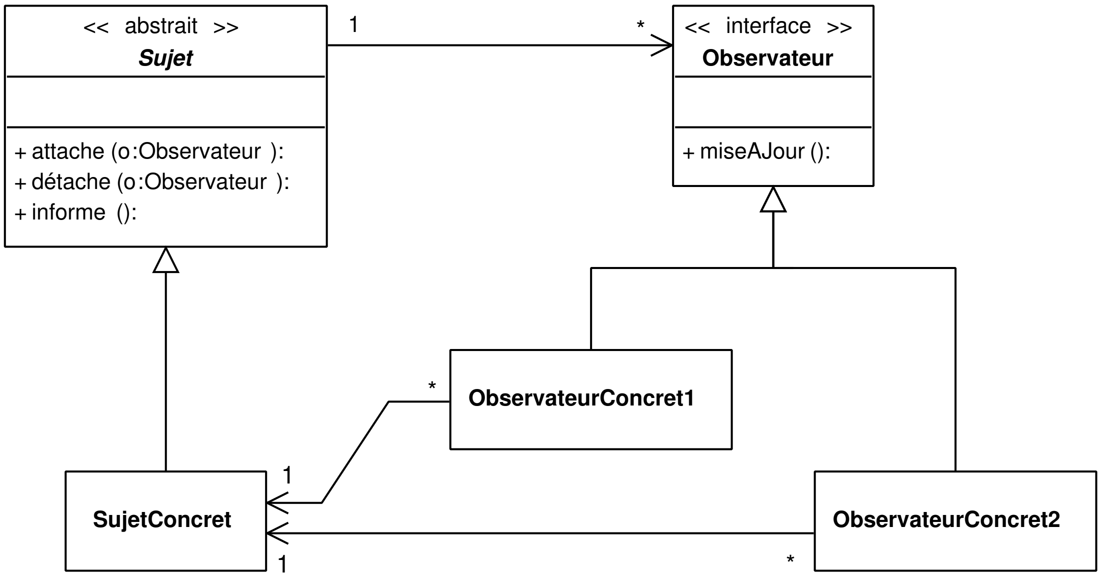


FIG. V. 12 – Patron Observateur

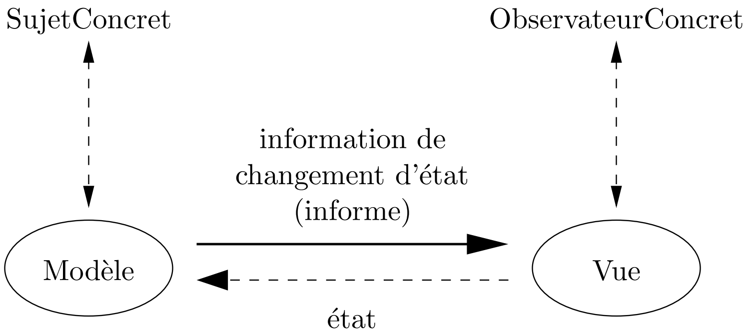


FIG. V. 13 – Implémentation d'une architecture MVC à l'aide du patron Observateur

g) Interprète

L'Interprète est un patron de conception comportemental. Il a pour but, étant donné un langage, de définir une représentation pour sa grammaire abstraite (autrement dit, une structure d'arbre abstrait), ainsi qu'un interprète utilisant cette représentation.

On suppose qu'on a une grammaire abstraite, avec des règles de la forme suivante :

$$\begin{array}{rcl} A & \rightarrow & \text{Noeud}_1(A_{1,1}, \ldots, A_{1,n_1}) \\ & & \mid \ \text{Noeud}_2(A_{2,1}, \ldots, A_{2,n_2}) \\ & & \mid \ \ldots \\ & & \mid \ \text{Noeud}_k(A_{k,1}, \ldots, A_{k,n_k}) \end{array}$$

On associe à chaque non terminal une classe abstraite, et à chaque nœud une classe concrète qui hérite de cette classe abstraite.

Pour chaque classe abstraite A, on écrit une méthode abstraite `interp(c:Contexte)`.

La classe `Contexte` correspond à des paramètres nécessaires à l'interprétation, qui peut varier d'une classe abstraite à l'autre.

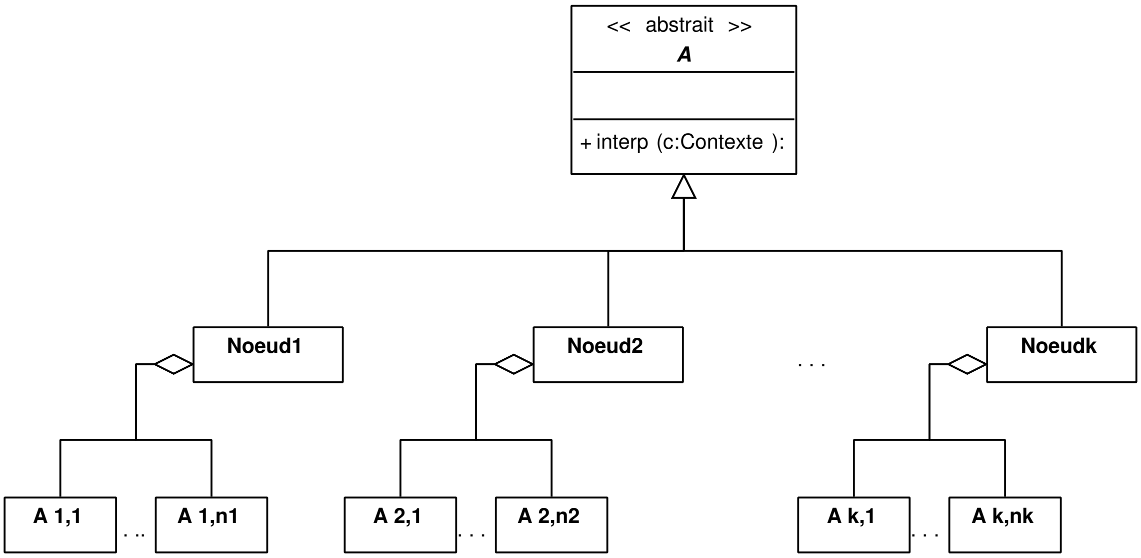


FIG. V. 14 – Patron Interprète

On peut appliquer ce patron de conception chaque fois qu'on doit définir des opérations ou des calculs dirigés par la syntaxe, en particulier en compilation :

- évaluation (pour écrire un interprète, au sens propre) ;
- vérification de type ;
- génération de code ;
- ... etc.

On peut par exemple appliquer le patron Interprète pour écrire un compilateur pour le langage Mini-Pascal.

Le patron Interprète a les avantages suivants :

- on peut facilement modifier et étendre la grammaire. Par exemple, on peut facilement ajouter de nouvelles expressions en définissant de nouvelles classes.
- l'implémentation de la grammaire est simple, et peut être réalisée automatiquement à l'aide d'outils de génération.

Le patron Interprète a certains inconvénients :

- lorsque la grammaire est complexe, on a une multiplication des classes.
- il devient alors délicat d'ajouter de nouvelles opérations, car celles-ci doivent être ajoutées dans toutes les classes de la hiérarchie.

h) Visiteur

Le Visiteur est un patron de conception comportemental.

L'objectif du patron Visiteur est de représenter une opération définie en fonction de la structure d'un objet. Le Visiteur permet alors de définir de nouvelles opérations sans modifier les classes qui définissent la structure des objets auxquelles elles s'appliquent.

On cherche à ne pas répartir les différents cas dans les classes qui définissent la structure des objets, mais au contraire à regrouper tout ce qui concerne une opération dans une seule classe.

Ne pas modifier les classes définissant la structure des objets présente les intérêts suivants :

- cela favorise la réutilisation (on récupère facilement, par exemple, une structure d'arbre abstrait pour un langage) ;
- cela permet le développement parallèle de code par plusieurs équipes. Par exemple, une équipe peut implémenter la vérification de types et une autre la génération de code sans travailler sur les mêmes classes ;
- cela permet de partager la structure des objets par plusieurs applications, qui peuvent alors coopérer plus facilement.

Principe

On considère une hiérarchie de classes définissant la structure d'objets. On suppose qu'on a une classe abstraite Elément, racine de cette hiérarchie (cf. figure V. 15).

On définit une interface Visiteur, qui contient une méthode *void* visite(ElémentX e) pour chaque sous-classe concrète ElémentX de Elément.

```
/**
 * Interface Visiteur, qui permet de définir des opérations qui s'appliquent
 * sur des éléments.
 */
interface Visiteur {
```

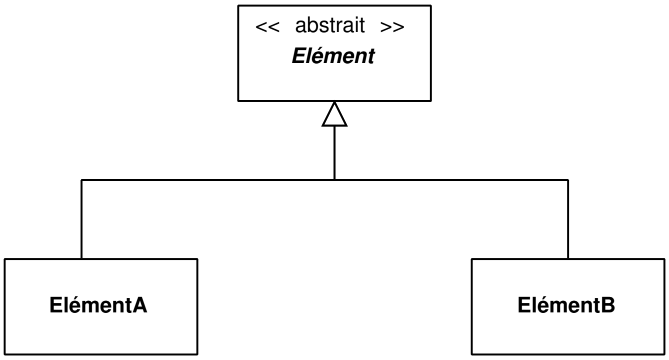


FIG. V. 15 – Hiérarchie Elément pour le patron Visiteur

```
    void visite(ElémentA e) ;
    void visite(ElémentB e) ;
}
```

Dans chaque classe de la hiérarchie de classes définissant la structure des objets, on définit une méthode

```
    void applique(Visiteur v) ;
```

Cette méthode est abstraite dans les classes abstraites. Dans les classes concrètes, elle appelle la méthode visite sur le visiteur v, et passe *this* en paramètre.

```
/**
 * Hiérarchie d'éléments.
 */

abstract class Elément {
    abstract void applique(Visiteur v) ;
}

class ElémentA extends Elément {
    void applique(Visiteur v) {
        v.visite(this) ; // Appel de la méthode visite(ElémentA)
    }
}

class ElémentB extends Elément {
    void applique(Visiteur v) {
        v.visite(this) ; // Appel de la méthode visite(ElémentB)
    }
}
```

Remarque

La définition de la méthode applique (c'est-à-dire v.visite(*this*)) est différente dans chaque classe concrète car la méthode visite appelée a une signature différente pour chaque classe : visite(ElémentA) ou visite(ElémentB). On ne peut donc pas définir la méthode applique une seule fois au niveau de Elément.

On appelle « visiteur » un objet d'une classe qui implémente l'interface Visiteur. Un visiteur définit une opération s'appliquant sur tout objet de type Elément. La méthode applique(Visiteur v) représente l'application de l'opération définie par le visiteur à l'objet.

On peut alors définir une opération qui s'applique sur Elément à l'aide d'une classe qui implémente Visiteur.

```
/**
 * Opération qui s'applique à un élément.
 */
class Opération implements Visiteur {

   public void visite(ElémentA e) {
      // Ce que l'opération doit faire sur un objet de type ElémentA
      . . .
   }

   public void visite(ElémentB e) {
      // Ce que l'opération doit faire sur un objet de type ElémentB
      . . .
   }

   // Les attributs de cette classe peuvent servir de paramètres
   // d'entrée ou de résultat pour l'opération.
}
```

Un appel de l'opération se fait de la façon suivante :

```
   Elément e = new ElémentA() ; // Un élément
   Visiteur v = new Opération() ; // Une opération
   e.applique(v) ; // L'opération v est appliquée sur l'élément e
```

L'appel « e.applique(v) » appelle « v.visite(e) », qui effectue le code correspondant à ElémentA, car visite a pour signature visite(ElémentA e).

Si on souhaite programmer une opération s'appliquant sur un objet de type Elément sans utiliser le patron Visiteur, on a deux possibilités :

1. Répartir tout le code de l'opération dans les différentes classes de la hiérarchie. Cela correspond à une application du patron Interprète.

2. Écrire une classe Opération qui teste les différents cas à l'aide de *instanceof* (il s'agit de programmation « classique », « non objet »).

```
class Opération {
    static void op(Elément e) {
        if (e instanceof ElémentA) {
            ElémentA aA = (ElémentA) e ;
            // Ce que l'opération doit faire sur ElémentA
            ...
        } else if (e instanceof ElémentB) {
            ElémentB eB = (ElémentB) e ;
            // Ce que l'opération doit faire sur ElémentB
            ...
        }
    }
}
```

Cette opération s'utilise de la façon suivante :

```
Elément e = new ElémentA() ; // Un élément
Opération.op(e) ; // Appel de l'opération op sur l'élément e
```

Le patron de conception Visiteur a les avantages suivants sur la programmation « classique » :

- le patron oblige à traiter tous les cas, et c'est vérifié à la compilation ;
- il s'agit d'une programmation « purement objet » ;
- le patron évite l'utilisation de *instanceof*, de devoir déclarer une variable initialisée à l'aide d'une conversion, et évite également ainsi le risque d'erreur de conversion à l'exécution ;
- le patron évite d'effectuer n tests pour trouver le code à exécuter (en particulier lorsque Elément a de nombreuses sous-classes) ;
- le patron permet l'ajout de nouvelles opérations sans modifier la hiérarchie de classes, et l'ajout de nouvelles classes (en modifiant l'interface).

Le patron présente les inconvénients suivants :

- ce patron est lourd à mettre en œuvre : il faut prévoir une méthode applique par classe.
- le traitement de méthodes comportant des paramètres et un résultat est également lourd ;
- on a un surcoût à chaque indirection applique $\rightarrow$ visite ;
- le code est peu lisible lorsqu'on ne connaît pas le patron.

Chapitre VI

Mise en œuvre

Dans ce chapitre, on s'intéresse à la mise en œuvre d'une conception basée sur un ensemble de diagrammes UML. On suppose que l'on cherche à produire du code Java. On se concentre sur deux sortes de diagrammes : les diagrammes de classes et les diagrammes d'états-transitions.

1. Classes et interfaces

Les notions de classe, d'interface, d'attribut et d'opération sont très similaires en UML et en Java.

classe UML	classe Java
classe abstraite UML	classe abstraite Java
interface UML	interface Java
attribut UMl	attribut Java
opération UML	méthode Java

On a en UML la notion de *classe utilitaire*, qui représente une classe qu'on ne pas instancier, et qui regroupe des attributs et des opérations de classe.

Par exemple, la classe utilitaire Math définit une constante pi et des fonctions mathématiques.

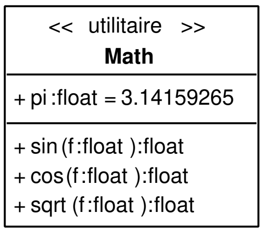


Fig. VI. 1 – Classe utilitaire Math

Pour coder une classe utilitaire en Java,

- on déclare le constructeur de classe comme privé, afin d'interdire l'instanciation de la classe par d'autres classes ;
- on utilise des constituants *static* car une classe utilitaire contient des attributs et des méthodes *de classe*.

```
class Math {
    // Constructeur privé
    private Math() { }

    // attribut final car pi est une constante
    static final float pi = 3.1415926535 ;

    static float sin(float x) { ... }
    static float cos(float x) { ... }
    static float sqrt(float x) { ... }
}
```

2. Relations binaires

Pour traduire une relation binaire en Java, il existe plusieurs choix possibles. Pour effectuer ce choix, il est nécessaire d'analyser ce dont on a besoin en termes de

- navigabilité,
- mutiplicités,
- évolution de la relation.

a) Navigabilité

Considérons une relation entre deux classes A et B.

Navigabilité de A vers B
: Si cette relation est navigable de A vers B, on doit pouvoir accéder, à partir d'une instance de A, aux instances de B qui sont en relation avec celle-ci.

Navigabilité de B vers A
: Si la relation est navigable de B vers A, on doit pouvoir accéder, à partir d'une instance de B, aux instances de A qui sont en relation avec celle-ci.

Navigabilité dans les deux sens
: Si la relation est navigable dans les deux sens, on doit à la fois pouvoir accéder aux instance de A à partir d'une instance de B et aux instances de B à partir d'une instance de A (cf. figure VI. 2).

b) Multiplicités

On peut distinguer plusieurs sortes de multiplicités : une multiplicité « simple », qui correspond à zéro ou une instance ; une multiplicité de cardinalité fixée, qui correspond à n instances, où n est un entier fixé ; et une multiplicité à cardinalité variable, qui correspond à un certain nombre, non fixé, d'instances.

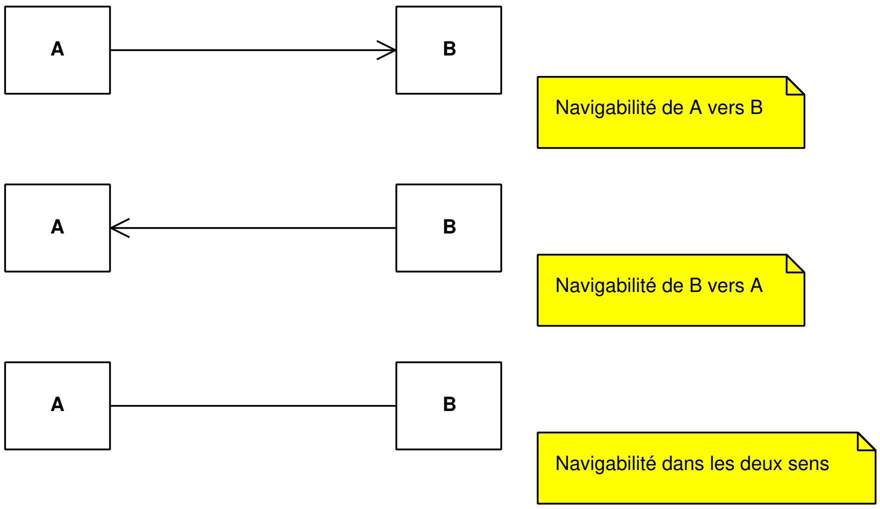


FIG. VI. 2 – Navigabilités possibles pour une relation

Multiplicité simple

On a une multiplicité simple lorsque au plus une instance de B est associée à une instance A. Cela peut correspondre à la multiplicité « `1` », ou « `0..1` ».

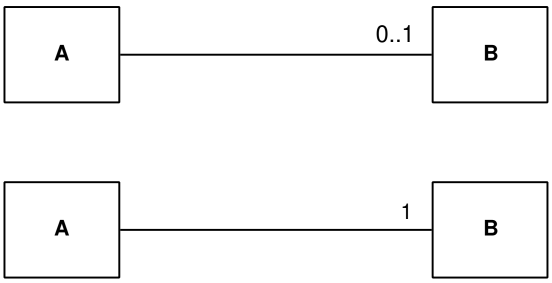


FIG. VI. 3 – Multiplicités simples

Multiplicité de cardinalité fixée

L'ensemble des instances de B associées à une instance de A a pour cardinalité n, où n est un entier fixé.

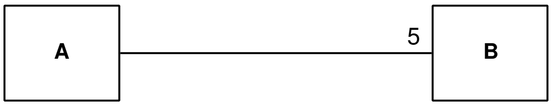


FIG. VI. 4 – Multiplicité de cardinalité fixée

Multiplicité de cardinalité variable

L'ensemble des instances de B associées à une instance de A a une cardinalité variable.

c) Évolution des liens au cours de l'exécution du programme

Considérons une relation binaire entre deux classes A et B. Au cours de l'exécution du programme, des instances des classes A et B peuvent être créées ; des liens entre ces instances

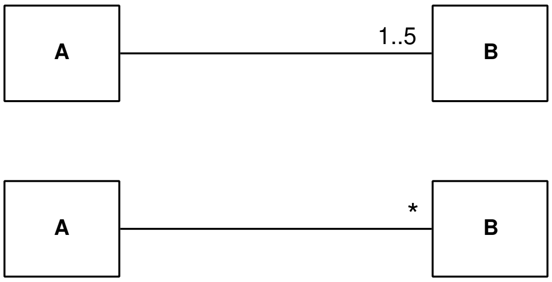


Fig. VI. 5 – Multiplicités de cardinalité variable

peuvent être créés. Ces liens peuvent

1. soit être créés une seule fois, et ne plus être modifiés par la suite ;
2. soit évoluer au cours de l'exécution du programme, par exemple être créés, supprimés, déplacés.

Les fonctionnalités offertes par les classes diffèrent dans les deux cas.

1. Dans le premier cas, les liens peuvent être créés au moment de la création des objets. C'est le constructeur qui peut créer les liens.
2. Dans le deuxième cas, les classes doivent fournir des méthodes qui permettent l'ajout, la suppression ou la modification d'un lien.

Dans la suite, nous développons quatre exemples de codage d'une relation binaire entre deux classes A et B.

Exemple 1

On considère une relation avec la multiplicité `0..1` de chaque coté.

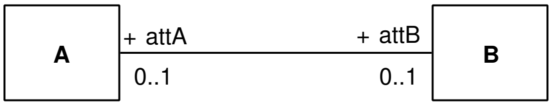


Fig. VI. 6 – Relation avec multiplicités `0..1`

Une instance de A peut être :

- soit isolée,
- soit liée à une instance de B.

De même, une instance de B peut être soit isolée, soit liée à une instance de A.

Le principe pour coder la relation est le suivant :

- pour coder un objet $b : B$ associé à un objet $a : A$, on utilise un attribut attB de type B dans la classe A. Cet attribut a la valeur *null* pour les instances isolées de A.
- pour coder un objet $a : A$ associé à un objet $b : B$, on utilise un attribut attA de type A dans la classe B. Cet attribut a la valeur *null* pour les instances isolées de B.

```
class A {
    B attB ; // Instance de B liée à cet objet
}

class B {
    A attA ; // Instance de A liée à cet objet
}
```

Si on ne prend pas de précaution, on peut obtenir des diagrammes d'objets incohérents, par exemple en écrivant le code suivant :

```
A a1 = new A() ;
A a2 = new A() ;
B b1 = new B() ;
a1.attB = b1 ;
b1.attA = a2 ;
a2.attB = null ;
```

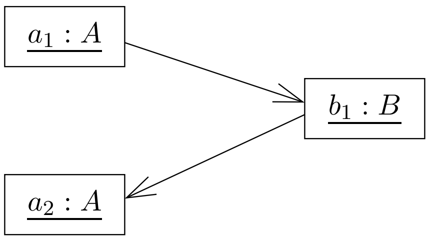


Fig. VI. 7 – Liens incohérents

Les contraintes à respecter pour que les liens soient cohérents sont les suivantes :

- $\forall a : A,\ a.attB \neq null \Rightarrow a.attB.attA = a$;
- $\forall b : B,\ b.attA \neq null \Rightarrow b.attA.attB = b$.

Afin d'interdire toute modification incontrôlée des attributs *attA* et *attB*, on les déclare comme privés et on définit des méthodes d'accès getA et getB, ainsi que des méthodes de modification setA et setB.

On définit une exception OperationIncorrecte qui est levée en cas d'opération incorrecte.

```
/**
 * Exception levée lorsqu'on tente d'effectuer une opération incorrecte.
 */
public class OperationIncorrecte extends RuntimeException { }

/**
 * Classe B.
 */
public class B {
```

```
    // L'instance de A associée à cette instance.
    private A attA ;

    /**
     * Constructeur d'instance isolée.
     */
    public B() {
        attA = null ;
    }

    /**
     * Constructeur d'instance liée à une instance a de A.
     * Précondition : a est une instance non nulle et isolée.
     * Postcondition : a est liée à cette instance.
     */
    public B(A a) {
        if (a == null || a.getB() != null) {
            throw new OperationIncorrecte() ;
        } else {
            a.attache(this) ;
        }
    }

    /**
     * Retourne l'instance de A associée à cette instance.
     */
    public A getA() {
        return attA ;
    }

    /**
     * Affecte l'instance de A associée à cette instance à l'objet a.
     * (Méthode non publique)
     */
    void setA(A a) {
        attA = a ;
    }
}

/**
 * Classe A.
 */
public class A {

    // L'instance de B associée à cette instance.
    private B attB ;

    /**
```

```
 * Constructeur d'instance isolée.
 */
public A() {
    attB = null ;
}

/**
 * Constructeur d'instance liée à une instance b de B.
 * Précondition : b est une instance non nulle et isolée.
 * Postcondition : b est liée à cette instance.
 */
public A(B b) {
    if (b == null || b.getA() != null) {
        throw new OperationIncorrecte() ;
    } else {
        attB = b ;
        b.setA(this) ;
    }
}

/**
 * Retourne l'instance de B associée à cet objet.
 */
public B getB() {
    return attB ;
}

/**
 * Attache l'instance b à cet objet.
 * Lève l'exception OperationIncorrecte si cet objet est lié à une instance
 * de B ou si b est null, ou si b est lié à une instance de A.
 */
public void attache(B b) {
    if (attB != null) {
        // Cet objet est déjà lié à une instance de A.
        throw new OperationIncorrecte() ;
    } else if (b == null) {
        // b est null
        throw new OperationIncorrecte() ;
    } else if (b.getA() != null) {
        // b ets déjà lié à une instance de A.
        throw new OperationIncorrecte() ;
    } else {
        attB = b ;
        b.setA(this) ;
    }
}

/**
```

```
    * Détache l'instance de B liée à cette instance.
    * Lève l'exception OperationIncorrecte si cet objet n'est pas lié à une
    * instance de B.
    */
   public void detache() {
      if (attB == null) {
         // Cet objet n'est pas lié à une instance de B.
         throw new OperationIncorrecte() ;
      } else {
         attB.setA(null) ;
         attB = null ;
      }
   }
}
```

La méthode setA est à visibilité *paquetage* (visibilité par défaut). On ne peut pas la déclarer privée car elle doit être utilisée par la classe A. L'idéal serait de pouvoir déclarer que *seule* la classe A a le droit d'utiliser cette méthode. Ceci n'est pas possible en Java (on pourrait le faire en C++ avec une méthode amie, ou « *friend* »).

À condition de ne pas réutiliser setA en dehors de la classe A, ce codage garantit qu'on ne formera pas de paires de pointeurs incorrectes.

Exemple 2

Dans ce deuxième exemple, on utilise la multiplicité `1`.

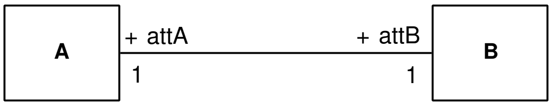


Fig. VI. 8 – Relation avec multiplicités `1`

Comme précédemment, on utilise un attribut *attA* de type A dans la classe B et un attribut *attB* de type B dans la classe A. Ces deux attributs ne doivent jamais avoir pour valeur *null*, car on ne peut pas avoir d'instance isolée de A ou de B.

```
class A {
   B attB ; // Instance de B liée à cet objet
}

class B {
   A attA ; // Instance de A liée à cet objet
}
```

Les contraintes à respecter sont les suivantes :

- $\forall a : A,\ a.attB \neq null$ et $a.attB.attA = a$;
- $\forall b : B,\ b.attA \neq null$ et $b.attA.attB = b$.

Par rapport à l'exemple 1, on ne peut plus attacher ou détacher une instance de B à une instance de A, ni l'inverse. On remarque donc qu'un codage « strict » de la multiplicité `1` est plus délicat que le codage de la multiplicité `0..1`.

On peut imaginer d'échanger les instances de B associées à deux instances de A.

```
/**
 * Classe B.
 */

public class B {

   // L'instance de A associée à cet objet.
   private A attA ;

   /**
    * Constructeur (non public).
    */
   B(A a) {
      attA = a ;
   }

   /**
    * Retourne l'instance de A associée à cet objet.
    */
   public A getA() {
      return attA ;
   }

   /**
    * Affecte l'instance a à attA (méthode non publique).
    * Précondition : a != null
    */
   void setA(A a) {
      if (a == null) {
         throw new OperationIncorrecte() ;
      } else {
         attA = a ;
      }
   }
}

/**
 * Classe A.
 */
```

```
public class A {

    // L'instance de B associée à cet objet.
    private B attB ;

    /**
     * Constructeur (public).
     */
    public A() {
        attB = new B(this) ;
    }

    /**
     * Retourne l'instance de B associée à cet objet.
     */
    public B getB() {
        return attB ;
    }

    /**
     * Affecte l'instance de B associée à cet objet à b.
     * (Méthode non publique)
     * Précondition : b != null
     */
    void setB(B b) {
        if (b == null) {
            throw new OperationIncorrecte() ;
        } else {
            attB = b ;
        }
    }

    /**
     * Echange les instances de B de cet objet et de l'objet a.
     * Précondition : a != null
     */
    public void echanger(A a) {
        if (a == null) {
            throw new OperationIncorrecte() ;
        } else {
            B temp = attB ;
            attB = a.getB() ;
            a.setB(temp) ;
            attB.setA(this) ;
            a.getB().setA(a) ;
        }
    }
}
```

Comme précédemment, les méthodes setA, setB, ainsi que le constructeur B(A) ont la visibilité *paquetage*, et ne doivent pas être utilisés ailleurs que dans les classes A et B. Si cette condition est respectée, on ne pourra pas construire de liens incorrects.

Exemple 3. Multiplicité de cardinalité fixée

Dans cet exemple, on considère une multiplicité à cardinalité fixée : la cardinalité de la relation est soit connue à la compilation, soit varie suivant les exécutions, mais reste fixée pour un objet et une exécution donnée.

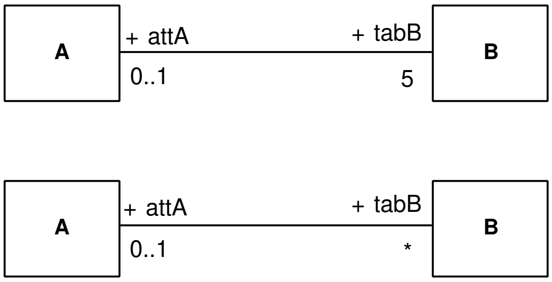


FIG. VI. 9 – Relation avec multiplicité fixée

On utilise un tableau pour stocker les instances de B liées à une instance de A.

```
class A {
    B[] tabB ; // Tableau des instances de B liées à cet objet
}

class B {
    A attA ; // L'instance de A liée à cet objet
}
```

Les contraintes à respecter sont les suivantes :

- $\forall a : A,\ a.tabB \neq null$ et $\forall i,\ (a.tabB[i] \neq null$ et $a.tabB[i].attA = a)$;
- $\forall b : B,\ b.attA \neq null \Rightarrow \exists i,\ b.attA.tabB[i] = b.$

```
/**
 * Classe B.
 */
public class B {

    // L'instance de A associée à cette instance.
    private A attA ;

    /**
     * Constructeur d'instance isolée.
     */
    public B() {
```

```
        attA = null ;
    }

    /**
     * Retourne l'instance de A associée à cette instance.
     */
    public A getA() {
        return attA ;
    }

    /**
     * Affecte l'instance de A associée à cette instance à l'objet a.
     * (Méthode non publique)
     */
    void setA(A a) {
        attA = a ;
    }
}

/**
 * Classe A.
 */
public class A {

    // L'ensemble des instances de B associées à cette instance.
    private B[] tabB ;

    /**
     * Constructeur d'instance liée à un ensemble d'instances de B.
     * Précondition : pour tout i, b[i] est une instance non nulle et isolée.
     * Postcondition : pour tout i, b[i] est liée à cette instance.
     */
    public A(B[] b) {
        if (b == null) {
            throw new OperationIncorrecte() ;
        } else {
            for (int i = 0 ; i < b.length ; i++) {
                if (b[i] == null || b[i].getA() != null) {
                    throw new OperationIncorrecte() ;
                }
            }
            tabB = b ;
            for (int i = 0 ; i < b.length ; i++) {
                b[i].setA(this) ;
            }
        }
    }

    /**
```

```
    * Retourne le tableau d'instances de B associées à cet objet.
    */
   public B[] getB() {
      return tabB ;
   }
}
```

Exemple 4. Multiplicité de cardinalité variable

Dans cet exemple, on considère une relation pour laquelle une instance de A est associée à un ensemble d'instances de B. Cet ensemble peut être modifié au cours de l'exécution du programme.

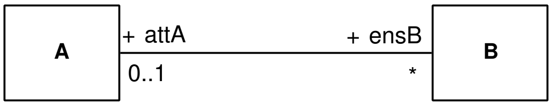


FIG. VI. 10 – Relation avec multiplicité variable

Pour stocker cet ensemble, on peut utiliser une collection Java, par exemple l'interface Set, et une implémentation de Set, comme HashSet.

```
class A {
   Set<B> ensB ; // Ensemble des instances de B liées à cet objet
   // Les éléments de ensB sont de type B
}

class B {
   A attA ; // L'instance de A liée à cet objet
}
```

Les contraintes à respecter sont les suivantes :

- $\forall a : A$,
  - $a.ensB \neq null$
  - $\forall o \in a.ensB,\ o \neq null$
  - $\forall b \in a.ensB,\ b.attA = a$;
- $\forall b : B,\ b.attA \neq null \Rightarrow b \in b.attA.ensB$.

```
/**
 * Classe B.
 */
public class B {

   // L'instance de A associée à cette instance.
   private A attA ;
```

```
    /**
     * Constructeur d'instance isolée.
     */
    public B() {
        attA = null ;
    }

    /**
     * Retourne l'instance de A associée à cette instance.
     */
    public A getA() {
        return attA ;
    }

    /**
     * Affecte l'instance de A associée à cette instance à l'objet a.
     * (Méthode non publique)
     */
    void setA(A a) {
        attA = a ;
    }
}

import java.util.Set ;
import java.util.HashSet ;

/**
 * Classe A.
 */
public class A {

    // L'ensemble des instances de B associées à cette instance.
    private Set<B> ensB ; // Ensemble d'éléments de type B.

    /**
     * Constructeur.
     */
    public A() {
        ensB = new HashSet<B>() ;
    }

    /**
     * Retourne l'ensemble d'instances de B associées à cet objet.
     */
    public Set<B> getB() {
        return ensB ;
    }
```

```
    /**
     * Attache l'instance b à cet objet.
     * Précondition : b est non null et est une instance isolée.
     */
    public void attache(B b) {
        if (b == null || b.getA() != null) {
            throw new OperationIncorrecte() ;
        } else {
            b.setA(this) ;
            ensB.add(b) ;
        }
    }

    /**
     * Détache l'instance b de cet objet.
     * Précondition : b est non null et est lié à cet objet.
     */
    public void detache(B b) {
        if (b == null || b.getA() != this) {
            throw new OperationIncorrecte() ;
        } else {
            b.setA(null) ;
            ensB.remove(b) ;
        }
    }
}
```

d) Héritage

Lors d'un héritage, les associations sont également héritées. Si on considère deux classes A et B, une relation R entre ces classes, et deux sous-classes A_2 et B_2 de A et B (cf. figure VI. 11). Alors la relation R est héritée au niveau de A_2 et B_2.

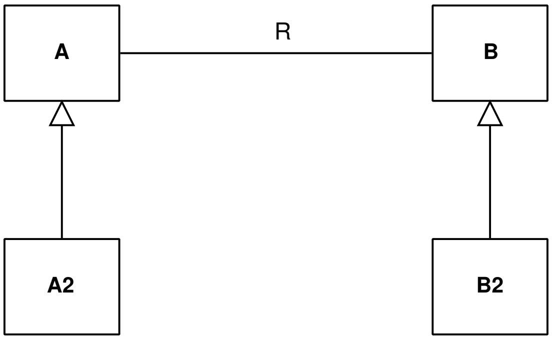


Fig. VI. 11 – Héritage de la relation R

Par exemple, le diagramme d'objets représenté figure VI. 12 doit pouvoir être construit.

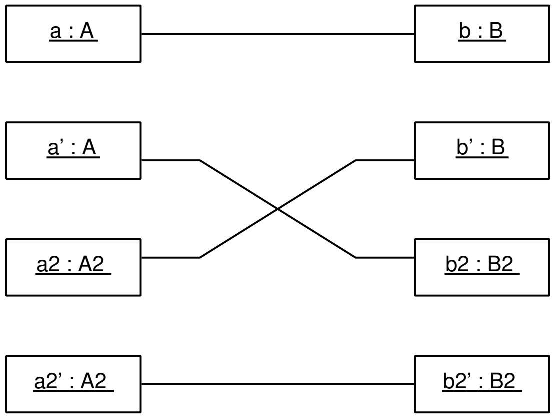


FIG. VI. 12 – Diagramme d'objets correspondant à la figure VI. 11

Nous avons proposé un codage de la forme suivante :

```
class A {
    B attB ; // Suivant la multiplicité, on utilise un type adéquat
}

class B {
    A attA ; // Suivant la multiplicité, on utilise un type adéquat
}

class A2 extends A { }
class B2 extends B { }
```

Avec ce codage, les attributs *attA* et *attB* sont hérités respectivement dans B et dans A. La relation R est donc héritée par A_2 et B_2. Le codage permet donc de construire le diagramme d'objets représenté figure VI. 12.

e) Agrégation

Il y a peu de différence entre une relation binaire quelconque et une agrégation. On peut noter que pour une agrégation, les cycles sont interdits dans un diagramme d'objets.

Si on veut éviter de pouvoir construire des cycles dans les diagrammes d'objets, il est nécessaire d'ajouter des tests dans les créations et les modifications de liens.

f) Composition

Les différences entre les agrégations et les compositions sont les suivantes :

- dans une composition, la multiplicité est `0..1` sur l'agrégat ;
- dans une composition, si un objet est détruit, alors ses composants sont également détruits.

En Java, on ne détruit jamais d'objet, car c'est le ramasse-miettes qui s'en charge. Si on code une composition en C++, il faut détruire les composants lorsqu'on détruit un objet.

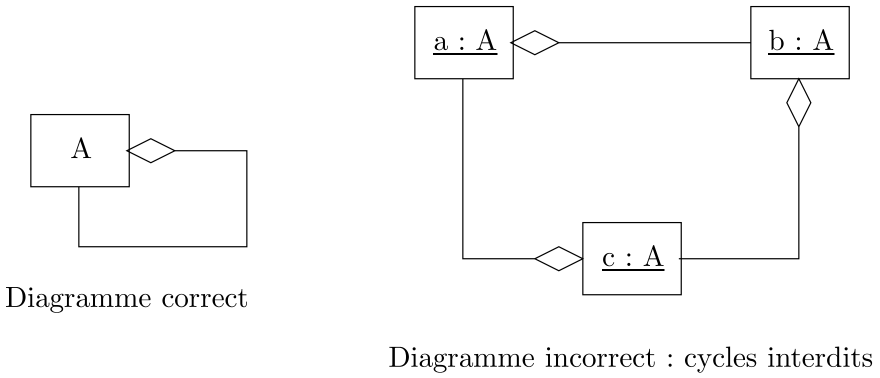


FIG. VI. 13 – Cycles et agrégations

3. Héritage multiple

Il y a *héritage multiple* lorsqu'une classe hérite de plusieurs classes. Les langages Java et Ada95 ne permettent pas de faire de l'héritage multiple, alors que C++ le permet.

a) Difficultés liées à l'héritage multiple

L'héritage multiple présente certaines difficultés, ce qui explique son interdiction dans certains langages :

- lorsqu'un un attribut est hérité par deux chemins différents (cf. figure VI. 14), doit-on avoir une seule valeur d'attribut ou cette valeur doit-elle être dupliquée ?
- lorsque deux méthodes de même nom sont héritées, quelle méthode doit-on choisir ?

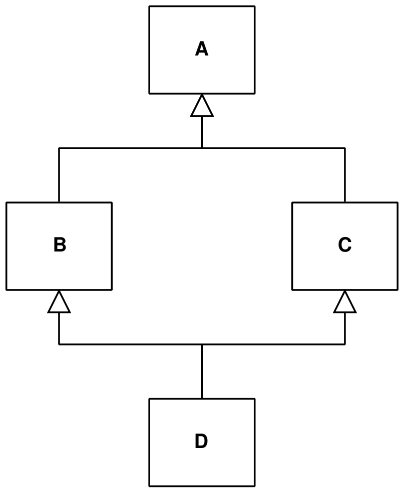


FIG. VI. 14 – Classe A héritée dans D par deux chemins

- les programmeurs ont des difficultés à utiliser l'héritage multiple à bon escient ;
- l'heritage multiple est plus compliqué à compiler, et il implique également un surcoût des programmes à l'exécution.

Méthode héritées par deux chemins différents

Si les classes B et C comportent deux méthodes qui ont la même signature, il y a une ambiguïté sur cette méthode au niveau de la classe D. La résolution de l'ambiguïté peut être réalisée de plusieurs façons différentes :

- on peut effectuer un choix « arbitraire » de l'une des deux méthodes (par exemple, choisir la première ayant été compilée) ;
- on peut obliger le renommage de l'une des deux méthodes dans la classe D (c'est par exemple le choix effectué dans le langage Eiffel) ;
- lors d'un appel de cette méthode, on peut obliger l'utilisateur à préciser la méthode appelée (c'est le choix effectué dans le langage C++).

Le choix effectué dans le langage Eiffel a les avantages suivants :

- il oblige le programmeur à prendre conscience des ambiguïtés au moment de la création de la classe (donc assez tôt) ;
- il permet d'écrire du code plus simple, puisque la désambiguïsation est réalisée une seule fois.

b) Simulation de l'héritage multiple en Java

Utilisation de la délégation

Pour simuler l'héritage multiple en Java, on peut remplacer l'un des deux héritages par une délégation (cf. figure VI. 15).

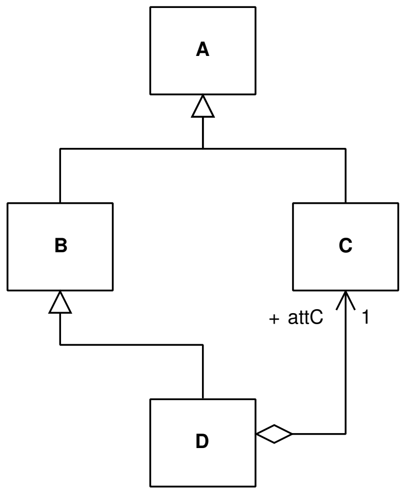


Fig. VI. 15 – Héritage multiple, simulé par délégation

```
class D extends B {

    // Attribut permettant la délégation.
    C attC ;

    // Pour chaque méthode m de C qui doit être héritée dans D, on réécrit
```

```
   // un appel (utilisation de la délégation).
   void m(...) {
      attC.m(...) ;
   }
}
```

Supposons que les classes B et C comportent deux méthodes de même nom. On a trois possibilités :

- si on ne réécrit pas la méthode de C dans la classe D, c'est la méthode de B qui sera choisie ;
- si on réécrit la méthode de C dans D, c'est la méthode de C qui sera choisie ;
- si on redéfinit la méthode dans D, les méthodes de B et de C seront cachées par cette redéfinition.

Avec ce codage, on peut appeler les méthodes de B et les méthodes de C sur un objet de type D. On a donc simulé l'*héritage*.

Exemple

Soient methA, methB, methC et methD des méthodes définies respectivement dans les classes A, B, C et D.

On peut écrire le code Java suivant :

```
D d = new D(...) ;
d.methA(...) ;  // Héritage "normal" de A
d.methB(...) ;  // Héritage "normal" de B
d.methC(...) ;  // Héritage d'une méthode de C, simulé par délégation
d.methD(...) ;  // Appel d'une méthode de D
```

Néanmoins, un objet de type C n'est pas un objet de type D. On n'a donc pas la propriété de *substitution* (ou *sous-typage*).

En particulier, on ne peut pas écrire :

```
C d2 = new D(...) ;  // Erreur de type à la compilation
```

A fortiori, on n'a pas de liaison dynamique, et on ne peut pas effectuer d'appel de la forme `d2.methD(...)`. Pour cette raison, on introduit une interface, qui permet de simuler en partie le sous-typage.

Utilisation d'interfaces

On ajoute une interface CInt qui contient (le profil de) toutes les méthodes de C, y compris les méthodes héritées de A. Les classes C et D implémentent l'interface CInt (cf. figure VI. 16).

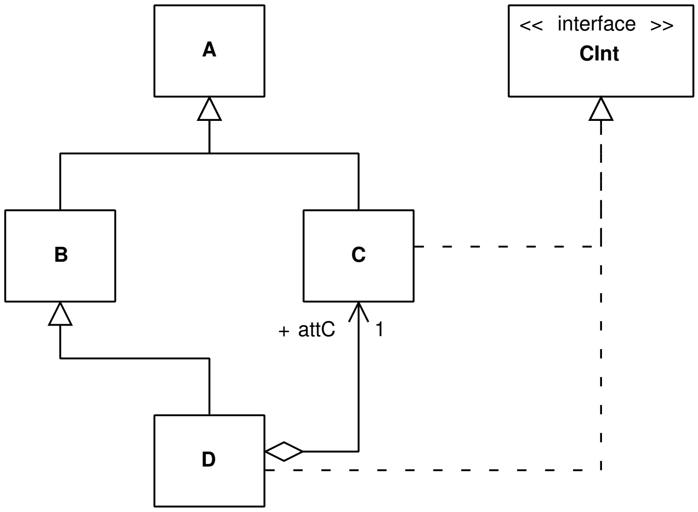


Fig. VI. 16 – Héritage multiple, simulé par délégation et interface

Comme précédemment, la délégation simule l'héritage. De plus, on peut utiliser CInt pour faire du sous-typage :

```
CInt d2 = new D(...) ; // ok
```

On peut appliquer les méthodes de C (et de A) à d2 :

```
d2.methA(...) ;
d2.methC(...) ;
```

L'objet d2 n'est pas de type C, par contre c'est un objet de type CInt. Si la méthode methC est déclarée dans C, et redéfinie dans D, on a bien le mécanisme de liaison dynamique :

```
CInt c1 = new C(...) ;
c1.methC(...) ; // Appel de methC dans C.

CInt c2 = new D(...) ;
c2.methC(...) ; // Liaison dynamique : appel de methC dans D.
```

Cette approche présente cependant certains inconvénients :

- Il faut recopier dans CInt tous les profils des méthodes de C, y compris les méthodes héritées. Le programme est donc difficile à maintenir, par exemple si on modifie une classe héritée dans C, comme A.
- Les attributs de classes hérités à la fois par B et C, comme ceux de la classe A, sont dupliqués :
  - on récupère un attribut par l'intermédiaire de la classe B, héritée dans D ;
  - on récupère un attribut par le délégué attC.

4. Diagrammes d'états-transitions

Il existe beaucoup de façons différentes de traduire un diagramme d'états-transitions dans un langage de programmation. Il y a différents choix à effectuer, concernant en particulier

- le codage des états : les états peuvent être codés soit implicitement, soit explicitement ;
- le codage des transitions : le code peut être soit dispersé dans plusieurs classes, soit regroupé dans une seule classe ;
- le comportement d'un objet, qui peut être soit actif, soit passif.

Dans la suite de ce paragraphe, on montre deux exemples de codage de diagrammes d'états-transitions.

a) Alarme à plusieurs niveaux

On considère un système comportant une alarme.

- L'alarme peut être active ou non.
- Lorsqu'elle est active, l'alarme est au niveau i pendant une minute, puis passe au niveau $i + 1$.
- Lorsque l'alarme atteint le niveau maximal, elle s'arrête au bout d'une minute.
- L'alarme peut être désactivée par l'utilisateur.

La figure VI. 17 montre le diagramme de classes du système, qui réutilise la classe Java Timer ; la figure VI. 18 montre le diagramme d'états-transitions associé à la classe Timer ; la figure VI. 19 montre le diagramme d'états-transitions associé à la classe Alarme.

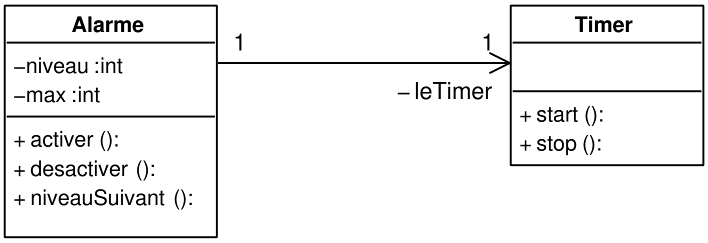


Fig. VI. 17 – Diagramme de classes de l'alarme

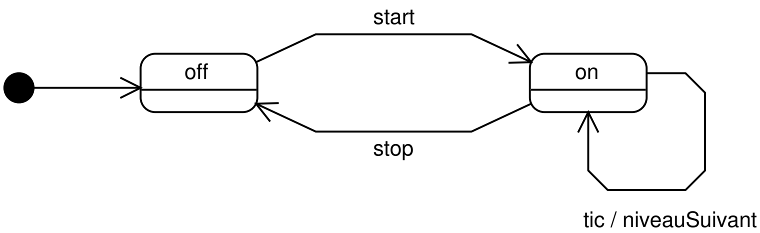


Fig. VI. 18 – Diagramme d'états-transitions de la classe Timer

Pour réaliser le codage en Java, on choisit de représenter les états de façon implicite, et de coder les transitions dans les méthodes.

L'état désactivée correspond à niveau $= 0$; l'état activée correspond à $1 \leq$ niveau $\leq$ max.

```
import javax.swing.Timer ;
```

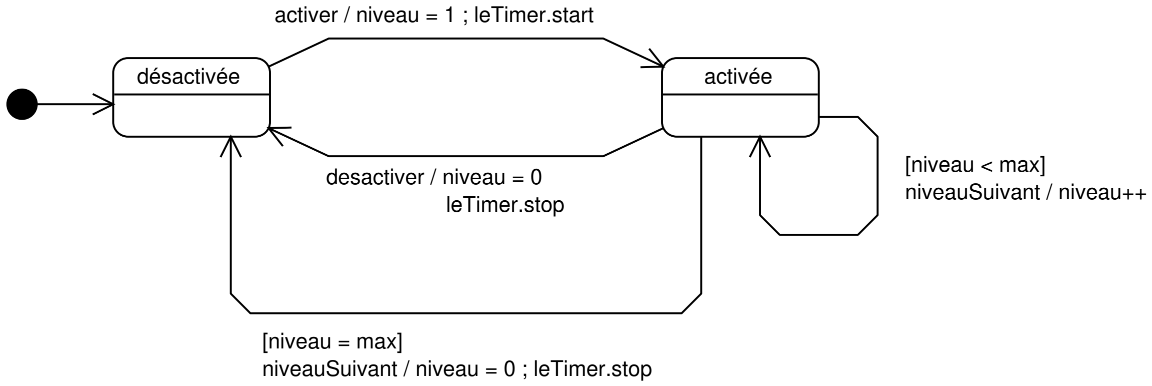


FIG. VI. 19 – Diagramme d'états-transitions de la classe Alarme

```
import java.awt.event.ActionListener ;
import java.awt.event.ActionEvent ;

/**
 * Classe des alarmes.
 */
public class Alarme {

   // Le niveau de cette alarme.
   private int niveau ;

   // Le niveau maximal de cette alarme.
   private int max ;

   // L'horloge associée à cette alarme.
   private Timer leTimer ;

   /**
    * Constructeur.
    */
   public Alarme(int max) {
      this.max = max ;
      niveau = 0 ;

      // Lorsqu'il est en marche, le timer appelle niveauSuivant chaque minute.
      // (1 minute = 60000 millisecondes).
      leTimer =
         new Timer(60000, new ActionListener() {
            public void
               actionPerformed(ActionEvent e) {
                  niveauSuivant() ;
               }
         }) ;
```

```
    }

    /**
     * Active cette alarme.
     */
    public void activer() {
        if (niveau == 0) {
            niveau = 1 ;
            leTimer.start() ;
        }
    }

    /**
     * Désactive cette alarme.
     */
    public void desactiver() {
        if (niveau != 0) {
            niveau = 0 ;
            leTimer.stop() ;
        }
    }

    /**
     * Passe au niveau suivant. L'alarme est désactivée si elle est au
     * niveau maximal.
     */
    public void niveauSuivant() {
        if (niveau == max) {
            niveau = 0 ;
            leTimer.stop() ;
        } else if (niveau != 0) {
            niveau++ ;
        }
    }
}
```

b) Activités s'effectuant en parallèle

Le but de ce deuxième exemple est de montrer comment des objets actifs peuvent communiquer par des envois d'événements.

On considère une application qui permet de réaliser deux activités (activité 1 et activité 2). On a une fenêtre (cf. figure VI. 20) qui comporte quatre boutons :

- le bouton *Choix* permet de sélectionner alternativement l'activité 1 et l'activité 2 ;
- le bouton *Démarrer* permet de démarrer l'activité sélectionnée ;
- le bouton *Arrêter* permet d'arrêter l'activité sélectionnée ;
- le bouton *Fermer* permet de quitter l'application.

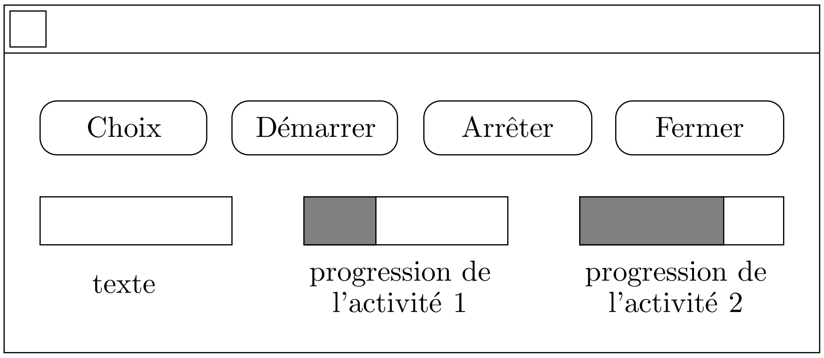


Fig. VI. 20 – Fenêtre permettant de commander les activités

Le champ texte permet d'afficher un message (sélection d'une activité, démarrage ou arrêt d'une activité).

Les deux barres de progression visualisent l'avancement des deux activités.

Le diagramme de classes de l'application est représenté figure VI. 21 ; les diagrammes d'états-transition de l'interface et d'une activité sont représentés figure VI. 22 et VI. 23.

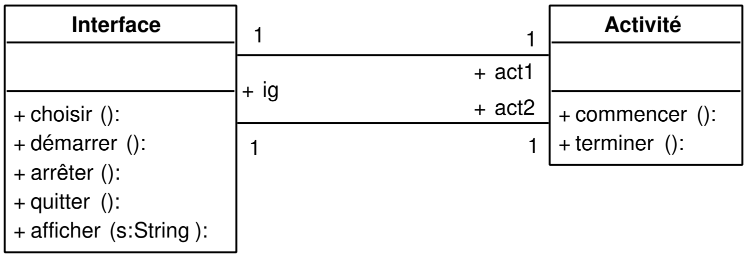


Fig. VI. 21 – Diagramme de classes de l'application permettant d'exécuter deux activités

Codage de l'automate associé à l'interface graphique

On code les états de l'interface de façon explicite.

```
/**
 * Classe des états de l'interface.
 */
class EtatInt {
   static final EtatInt e1 = new EtatInt() ;
   static final EtatInt e2 = new EtatInt() ;
   private EtatInt() { }
}
```

On code également les événements de l'automate de façon explicite : on introduit pour cela une classe EvtInt.

```
/**
 * Classe des événements de l'automate associé à l'interface.
```

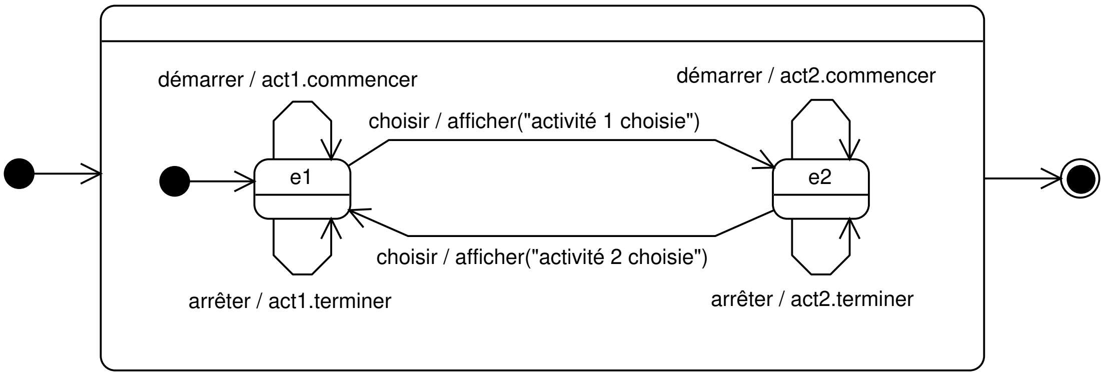


Fig. VI. 22 – Diagramme d'états-transitions de la classe Interface

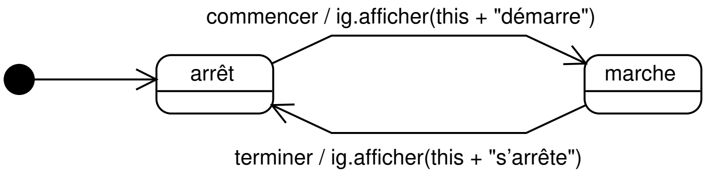


Fig. VI. 23 – Diagramme d'états-transitions de la classe Activité

```
 */
class EvtInt {
    // Le nom de l'événement
    String nom ;

    // Les paramètres de l'événement
    Vector param ;

    /**
     * Constructeur d'événement sans paramètre.
     */
    EvtInt(String s) {
        nom = s ;
        param = new Vector() ;
    }

    /**
     * Constructeur d'événement avec un paramètre.
     */
    EvtInt(String s, Object param1) {
        nom = s ;
        param = new Vector() ;
        param.add(param1) ;
    }

    /**
```

```
     * Accès au premier paramètre.
     */
    Object param1() {
        return param.elementAt(0) ;
    }

    /**
     * Teste si cet événement a pour nom la chaîne s.
     */
    boolean aPourNom(String s) {
        return nom.equals(s) ;
    }
}
```

Pour coder l'automate, on choisit de coder toutes les transitions de l'automate dans une seule classe. Il s'agit d'un codage « non objet ».

```
class ControleurInt {

    // L'interface associée à ce contrôleur
    Interface ig ;

    // L'état courant de l'automate
    EtatInt etatCourant ;

    /**
     * Constructeur.
     */
    ControleurInt(Interface i) {
        ig = i ;
        // L'état initial est e1
        etatCourant = EtatInt.e1 ;
    }

    /**
     * Passe à l'état suivant.
     */
    void etatSuivant(EvtInt ev) {
        if (etatCourant == EtatInt.e1) {
            if (ev.aPourNom("choisir")) {
                etatCourant = EtatInt.e2 ;
                ig.afficher("activité 2 choisie") ;
            } else if (ev.aPourNom("démarrer")) {
                ig.act1.commencer() ;
            } else if (ev.aPourNom("arrêter")) {
                ig.act1.arreter() ;
            } else if (ev.aPourNom("afficher")) {
                String mess = (String) ev.param1() ;
                ig.etiq.setText(mess) ;
```

```
            } else {
                throw new EvtIncorrect(ev) ;
            }
        } else if (etatCourant == EtatInt.e2) {
            if (ev.aPourNom("choisir")) {
                etatCourant = EtatInt.e2 ;
                ig.afficher("activité 1 choisie") ;
            } else if (ev.aPourNom("démarrer")) {
                ig.act2.commencer() ;
            } else if (ev.aPourNom("arrêter")) {
                ig.act2.arreter() ;
            } else if (ev.aPourNom("afficher")) {
                String mess = (String) ev.param1();
                ig.etiq.setText(mess) ;
            } else {
                throw new EvtIncorrect(ev) ;
            }
        } else {
            throw new EtatIncorrect(etatCourant) ;
        }
    }
}
```

On code enfin l'interface :

```
/**
 * Classe Interface.
 */
class Interface {
    Activite act1, act2 ; // Les deux activités contrôlées par l'interface
    ControleurInt ctrlI ; // L'automate associé à cette interface

   ... // Eléments de l'interface

   // Choix d'une activité
   void choisir() {
      ctrlI.etatSuivant(new EvtInt("choisir")) ;
   }

   // Démarre l'activité sélectionnée
   void demarrer() {
      ctrlI.etatSuivant(new EvtInt("démarrer"));
   }

   // Arrête l'activité sélectionnée
   void arreter() {
      ctrlI.etatSuivant(new EvtInt("arrêter")) ;
   }
```

```
  // Affiche la chaîne s
  void afficher(String s) {
    ctrlI.etatSuivant(new EvtInt("afficher",s));
  }
}
```

Codage de l'automate qui contrôle une activité

Pour coder l'automate qui contrôle une activité, on choisit de coder les états et les événements de façon explicite, de regrouper le codage des transitions dans une seule classe (codage « non objet »).

```
/**
 * Classe des états de l'automate qui contrôle une activité.
 */
class EtatAct {
    static final EtatAct marche = new EtatAct();
    static final EtatAct arret  = new EtatAct();
    private EtatAct() { }
}

/**
 * Classe des événements de l'automate.
 */
class EvtAct {
    static final EvtAct commencer= new EvtAct();
    static final EvtAct terminer = new EvtAct();
    private EvtAct() { }
}
```

Les activités doivent pouvoir s'exécuter en parallèle. Pour cela, on associe à chaque activité un « thread », que l'on démarre lorsqu'on construit une activité.

```
class Activite implements Runnable {

    // Le contrôleur associé à cette activité.
    ControleurActivite ctrlA ;

    // L'interface graphique associée à cette activité.
    Interface ig ;

    /**
     * Constructeur.
     */
    Activite(Interface i) {
        ig = i ;
```

```
        ctrlA = new ControleurActivite(this) ;
        // Démarre un nouveau thread associé à cette activité.
        // La méthode start appelle run.
        new Thread(this).start() ;
    }

    /**
     * Commence cette activité.
     */
    void commencer() {
        ctrlA.envoyer(EvtAct.commencer) ;
    }

    /**
     * Arrête cette activité.
     */
    void terminer() {
        ctrlA.envoyer(EvtAct.terminer) ;
    }

    /**
     * Méthode que doit implémenter tout thread, appelée par start.
     */
    public void run() {
        while (true) {
            ctrlA.etatSuivant() ; // Fait avancer l'automate associé à cette activité
        }
    }
}
```

Pour coder l'automate associé à une activité, on doit dissocier l'envoi d'un message (événement) et le fait de passer à l'état suivant. Pour cela, on utilise une file d'attente, qui sert à stocker les événements envoyés à une activité.

```
class ControleurActivite {

    // L'activité contrôlée par cet automate
    Activite lActivite ;

    // L'état courant de l'automate
    EtatAct etatCourant ;

    // La file d'événements reçus par l'automate
    private Vector fileEvt ;

    /**
     * Constructeur.
     */
    ControleurActivite(Activite a) {
```

```
        lActivite = a ;
        etatCourant = EtatAct.arret ;
        fileEvt = new Vector() ;
    }

    /**
     * Envoie l'événement ev au contrôleur :
     *  ev est ajouté dans la file d'attente.
     */
    synchronized void envoyer(EvtAct ev) {
        fileEvt.add(ev) ;
    }

    /**
     * Lit un événement dans la file d'attente.
     * Retourne null si la file est vide.
     */
    synchronized EvtAct lireEvenement() {
        EvtAct ev = null ;
        if (!fileEvt.isEmpty()) {
            ev = (EvtAct) fileEvt.elementAt(0) ;
            fileEvt.removeElementAt(0) ;
        }
        return ev ;
    }

    /**
     * Attend qu'un événement arrive dans la file d'attente.
     */
    EvtAct attendreEvenement() {
        EvtCA ev = lireEvenement() ;
        while (ev == null) {
            // Comme on ne fait rien, on laisse les autres threads s'exécuter.
            Thread.yield() ;
            ev = lireEvenement() ;
        }
        return ev ;
    }

    /**
     * Passe à l'état suivant.
     */
    void etatSuivant() {
        EvtAct ec ;
        if (etatCourant == EtatAct.arret) {
            ev = attendreEvenement() ;
            if (ev == EvtAct.commencer) {
                etatCourant = EtatAct.marche ;
                lActivite.ig.afficher(lActivite + "démarre") ;
```

```
            }
        } else if (etatCourant == EtatAct.marche) {
            ev = lireEvenement() ;
            if (ev = EvtAct.terminer) {
                etatCourant = EtatAct.arret ;
                lActivite.ig.afficher(lActivite + "s'arrête") ;
            } else if (ev == null) {
                // Pas d'événement dans la file : on effectue l'activité.
                ...
            }
        }
    }
}
```

Les méthodes envoyer et lireEvenement sont en exclusion mutuelle (« *synchronized* ») car elles agissent toutes les deux sur la file d'événements. Ainsi, le code des deux méthodes ne peut pas être entrelacé. Si ce n'était pas le cas, certains événements pourraient être dupliqués ou perdus.

Bibliographie

M.-C. Gaudel, B. Marre, F. Schlienger, G. Bernot. *Précis de génie logiciel.* Masson 1996.

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

P.-A. Muller, N. Gaertner. *Modélisation objet avec UML. Deuxième édition.* Eyrolles, 2000.

J. Rumbaugh, I. Jacobson, G. Booch. *Unified Modeling Language Reference Manuel.* Addison Wesley, 1999.

E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley. 1995.

Rémy Fannader, Hervé Leroux. *UML, Principes de modélisation.* Dunod, 1999.

C. Larman. *UML et les Design Patterns.* Campus Press, 2002.

Robert C. Martin. *Agile Software Development. Principles, Patterns and Practices.* Prentice Hall 2002.

Robert C. Martin. *UML for Java Programmers.* Prentice Hall 2003.

Sinan Si Alhir. *Introduction à UML.* O'Reilly, 2004.

F. Buschman, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns.* Wiley, 1996.

Unified Modeling Language Specification (1.5). OMG, 2003.

Site sur UML : `http://www.uml.org`

Site sur JUnit : `http://www.junit.org`

Une grande partie des diagrammes de ces notes a été dessinée à l'aide de l'outil Poseidon, qui permet de dessiner différents diagrammes UML (Site : `http://www.gentleware.com`).

Exercices

Exercice 1. Cycle de vie du logiciel
Dans le projet Génie Logiciel réalisé au mois de janvier, identifier les activités, documents et programmes (réalisés par les enseignants et les étudiants) correspondant aux différentes étapes du cycle de vie du logiciel.

Exercice 2. Modèle incrémental
On suppose qu'on veut réaliser le projet Génie Logiciel avec un processus itératif comportant une dizaine d'itérations.

1. Comment peut-on spécifier un incrément ?
2. Proposer les incréments correspondant à ces itérations.
3. Discuter les avantages et inconvénients de cette approche.

Exercice 3. Programmation extrême
Quels sont les pratiques de la programmation extrême que vous avez appliquées ou qui vous semblent intéressants à appliquer pendant le projet Génie Logiciel ?

Exercice 4.
Proposer un diagramme d'objets correspondant au diagramme de classes représenté figure VI. 24.

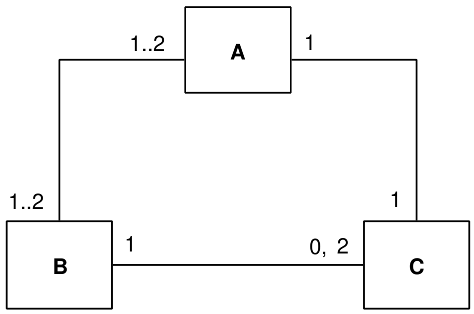


Fig. VI. 24 – Diagramme de classes de l'exercice 4

Exercice 5.
Proposer un diagramme d'objets correspondant au diagramme de classes représenté figure VI. 25.

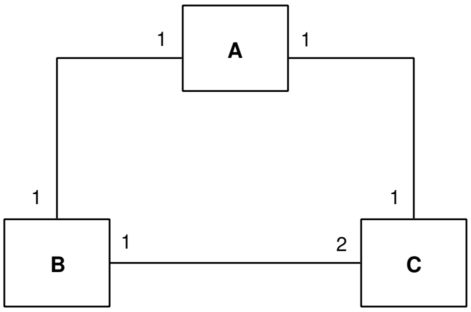
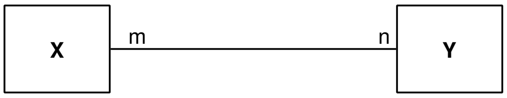


FIG. VI. 25 – Diagramme de classes de l'exercice 5

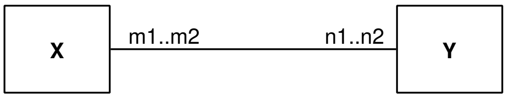
Exercice 6.
On s'intéresse aux contraintes de multiplicités associées à une relation.

1. On considère le diagramme de classes suivant.

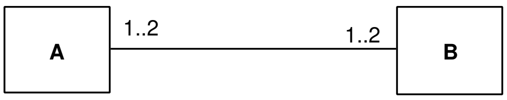


   Soit un diagramme d'objets correspondant à ce diagramme de classes. Donner une relation entre m, n, $\operatorname{Card} X$ et $\operatorname{Card} Y$.

2. Même question avec le diagramme de classes suivant.

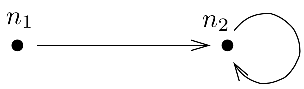


3. Soit le diagramme de classes suivant.



   Donner tous les diagrammes d'objets, utilisant des objets anonymes, tels que $\operatorname{Card} A = 3$ et $\operatorname{Card} B = 3$.

Exercice 7. Graphes
Dessiner un diagramme de classes permettant de modéliser un graphe orienté. Dessiner un diagramme d'objets correspondant au graphe suivant.

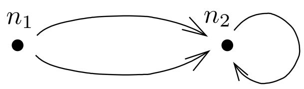


Rappel : un graphe comporte un ensemble de sommets et une relation entre ces sommets.

Exercice 8. Multi-graphes
On cherche à modéliser des multi-graphes.

Rappel : dans un multi-graphe, on peut avoir plusieurs arcs qui relient le même couple de sommets.

Dessiner un diagramme de classes permettant de modéliser un multi-graphe. Dessiner un diagramme d'objets correspondant au multi-graphe suivant.



Exercice 9. Figures géométriques
Une figure géométrique peut être un rectangle, un segment ou un cercle ou composée d'un ensemble de figures. Une figure peut être affichée, déplacée ou agrandie.

1. Dessiner un diagramme de classes qui modélise les figures géométriques.
2. Dessiner un diagramme d'objets modélisant la figure VI. 26.
3. Dessiner un diagramme de séquence correspondant à un déplacement de la Figure VI. 26.
4. Dessiner un diagramme de collaboration correspondant à un déplacement de la Figure VI. 26.
5. Écrire un programme Java codant les classes et le déplacement d'une figure.

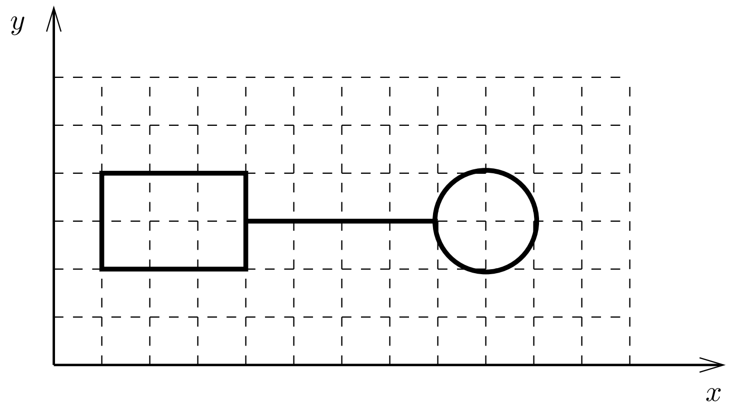


Fig. VI. 26 – Figure géométrique

Exercice 10. Classes et interfaces Java
On cherche à modéliser la notion de classe et d'interface du langage de programmation Java. On retient les informations suivantes.

- Une classe (ou une interface) contient des attributs et des méthodes.

- Un type est soit un type de base (int, float, boolean, void), soit une classe, soit une interface.
- Un attribut a un nom et un type.
- Une méthode a un nom, un type de retour, et des paramètres.
- Un paramètre a un nom et un type.
- Une classe (ou une interface) a un nom.
- Un paquetage a un nom, et contient des classes et des interfaces.

Questions

1. Dessiner un diagramme de classes qui modélise les éléments Java définis ci-dessus.
2. Dessiner un diagramme d'objets correspondant à la classe Java suivante :

```
class Point {
    int x ; // abscisse
    int y ; // ordonnée
    boolean mêmePos(Point P){ ... }
}
```

On cherche à afficher la classe Java Point sous la forme suivante :

```
class Point {
    int x ;
    int y ;
    boolean mêmePos(Point P){ }
}
```

On suppose qu'on dispose des opérations print(String) et println(String) pour effectuer un affichage et que l'opération + effectue la concaténation des chaînes de caractères.

3. Dessiner un diagramme de séquence correspondant à l'affichage de la classe Point.
4. Dessiner un diagramme de collaboration correspondant à l'affichage de la classe Point.
5. Écrire un programme Java qui implémente les différentes classes ainsi que la méthode afficher.

Exercice 11. Clics et doubles clics d'une souris à trois boutons
On considère une souris qui comporte trois boutons : B_1, B_2 et B_3.

L'objectif de l'exercice consiste à définir un automate qui transforme les clics physiques (appuis sur les boutons B_1, B_2 ou B_3) en clics logiques. Un clic logique est de la forme clic(b, d) où $1 \leq b \leq 3$ indique le numéro du bouton et $1 \leq d \leq 2$ indique s'il s'agit d'un simple clic ou d'un double clic. On a un double clic lorsqu'on appuie deux fois sur le même bouton successivement et à moins de t milli-secondes d'intervalle.

1. Donner les événements produits par la séquence d'appels représentée figure VI. 27.
2. Dessiner un automate qui modélise le comportement de la souris, plus précisément, qui modélise la transformation des clics physiques en clics logiques.

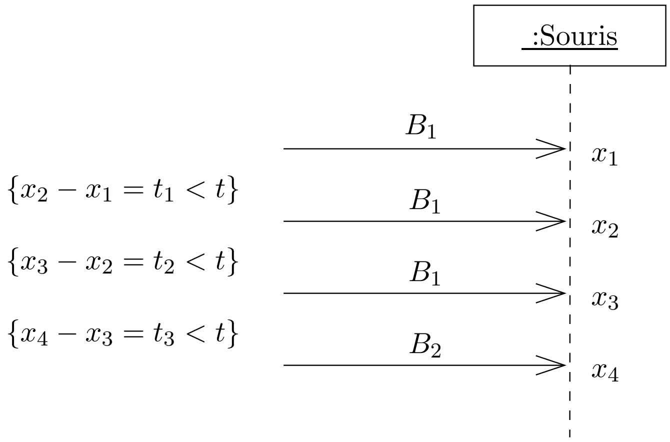


FIG. VI. 27 – Diagramme représentant une séquence de clics

Exercice 12. Comportement d'une platine cassettes
On considère une platine cassettes comportant les boutons suivants : on_off, play, stop, pause, av_r (avance rapide), ret_r (retour rapide). Le fonctionnement de la platine cassettes est le suivant :

- Lorsque la platine est éteinte, on_off permet de l'allumer (les autres boutons sont inopérants).
- Lorsque la platine est allumée, on_off permet de l'éteindre ; play permet de la démarrer. La platine se met alors en marche, en avance normale.
- Lorsque la platine est en marche, en avance normale, le bouton av_r permet de passer en avance rapide ; le bouton ret_r permet de passer en retour rapide (dans les deux cas, on revient en avance normale avec stop) ; pause permet de faire une pause (on revient en avance normale avec pause).
- Lorsque la platine est en avance normale ou en pause, un appui sur stop permet de l'arrêter.

Dessiner un diagramme d'états-transition qui modélise le comportement de la platine.

Exercice 13. Comportement d'un bouton
On cherche à modéliser le comportement d'un bouton (classe Java JButton). À un bouton est associée une action, qui est exécutée lorsque l'on clique dessus. La figure VI. 28 montre le comportement simplifié d'un bouton.

On cherche à modéliser plus finement le comportement d'un bouton, en tenant compte des éléments suivants.

- Un bouton peut être visible ou invisible. On passe d'un état à l'autre avec la méthode setVisible(*boolean*).
- Un bouton peut être actif ou inactif. On passe d'un état à l'autre avec la méthode setEnabled(*boolean*).
- Un clic peut être effectué uniquement si le bouton est visible et actif.

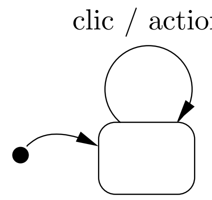


Fig. VI. 28 – Comportement simplifié d'un bouton

- Pour effectuer un clic, la souris doit pointer sur le bouton et on appuie sur le bouton, qui devient « armé ». L'action se déclenche lorsque l'on relâche le bouton.
- Si la souris quitte le bouton armé, celui-ci se désarme, et l'action ne se déclenche pas lorsqu'on relâche le bouton. Si la souris pointe à nouveau sur le bouton, celui-ci se réarme.

1. Décrire les événements auxquels le bouton peut réagir.
2. Dessiner un diagramme d'états-transitions qui modélise le comportement d'un bouton.
3. Que se passe-t-il lorsque l'on appuie sur le bouton, puis que l'on pointe sur le bouton avec la souris, puis que l'on relâche le bouton ?

Exercice 14. Fenêtre d'impression

Un logiciel comporte une fonctionnalité permettant d'imprimer un document. La figure VI. 29 montre la forme de la fenêtre.

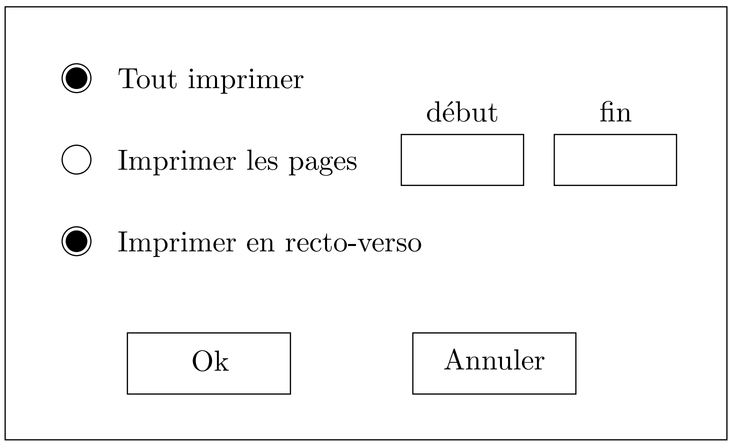


Fig. VI. 29 – Fenêtre d'impression

Les boutons « tout imprimer » et « imprimer les pages » ne peuvent pas être sélectionnés en même temps.

On peut indiquer la page de début et la page de fin uniquement lorsque le bouton « imprimer les pages » est sélectionné.

1. Définir les événements auxquels le système peut réagir.
2. Dessiner un diagramme d'états-transitions représentant le comportement de la fenêtre d'impression.

3. « Aplatir » ce diagramme d'états-transitions (c'est-à-dire : donner un automate équivalent qui ne comporte ni sous-état, ni sous-automates mis en parallèle).

Exercice 15. Montre digitale
On considère une montre digitale comportant quatre boutons A, B, C et D. Le bouton A est un bouton de mode, qui permet de passer du mode heure au mode date, puis au mode chronomètre, puis au mode alarme et enfin de revenir au mode heure.

Lorsque la montre est en mode heure :

- le bouton C permet de faire clignoter alternativement les heures et les minutes ;
- le bouton D permet de faire avancer les heures (si les heures clignotent) ou les minutes (si les minutes clignotent) ;
- le bouton B permet de remettre les secondes à zéro, lorsque les heures ou les minutes clignotent ;
- le bouton A permet de revenir au mode heure standard.

Lorsque la montre est en mode date :

- le bouton C permet de faire clignoter alternativement le jour et le mois ;
- le bouton D permet de faire avancer le jour et le mois ;
- le bouton A permet de revenir au mode date standard.

Lorsque la montre est en mode chronomètre :

- le bouton C permet le démarrage et l'arrêt du chronomètre ;
- le bouton D permet d'afficher le temps intermédiaire (lorsque le chronomètre est en marche) ; une seconde pression permet de voir à nouveau le chronomètre défiler ;
- le bouton B permet de remettre le chronomètre à zéro, lorsque celui-ci est arrêté.

Lorsque la montre est en mode alarme :

- le bouton C permet de faire clignoter alternativement les heures et les minutes ;
- le bouton D permet de faire avancer les heures ou les minutes ;
- le bouton B permet d'activer et de désactiver l'alarme ;
- le bouton A permet de revenir au mode alarme non clignotant.

Questions

1. Quels sont les événements extérieurs auxquels le système doit réagir ?
2. Décrire un diagramme de séquence correspondant au passage de l'heure d'hiver à l'heure d'été.
3. Décrire un diagramme de séquence correspondant à un chronométrage avec une prise de temps intermédiaire.
   Pour les questions 2 et 3, donner les pré- et les post-conditions correspondantes.
4. Décrire un diagramme d'états-transitions correspondant au comportement de la montre.

Exercice 16. Analyse et expression des besoins de la minuterie
Le but de l'exercice est de réaliser l'analyse des besoins du système « Minuterie ».

1. Décrire les acteurs.
2. Décrire les événements auxquels le système doit réagir.
3. Identifier les passages imprécis, ambigus ou incomplets du cahier des charges, proposer une question correspondante (qui pourrait être posée aux utilisateurs) et proposer une réponse. Ce point pourra être complété au fur et à mesure qu'on répondra aux questions 4, 5 et 6.
4. Décrire les cas d'utilisation du système : dessiner un diagramme de cas d'utilisation et décrire les différents cas d'utilisation.
5. Documenter les cas d'utilisation à l'aide de diagrammes de séquence système.
6. Spécifier le comportement du système à l'aide d'un diagramme d'états-transitions.

Exercice 17. Analyse et expression des besoins du système de traitement des ventes
Le but de l'exercice est de réaliser l'analyse des besoins du système de traitement des ventes.

1. Décrire les acteurs.
2. Décrire les cas d'utilisation du système : dessiner un diagramme de cas d'utilisation et décrire les différents cas d'utilisation.
3. Identifier une imprécision du cahier des charges concernant la gestion des clients. Proposer une évolution du cahier des charges de façon à permettre une gestion ergonomique des fiches clients.
4. Documenter les cas d'utilisation à l'aide de diagrammes de séquence système.

Exercice 18. Modélisation objet du domaine pour la minuterie
Le but de l'exercice est de réaliser la modélisation objet du domaine pour le système « Minuterie ».

1. Dessiner un diagramme de classes du domaine.
2. Dessiner un diagramme d'objets correspondant à une minuterie.

Exercice 19. Modélisation objet pour le système de traitement des ventes
Le but de cet exercice est de réaliser la modélisation objet du domaine pour le système de traitement des ventes.

1. Dessiner un diagramme de classes du domaine.
2. Dessiner un diagramme d'objets correspondant à ce diagramme de classes.

Exercice 20. Architecture pour la minuterie
Proposer une architecture pour le système « Minuterie ».

Exercice 21. Architecture pour le système de traitement des ventes
Proposer une architecture pour le système de traitement des ventes.

Exercice 22. Patron de conception Observateur
Le but de l'exercice est d'instancier le patron de conception Observateur sur les affichages

de pourcentage (affichage textuel, par histogramme et par camemberts). En Java, écrire les classes et interfaces Sujet, Observateur, Pourcentage et Texte.

Exercice 23. Patron de conception Interprète
On considère la grammaire abstraite suivante :

$$\begin{array}{lcl} \text{Exp} & \rightarrow & \text{ExpBin} \mid \text{ExpUn} \mid \underline{\text{num}} \mid \underline{\text{var}} \\ \text{ExpBin} & \rightarrow & \text{Exp} + \text{Exp} \mid \text{Exp} - \text{Exp} \mid \text{Exp} * \text{Exp} \\ \text{ExpUn} & \rightarrow & + \text{Exp} \mid - \text{Exp} \end{array}$$

1. Écrire un diagramme de classes correspondant à cette grammaire.
2. Écrire le diagramme d'objets correspondant à l'expression arithmétique `2 * - a`.
3. Écrire les classes Java, ainsi qu'une méthode évaluer permettant d'évaluer une expression arithmétique dans un environnement.
   On rappelle qu'un environnement associe à une variable sa valeur.

Exercice 24. Patron de conception Visiteur
Appliquer la patron Visiteur pour définir l'opération évaluer de l'exercice précédent.

Exercice 25. Conception de la minuterie
Le but de cet exercice est de réaliser la conception du système « Minuterie ». On décide d'utiliser différents patrons de conception : Observateur, Visiteur et État.

1. Montrer comment le patron Observateur peut être utilisé pour gérer les mises à jour de l'interface graphique.
2. Montrer comment les différents états du contrôleur peuvent être représentés à l'aide du patron État.
3. Montrer comment les actions à effectuer en réponse à des clics sur les boutons Incr, Mode et Start/Stop peuvent être représentées en utilisant le patron Visiteur.
4. Préciser le contenu des principales classes.

Exercice 26. Codage des classes-associations
Proposer une implémentation Java pour la classe-association représentée figure VI. 30. Préciser les contraintes qui doivent être respectées.

Exercice 27. Codage des relations n-aires
Proposer une implémentation Java pour la relation ternaire représentée figure VI. 31.

La traduction proposée se comporte-t-elle bien par rapport à l'héritage ?

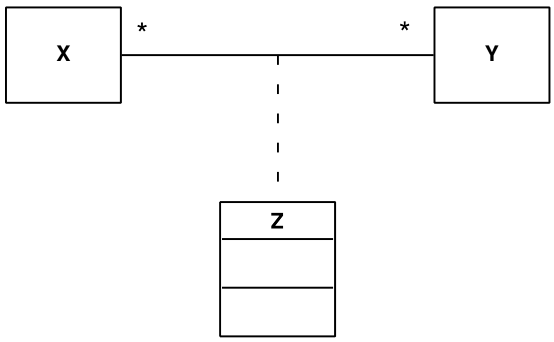


FIG. VI. 30 – Une classe-association

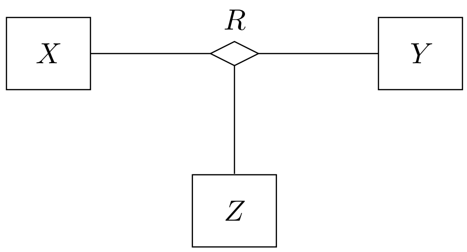


FIG. VI. 31 – Une relation ternaire

Études de cas

Minuterie

On s'intéresse à un système « Minuterie », qui permet de déclencher une alarme après une période de temps spécifiée.

1. Une minuterie peut être pilotée par une interface graphique (cf. figure VI. 32) qui contient :
   - trois compteurs permettant d'afficher les heures, les minutes et les secondes ;
   - quatre boutons : Mode, Incr, Start/Stop, Fermer ;
   - un indicateur d'alarme.

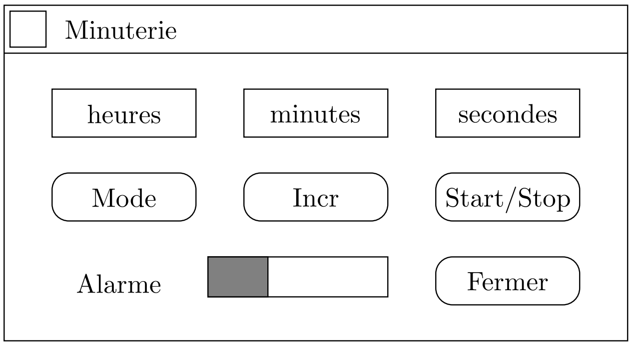


Fig. VI. 32 – Interface de la minuterie

2. Lorsque la minuterie est arrêtée, le bouton Mode permet de passer du mode « normal » au mode « édition heures », puis au mode « édition minutes » et enfin de revenir au mode « normal ».
3. Lorsque la minuterie est dans le mode « édition heures », le bouton Incr permet d'incrémenter ce compteur.
4. Lorsque la minuterie est dans le mode « édition minutes », le bouton Incr permet d'incrémenter ce compteur.
5. Lorsque la minuterie est arrêtée, le bouton Start/Stop permet de la démarrer.
6. Lorsque la minuterie est en marche, le bouton Start/Stop permet de l'arrêter. On se retrouve alors dans le mode « normal ». Lorsque la minuterie est en marche, elle décrémente ses compteurs d'une seconde à chaque seconde. Lorsque la minuterie atteint zéro, l'alarme est activée.

7. L'alarme, lorsqu'elle est active, peut être au niveau 1, 2 ou 3. Une fois activée, l'alarme change de niveau chaque minute, puis est désactivée par le système.
8. Lorsque l'alarme est active, le bouton Start/Stop permet de la désactiver. Après désactivation de l'alarme par l'utilisateur ou par le système, la minuterie est dans le mode « normal ».
9. À tout moment, on peut fermer la fenêtre avec le bouton Fermer.

Logiciel de traitement des ventes

1. Introduction

Ce document est un cahier des charges pour un logiciel de traitement des ventes d'un magasin. Le système doit pouvoir traiter les ventes, gérer le stock, analyser l'activité du magasin, aider à faire l'inventaire et maintenir une liste de clients.

2. Utilisateurs

Plusieurs personnes peuvent utiliser le logiciel. On distingue plusieurs types d'utilisateurs. À chaque type d'utilisateur est associé un ensemble de fonctionnalités du système. Une personne physique peut être liée à plusieurs types d'utilisateurs. On distingue les types d'utilisateurs suivants.

2. 1. Vendeur
Un vendeur doit pouvoir effectuer une vente, demander si un article est en stock, gérer le retour d'une marchandise.

2. 2. Magasinier
Un magasinier doit pouvoir enregistrer dans le système les articles reçus, et réaliser l'inventaire.

2. 3. Gestionnaire
Un gestionnaire doit pouvoir analyser l'activité du magasin.

2. 4. Administrateur
Un administrateur doit pouvoir gérer la liste des utilisateurs.

3. Logins et mots de passe

Une personne est identifiée par un « nom de login » et possède un mot de passe.

3. 1. Connexion au système
La connexion consiste à entrer le login, puis le mot de passe. Le système identifie la personne, vérifie le mot de passe et affiche un menu correspondant aux fonctionnalités du système que la personne peut utiliser. Si le login n'existe pas ou si le mot de passe est incorrect (c'est-à-dire ne correspond pas au mot de passe du login), la connexion échoue.

3. 2. Changement de mot de passe
L'administrateur peut modifier le mot de passe d'un utilisateur. Un utilisateur peut modifier son mot de passe.

- 3. 2. a. Pour modifier un mot de passe, l'administrateur entre le login concerné, puis entre deux fois le nouveau mot de passe. Si les deux entrées sont différentes, le mot de passe n'est pas modifié.
- 3. 2. b. Pour modifier son mot de passe, l'utilisateur entre d'abord l'ancien mot de passe. Si celui-ci n'est pas correct, la modification échoue. Si l'ancien mot de passe est correct, l'utilisateur doit entrer deux fois son nouveau mot de passe. Si les deux entrées sont différentes, le mot de passe n'est pas modifié.

4. Articles

Un article a une désignation (un nom), et une référence.

4.1. Articles neufs et d'occasion
Le magasin vend des articles neufs et des articles d'occasion. Les articles neufs sont identifiés par une référence (référence catalogue) et un numéro de série. Les articles d'occasion sont identifiés par une référence interne au magasin. Les articles d'occasion proviennent de « reprises » : le magasin peut en effet racheter au client un article déjà utilisé.

4.2. Prix
Un article, neuf ou d'occasion, a un prix d'achat (hors taxes) et un prix maximal de vente (prix toutes taxes comprises ou TTC). Un article peut être vendu à un prix inférieur au prix maximal de vente si on y applique une réduction.

4.3. Retour sous garantie
Un article défectueux sous garantie peut être retourné. Il y a alors échange avec le même article si la référence existe encore, avec un article similaire si la référence n'existe plus.

4.4. Articles commandés
Si un article n'est pas en stock, il peut être commandé. Dans ce cas, le client règle un acompte, égal au minimum à dix pour cent du prix de l'article. Le solde est réglé à la livraison de l'article.

5. Client

Le système tient à jour une liste de clients. Un client a un nom, un prénom, une adresse et un numéro de téléphone.

5.1. Création d'une fiche client
Pour les nouveaux clients, une nouvelle fiche client peut être créée.

5.2. Modification d'une fiche client
L'adresse d'un client peut être modifiée.

5.3. Destruction d'une fiche client
Une fiche client peut être détruite, par exemple sur demande d'un client.

5.4. Impression d'étiquettes
La liste client permet au gestionnaire d'imprimer des étiquettes pour enveloppes afin de réaliser périodiquement un envoi de publicités.

6. Vente

6.1. Ligne d'articles

Une ligne d'articles contient un article (avec sa référence et, si l'article est neuf, son numéro de série ; si l'article est d'occasion, la mention « occasion »), une réduction éventuelle, le prix de l'article avant réduction, et le prix de l'article après réduction. En cas de reprise, seul le prix de rachat est indiqué, en négatif.

6.2. Saisie d'un article

6. 2. a. Pour saisir un article, le vendeur peut soit utiliser le lecteur de code barre, qui permet de retrouver la référence et le numéro de série d'un article, soit directement taper la référence de l'article, avec le numéro de série si l'article est neuf.

6. 2. b. Pour chaque article saisi, le caissier entre la réduction à appliquer sur le prix maximal de vente.

6. 2. c. En cas de reprise, le système créé un nouvel article d'occasion, et lui affecte une référence interne au magasin.

6. 3. Prix total
Une fois tous les articles saisis, le vendeur demande l'addition. Le système affiche alors le prix total de la vente devant être réglé par le client.

6. 4. Identification du client
Le caissier entre le nom du client.

6. 4. a. S'il s'agit d'un client répertorié, le système affiche sa fiche, qui peut être modifiée par le vendeur.

6. 4. b. S'il s'agit d'un client non répertorié, le système crée une nouvelle fiche que le vendeur complète.

6. 5. Moyen de paiement
Le vendeur demande au client son moyen de paiement et l'indique au système.

6. 6. Facture
Une facture est imprimée. Cette facture contient les éléments suivants :

6. 6. a. Nom du client

6. 6. b. Date de la vente

6. 6. c. Pour chaque article : désignation et référence. Si l'article est neuf, son numéro de série est indiqué ; s'il est d'occasion, la mention « occasion » est indiquée. La réduction éventuelle, le pris de l'article avant réduction et le prix de l'article après réduction (prix de vente). En cas de reprise, le prix de rachat est indiqué, en négatif.

6. 6. d. Le prix total TTC de la vente (prix payé par le client), qui correspond à la somme des prix de vente des différents articles.

6. 6. e. Le prix hors-taxes de la vente.

7. Gestion du stock et inventaire

Le magasinier s'occupe de la gestion du stock : il doit entrer dans le système les articles qui arrivent. Lorsqu'un article arrive, il faut entrer dans le système sa désignation, sa référence catalogue, son numéro de série et son code barre. Lors de l'inventaire, il s'agit de lister tous les articles disponibles. Cette liste est mise en relation avec l'ensemble des articles effectivement présents dans le magasin. Les articles présents dans le système, mais absents du magasin sont perdus ou volés.

8. Analyse de l'activité

Le gérant doit pouvoir analyser l'activité du magasin. L'analyse s'appuie sur une liste de ventes réalisées entre deux dates spécifiées. Cette liste est chronologique, et par vendeur sur une durée spécifiée. Le chiffre d'affaire et la marge effectuée sont affichés. Après inventaire, on peut connaître les articles volés et le montant correspondant.

Sujets de TP

Ce TP est à réaliser par groupes de trois ou quatre étudiants. Il y a trois sujets au choix.

Sujet 1 : jeu de cartes

Le but de ce TP est de réaliser l'analyse et la conception objet d'un jeu de cartes. Un incrément significatif du jeu sera choisi et implémenté en Java ou C++.

Le but n'est pas d'obtenir un jeu complet, mais d'être capable d'en développer une partie qui pourrait correspondre à une première version à livrer au client.

Sujet 2 : service vocal interactif d'orientation

Ce TP a pour but de de réaliser l'analyse et la conception d'un système d'orientation par la voix. Un incrément significatif de ce système sera réalisé en Java ou C++. Ce système fournit aux automobilistes un service comparable aux systèmes GPS embarqués dans les voitures, à ceci près qu'il est accessible via un simple téléphone portable en kit main libre. Au lieu d'utiliser un positionnement GPS, l'utilisateur tient le système informé de sa progression par la voix.

À titre d'illustration, un scénario d'utilisation possible est représenté figure VI. 33.

Le degré de sophistication des indications données par le système, le niveau de complexité des interactions avec l'utilisateur, la précision du modèle géographique sont laissés libres.

On peut également traiter des situations particulières dans lesquelles le système se comporte différemment. Par exemple, si la distance entre deux carrefours où l'automobiliste doit tourner est inférieure à un certain seuil, le système doit annoncer ces deux carrefours en même temps afin de permettre à l'automobiliste d'anticiper. Ce principe peut être généralisé à n carrefours consécutifs.

On peut également envisager que le système rende des services connexes simples, comme informer l'utilisateur de la distance à parcourir pour se rendre d'un lieu à un autre.

Mise en œuvre
Selon l'incrément choisi (cf. documents à rendre), les échanges entre le système et l'utilisateur peuvent être purement textuels, ou passer par la voix. Dans le second cas, il est possible d'utiliser des bibliothèques Java gratuites qui assurent la synthèse vocale (Text To Speech : par exemple FreeTTS), et/ou la reconnaissance de la parole basée sur une grammaire et un dictionnaire (Speech Recognition : par exemple CMU Sphinx-4). L'utilisation du téléphone portable en kit main libre est alors simulée par l'usage d'un casque-micro sur l'ordinateur. À noter que ces bibliothèques tendent à imposer la langue anglaise.

Système	“Bonjour. Où vous trouvez-vous ?”
Utilisateur	“Ecole d’informatique.”
Système	“Vous êtes à l’école d’informatique.”
Utilisateur	“Oui.”
Système	“Où voulez-vous aller ?”
Utilisateur	“Club de tennis.”
Système	“Vous voulez aller à : club de billard.”
Utilisateur	“Non. Club de tennis.”
Système	“Vous voulez aller à : club de tennis.”
Utilisateur	“Oui.”
Système	“Etes-vous prêt à partir ?”
Utilisateur	“Oui.”
Système	“Prenez la rue à gauche, tournez à droite au deuxième carrefour.”
Utilisateur	“... Ok.”
Système	“Tournez à droite au troisième carrefour.”
Utilisateur	“... Ok.”
Système	“Passez la ligne de tramway, tournez à gauche au troisième carrefour.”
Utilisateur	“... Je suis perdu.”
Système	“Où êtes-vous ?”
Utilisateur	“Restaurant universitaire.”
Système	“Vous êtes à : restaurant universitaire.”
Utilisateur	“Oui.”
Système	“Faites demi-tour, tournez à gauche au premier carrefour.”
Utilisateur	“... Ok.”
Système	“Continuez sur deux cent mètres jusqu’à : club de tennis.”
Utilisateur	“... Ok.”
Système	“Vous êtes arrivé. Au revoir et à bientôt.”

FIG. VI. 33 – Un scénario d’interaction pour le service vocal d’orientation

Sujet 3 : sujet “Amnesty”

Le conseil national de la section française d’Amnesty International a commencé début janvier l’étude d’un projet de site Internet européen. Celui-ci a de multiples objectifs, dont la création d’outils pour l’éducation aux droits humains, pour le recensement des groupes actifs en Europe, et pour le lobbying envers les Etats.

Les acteurs sont nombreux : membres des groupes locaux, membres des commissions d’études, chercheurs sur le terrain, service de documentation, service des relations extérieures, et bien sûr le grand public. Les défis sont également nombreux : parmi les outils on trouve des générateurs de cartes et de frises chronologiques automatiques et la mise en place de bases de données spatio-temporelles.

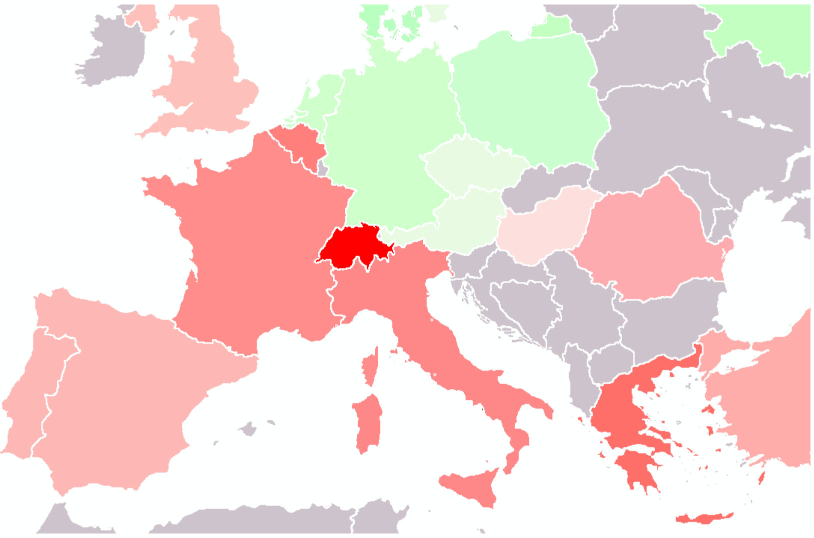


Fig. VI. 34 – L’accès au droit de vote pour les femmes en Europe

Dans un premier temps le travail consistera à modéliser ce qu’est un groupe de personnes dans l’espace et dans le temps et la notion de représentation (par exemple modélisation des groupes d’amnesty mais aussi des parlements des Etats). Il faudra également modéliser ce qu’est une carte, les élements qui la compose, les problèmes de projection. Peut-être aurez vous aussi à modéliser ce qu’est une ontologie, un thème. Il faudra réfléchir à ce qu’est une bibliographie, une bibliothèque de documents, aux problèmes du multilinguisme.

Dans un deuxième temps le travail portera sur la conception de l’architecture logicielle et le choix d’un incrément significatif que vous implémenterez. Il sera présenté au client (fin mai, à Paris), puis déployé sur un serveur web.

À noter qu’un groupe d’étudiants du cours de Management de Projet Appliqué travaillera sur le plan de développement de ce projet jusqu’à la fin mars. Vous pourrez profiter de leur travail parallèle.

Enfin, un prototype plus avancé pourra être poursuivi en projet de spécialité par vous ou d’autres sur la base de votre travail.

Contact pour ce sujet : Mathias Péron (`mathias point peron at imag point fr`)

À rendre

1. Manuel utilisateur détaillé.
2. Document d'analyse.
   (a) Cas d'utilisation du jeu. Chaque cas d'utilisation sera illustré par un ou plusieurs diagrammes de séquence système.
   (b) Modélisation objet du domaine d'application.
3. Document de conception.
   (a) Architecture logique du logiciel.
   (b) Description de l'incrément choisi (sous-ensemble significatif de fonctionnalités). Le reste du document de conception ainsi que la mise en œuvre portent alors sur cet incrément.
   (c) Conception détaillée. Pour l'incrément choisi : diagramme de classes logicielles, diagrammes de séquence et, éventuellement, diagrammes d'états-transitions. Un soin particulier devra être apporté à la cohérence entre ces diagrammes.
4. Programmes Java (ou `C++`) correspondant à l'incrément choisi.
5. Bilan sur les outils de modélisation utilisés, en particulier les problèmes rencontrés, ainsi que les solutions trouvées.

   Il vous est demandé dans cette partie de bien préciser les logiciels, en particulier les modeleurs UML que vous avez utilisés.

Rendre un fichier `.tar` contenant les différents documents, les fichiers sources et un fichier `A_Lire.txt` indiquant comment compiler et exécuter le programme.

Remarque
Il sera tenu compte de la cohérence des documents rendus : cohérence entre les différents diagrammes, cohérence entre les diagrammes et le programme.

Table des figures

Table des matières