Annexe B

Cas Association de Loisirs

Solution 6.1 du chapitre 2

AVERTISSEMENT : compte-tenu de la place limitée dont nous disposons, nous ne traiterons, dans cette solution, qu'une partie du cas. Seule sera étudiée la gestion des membres de l'association, ceci ne comprenant PAS la tenue de leur "compte". Pour éviter, en outre, toute confusion avec une gestion de comptes bancaires, nous employerons dans la suite de ce texte le vocable "situation" en lieu et place de celui de "compte".

a) Une première lecture de ce texte permet de dégager trois grands secteurs d'activité : administration, sections et ateliers. Le premier traite de tout ce qui se rapporte aux assemblées générales, à la gestion des membres et de leurs situations. Le deuxième comprend les activités durables, celles qui ne le sont pas étant regroupées dans le troisième. Nous allons donc découper notre système en trois paquetages que nous étudierons séparément. Nous supposons, en effet, que l'étude est menée par trois équipes indépendantes qui travaillent en parallèle. Une fois les paquetages étudiés, il restera un travail de mise en cohérence à effectuer. Chaque équipe mettre en évidence les interactions avec les autres paquetages qu'elle aura recensé[1], des contrôles devront être mis en œuvre (pour vérifier l'existence dans un paquetage des éléments signalés par les autres), des interfaces définis. Les attributs partagés devront également être publics.

Hypothèse 0.1 (organisation du travail) *Il existe trois équipes qui travaillent en parallèle. Chaque paquetage est pris en charge par une équipe.*

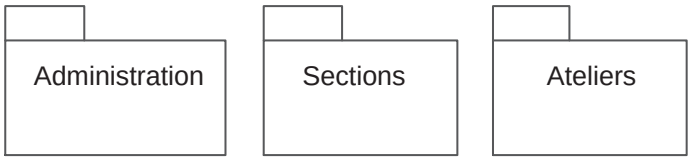


Figure 213 : *Diagramme de classes général - ALEMEC*

Nous mettrons l'accent, dans ce corrigé, sur la cohérence entre les différents diagrammes, introduisant, par là, ce qui sera développé dans le chapitre 4.

L'administration de l'association consiste à gérer ses membres (et notamment, mais pas seulement, son conseil d'administration et son bureau) et leurs situations. Essayons d'y voir un peu plus clair en modélisant son fonctionnement grâce à des cas d'utilisation. Deux parties se distinguent nettement, celle qui permet de gérer les membres et celle qui permet de gérer les situations. Une première description en cas d'utilisation peut donc être obtenue :

[1] Un exemple de ceci peut être trouvé en première partie de page 320.

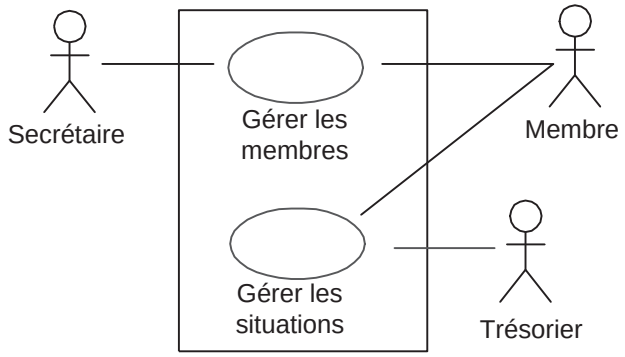


Figure 214 : *Cas d'utilisation général - ALEMEC*

a-1) Gestion des membres La gestion des membres comprend une partie gestion des réunions du conseil, une partie gestion des assemblées générales et une partie gestion des membres eux-mêmes.

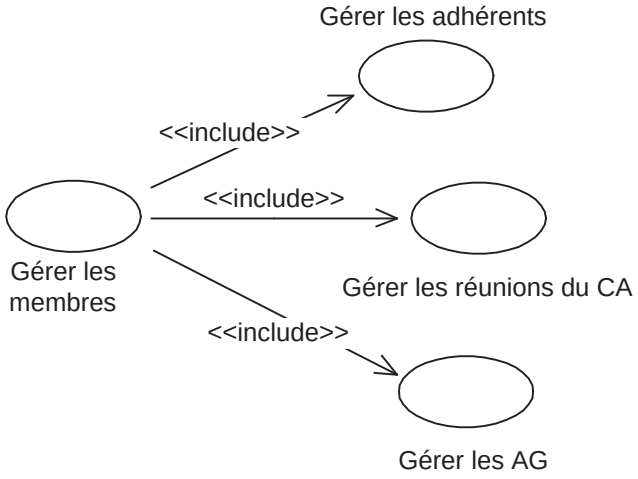


Figure 215 : *Cas d'utilisation Gérer les membres - ALEMEC*

La gestion des adhérents est la partie la plus classique (enregistrer un nouvel adhérent, prendre en compte un changement d'adresse, radier un adhérent... en constituent les composants) et la moins sujette à interactions importantes entre le système et son environnement. A titre d'illustration, et sans que cela épuise le sujet, nous pouvons décrire brièvement le cas comme suit :

Cas d'utilisation : `Gérer les adhérents`
acteurs primaires : `Secrétariat, Membre`
description
La gestion des adhérents comprend l'ajout d'un membre, la définition du bureau de l'association, la démission d'un membre, son départ, l'édition de listes diverses.
cas : `Ajout d'un membre`
précondition : Le membre à ajouter n'existe pas
Saisir les informations sur le membre (nom, prénoms, adresse, date de naissance). Vérifier qu'il n'existe pas et, si tel est le cas, enregistrer les informations sur ce membre.
cas : `Définir le bureau`
précondition : Les membres du bureau sont membres de l'association et du CA
A la suite de l'AG, la composition du bureau est fourni au "système". Pour chaque fonction, la personne choisie est mentionnée. Elle doit obligatoirement être membre du CA. Le choix est effectué pendant une AG. La date d'entrée en fonction du membre est la date de l'assemblée générale durant laquelle le choix a eu lieu.
cas : `Démettre un membre du bureau`
La démission d'un membre du bureau passe d'abord par l'enregistrement d'une demande de démission émanant d'un membre actif de l'association. Une fois cette demande mémorisée, le nombre de demandes similaires est comptabilisé. S'il y a plus de la moitié de ce bureau qui réclame la démission de cette personne, le secrétariat est prévenu : le conseil doit se réunir en urgence. Il (ie. le secrétariat) choisit une date et le système programme une réunion (état "prévue", nature "provoquée", ordre du jour "démission membre"). Les personnes concernées sont les membres du CA. Le délai de convocation est de 8 jours. Appel UC Programmer une réunion
cas : `Editer les listes`
L'édition de listes correspond à la production d'une liste (!!) de personnes appartenant à un groupe donné : liste des membres de l'association, liste des membres du bureau, liste des membres du CA, liste des membres sortants...
...
...
cas : `Départ d'un membre`
Il y a plusieurs cas de départ : un membre peut quitter l'association ou bien seulement quitter la fonction qu'il occupait. Un membre qui quitte la fonction qu'il exerce se signale au secrétariat. Celui-ci fournit l'information au système qui enregistre le départ. La date de signalement est notée comme la date de sortie de la fonction du membre. Le membre qui part "perd" toutes les demandes de démission déposées contre lui. Celles-ci sont détruites. Un membre qui quitte l'association est supprimé (il s'agit d'un marquage, son état prenant la valeur "parti"), les fonctions qu'il occupait deviennent vacantes, les demandes de démission qu'il avait éventuellement sont supprimées (elles, réellement).
exceptions
cas : `Proposer un inconnu`
La proposition de membre du bureau concerne quelqu'un qui n'est pas membre de l'association, ni du conseil d'aministration. La proposition est refusée.
cas : `Ajouter un membre connu`
Le membre qui doit être ajouté existe déjà. L'ajout est refusé.

Dans cette description, nous avons pris en compte certaines hypothèses, qu'il est bon de rappeler ici :

Hypothèse 0.2 (fonctions concernées par une demande de démission) *Une demande concerne un membre du bureau. Si cette personne est également responsable, par exemple, d'une section ou d'une activité, cette deuxième responsabilité n'est pas en jeu.*

Hypothèse 0.3 (portée d'une demande de démission) *Les demandes sont conservées jusqu'au départ du bureau de la personne. A ce moment-là, elles sont supprimées.*

La description textuelle recense cinq cas normaux et deux cas exceptionnels. Les interactions entre le système et les utilisateurs doivent donc être décrites sous la forme de sept

scénarios. Un seul est présenté ici.

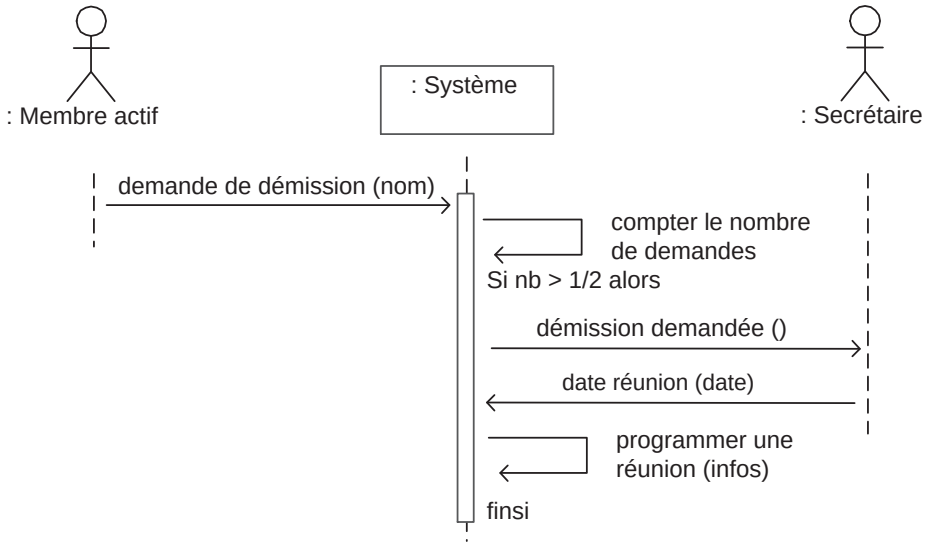


Figure 216 : *Scénario Démettre un membre du bureau - ALEMEC*

Nous pourrions ajouter des cas permettant à un membre de se porter candidat à une élection, de retirer sa candidature... Nous ne développerons pas davantage. Le lecteur intéressé pourra regarder comment nous avons traité un problème similaire dans notre présentation de la solution de l'exercice précédent 5.1 (cf page 146).

Par contre, nous allons détailler les deux autres, gestion des assemblées générales et gestion des réunions du conseil d'administration.

La gestion des assemblées générales comprend la convocation à l'assemblée générale ordinaire -AG- (le cas "standard") ou extraordinaire -AGE- (le cas "anormal") et la gestion des élections. Celles-ci se déroulent en deux temps, vote puis dépouillement. Celui-ci se fait en deux étapes, au cours desquelles sont comptabilisés les votants puis les voix. Nous devons faire, ici, quelques hypothèses supplémentaires :

Hypothèse 0.4 (informatisation des élections) *Les élections proprement dites seront informatisées, ce qui signifie que les votes seront saisis, les voix attribuées aux membres enregistrées.*

Hypothèse 0.5 (organisation de l'AG) *Une AG (ou une AGE) est préparée un mois avant la date fixée pour sa tenue. Elle est programmée en début d'année.*

Hypothèse 0.6 (remplacement des membres lors d'une AGE) *Lors d'une AGE, le remplacement se fait membre par membre : un membre élu prend la place (dans le tiers d'affectation) du membre remplacé.*

Hypothèse 0.7 (procuration) *Il n'y a pas de vote par procuration. Les votes se font à bulletins secrets.*

Hypothèse 0.8 (membres électeurs) *Seuls les membres à jour de leur cotisation peuvent voter.*

Hypothèse 0.9 (égalité) *Si plusieurs membres recueillent, lors du vote, le même nombre de voix, le plus jeune est déclaré élu.*

La structuration de cette gestion en cas comprend une partie "évidente", comprenant les deux cas `Programmer une AG` et `Convoquer une AG` et une partie plus "floue", correspondant au vote. Nous pouvons structurer ce vote en trois cas normaux :

1. `Voter`, de précondition "bureau ouvert" ;

2. `Comptabiliser les votants`, de précondition "bureau ouvert" et de post-condition "bureau fermé" ;
3. `Comptabliser les voix`, de précondition "bureau fermé".

ou bien en un seul (`Voter`), qui englobe les deux autres. Nous choisissons la seconde solution. Nous complèterons la structuration par deux cas exceptionnels (`Convoquer une AGE` et `Reconvoquer une assemblée`).

Cas d'utilisation : `Gérer les AG`	
acteurs primaires :	`Membre`
invariant :	`L'ensemble des membres de l'association ne change pas.`

description

La gestion des assemblées générales comprend la programmation de ces assemblées, la convocation à l'assemblée générale (le cas "standard") ou extraordinaire (le cas "anormal") et la gestion des élections. Cette dernière comprend la gestion des votes, la comptabilisation des votants et des voix, regroupées en un seul cas, `Voter`.

cas : `Programmer une AG`

Tous les ans, en début d'année, la date de l'assemblée générale est choisie. La programmation de cette AG consiste à créer une réunion ayant pour nature la valeur "AG", pour code la valeur "prévue". Son ordre du jour peut être, au moment de la programmation, encore indéfini. Il est fourni par le secrétariat. Tous les membres de l'association sont concernés par cette réunion. Le délai de convocation est d'un mois.

...

...

cas : `Convoquer une AG`

Tous les ans, un mois avant la date choisie, une convocation est envoyée à tous les membres de l'association. Cette convocation comprend l'ordre du jour de l'AG (défini par le président et fourni par le secrétariat), la date de l'AG (fournie par le secrétariat en début d'année) et la liste des membres actifs à remplacer. Cette liste recense les membres élus trois années auparavant (le "tiers sortant"). Il s'agit d'une AG ordinaire.

cas : `Voter`

Les votes ne peuvent avoir lieu que si le bureau est ouvert. Cette ouverture faite, quand un membre de l'association se présente pour voter, il y a vérification de sa situation. Seuls, en effet, les membres à jour de leur cotisation reçoivent les bulletins de vote. Si ce membre n'est pas en règle, il peut payer immédiatement sa cotisation. Il recevra ensuite les bulletins lui permettant de s'exprimer. A la clôture du bureau de vote, le nombre de votants est calculé. S'il y a eu au moins deux tiers de membres à jour de leur cotisation qui ont voté, le vote est déclaré licite. Le nombre de voix attribuées à chaque candidat est comptabilisé. Ceux qui ont recueilli le plus grand nombre de suffrages (majorité absolue) sont déclarés élus membres actifs de l'association. Si deux membres recueillent un nombre de voix identique, le plus jeune des deux est déclaré élu.

exceptions

cas : `Convoquer une AGE`

précondition : Plus du quart des membres actifs à remplacer

Si plus d'un quart des membres actifs fait défaut (par suite de décès, démission ou radiation dans l'année écoulée), alors le président devra convoquer une assemblée générale extraordinaire (AGE) afin de pourvoir à leur remplacement. Le traitement est similaire à celui d'une AG, mais les membres à remplacer sont ceux qui font défaut et non le tiers-sortant. Lors d'une AGE, le remplacement se fait membre par membre : un membre élu prend la place (dans le tiers d'affection) du membre remplacé.

cas : `Reconvoquer une assemblée`

précondition : Moins de $2/3$ des membres à jour ont voté

Le vote est annulé. Une nouvelle AG est programmée, à une nouvelle date, communiquée par le secrétariat. L'ordre du jour, la liste des membres actifs à remplacer sont inchangés. Dans la "foulée", la convocation est envoyée.

Nous avons, dans cette description textuelle, traité différemment les cas "anormaux" : exception dans certaines situations, alternative dans d'autres. C'est volontaire et ne relève

pas seulement de la simple volonté pédagogique de mettre en évidence toutes les possibilités offertes par UML à l'analyste. Nous avons également pris en compte, dans notre choix, la "rareté" d'apparition de chaque situation. Le cas où plus du quart des membres actifs font défaut, celui dans lequel moins des deux tiers des membres s'expriment sont moins fréquents que celui d'un membre non à jour de sa cotisation ou bien encore celui d'une égalité entre deux candidats élus. Ceci est, bien évidemment, discutable.

La suite logique de cette étude des besoins est d'apporter des précisions sur les interactions entre les utilisateurs et le système. Ceci se fait au travers des scénarios. Il y a cinq cas normaux et deux cas exceptionnels. Nous devons donc avoir au moins sept scénarios. Quatre sont présentés ici.

Convoquer une AG ou convoquer une AGE, globalement, c'est le même traitement. Seuls l'événement déclencheur (annuel dans le premier cas, au moins un quart manquant dans le second) et la nature de la convocation adressée aux membres (AG dans le premier cas, AGE dans le second) changent. Nous avons donc deux scénarios (à message près) pour la convocation :

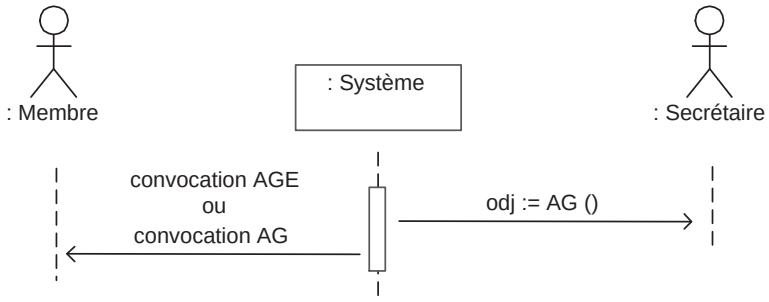


Figure 217 : *Scénarios Convoquer - ALEMEC*

Lors d'un vote, il y a, à la clôture, comptablisation des votants. Selon le résultat de ce comptage, la consultation sera déclarée licite ou non. Il y a donc, là encore, deux scénarios. Le premier traite de la convention licite :

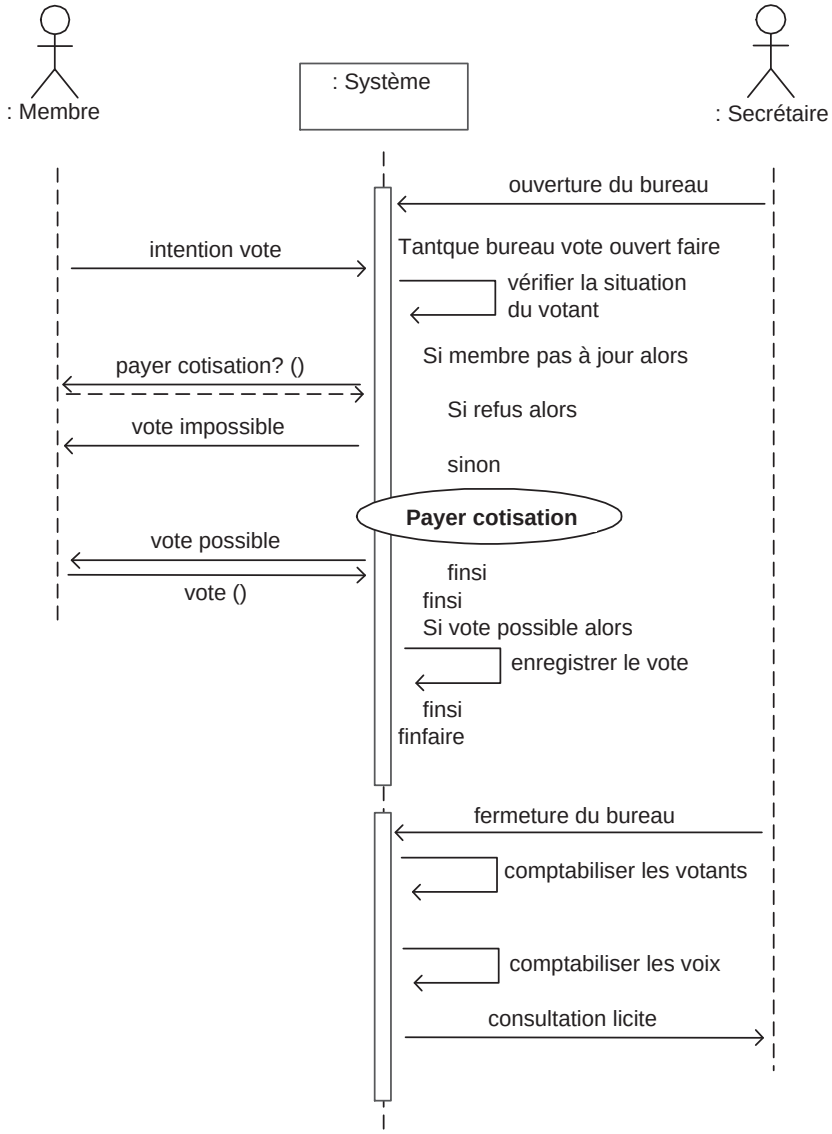


Figure 218 : *Scénarios Voter, consultation licite- ALEMEC*

Le second scénario ne diffère que par la partie "exécutée" à la fermeture du bureau. Celle-ci correspond aux échanges suivants :

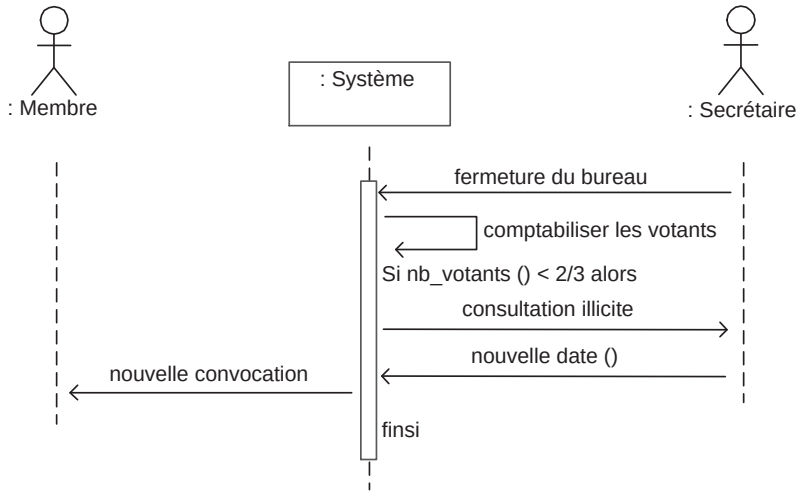


Figure 219 : *Scénarios Voter, consultation illicite - ALEMEC*

La gestion des réunions du conseil d'administration comprend la programmation des réunions (le texte parle clairement d'automatisation). Elle comprend aussi la saisie de la composition du bureau, après le renouvellement du CA, la demande de démission d'un membre du bureau, la démission effective de celui-ci, la convocation à une nouvelle réunion, extraordinaire. Elle comprend enfin la mise en évidence du tiers-sortant et l'édition de la liste des membres de ce bureau. Nous avons également émis une hypothèse :

Hypothèse 0.10 (convocation) *Les réunions du CA sont précédées d'une convocation qui doit être envoyée aux personnes concernées huit jours avant la date de la réunion.*

Cas d'utilisation : `Gérer les réunions du CA`
acteurs primaires : `Secrétariat, Membre`
invariant :
inclut `Programmer une réunion`
description
La gestion des réunions du Conseil d'Administration regroupe les opérations permettant de programmer des réunions mensuelles, d'enregistrer la composition du bureau après une assemblée générale, de gérer la démission d'un membre du bureau à la demande des membres, d'effectuer le travail de convocation à une réunion des personnes devant y assister et, enfin, d'éditer des listes. Il faut y ajouter la prise en compte du départ d'un membre du bureau ou du CA.
cas : `Programmer les réunions`
Chaque année, le secrétariat fournit une liste de dates auxquelles devront se tenir des réunions du conseil d'administration. La "programmation" consiste simplement à appeler l'UC *Programmer une réunion* pour chacune des dates sur la liste. Chaque réunion à programmer a une nature (par exemple "mensuelle"), un état (par exemple "prévue") et un ordre du jour (par exemple "à définir"). Les personnes concernées sont les membres du CA. Le délai est de 8 jours.
cas : `Convoquer à une réunion`
précondition : La réunion a été programmée
Une convocation est adressée à toutes les personnes concernées par la réunion. Celle-ci contient la date de la réunion, l'ordre du jour. La nature de la réunion est également fournie. La convocation est lancée un certain temps avant la date prévue pour la tenue de la réunion, ce laps de temps pouvant varier d'une réunion à une autre.
exceptions
cas : `Convoquer à une réunion non programmée`
La réunion n'ayant pas été programmée, la convocation n'est pas envoyée.

Cas d'utilisation : `Programmer une réunion`
acteurs primaires : `Secrétariat, Membre`
invariant :
description
Programmer une réunion consiste à créér une réunion en lui associant une date, une nature, un ordre du jour et un état. Cela consiste aussi à y associer toutes les personnes concernées. Un délai de convocation est fourni.

a-2) Gestion des situations Il a été décidé, en début d'exercice, de ne pas développer cette partie. Nous ne le ferons donc pas. Nous savons toutefois, par la description du cas `Gérer les assemblées générales` (cf page 315), que cette gestion comporte au moins l'encaissement des cotisations.

a-3) Première version du diagramme de classes De ces modèles et des informations contenues dans l'énoncé, nous pouvons extraire le diagramme de classes suivant :

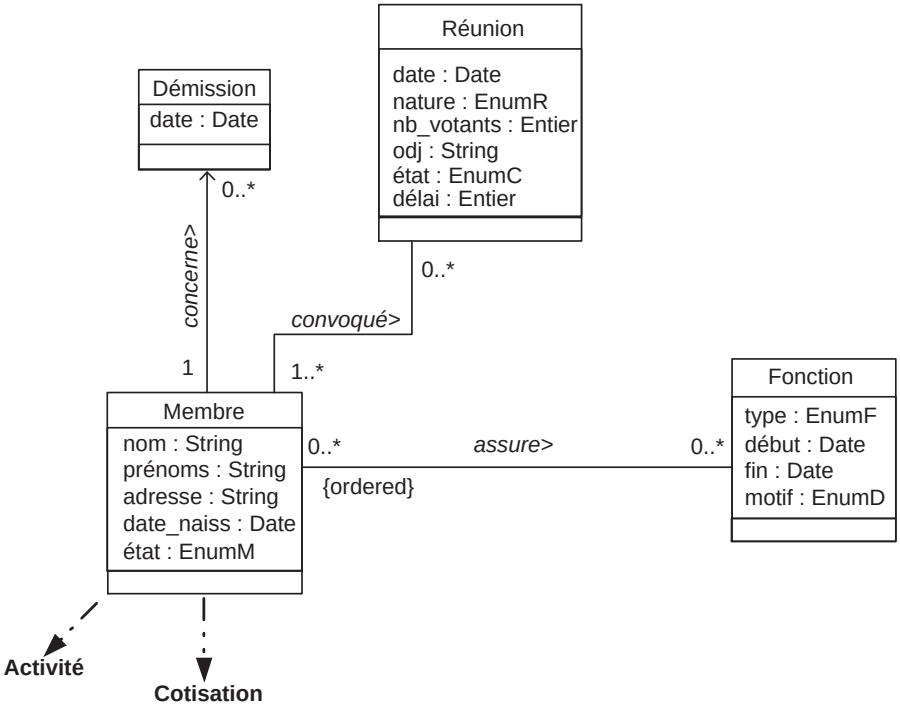


Figure 220 : *Diagramme de classes métier V1 - ALEMEC*

Un membre est décrit par son nom, ses prénoms, son adresse et sa date de naissance. Il lui est attribué un statut (voir les diagrammes états-transitions, page 327, pour connaître les valeurs prises par cet attribut). Un membre peut avoir plusieurs fonctions, mais il ne peut cumuler celles qu'il occupe au sein du bureau. Ceci milite en faveur d'une énumération de ces fonctions[2] :

```
EnumF = enum {président, vice_président, président_honneur, secrétaire,
         secrétaire_adjoint, trésorier, trésorier_adjoint, membre_CA}
```

et d'une mémorisation des dates d'entrée et de sortie d'une fonction. Nous avons également enregistré le motif de départ d'une fonction, à l'aide d'un type `EnumD` qui prend les valeurs suivantes :

```
EnumD = enum {démission, décès, radiation, fin, encours}
```

une fonction actuellement occupée par un membre ayant la valeur `encours` pour l'attribut `motif` et une date de sortie nulle. De façon à accélérer un peu les traitements, les membres ayant des fonctions (ou ayant eu des fonctions) sont classés (en ordre croissant ?) par rapport à leur date d'entrée en fonction.

Nous pouvons ici faire plusieurs remarques concernant les fonctions :

- Les fonctions assurées par les membres peuvent être modélisées sous forme d'une classe `Type_fonction`, ayant autant d'instances qu'il y a de valeurs dans `EnumF`, reliée à la classe `Membre` par des associations. Celles-ci expriment, lorsqu'elles existent, l'implication des membres. Un membre relié à un type de fonction assure actuellement la fonction (ou a assuré un jour la fonction, si nous mettons en place un historique). Ces fonctions peuvent aussi être modélisées sous la forme d'une classe `Fonction`, correspondant à une fonction assurée actuellement par un membre (ou ayant été assurée un jour par un membre, si nous mettons en place un historique). Nous avons choisi la seconde façon de faire.

[2]Celle-ci pourra sans doute être complétée lors de l'étude des autres paquetages et notamment des responsabilités de sections et/ou d'ateliers.

- Pour éviter toute ambiguÔté, et dans la mesure où nous ne traitons que la partie `Administration`, nous avons renommé la classe `Fonction` en `FonctionCA`. D'autres fonctions pourront émerger lors de l'intégration des divers paquetages.
- Nous ne mettrons pas en place un historique des fonctions assurées par les membres, bien qu'elle soit rendue possible par la structure des données mise en place. En d'autres termes, nous pouvons le faire, mais ne le faisons pas pour l'instant.

Les règles de non-cumul sont multiples :

- elles ne portent que sur les sept premières fonctions ;
- une personne ne peut occuper deux fonctions simultanément ;
- elle ne peut pas non plus occuper la même fonction pendant deux intervalles de temps (l'intervalle étant défini comme le temps qui s'écoule entre la date d'entrée en fonction et la date de sortie de celle-ci) se chevauchant ;
- deux personnes, enfin, ne peuvent occuper la même fonction simultanément, sauf s'il s'agit de vice-président (auquel cas il ne peut y en avoir que deux).

Tout ceci doit faire l'objet d'une description formelle en OCL.

Un certain nombre de traitements "sur" les membres sont mentionnés : liste des membres, liste des activités d'un membre, liste des noms des membres assurant actuellement une fonction, état des cotisations... Ils suggèrent une liaison entre la classe `Membre` et les classes `Activité` (qui appartient sans doute aux paquetages `Ateliers` ou `Sections`) et `Cotisation`.

Le système à développer devant, ceci est expressément demandé, automatiser les convocations aux réunions mensuelles du CA, une classe `Réunion` a été ajoutée. Chaque réunion est datée ; elle est d'une nature donnée, pouvant prendre une valeur parmi celles de l'ensemble `EnumR` suivant :

```
EnumR = enum {mensuelle, AG, AGE, provoquée}
```

cette dernière valeur correspondant à une convocation faite pour remplacer un membre du bureau. Un état lui est associé. Il peut prendre sa valeur dans l'ensemble suivant :

```
EnumC = enum {passée, prévue}
```

Un délai de convocation, exprimé en nombre de jours, est associé à chaque réunion. Enfin, à toute réunion est associée une liste de personnes convoquées. Des votes ayant lieu lors des assemblées générales, un attribut `nb_votants` complète la description de cette classe. Cet attribut a une valeur nulle pour toutes les réunions qui ne sont pas des AG ou des AGE.

La prise en compte, par le logiciel, de la démission d'un membre du bureau à la demande d'au moins la moitié des membres actifs nécessite la création d'une classe `Démission`. Celle-ci correspond à la demande déposée par un (quelconque) membre actif. Son auteur n'est pas mémorisé. Seule la date de dépôt de la demande est conservée. La relation `concerne` qui lie `Démission` à `Membre` ne sert que dans le sens `Membre - Démission` pour compter le nombre de demandes déposées. Elle est donc unidirectionnelle.

Nous n'avons, pour l'instant, pas distingué les membres entre eux. Il n'est pas impossible que nous y soyons "contraints" et que nous ayons recours à une spécialisation des membres entre membres ordinaires, membres candidats, membres du CA et membres du bureau. L'avancement de l'étude va nous permettre de prendre position sur ce recours. De la même façon, les réunions sont probablement multiples, certaines nécessitant des votes, d'autres non. Une spécialisation est donc probable, permettant de mettre en évidence des réunions avec votes et d'autres sans.

b) Affiner la solution développée dans la question précédente revient d'abord à prendre les scénarios et à préciser le rôle du système. L'objectif essentiel est de mettre en évidence les classes qui sont impliquées dans les traitements. Nous allons également faire apparaître plusieurs classes, certaines chargées de gérer l'interface (`F. Gestion des Membres`) ou les groupes d'instances (`Ens. Membres`). Nous obtiendrons des diagrammes de séquence.

La convocation des membres de l'association à l'assemblée générale (scénario `Convoquer`) est définie par les scénarios de la figure 217. L'ordre du jour est fourni par le secrétariat ; la convocation est adressée à chaque membre de l'association, à jour ou non de sa cotisation ; la liste des membres à remplacer y est jointe. Une occurrence de la classe `Réunion` a été créée en début d'année. Le diagramme de séquence est donc le suivant :

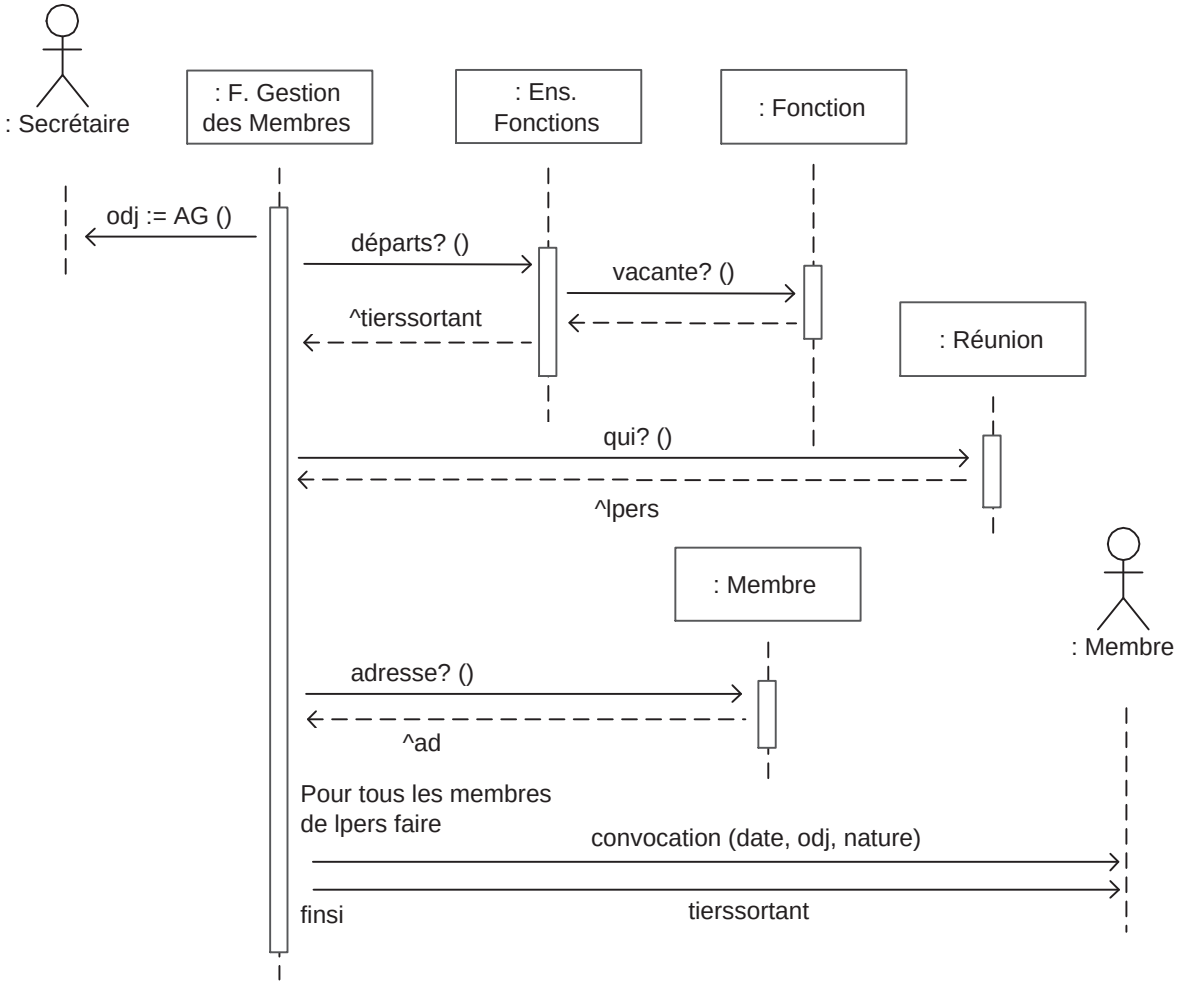


Figure 221 : *Diagramme de séquence Convoquer une AG - ALEMEC*

Pour faire comptabiliser, par le système, les votants (ceci ne se fait qu'après la fermeture du bureau de vote), il faut avoir, auparavant, mémorisé les votes. Ceci n'était pas prévu, ni dans le premier scénario de la figure 217, ni dans le diagramme de classes de la figure 220. Nous allons ajouter dans la classe `Membre`[3] un attribut, `nb_voix`, qui servira à enregistrer le nombre de voix obtenues. Cet attribut ne servira que pendant les élections se déroulant lors d'une assemblée générale (AG). Il est d'une portée très restreinte et a une durée de vie (très) limitée.

Lors du vote, il y a vérification de la situation du membre votant. S'il a payé sa cotisation, son vote peut être comptabilisé. Il y a un votant de plus durant cette AG. Chaque voix est enregistrée. S'il ne l'a pas fait, il est invité à le faire. S'il paye sa cotisation, il peut voter, son vote sera comptabilisé. S'il ne le veut pas, il ne pourra voter. La première partie du scénario `Voter` devient donc :

[3] ou, mieux, dans la classe `Membre candidat` spécialisant `Membre`

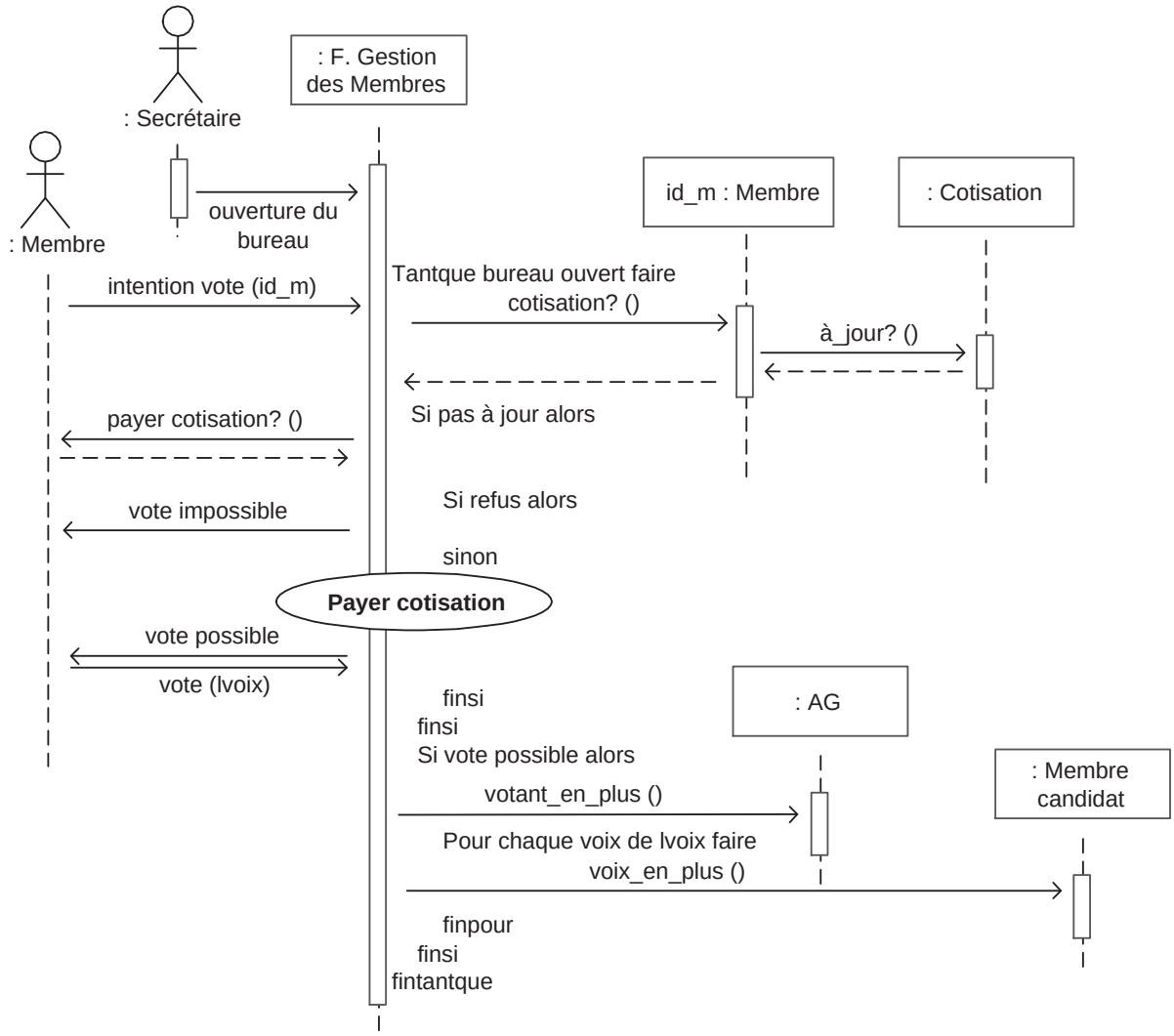
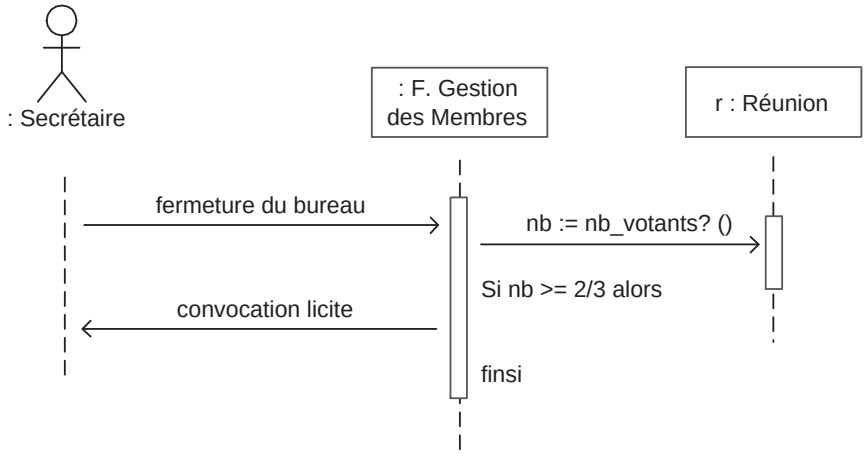


Figure 222 : *Diagramme de séquence Voter, consultation licite - ALEMEC*

Le décompte des votants est alors assez simple, puisque déjà réalisé dans la classe `Réunion`.



Nous venons de redéfinir deux des scénarios précédemment présentés. Ce travail de raffinage, nous allons le continuer, mais en changeant de formalisme.

La comptabilisation des voix (il s'agit d'une partie du scénario `Voter`) se fait en interrogeant `Ens. Membres` pour obtenir la liste des membres candidats, puis en interrogeant chaque instance de la classe `Membre candidat`. Il y a ensuite tri selon le nombre de voix et l'âge.

Tous ceux qui ont obtenus la majorité absolue sont proclamés élus. Tous les élus changent de statut (ils passent de `candidat` à `actif`). Tous les perdants[4] changent également de statut. Ils redeviennent membres "ordinaires". La classe `Ens. Membres` est informée de ces changements de statuts.

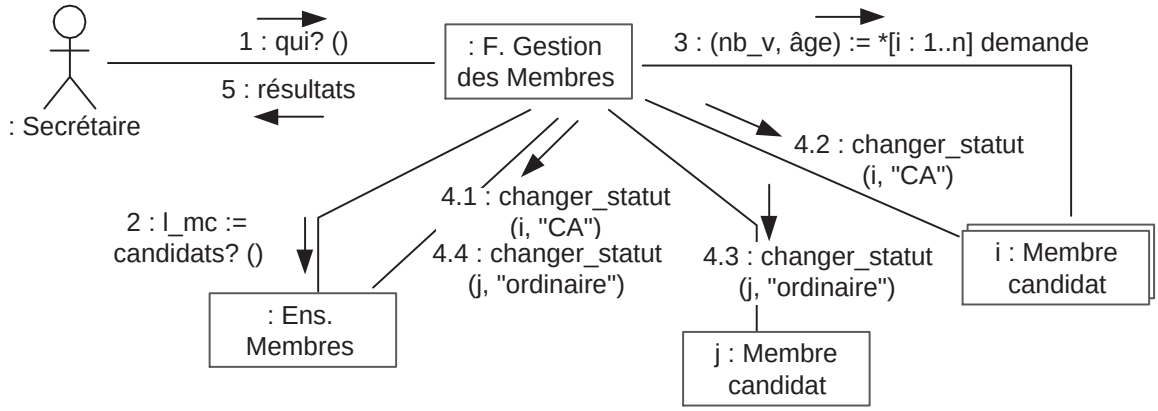


Figure 223 : *Diagramme de collaboration Voter, Comptabiliser les voix - ALEMEC*

NB : nous avons mené cette étude des collaborations en parallèle avec celles de la structure de données (le diagramme de classes) et des diagrammes états-transitions. Cela explique pourquoi certaines références à des transitions apparaissent ici.

La variable `i`, qui sert à exprimer le fait que tous les membres candidats sont consultés, prend ses valeurs dans la liste des membres candidats (`l_mc`) fournie par la classe `Ens. Membres`.

Une fois les membres du CA élus, traditionnellement, ils se réunissent en "conclave" et choisissent en leur sein le bureau. Le résultat de ce "travail" est fourni par le ou la secrétaire, sous la forme d'une liste de personnes associées, chacune, à une fonction. Nous appelons `lcouples` la liste de ces couples (fonction, personne).

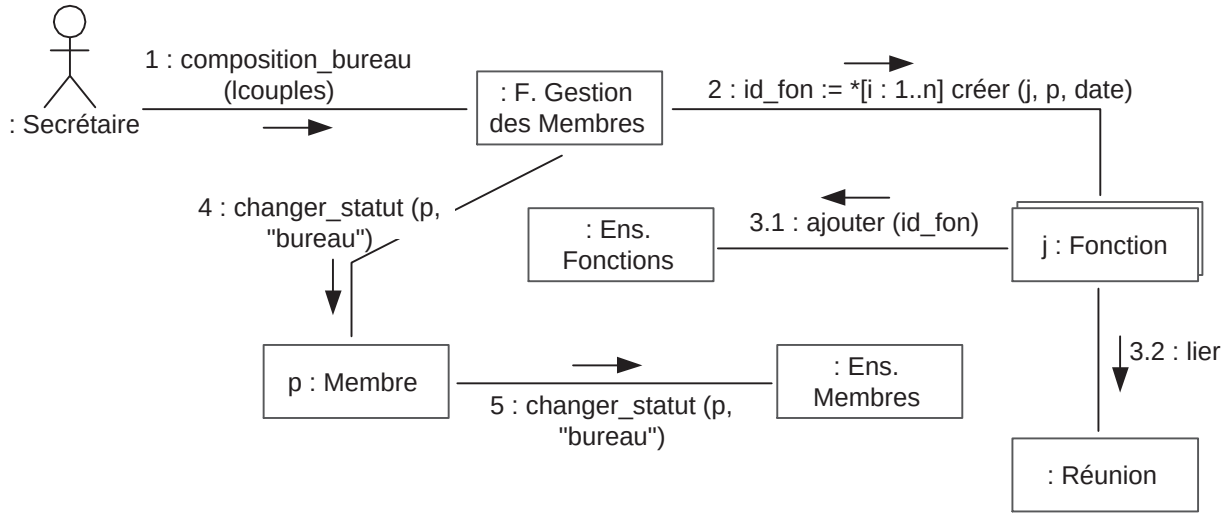


Figure 224 : *Diagramme de collaboration Définir le bureau - ALEMEC*

Dans ce schéma, la variable `i` correspond à un élément de la liste fournie par le secrétariat. `f` correspond au premier élément du couple i, `p` au second. En d'autres termes, `f` est la fonction concernée, `p` la personne qui assure `f`.

Pour chaque couple, il y a création d'une nouvelle fonction, avec pour date d'entrée la date de l'AG -ce qui permet de lier l'entrée en fonction et la réunion durant laquelle cette entrée a été votée. La classe `Ens. Fonctions` est informée de cette création. Le membre `p` change

[4]Ils sont nommés "j" dans le schéma.

de statut et met à jour son "CV" -la liste des fonctions qu'il a occupé et/ou qu'il occupe. La classe `Ens. Membres` est informée de ce changement (elle maintient une liste des membres du CA et une liste des membres du bureau, en plus de celle des membres de l'association).

La démission forcée d'un membre du bureau nécessite une demande d'au moins la moitié des membres actifs. A chaque demande, il y a mémorisation et vérification du membre. Si le nombre de demandes l'impose, il y a programmation d'une réunion pour procéder à son remplacement. Cette réunion est placée à une date fournie par le secrétariat ; elle est de nature `provoquée`, a pour état `prévue` et pour ordre du jour `démission membre`. Le délai de convocation des membres est de 8 jours.

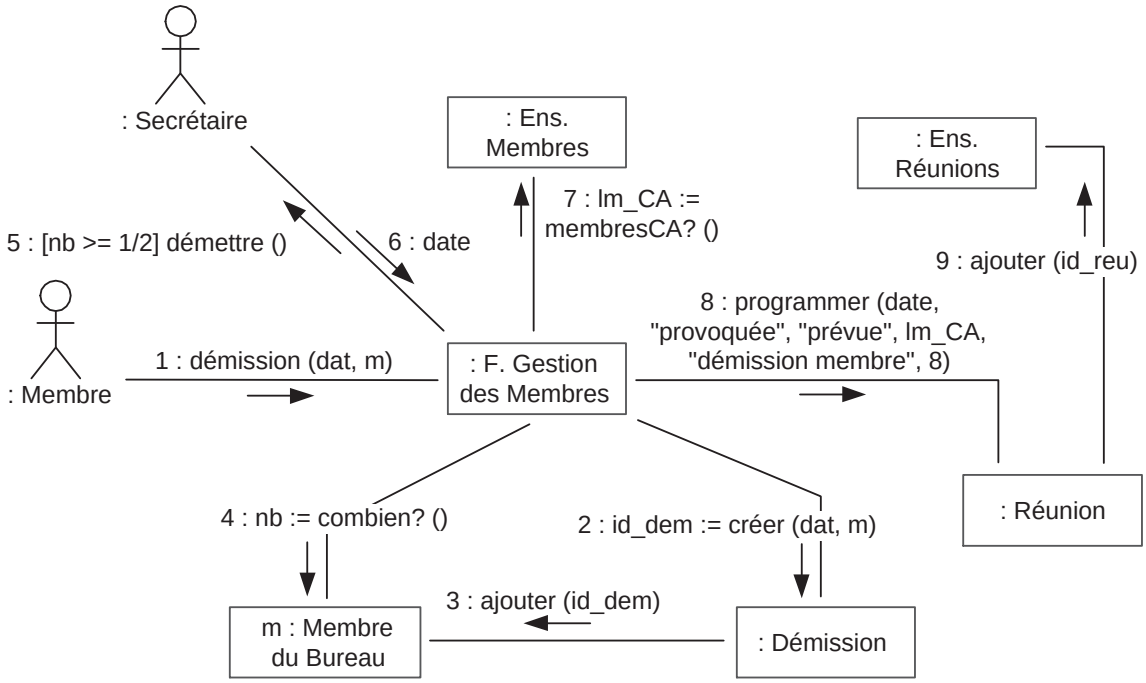


Figure 225 : *Diagramme de collaboration Démettre un membre du bureau - ALEMEC*

La liste des demandes concernant un membre est supposée conservée dans la classe `Membre` elle-même.

La convocation à une réunion mensuelle est toujours précédée de la programmation de celle-ci. Une réunion du CA est une réunion de nature mensuelle, d'état prévue et qui concerne les membres du CA. Son ordre du jour sera fourni par le secrétariat, juste avant la convocation. Il est valué à "à définir" lors de la programmation.

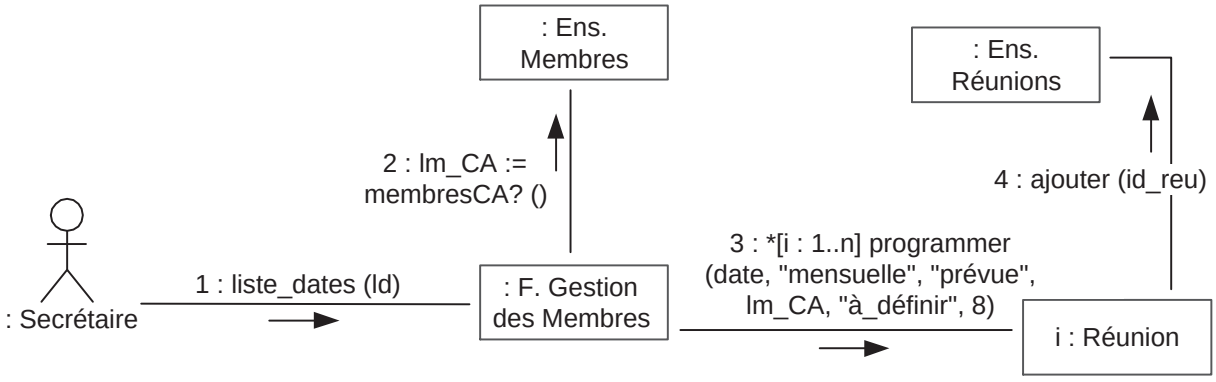


Figure 226 : *Diagramme de collaboration Programmer les réunions du CA - ALEMEC*

La variable `i` prend ses valeurs dans la liste de dates `ld` fournie par le secrétariat.

La convocation proprement dite est déclenchée par la classe `F. Gestion des Membres`. Régulièrement, elle demande à la classe `Ens. Réunions` la liste des réunions prévues. Pour

chacune, elle récupère les renseignements mémorisés et vérifie la date. Si la date de la réunion est égale à la date du jour + le délai, il y a récupération de l'adresse des personnes concernées (auprès de la classe `Membre`), de l'ordre du jour (c'est le secrétariat qui le fournit) et envoi d'une convocation.

La réunion est alors considérée comme `passée` (ie. une réunion passée est une réunion pour laquelle on a envoyé une convocation).

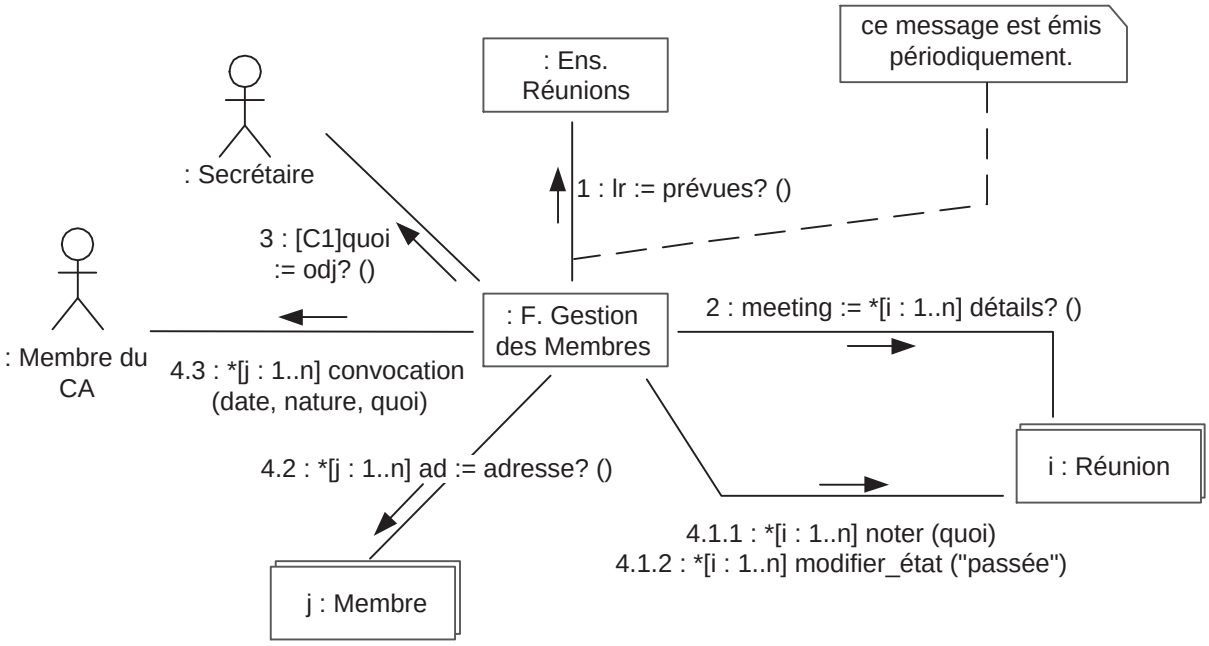


Figure 227 : *Diagramme de collaboration Convoquer à une réunion du CA - ALEMEC*

La variable `i` prend ses valeurs dans la liste `lr` des réunions prévues fournie par la classe `Ens. Réunions` ; `j`, quant à elle, est tirée de la liste des membres convoqués à la réunion, cette liste étant fournie dans `meeting`, avec `date` et `nature`. La condition C1 qui garde la transition `3 : quoi := odj ? ()` permet de faire le "tri" entre toutes les réunions prévues et de sélectionner seulement celles qui sont prévues dans le délai. Elle vaut `date_jour = meeting.date - meeting.délai`.

Ce traitement est déclenché régulièrement (par exemple, à chaque démarrage du logiciel).

c) Le diagramme de classes métier de la première version (voir la figure 220) s'est enrichi au cours de l'étude. Il est maintenant le suivant :

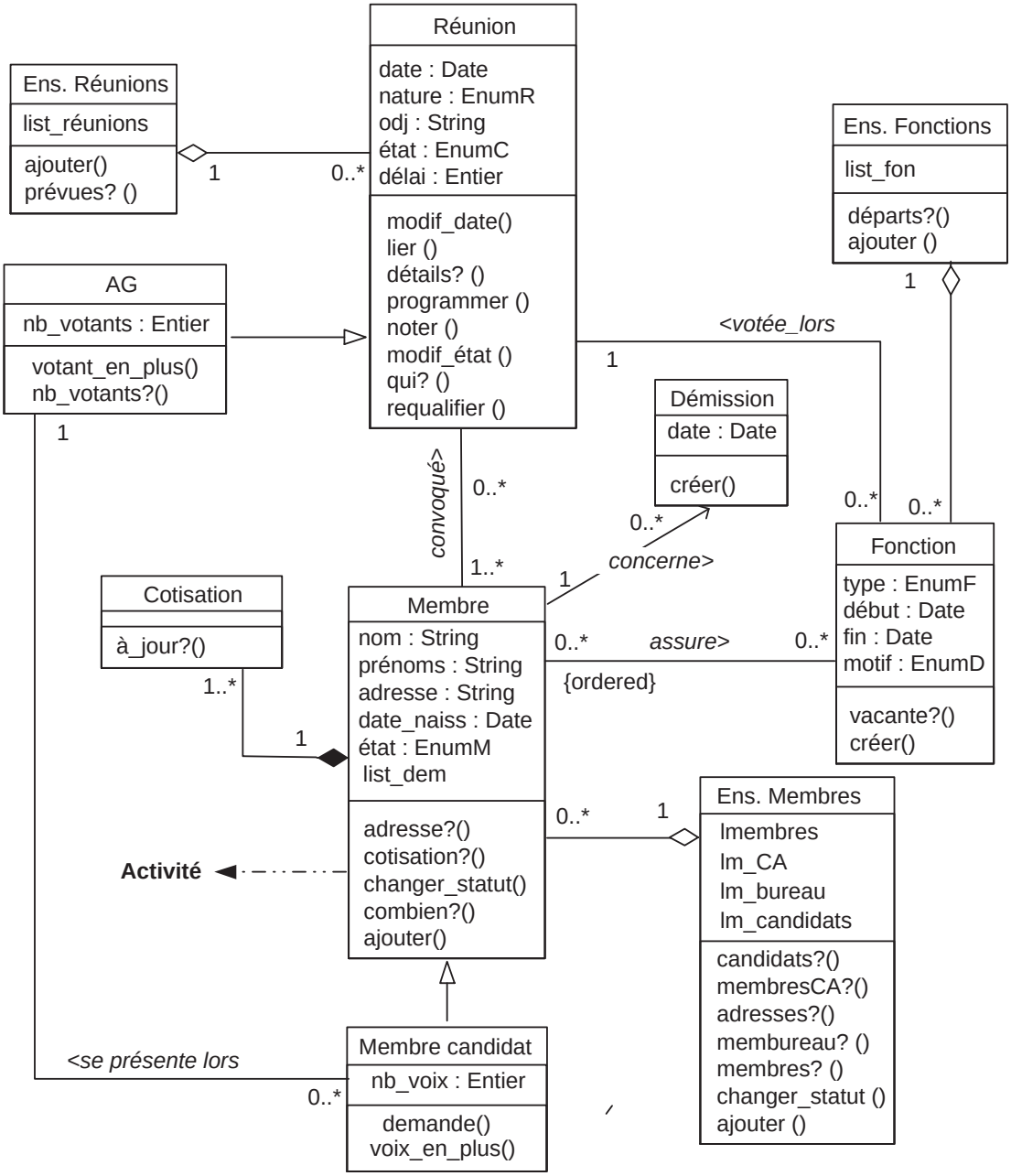


Figure 228 : *Diagramme de classes métier V2 - ALEMEC*

Nous y avons ajouté les spécialisations de `Membre` et de `Réunion`, les classes `Ens. xxx`, la relation entre `Fonction` et `Réunion`, celle entre `AG` et `Membre candidat`, les diverses opérations. Pour ces dernières, nous avons appliqué la "règle" qui veut que les messages soient pris en compte par le receveur et que cette prise en charge se traduise par une méthode, une opération. Chaque classe `Ens. xxx` est traitée de façon uniforme, avec un attribut correspondant à une liste de `xxx` et une méthode `ajouter ()` qui permet d'enregistrer une nouvelle occurrence à cette liste. Il conviendrait d'y adjoindre aussi une opération de retrait. Nous laissons le lecteur le faire.

Un certain nombre de propriétés sont non persistantes. D'autres sont conservées plus longtemps. Il serait bon de mettre en évidence cet état de fait, en introduisant une notation particulière pour un des deux types de données.

La classe `F : Gestion des Membres` peut, elle aussi, être décrite, dans sa partie méthodes du moins :

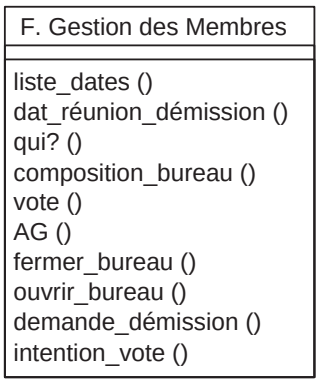


Figure 229 : *Diagramme de classes F.Gestion des Membres - ALEMEC*

De toutes ces classes, celles dont le comportement dynamique est le plus intéressant sont sûrement `Réunion` et `Membre`. Modélisons leurs réactions par un (ou plusieurs) diagramme(s) états-transitions.

La classe `Réunion` dispose d'une variable `état` qui peut prendre deux valeurs, `prévue` et `passée`. Il est possible de décrire son comportement par un diagramme états-transitions :

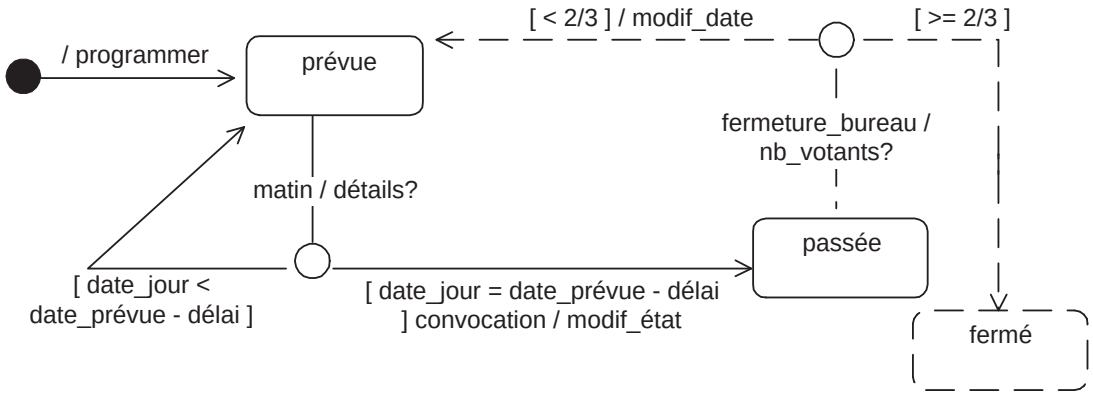


Figure 230 : *Diagramme états-transitions de la classe `Réunion` - ALEMEC*

La partie en pointillés correspond à la sous-classe `AG`. Nous avons là un exemple d'héritage d'automates, cette classe `AG` ayant un automate composé d'une partie propre et d'une partie héritée.

La classe `Membre` "oscille", au premier niveau, entre les états `à jour` et `pas à jour` :

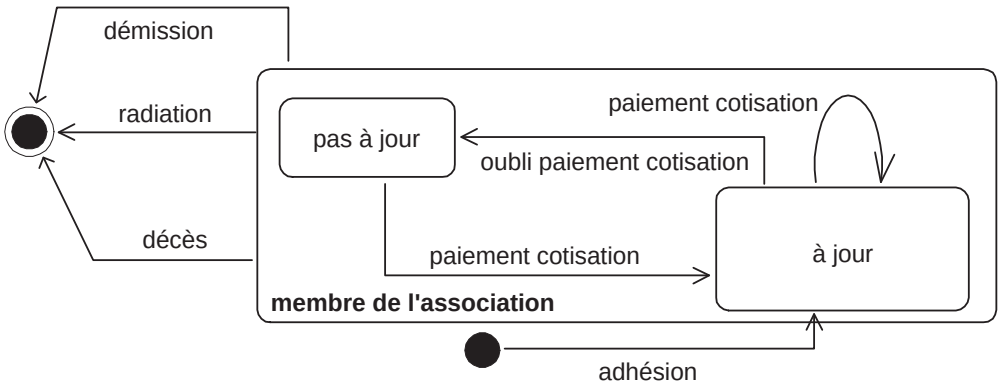


Figure 231 : *Diagramme états-transitions de la classe `Membre` - ALEMEC*

Les différents événements évoqués dans ce diagramme rythment la vie des membres. Il appartient à la personne qui reprendra cette modélisation en l'approfondissant d'associer, le cas échéant, des actions, des opérations, à ces transitions et de penser à modifier le diagramme des classes en conséquence.

L'état `à jour` peut être défini plus précisément : un membre à jour peut être ordinaire, candidat à une fonction[5], actif -il assure une fonction au sein du CA- ou membre du bureau.

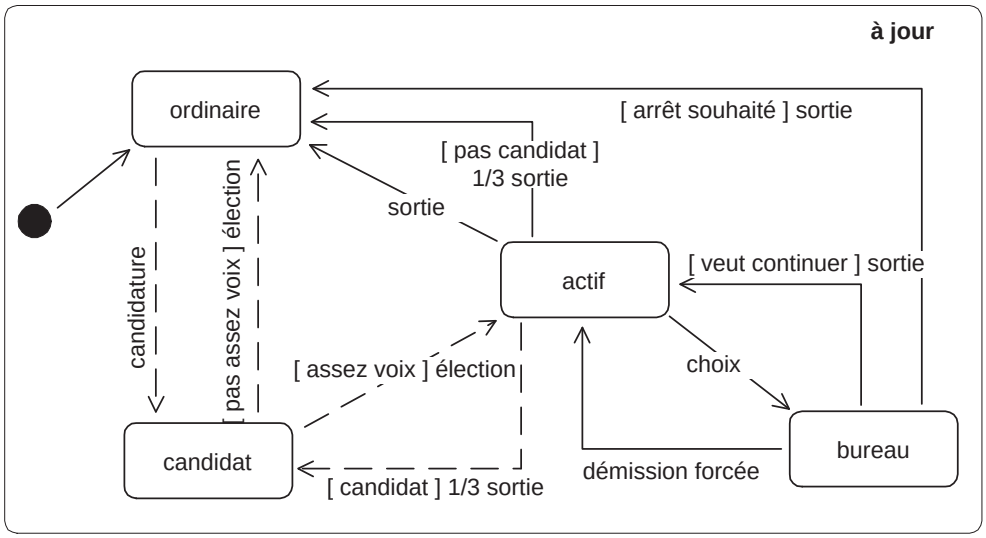


Figure 232 : *Diagramme états-transitions de la classe `Membre`, état `à jour` - ALEMEC*

Comme pour le diagramme états-transitions de la classe `Réunion` (voir la figure 230), la partie en pointillés est spécifique de `Membre candidat`, celle-ci héritant de la super-classe `Membre` tout le reste. Les gardes sont exprimées en clair. Il s'agit davantage de commentaires que de véritables conditions.

Les événements du diagramme sont, le plus souvent, associés à des opérations de la classe. Nous ne les avons pas fait figurer sur le diagramme dans un souci de lisibilité.

EVENEMENT	OPERATION ASSOCIEE
candidature	dépôt_candidature
élection	changer_statut
1/3 sortie	changer_statut
sortie	cessation_fonction
choix	changer_statut
démission forcée	radier_fonction

Sur ces quatre opérations, seule `Changer_statut` est dans la classe. Les trois autres doivent y être ajoutées.

Le diagramme d'activités sert à préciser l'ordre dans lequel les activités sont réalisées au sein d'un état (diagramme d'activité d'un état) ou entre les diverses classes (diagramme d'activité d'un processus). Nous l'avons associé (mais cela n'est pas une pratique conventionnelle) à la classe `F. Gestion des Membres` et obtenu le schéma suivant :

[5] dans le cadre d'une AG

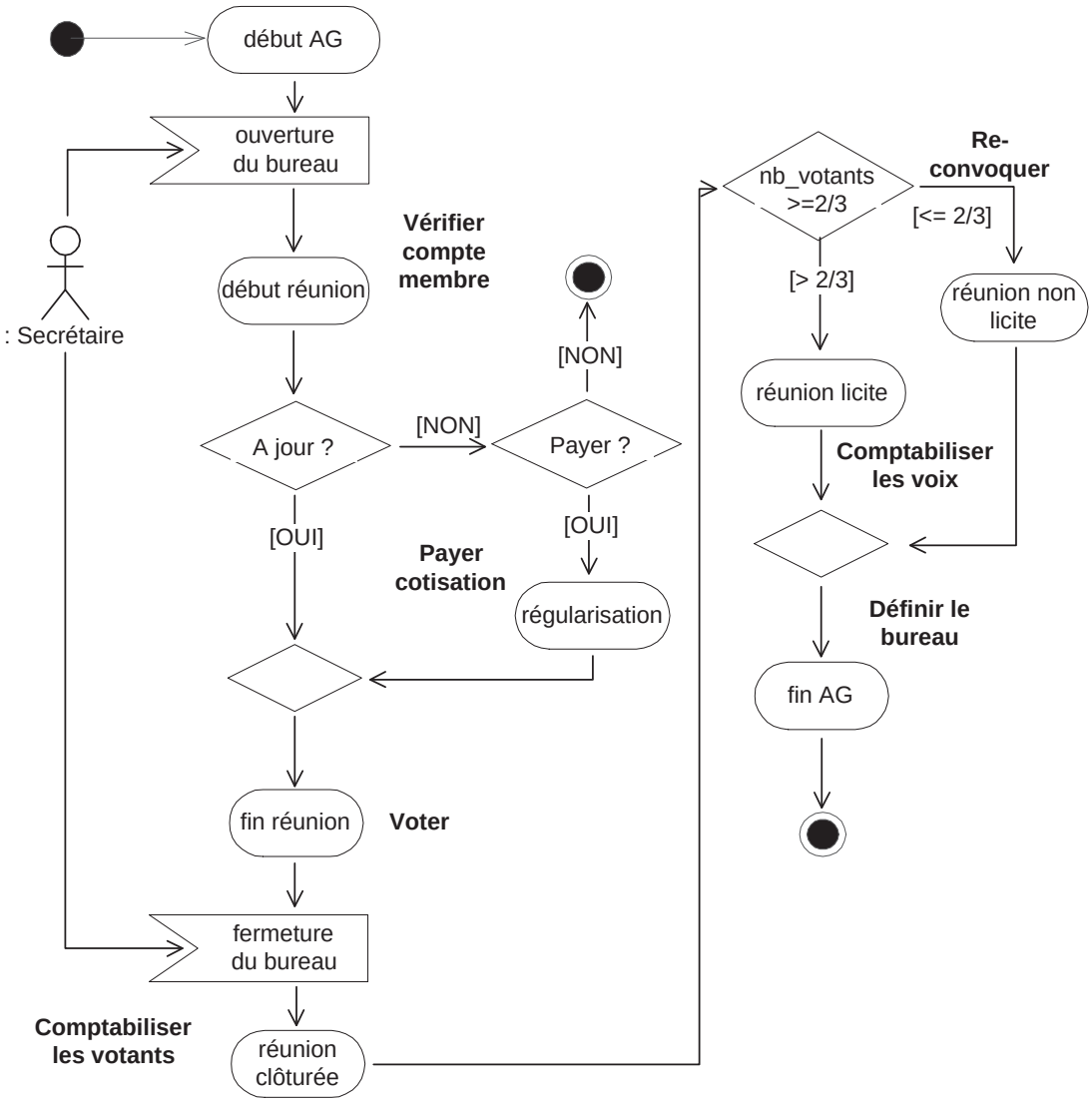


Figure 233 : *Diagramme d'activité explicitant le vote - ALEMEC*

On peut ainsi préciser le "fonctionnement" général de cette classe et associer à chaque activité un scénario ou une partie de scénario (ils apparaissent en gras, à côté des activités).

d) La conception préliminaire définit l'architecture du système. Celle-ci précise l'organisation logique de ses éléments logiciels et matériels. Ceci peut être fait à l'aide de diagrammes de classes, de composants ou de déploiement.

L'énoncé contient peu d'éléments permettant de situer les différents moyens de calcul. La constitution d'un diagramme de déploiement n'est donc pas aisée. Au tout début du texte décrivant la solution (voir la figure 213), nous avons mis en évidence trois paquetages, correspondant à l'administration, aux sections et aux ateliers. Le texte de l'énoncé, d'un autre côté, précise plusieurs choses et notamment que la trésorerie des sections et ateliers n'est pas à informatiser. Il précise également que la trésorerie de l'association elle-même est effectuée par le trésorier, à l'aide d'un *tableur standard du marché*. Les activités des sections et ateliers fait l'objet d'un consignement ... électronique. Il y a, enfin, une dernière activité informatique, celle de gestion des réunions, convocations... Le diagramme de déploiement suivant peut donc être mis en avant (nous avons envisagé par hypothèse que le travail du trésorier se faisait sur un poste séparé de celui du secrétariat) :

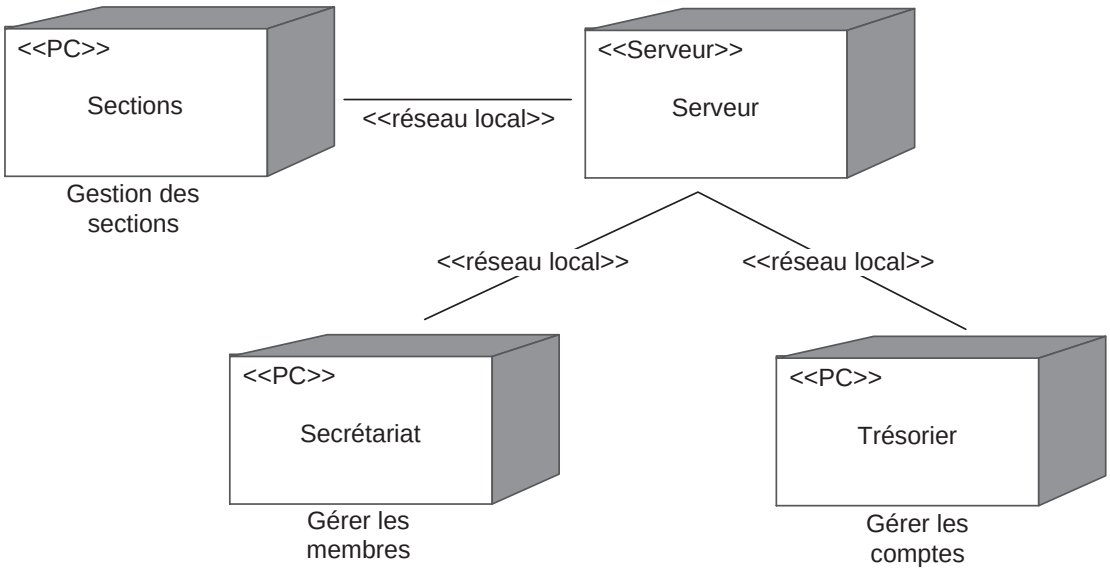


Figure 234 : *Diagramme de déploiement - ALEMEC*

Si l'on prend en compte le fait que, traditionnellement, les associations n'ont pas trop d'argent en caisse, cette architecture "riche", avec 3 postes de travail, peut être réduite d'une unité, la gestion des sections et ateliers pouvant très bien se faire sur le poste du secrétariat... voire même de deux, tout étant regroupé sur une seule machine.

Toute l'application n'ayant pas été étudiée, la mise en évidence des différents composants ne peut être que partielle. Nous avons toutefois déjà pris quelques décisions et pouvons donc dès maintenant en proposer une première ébauche :

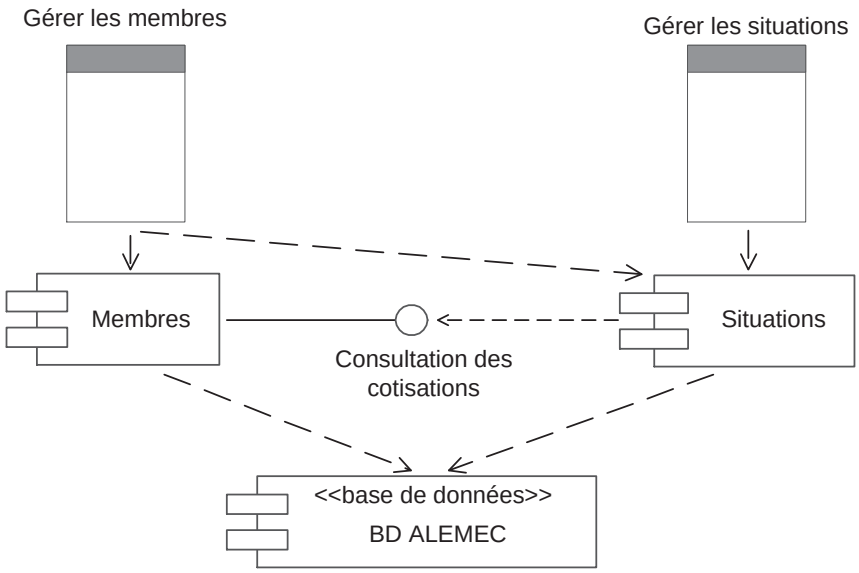


Figure 235 : *Diagramme de composants V1 - ALEMEC*

Il manque à ce diagramme toute la partie se rattachant à la gestion des sections et ateliers. Nous laissons au lecteur le soin de compléter l'étude.

Cette solution est, nous en sommes parfaitement conscients, incomplète. Nous avons essayé de montrer comment, à partir d'un texte, les besoins pouvaient, petit à petit, être mis en forme. Les limites de toute sorte (de temps, de place...) qui ont pesé sur nous lors de la rédaction de cette solution ont fait que nous n'avons pas été au bout de la modélisation. Nous avons un autre regret, celui de ne pas avoir pu mieux prendre en compte certains aspects de la solution qui suggèrent une "factorisation". Quand on regarde les programmations de réunions (qu'il s'agisse de réunions mensuelles, d'AG, d'AGE...), les convocations (à une réunion du bureau, à l'AG...), il y a des similitudes évidentes. Elles doivent pouvoir être modélisées.

Annexe C

Conception et programmation d'une compétition de tennis

Le cas d'étude est la gestion d'un match de tennis. Cet exemple est librement inspiré de [Ner90].

1 Enoncé informel

On souhaite écrire un programme qui simule une partie de tennis entre deux joueurs, selon les règles classiques du tennnis (inspiré librement de [Ner90]. Un match se joue en 3 sets gagnants (soit au maximum 3 ou 5 jeux).

Un joueur gagne un set s'il a au moins 6 jeux gagnés et deux jeux d'écart avec son adversaire. Si le score est de 6 jeux à 5 alors plusieurs cas sont possibles : soit le premier joueur gagne le jeu et il remporte le set 7-5, soit le second joueur gagne le jeu et un *tie-break* est joué, le gagnant du *tie-break* remporte le set par le score de 7 à 6.

Chaque jeu se joue en 4 points maximum pour le gagnant et deux points d'écart avec son adversaire. les points sont notés 0, 15, 30, 40, jeu. En cas d'égalité 40-40, les points sont notés : égalité, avantage au serveur ou avantage au receveur.

Le *tie-break* se joue en 7 points minimum pour le gagnant et deux points d'écart avec son adversaire (par exemple, 7-4, 9-7...).

C'est le même joueur qui sert pour tout un set, hormis le *tie-break* pour lequel, le serveur du set sert une fois puis les deux joueurs servent alternativement Le joueur qui sert en premier est choisi au hasard. On ne tiendra pas compte des fautes ou des deux balles de service. Pour chaque balle engagée, le point sera simplement gagnant ou perdant.

On ne tient pas compte pour l'instant des impondérables : blessures des joueurs, intempéries, abandons.

On souhaite pouvoir adapter le programme aux évolutions suivantes.

1. Dans un tournoi certains matchs se jouent en deux sets gagnants (début du tournoi, tournoi féminin...).
2. On souhaite inclure ce programme dans le cadre d'un tournoi complet et conserver tous les scores jusqu'à la finale.
3. On souhaite modifier le programme pour des matchs de doubles.
4. On souhaite écrire un programme similaire pour un tournoi de tennis de table.
5. Un joueur peut abandonner la partie.

2 Conception abstraite à objets

La lecture de l'énoncé fait apparaître quatre abstractions distinctes : les joueurs, les matchs, les sets et les jeux. En fait, on remarque une imbrication des différents objets entre eux : les joueurs d'un set sont ceux du matchs du set.

- Joueur. Un joueur est simplement caractérisé par un nom, un prénom et son classement.
- Match. Un match est une partie entre deux joueurs. Le match est défini par un ensemble de sets (une liste si on souhaite mémoriser la progression). Le match est terminé dès qu'un joueur a remporté le nombre de sets gagnants.
- Un set est une partie d'un match. Un set est défini par un ensemble de jeux (une liste si on souhaite mémoriser la progression). O souhaite ici ne mémoriser que le score de chaque jeu du set. Le set est terminé lorsqu'un joueur a remporté six jeux avec deux jeux d'écart ou un tie-break.
- Un jeu met en compétition deux joueurs, le serveur et le receveur. Le jeu se joue en 4 points minimum. Le vainqueur a au moins 4 points avec deux points d'écart. les points sont notés selon les convention 0, 15, 30, 40 et la règle de l'égalité ou de l'avantage. Un jeu gagné par le receveur est un avantage psychologique, appelé *break*.

Nous proposons dans la figure 236 une conception abstraite du problème. Nous avons adopté une notation à la Smalltalk des variables et associations pour faciliter l'implantation. La navigation des associations indique une orientation non symétrique des liens entre objets. Nous proposons, comme pour le programme structuré, des constantes pour paramétrer le programme.

Nous avons mis en évidence un schéma d'abstraction : match, set, jeu et tie-break sont des parties entre adversaires. Il y a `NbJoueurs` adversaires (variable de classe). Pour simplifier, concerne deux joueurs (opération `adversaires`) dont l'un doit servir (opération `serveur`). Ces joueurs sont indexés (à partir du joueur, on trouve son score). Le receveur est l'adversaire qui ne sert pas (opération `receveur`). Le principe est le suivant, la partie est une suite de coups (opération `jouerUnePartie`). Chaque coup est gagné par un joueur qui augmente son score (opération `jouerUnCoup`). La fin de la partie est déterminée dans la mise-à-jour de la partie (opération `majPartie`) et stockée dans la variable `fin` (opération `fin`). Elle est fonction de nombre de points à atteindre (variable de classe `NbPtsMax`). Cette variable évoluant dans les sous-classe, elle sera implantée comme une variable d'instance de la métaclasse en Smalltalk. Un message est affiché en début de partie (opération `afficherMsg`), à chaque coup (opération `afficherScore`) et à la fin de la partie (opération `afficherGagnant`).

```
context Partie
inv:
   self.adversaires->size = Partie.nbJoueurs
   -- nous utilisons une notation propre pour accéder à la variable de classe
   self.adversaires->includesAll(self.serveur)
   -- le serveur est un des adversaires
   self.adversaires->forAll(j : JoueurTen | self.score[j] <= Partie.nbPtsMax)
   -- le score est limité, s'agissant d'une collection indexée,
   -- nous utilisons la notation des associations qualifiées
```

Cette structure est ensuite redéfinie au besoin dans les sous-classes. Nous avons ajouté un niveau d'abstraction pour marquer les éléments communs entre jeu et tie-break. Jouer un coup signifie jouer un points. L'affichage des scores, l'évolution de la partie et le nombre de points diffèrent entre jeu et tie-break. L'affichage des points des jeux ordinaires est noté 0, 15, 30, 40, égalité, avantage (opération `valPoints`). Le changement de service intervient dans le tie-break (opération `majPartie`). Les messages sont personnalisés.

Un set est une partie dont les coups sont des jeux. On ne mémorise dans le score que les résultats des jeux. Le set s'arrête si le nombre de jeux gagnants est atteint et qu'il y a un écart

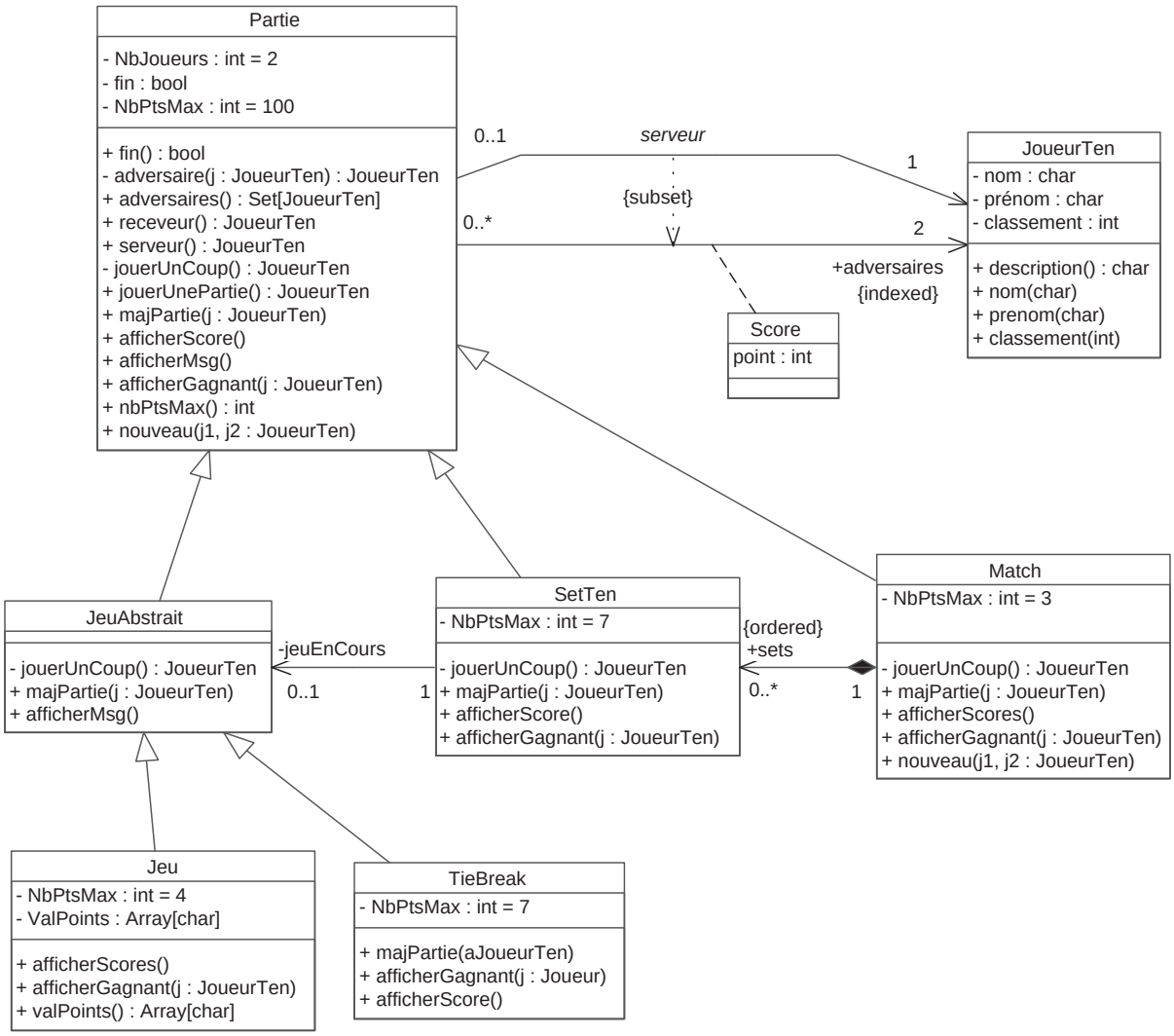


Figure 236 : *Conception objet du programme* ***tennis***

de deux jeux. Un tie break est joué si le score est de 6-6. Le changement de service intervient à chaque jeu (opération `majPartie`). Les affichages de scores varient aussi.

Un match est une partie dont les coups sont des sets. On ne mémorise dans l'ordre les résultats des sets (association `sets`). Le match s'arrête si le nombre de sets gagnants est atteint. Le changement de service est fonction du set précédent. Au départ, le serveur est tiré au sort (opération `nouveau`). Les affichages de scores varient aussi.

Les contraintes suivantes sont ajoutées aux contraintes de la classe `Partie`. Les joueurs d'un set sont les joueurs du match du set. Le nombre maximum de sets d'un match est de (Match.nbPtsMax * 2) -1 par exemple 5 avec 3 sets gagnants. Si le match est fini, le vainqueur a exactement nbPtsMax sets gagnants.

```
context Match inv:
   self.sets->forAll(s : SetTen | s.adversaires = self.adversaires)
   self.sets->size <= (Match.nbPtsMax * 2) -1
   -- nbPtsMax est le nombre de sets gagnants d'un match (var. de classe)
   self.fin() implies
      self.sets.adversaires->exists(j : JoueurTen |
         self.score[j] = Match.nbPtsMax and
        (self.sets.adversaires->forAll(i : JoueurTen |
```

```
            i <> j and self.score[i] < Match.nbPtsMax))
   -- on vérifie l'unicité
```

Pour le set en cours, les adversaires du jeu en cours sont ceux du set. Si le jeu est fini, alors au moins un des scores est supérieur à `nbPtsMax - 1` (en principe le set se joue en 6 jeux).

```
context SetTen inv:
   self.jeuEnCours.adversaires = self.adversaires
   self.fin() implies
      self.sets.adversaires->exists(j : JoueurTen |
         self.score[j] >= SetTen.nbPtsMax - 1 and
         (self.sets.adversaires->forAll(i : JoueurTen |
             i <> j and  ((self.score[i] <= self.score[j] - 2) or
                (self.score[i] = SetTen.nbPtsMax - 1 and
                 self.score[j] = SetTen.nbPtsMax)))))
   -- on vérifie l'unicité avec un éventuel tie-break
```

Dans la classe `JeuAbstrait`, nous avons la contrainte suivantre : si le jeu ou le tue-break est fini, alors au moins un des scores est supérieur ou égal à `nbPtsMax`.

```
context SetTen inv:
   self.fin() implies
      self.sets.adversaires->exists(j : JoueurTen |
          self.score[j] >= SetTen.nbPtsMax and
         (self.sets.adversaires-> forAll(i : JoueurTen |
             i <> j and (self.score[i] <= self.score[j] - 2)))
```

3 Implantation en Smalltalk

Implanter la conception à objets en Smalltalk. Discuter des alternatives de codage.

Le code ci-après est une traduction quasi-systématique de la conception. Les assertions n'ont pas été programmées sous cette forme, mais on peut vérifier qu'elles sont valides. Un certain nombre d'optimisations dues à l'héritage ont été effectuées.

Les associations sont représentées par des variables d'instances, sauf `jeuEnCours` car c'est une variable locale. Du fait des choix de navigation, une seule variable suffit. Elle porte le nom du rôle associé et se trouve dans la classe à l'origine de la navigation. La contrainte `indexed` est représentée en utilisant les dictionnaires de Smalltalk :
`score : Dictionary [JoueurTen] of Integer`.

L'héritage est simple et ne pose pas de difficultés. Noter que la variable de classe `NbPtsMax` est définie par une variable d'instance de ma métaclasse car sa valeur est redéfinie par la méthode de classe `initialize` dans les sous-classes de `Partie`.

3.1 Code Smalltalk de l'application tennis

```
Object subclass: #JoueurTen
   instanceVariableNames: 'nom prenom classement '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Tennis'!
JoueurTen comment:
'Cette classe implante la classe JoueurTen du match de tennis.

Les variables d''instance sont :
nom : String
prenom : String
```

```
classement : Number'!

!JoueurTen methodsFor: 'printing'!

description
   "Ecrit le nom, le prénom et le classement dans le flot."

   | aStream |
   aStream := WriteStream on: (String new: 16).
   self description: aStream.
   ^aStream contents!

description: aStream
   "Ecrit le nom et le prénom dans le flot."

   aStream cr; nextPutAll: 'Je m''appelle ' , nom , ' ' , prenom.
   aStream nextPutAll: ' et je suis classé ' , classement printString!

printOn: aStream
   "Ecrit le nom et le prénom dans le flot."

   aStream nextPutAll: nom , ' ' , prenom! !

!JoueurTen methodsFor: 'accessing'!

classement
   "rend le classement du JoueurTen"

   ^classement!

classement: aNumber
   "affecte le classement du joueur"

   classement := aNumber!

nom
   "rend le nom du joueur"

   ^nom!

nom: aString
   "affecte le nom du joueur"

   nom := aString!

prenom
   "rend le prenom du joueur"

   ^prenom!

prenom: aString
   "affecte le prenom du joueur"

   prenom := aString! !

"******************************** "!
Object subclass: #Partie
   instanceVariableNames: 'score fin serveur '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Tennis'!
Partie comment:
'Cette classe implante une version abstraite d''une partie du match de tennis.
```

```
Elle rassemble le comportement commun à un set, un jeu et un tie break.
La partie est définie par des adversaires dont on mémorise les scores
dans un dictionnaire, l''adversaire qui sert pour le gain de la partie
(serveur) un booléen indiquant si la partie est terminée ou pas.

Les variables d''instance sont :
     score : Dictionary [JoueurTen] : Integer
     serveur : JoueurTen
     fini : Boolean

 - la variable score mémorise le score actuel de chaque adversaire du
   jeu dans un dictionnaire
 - la variable fini mémorise la fin du jeu. Sa valeur pourrait être
   calculée à partir du score, mais nous utilisons une variable pour
   optimiser le temps de traitement.
 - la variable serveur indique le joueur qui sert pour ce jeu.

Les variables d''instance de la métaclasse sont :
   NbPtsMax : Integer  nombre de coups nécessaires pour gagner la partie.'!

!Partie methodsFor: 'initialize-release'!

initializeJ1: aJoueurTen1 j2: aJoueurTen2
   "Démarre un nouveau jeu, les scores sont mis à zéro pour chaque
    joueur. Par défaut, le serveur est le premier joueur."

   fin := false.
   score := Dictionary withKeysAndValues: (Array
               with: aJoueurTen1
               with: 0
               with: aJoueurTen2
               with: 0).
   serveur := aJoueurTen1.
   ^self! !

!Partie methodsFor: 'printing'!

afficherGagnant: aJoueurTen
   Transcript cr; show: aJoueurTen printString , ' gagne'!

afficherMsg
   "Affichage d''un message. Version abstraite."!

afficherScores
   "Affichage du score de la partie. Version abstraite."

   Transcript show: (score at: self serveur) printString , ' - ' ,
        (score at: self receveur) printString! !

!Partie methodsFor: 'update'!

jouerUnCoup
   "On tire le gagnant au sort.
   Si le gagnant est 1 alors le serveur gagne, sinon le receveur gagne."

   | gagnant sort |
   sort := (Random new next * 10000 rem: 2) truncated + 1.
   sort = 1
      ifTrue: [gagnant := self serveur]
      ifFalse: [gagnant := self receveur].
   score at: gagnant put: (score at: gagnant)
         + 1.
   ^gagnant!
```

```
jouerUnePartie
   "On joue les coups jusqu'à ce que la partie soit terminée."
   "On rend le gagnant"

   | gagn |
   self afficherMsg.
   [self fin]
      whileFalse:
         [gagn := self jouerUnCoup.
         self afficherScores.
         self majPartie: gagn].
   self afficherGagnant: gagn.
   ^gagn!

majPartie: aJoueurTen
   "On met à jour la partie connaissant le gagnant d'un coup"

   fin := true! !

!Partie methodsFor: 'private'!

adversaire: aJoueurTen
   "rend l'adversaire d'un joueur"

   | adv |
   (score keys includes: aJoueurTen)
      ifFalse: [^nil]
      ifTrue: [self adversaires do: [:a | a == aJoueurTen
           ifFalse: [adv := a]]].
   ^adv! !

!Partie methodsFor: 'accessing'!

adversaires
   "rend les adversaires opposés dans le jeu"

   ^score keys!

fin
   "Détermine la fin de la partie. Dans cette version on rend la valeur
    de la variable"

   ^fin!

receveur
   "rend le serveur du jeu"

   ^self adversaire: self serveur!

serveur
   "rend le serveur du jeu"

   ^serveur! !
"- - - - - - - - - - - - - - - - - - "!

Partie class
   instanceVariableNames: 'NbPtsMax '!

!Partie class methodsFor: 'instance creation'!

nouveauJ1: aJoueurTen1 j2: aJoueurTen2
   "Démarre une nouvelle partie"
```

```
   ^self new initializeJ1: aJoueurTen1 j2: aJoueurTen2! !

!Partie class methodsFor: 'accessing'!

nbPtsMax
   "accès à la variable d'instance de la métaclasse"

   ^NbPtsMax! !

!Partie class methodsFor: 'class initialization'!

initialize
   "Partie initialize"

   NbPtsMax := 0.! !

"******************************** "!
Partie subclass: #JeuAbstrait
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Tennis'!
JeuAbstrait comment:
'Cette classe implante une version abstraite de la classe Jeu du
 match de tennis. Elle est définie par spécialisation de la classe
 abstraite Partie. Elle rassemble le comportement commun à un jeu et
 un tie break.
    jouerUnCoup signifie jouerUnPoint ici
    jouerUnePartie signifie jouerUnJeu ici'!

!JeuAbstrait methodsFor: 'update'!

jouerUnCoup
   "On tire le gagnant au sort.
   Si le gagnant est 1 alors le serveur gagne, sinon le receveur gagne."

   | gagnant sort |
   sort := (Random new next * 10000 rem: 2) truncated + 1.
   sort = 1
      ifTrue: [gagnant := self serveur]
      ifFalse: [gagnant := self receveur].
   score at: gagnant put: (score at: gagnant)
         + 1.
   ^gagnant!

majPartie: aJoueurTen
   "On met à jour la partie connaissant le gagnant d'un coup"

   | gagn sg sp |
   gagn := aJoueurTen.
   sg := score at: gagn.
   sp := score at: (self adversaire: gagn).
   sg >= self class nbPtsMax & (sg - sp >= 2) ifTrue: [fin := true]! !

!JeuAbstrait methodsFor: 'printing'!

afficherMsg
   "Affichage d'un message en début de partie."

   Transcript cr; show: self serveur printString , ' sert'! !

"******************************** "!
```

```
JeuAbstrait subclass: #Jeu
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Tennis'!
Jeu comment:
'Cette classe implante la classe Jeu du match de tennis.
Elle est définie par héritage de JeuAbstrait.
Le jeu évolue selon la spécification de la classe abstraite JeuAbstrait.
Seul l''affichage des scores change.

Les variables d''instance de la métaclasse sont :
   NbPtsMax : Integer     est redéfinie avec la valeur 4.
   ValPoints : Array[String]    valeur d''affichage des points
Cette dernière variable permet de transcrire le score selon les règles habituelles :
elle contient une table de codage : score -> affichage.
Elle est utilisée par une méthode de classe valPoints et initialisée à la création de la classe.'!

!Jeu methodsFor: 'printing'!

afficherGagnant: aJoueurTen
   super afficherGagnant: aJoueurTen.
   Transcript show: ' le jeu'!

afficherScores
   "Affichage du score du jeu"

   | gagn |
   (score at: self serveur)
      > (score at: self receveur)
      ifTrue: [gagn := self serveur]
      ifFalse: [gagn := self receveur].
   (score at: gagn)
      >= self class nbPtsMax
      ifTrue:
         [| diff |
         diff := (score at: gagn)
                  - (score at: (self adversaire: gagn)).
         diff = 0
            ifTrue: [Transcript cr; show: 'égalité']
            ifFalse: [diff = 1
               ifTrue: [Transcript cr; show: 'avantage ' , gagn printString]
               ifFalse: [fin := true]]]
      ifFalse: [Transcript cr; tab; show:
           (self class valPoints at: (score at: self serveur) + 1)
           , ' - ' , (self class valPoints at:
                 (score at: self receveur)+ 1)]! !
"- - - - - - - - - - - - - - - - - - - - "!

Jeu class
   instanceVariableNames: 'ValPoints '!

!Jeu class methodsFor: 'class initialization'!

initialize
   "Jeu initialize"

   NbPtsMax := 4.
   ValPoints := #('0'  '15'  '30'  '40' 'jeu')! !

!Jeu class methodsFor: 'accessing'!

valPoints
```

```
      "accès à la variable d'instance de la métaclasse"

      ^ValPoints! !

   "****************************** "!
   JeuAbstrait subclass: #TieBreak
      instanceVariableNames: ''
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Tennis'!
   TieBreak comment:
   'Cette classe implante la classe Tie Break du match de tennis.
   Elle est définie par héritage de JeuAbstrait.

   Seule l''évolution du jeu change : le serveur change tous les points pairs.

   Les variables d''instance de la métaclasse sont :
      NbPtsMax : Integer     est redéfinie avec la valeur 7. '!

   !TieBreak methodsFor: 'update'!

   majPartie: aJoueurTen
      "On met à jour la partie connaissant le gagnant d'un coup.
      Cette méthode est redéfinie pour inclure le changement de service."

      | gagn sg sp |
      gagn := aJoueurTen.
      sg := score at: gagn.
      sp := score at: (self adversaire: gagn).
      sg >= self class nbPtsMax & (sg - sp >= 2)
         ifTrue: [fin := true]
         ifFalse: [(sg + sp rem: 2)
               = 1
               ifTrue:
                  [serveur := self receveur.
                  Transcript tab; show: 'changement de service ' , self serveur printString]]!
!

   !TieBreak methodsFor: 'printing'!

   afficherGagnant: aJoueurTen
      super afficherGagnant: aJoueurTen.
      Transcript show: ' le tie break'!

   afficherScores
      "Affichage du score de la partie. Version abstraite."

      Transcript cr; tab.
      super afficherScores! !
   "- - - - - - - - - - - - - - - - - - "!

   TieBreak class
      instanceVariableNames: ''!

   !TieBreak class methodsFor: 'class initialization'!

   initialize
      "TieBreak initialize"

      NbPtsMax := 7.! !

   "****************************** "!
   Partie subclass: #SetTen
```

```
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Tennis'!
SetTen comment:
'Cette classe implante la classe SetTen (pour ne pas confondre avec la
 classe Set) du match de tennis. Elle est définie par héritage de JeuAbstrait.

jouerUnCoup signifie jouerUnJeu ici
jouerUnePartie signifie jouerUnSet ici

L''évolution du jeu et l''affichage des scores changent.
Le serveur change à chaque jeu.

Les variables d''instance de la métaclasse sont :
   NbPtsMax : Integer     est redéfinie avec la valeur 7. '!

!SetTen methodsFor: 'printing'!

afficherGagnant: aJoueurTen
   super afficherGagnant: aJoueurTen.
   Transcript show: ' le set'!

afficherScores
   "Affichage du score du jeu. Version Set."

   Transcript cr; tab; show: 'Set : ' , self serveur printString , ' '.
   super afficherScores.
   Transcript show: ' ' , self receveur printString! !

!SetTen methodsFor: 'update'!

jouerUnCoup
   "Jouer un coup signifie jouer un jeu dans le set.
   Le serveur est donné en premier. "

   | jeuEnCours gagnant |
   jeuEnCours := Jeu nouveauJ1: self serveur j2: self receveur.
   gagnant := jeuEnCours jouerUnePartie.
   score at: gagnant put: (score at: gagnant)
         + 1.
   ^gagnant!

majPartie: aJoueurTen
   "On joue un set, on change de serveur entre chaque jeu.
   Si le score est de 6 à 6, on joue un tie-break.   "

   | gagn sg sp |
   gagn := aJoueurTen.
   gagn = self receveur ifTrue: [Transcript tab; tab; show:
            ' / Break pour ' , self receveur printString].
   sg := score at: gagn.
   sp := score at: (self adversaire: gagn).
   sg >= (self class nbPtsMax - 1) & (sg - sp = 1) not
      ifTrue:
         [sg - sp >= 2
            ifTrue: [fin := true]
            ifFalse:
               ["sg = sp"
               Transcript cr; cr; show: 'Tie break'.
               gagn := (TieBreak nouveauJ1: self receveur j2: self
                     serveur) jouerUnePartie.
               score at: gagn put: (score at: gagn)
```

```
                    + 1].
          fin := true]
      ifFalse: [serveur := self receveur]! !
"- - - - - - - - - - - - - - - - - - - "!

SetTen class
   instanceVariableNames: ''!

!SetTen class methodsFor: 'class initialization'!

initialize
   "SetTen initialize"

   NbPtsMax := 7.! !

"******************************** "!
Partie subclass: #Match
   instanceVariableNames: 'sets adversaires '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Tennis'!
Match comment:
'Cette classe implante la classe Match de tennis. Elle est définie
 par héritage de JeuAbstrait car elle définit deux adversaires et
 le déroulement d''une partie.

Les variables d''instance sont :
     sets : OrderedCollection [SetTen]

 - la variable sets mémorise dans l''ordre les sets joués
 - le score contient le nombre de sets gagnés par joueur.

Les variables d''instance de la métaclasse sont :
   NbPtsMax : Integer     nombre de sets gagnants pour gagner le match.

jouerUnCoup signifie jouerUnSet ici
jouerUnePartie signifie jouerUnMatch ici'!

!Match methodsFor: 'printing'!

afficherGagnant: aJoueurTen
   super afficherGagnant: aJoueurTen.
   Transcript show: ' le match'!

afficherScores
   "Affichage du score du match.
   Problème : l'ordre des joueurs dans les sets peut varier d'un set à l'autre."

   Transcript cr; tab; show: 'Score du match : '.
   sets do: [:s | s afficherScores]! !

!Match methodsFor: 'update'!

jouerUnCoup
   "Jouer un coup signifie jouer un set dans le match.
   Le serveur est donné en premier. Les scores sont mémorisés.
   La fin du set détermine le changement de service."

   | gagnant setEnCours |
   setEnCours := SetTen nouveauJ1: self serveur j2: self receveur.
   gagnant := setEnCours jouerUnePartie.
   score at: gagnant put: (score at: gagnant)
         + 1.
```

```
   sets add: setEnCours.
   serveur := setEnCours receveur.
   ^gagnant!

majPartie: aJoueurTen
   "On met à jour le match connaissant le gagnant d'un set.
   Cette méthode est redéfinie pour inclure le changement de service.
   On s'arrête lorsque le nombre de set gagnants est atteint."

   (score at: aJoueurTen)
      = self class nbPtsMax ifTrue: [fin := true]! !

!Match methodsFor: 'initialize-release'!

initializeJ1: aJoueurTen1 j2: aJoueurTen2
   "Démarre un nouveau match.
   Par défaut, le serveur est tiré au sort."

   super initializeJ1: aJoueurTen1 j2: aJoueurTen2.
   (Random new next * 10000 rem: 2) truncated + 1 = 1
      ifTrue: [serveur := aJoueurTen1]
      ifFalse: [serveur := aJoueurTen2].
   Transcript cr; show: 'Le tirage au sort donne le service à ' ,
       serveur printString , '.'.
   sets := OrderedCollection new.
   ^self! !
"- - - - - - - - - - - - - - - - - - "!

Match class
   instanceVariableNames: ''!

!Match class methodsFor: 'class initialization'!

initialize
   "Match initialize"

   NbPtsMax := 3.! !

!Match class methodsFor: 'examples'!

unMatch
   "Match unMatch"

   | joueur1 joueur2 leMatch |
   joueur1 := (JoueurTen new) nom: 'Pierre'; prenom: 'Afeux'; classement: 12.
   joueur2 := (JoueurTen new) nom: 'René'; prenom: 'Gossie'; classement: 11.
   Transcript cr; show: joueur1 description; show: joueur2 description.
   leMatch := self nouveauJ1: joueur1 j2: joueur2.
   leMatch jouerUnePartie! !

Partie initialize!
Match initialize!
SetTen initialize!
Jeu initialize!
TieBreak initialize!
```

Figure 237 : *Code Smalltalk de l'application tennis*

3.2 Exécution

```
%inphb.im created at February 26, 2001 10:51:05 am

Je m'appelle Pierre Afeux et je suis classé 12
Je m'appelle René Gossie et je suis classé 11
Le tirage au sort donne le service à René Gossie.
René Gossie sert
   15 - 0
   15 - 15
   15 - 30
   30 - 30
   40 - 30
René Gossie gagne le jeu
   Set : René Gossie 1 - 0 Pierre Afeux
Pierre Afeux sert
   0 - 15
   0 - 30
   0 - 40
   15 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 0 - 2 René Gossie       / Break pour René Gossie
René Gossie sert
   15 - 0
   15 - 15
   30 - 15
   30 - 30
   30 - 40
Pierre Afeux gagne le jeu
   Set : René Gossie 2 - 1 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux sert
   15 - 0
   30 - 0
   40 - 0
   40 - 15
   40 - 30
Pierre Afeux gagne le jeu
   Set : Pierre Afeux 2 - 2 René Gossie
René Gossie sert
   0 - 15
   0 - 30
   15 - 30
   30 - 30
   40 - 30
René Gossie gagne le jeu
   Set : René Gossie 3 - 2 Pierre Afeux
Pierre Afeux sert
   15 - 0
   30 - 0
   30 - 15
   30 - 30
   30 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 2 - 4 René Gossie       / Break pour René Gossie
René Gossie sert
   0 - 15
   0 - 30
   15 - 30
   30 - 30
   40 - 30
   40 - 40
avantage Pierre Afeux
Pierre Afeux gagne le jeu
```

```
   Set : René Gossie 4 - 3 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux sert
   0 - 15
   0 - 30
   0 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 3 - 5 René Gossie       / Break pour René Gossie
René Gossie sert
   0 - 15
   0 - 30
   0 - 40
   15 - 40
   30 - 40
Pierre Afeux gagne le jeu
   Set : René Gossie 5 - 4 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux sert
   15 - 0
   30 - 0
   40 - 0
   40 - 15
   40 - 30
   40 - 40
avantage René Gossie
égalité
avantage Pierre Afeux
Pierre Afeux gagne le jeu
   Set : Pierre Afeux 5 - 5 René Gossie
René Gossie sert
   0 - 15
   15 - 15
   30 - 15
   40 - 15
René Gossie gagne le jeu
   Set : René Gossie 6 - 5 Pierre Afeux
Pierre Afeux sert
   15 - 0
   30 - 0
   30 - 15
   30 - 30
   30 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 5 - 7 René Gossie       / Break pour René Gossie
René Gossie gagne le set
   Score du match :
   Set : Pierre Afeux 5 - 7 René Gossie
René Gossie sert
   15 - 0
   30 - 0
   30 - 15
   30 - 30
   30 - 40
Pierre Afeux gagne le jeu
   Set : René Gossie 0 - 1 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux sert
   15 - 0
   15 - 15
   15 - 30
   15 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 1 - 1 René Gossie       / Break pour René Gossie
René Gossie sert
   15 - 0
   30 - 0
```

```
   30 - 15
   30 - 30
   30 - 40
   40 - 40
avantage René Gossie
René Gossie gagne le jeu
   Set : René Gossie 2 - 1 Pierre Afeux
Pierre Afeux sert
   15 - 0
   15 - 15
   15 - 30
   15 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 1 - 3 René Gossie       / Break pour René Gossie
René Gossie sert
   15 - 0
   15 - 15
   15 - 30
   15 - 40
   30 - 40
Pierre Afeux gagne le jeu
   Set : René Gossie 3 - 2 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux sert
   0 - 15
   0 - 30
   15 - 30
   30 - 30
   40 - 30
Pierre Afeux gagne le jeu
   Set : Pierre Afeux 3 - 3 René Gossie
René Gossie sert
   0 - 15
   15 - 15
   30 - 15
   40 - 15
René Gossie gagne le jeu
   Set : René Gossie 4 - 3 Pierre Afeux
Pierre Afeux sert
   0 - 15
   0 - 30
   15 - 30
   30 - 30
   40 - 30
Pierre Afeux gagne le jeu
   Set : Pierre Afeux 4 - 4 René Gossie
René Gossie sert
   0 - 15
   15 - 15
   30 - 15
   30 - 30
   30 - 40
Pierre Afeux gagne le jeu
   Set : René Gossie 4 - 5 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux sert
   15 - 0
   30 - 0
   30 - 15
   30 - 30
   30 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 5 - 5 René Gossie       / Break pour René Gossie
René Gossie sert
   15 - 0
```

```
   15 - 15
   15 - 30
   15 - 40
Pierre Afeux gagne le jeu
   Set : René Gossie 5 - 6 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux sert
   15 - 0
   15 - 15
   15 - 30
   15 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 6 - 6 René Gossie       / Break pour René Gossie

Tie break
René Gossie sert
   0 - 1   changement de service Pierre Afeux
   1 - 1
   1 - 2   changement de service René Gossie
   3 - 1
   4 - 1   changement de service Pierre Afeux
   2 - 4
   2 - 5   changement de service René Gossie
   5 - 3
   6 - 3   changement de service Pierre Afeux
   3 - 7
René Gossie gagne le  tie break
René Gossie gagne le set
   Score du match :
   Set : Pierre Afeux 5 - 7 René Gossie
   Set : Pierre Afeux 6 - 7 René Gossie
René Gossie sert
   0 - 15
   15 - 15
   15 - 30
   15 - 40
Pierre Afeux gagne le jeu
   Set : René Gossie 0 - 1 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux sert
   0 - 15
   0 - 30
   15 - 30
   15 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 1 - 1 René Gossie       / Break pour René Gossie
René Gossie sert
   0 - 15
   15 - 15
   30 - 15
   40 - 15
René Gossie gagne le jeu
   Set : René Gossie 2 - 1 Pierre Afeux
Pierre Afeux sert
   0 - 15
   15 - 15
   30 - 15
   40 - 15
Pierre Afeux gagne le jeu
   Set : Pierre Afeux 2 - 2 René Gossie
René Gossie sert
   0 - 15
   0 - 30
   0 - 40
   15 - 40
```

```
   30 - 40
Pierre Afeux gagne le jeu
   Set : René Gossie 2 - 3 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux sert
   15 - 0
   30 - 0
   40 - 0
   40 - 15
   40 - 30
   40 - 40
avantage Pierre Afeux
Pierre Afeux gagne le jeu
   Set : Pierre Afeux 4 - 2 René Gossie
René Gossie sert
   0 - 15
   0 - 30
   15 - 30
   30 - 30
   40 - 30
René Gossie gagne le jeu
   Set : René Gossie 3 - 4 Pierre Afeux
Pierre Afeux sert
   15 - 0
   30 - 0
   40 - 0
Pierre Afeux gagne le jeu
   Set : Pierre Afeux 5 - 3 René Gossie
René Gossie sert
   15 - 0
   30 - 0
   40 - 0
   40 - 15
   40 - 30
   40 - 40
avantage Pierre Afeux
égalité
avantage René Gossie
René Gossie gagne le jeu
   Set : René Gossie 4 - 5 Pierre Afeux
Pierre Afeux sert
   0 - 15
   0 - 30
   0 - 40
   15 - 40
   30 - 40
   40 - 40
avantage Pierre Afeux
Pierre Afeux gagne le jeu
   Set : Pierre Afeux 6 - 4 René Gossie
Pierre Afeux gagne le set
   Score du match :
   Set : Pierre Afeux 5 - 7 René Gossie
   Set : Pierre Afeux 6 - 7 René Gossie
   Set : Pierre Afeux 6 - 4 René Gossie
René Gossie sert
   0 - 15
   15 - 15
   15 - 30
   15 - 40
Pierre Afeux gagne le jeu
   Set : René Gossie 0 - 1 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux sert
   15 - 0
```

```
   30 - 0
   40 - 0
   40 - 15
   40 - 30
   40 - 40
avantage Pierre Afeux
Pierre Afeux gagne le jeu
   Set : Pierre Afeux 2 - 0 René Gossie
René Gossie sert
   0 - 15
   0 - 30
   0 - 40
   15 - 40
   30 - 40
   40 - 40
avantage Pierre Afeux
Pierre Afeux gagne le jeu
   Set : René Gossie 0 - 3 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux sert
   15 - 0
   15 - 15
   30 - 15
   30 - 30
   30 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 3 - 1 René Gossie       / Break pour René Gossie
René Gossie sert
   0 - 15
   0 - 30
   15 - 30
   15 - 40
Pierre Afeux gagne le jeu
   Set : René Gossie 1 - 4 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux sert
   0 - 15
   0 - 30
   0 - 40
   15 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 4 - 2 René Gossie       / Break pour René Gossie
René Gossie sert
   0 - 15
   0 - 30
   0 - 40
Pierre Afeux gagne le jeu
   Set : René Gossie 2 - 5 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux sert
   0 - 15
   0 - 30
   0 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 5 - 3 René Gossie       / Break pour René Gossie
René Gossie sert
   15 - 0
   30 - 0
   30 - 15
   30 - 30
   30 - 40
Pierre Afeux gagne le jeu
   Set : René Gossie 3 - 6 Pierre Afeux       / Break pour Pierre Afeux
Pierre Afeux gagne le set
   Score du match :
   Set : Pierre Afeux 5 - 7 René Gossie
```

```
   Set : Pierre Afeux 6 - 7 René Gossie
   Set : Pierre Afeux 6 - 4 René Gossie
   Set : René Gossie 3 - 6 Pierre Afeux
Pierre Afeux sert
   15 - 0
   30 - 0
   40 - 0
Pierre Afeux gagne le jeu
   Set : Pierre Afeux 1 - 0 René Gossie
René Gossie sert
   15 - 0
   30 - 0
   40 - 0
   40 - 15
   40 - 30
   40 - 40
avantage Pierre Afeux
égalité
avantage René Gossie
René Gossie gagne le jeu
   Set : René Gossie 1 - 1 Pierre Afeux
Pierre Afeux sert
   0 - 15
   15 - 15
   30 - 15
   40 - 15
Pierre Afeux gagne le jeu
   Set : Pierre Afeux 2 - 1 René Gossie
René Gossie sert
   0 - 15
   0 - 30
   15 - 30
   15 - 40
   30 - 40
   40 - 40
avantage René Gossie
René Gossie gagne le jeu
   Set : René Gossie 2 - 2 Pierre Afeux
Pierre Afeux sert
   15 - 0
   30 - 0
   30 - 15
   30 - 30
   30 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 2 - 3 René Gossie       / Break pour René Gossie
René Gossie sert
   15 - 0
   30 - 0
   40 - 0
René Gossie gagne le jeu
   Set : René Gossie 4 - 2 Pierre Afeux
Pierre Afeux sert
   15 - 0
   30 - 0
   30 - 15
   40 - 15
Pierre Afeux gagne le jeu
   Set : Pierre Afeux 3 - 4 René Gossie
René Gossie sert
   15 - 0
   30 - 0
   40 - 0
```

```
René Gossie gagne le jeu
   Set : René Gossie 5 - 3 Pierre Afeux
Pierre Afeux sert
   0 - 15
   0 - 30
   0 - 40
René Gossie gagne le jeu
   Set : Pierre Afeux 3 - 6 René Gossie       / Break pour René Gossie
René Gossie gagne le set
   Score du match :
   Set : Pierre Afeux 5 - 7 René Gossie
   Set : Pierre Afeux 6 - 7 René Gossie
   Set : Pierre Afeux 6 - 4 René Gossie
   Set : René Gossie 3 - 6 Pierre Afeux
   Set : Pierre Afeux 3 - 6 René Gossie
René Gossie gagne le match
```

Figure 238 : *Exécution d'un match de tennis*

4 Evolution de la conception

Nous avons utilisé des variables de classes pour paramétrer le programme. Une factorisation a été obtenue par abstraction dans la classe `Partie`. Une fois cette classe conçue et programmée, l'écriture des sous-classe est rapide. Il faut noter qu'à chaque fois qu'un comportement commun est mis en évidence, il est factorisé dans les super-classes. La structure du programme est bien visible par le modèle des classes de la figure 236.

Discutons maintenant des différentes évolutions proposées.

1. Un joueur peut abandonner la partie. L'abandon est une fonction aléatoire à l'intérieur de la procédure `jouerUnePartie`. A priori, il suffit de mettre à jour la variable fin pour arrêter le jeu. Deux précautions sont à prendre :
   (a) Pour répercuter, l'abandon dans un jeu au niveau du set puis du match, on rend un résultat supplémentaire dans la méthode `jouerUnCoup`. Dans la méthode appelante, on teste ce résultat.
   (b) Les contraintes posées dans la conception ne sont valables que si le jeu est fini normalement.

   La solution consiste donc à ajouter une variable d'instance `abandon` dans la classe `Partie` et à modifier les méthodes suivantes :
   (a) La méthode `jouerUnCoup` rend un paramètre supplémentaire.
   (b) La méthode `jouerUnePartie` boucle jusqu'à la fin ou l'abandon.

   Il faut vérifier la cohérence avec la redéfinition de ces méthodes dans les sous-classes.

   Seule une partie (verticale) du code est modifiée.
2. Matchs en deux sets gagnants.

   Il suffit de changer la constante `NbPtsMax` en 2 au lieu de 3. dans la classe `Match`.
3. Tournoi complet.

   Un tournoi complet est un ensemble de matchs. Des contraintes doivent être ajoutées sur les matchs joués : tours, adversaires qualifiés, etc. Le code existant n'est nullement remis en cause : c'est la réutilisation horizontale.
4. Matchs en doubles.

   Soit on considère l'équipe et non le joueur et il n'y a rien à changer. Soit il faut dissocier la notion de joueur (conservée pour le changement de service) de la notion d'équipe (mémorisation des scores).

5. Tennis de tables.

   On change les règles de calcul des points (constantes) et la règle de jeu d'un set. Le changement de service est aussi différent.

Les deux dernières évolutions implique des modifications assez subtiles dans l'ensemble du programme. Nous avons deux possibilités :

- Soit on recopie l'ensemble de la hiérarchie de classe, en renommant les classes. Puis on modifie localement les méthodes `jouerUnCoup`, `jouerUnePartie` `majPartie` et les variables de classes.
- Soit on définit un schéma d'héritage multiple comme le montre partiellement la figure 239. Pour être exploitable, il faudrait définir un héritage de groupes de classes, qu'on peut qualifier de *pattern* ici.

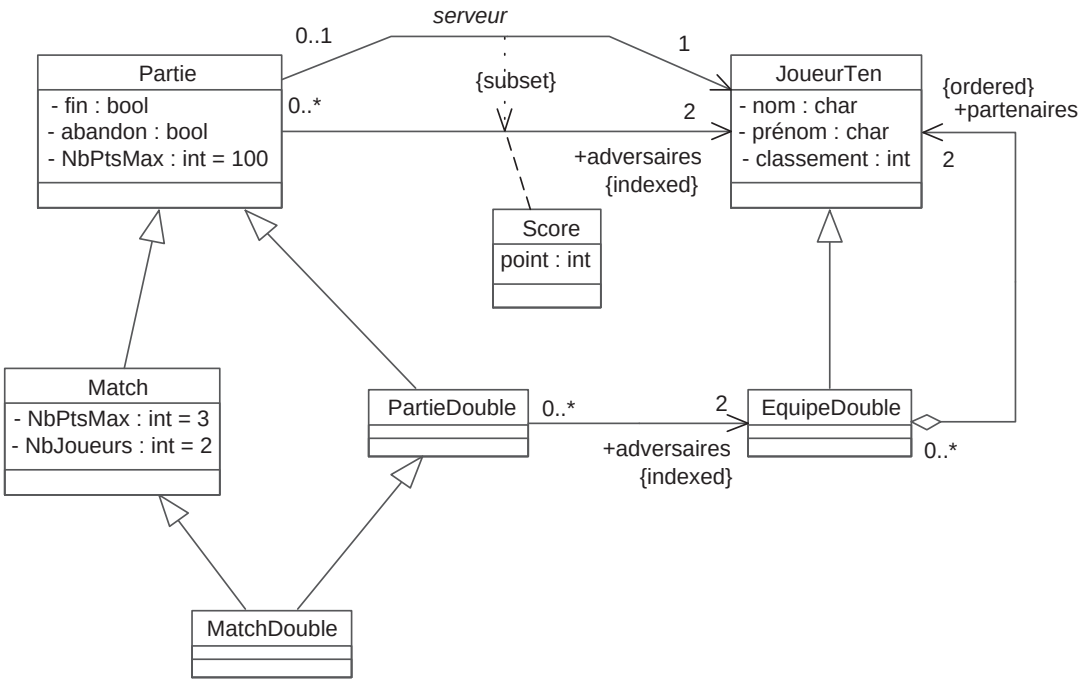


Figure 239 : *Conception objet du programme* `tennis en double`

4.1 Enseignements

La conception objet est plus perméable aux évolutions que la conception structurée. Par réutilisation verticale (héritage) ou hrorizontale (copie ou composition), il est souvent aisé de modifier une conception. Néanmoins, l'ajout de nouvelles sous-classe peut amener, pour une utilisation optimale de l'héritage et une meilleur lisibilité du code, à restructurer les classes de niveau intermédiaire.

5 Vers un pattern

Nous avons mis en évidence deux *patterns*. Le premier regroupe l'ensemble des classes de la figure 236. Deux personnalisations sont : parties de doubles et parties de tennis de table. Le second regroupe l'ensemble des classes relative à une partie d'un jeu dans la figure 240.

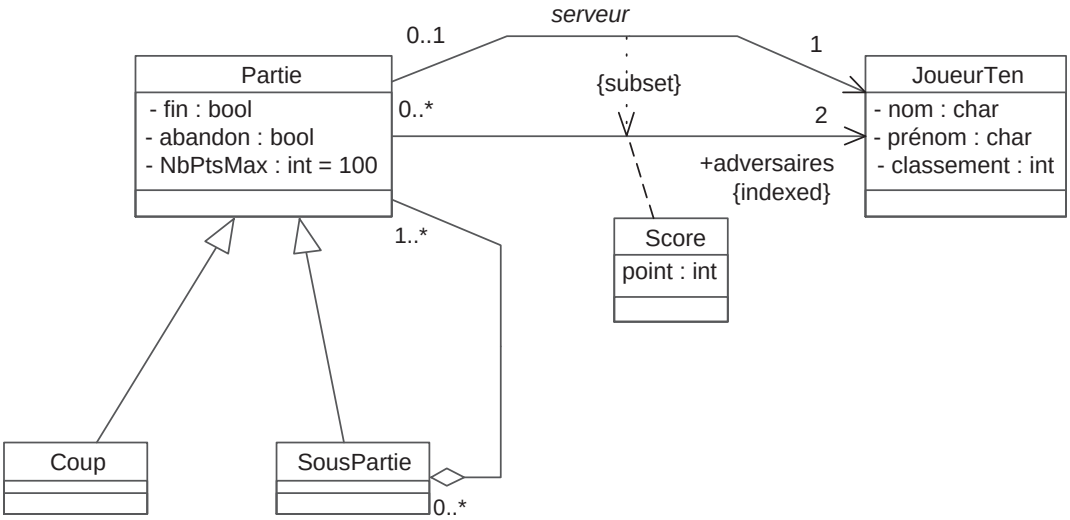


Figure 240 : *Conception objet du programme* `tennis en double`

6 Bilan

La conception abstraite est en fait issue d'une suite d'itérations, `conception - implantation`. Les classes de Smalltalk jouent un rôle dans la réutilisation.

A travers ce petit exemple, nous avons mis en évidence qu'un découpage en objet permet de définir une bonne abstraction du code. Ce découpage facilite la lisibilité, la conception, le test des différents modules. La cohésion des objets et l'héritage facilitent l'extension de l'application et la réutilisation de code. Noter aussi que la notation UML est un bon outil de documentation du code.