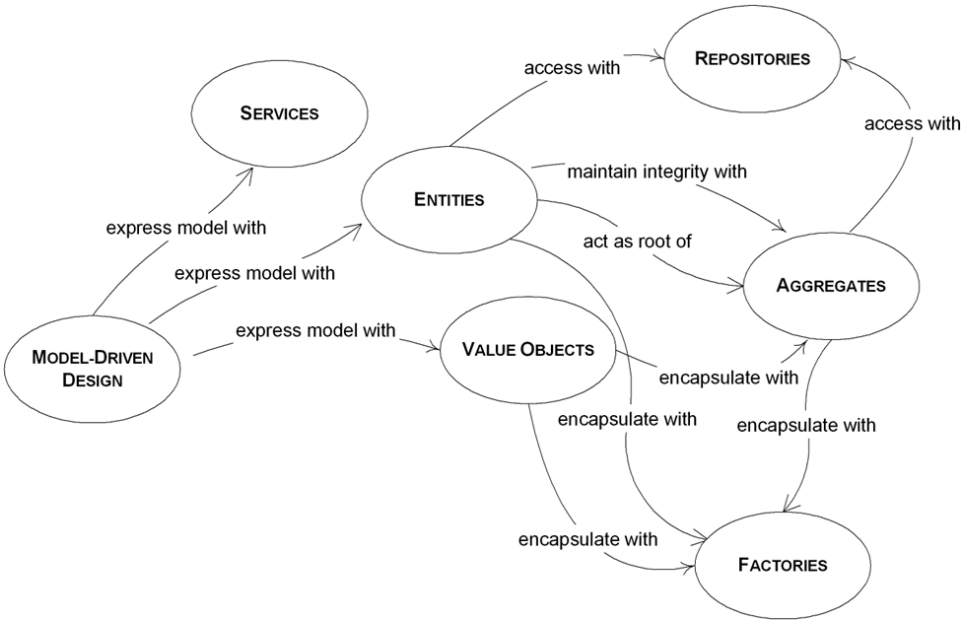
Travaux dirigés 3

# Conception dirigée par le domaine

La conception dirigée par le domaine est une méthode de développement basée sur le principe que, pour créer un bon logiciel, nous devons nous concentrer sur le domaine dans lequel il opère. La meilleure façon d'y arriver est de faire du logiciel un reflet du domaine. Il doit incorporer les concepts et les éléments qui sont au coeur du domaine, et saisir avec précision les relations entre eux. Autrement dit, le logiciel doit modéliser le domaine. Pour ce faire, nous devons organiser l'information qui nous intéresse, la systématiser, la diviser en plus petits morceaux, et regrouper ceux-ci dans des modules logiques. Le résultat est notre vision du domaine exprimée sous forme de descriptions textuelles, dessins, diagrammes UML, etc. Au début, cette vision est toujours incomplet. Mais avec le temps, en travaillant dessus, nous l'améliorons et elle devient de plus en plus clair. La figure suivante illustre les principales activités menées en conception dirigée par le domaine.




Le présent TD est organisé en trois sections, dont les deux premières proposent des exercices qui visent à faire comprendre les principes fondamentaux de la conception dirigée par le domaine, tandis que la troisième section porte sur un mini-projet concernant la réalisation du jeu d'échecs.

## 3.1 Modélisation du domaine

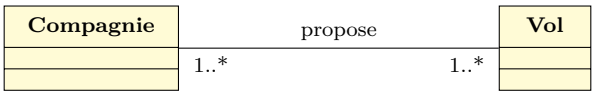
La conception pilotée par le domaine préconise de commencer par modéliser le domaine dans lequel opère le logiciel. Une possible manière d'exprimer le modèle du domaine est par le diagramme de classes. Nous avons déjà vu que ceci peut représenter la structure d'un code orienté objet. Néanmoins, il peut tout aussi bien décrire des concepts du monde réel, de la façon suivante.

— **Classe** : description abstraite d'un ensemble d'éléments ;
— **Attribut** : donnée logique contenu dans une classe ;
— **Association** : relation sémantique durable entre deux classes ;
— **Opération** : élément dynamique ou comportement d'une classe.

Une technique utile pour identifier les concepts fondamentaux du domaine est l'analyse linguistique, qui consiste à repérer les groupes nominaux et verbales dans une description textuelle. Par exemple, considérons un système de réservation de vols pour une agence de voyage, dont la description du domaine est résumée dans les phrases suivantes.

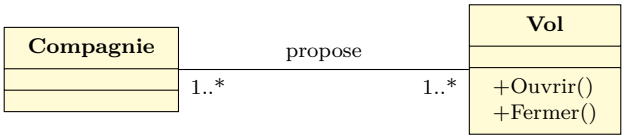
*1. Des compagnies aériennes proposent différents vols.*

Les substantifs *compagnie aérienne* et *vol* sont des concepts important du monde réel, car ils ont des propriétés et des comportements. Ils sont donc des classes candidates pour le modèle du domaine. Cependant, la phrase ne nous donne pas d'indication sur la multiplicité du coté de la classe `Compagnie`. Dans la suite, nous partirons du principe qu'un vol est proposé par une seul compagnie, mais il peut être partagé entre plusieurs affréteurs.



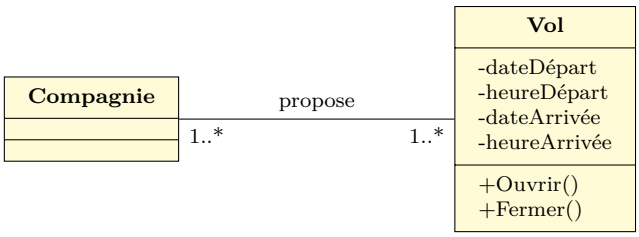

*2. Un vol est ouvert ou fermé à la réservation par la compagnie.*

Les notions d'ouverture et fermeture représentent des concepts dynamiques. Il s'agit en effet de changements d'état d'un objet `Vol` sur ordre d'un objet `Compagnie`. Dans le diagramme de classes, un élément dynamique est modélisé par une opération. En orienté objet, on considère que l'objet sur lequel on pourra réaliser un traitement doit le déclarer en tant qu'opération, afin que les autres objets puissent lui envoyer un message qui invoque cette opération. Nous plaçons donc les opérations d'ouverture et fermeture dans la classe `Vol`.



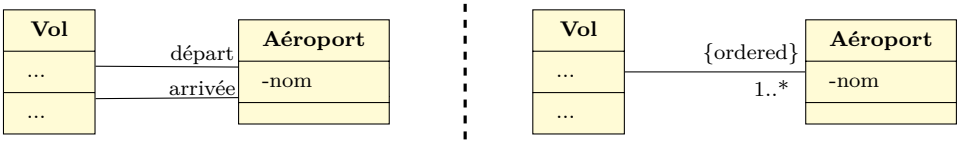
*3. Un vol a une date/heure de départ et une date/heure d'arrivée.*

Les notions de *date de départ* et *date d'arrivée* représentent des valeurs pures. Nous les modélisons donc comme des attributs. A cet égard, notons que l'erreur de modélisation la plus courante consiste à représenter quelque chose comme un attribut alors qu'il devrait s'agir d'une classe. Une bonne règle à suivre est la suivante : si l'on ne peut demander à un élément que sa valeur, il s'agit d'un attribut ; si plusieurs questions s'appliquent, c'est une classe.




*4. Un vol a un aéroport de départ et un aéroport d'arrivée.*

Contrairement à la notion de date, qui est un type « simple », le concept d'*aéroport* est complexe. Ceci ne possède pas seulement un nom, il a aussi une capacité, dessert des villes, etc. C'est la raison pour laquelle nous préférons créer une classe `Aeroport` plutôt que d'ajouter des simples attributs dans la classe `Vol`. En l'occurrence, la façon la plus précise de procéder est de créer deux associations entre `Aeroport` et `Vol`, chacune affectée d'un rôle différent. Notons qu'il existe une modélisation alternative, qui consiste à créer une association un-à-plusieurs. Mais nous perdons les notions de départ et d'arrivée. Une astuce serait alors d'ajouter une contrainte `{ordered}` du coté de la classe `Aéroport`, pour indiquer que les deux aéroports liés à un vol sont ordonnés (l'arrivée à lie après le départ). Voici les deux solutions.




## Exercice 3.1

Continuez la modélisation en prenant en compte les phrases suivantes.

*5. Un vol peut comporter des escales dans des aéroports.*

*6. Un escale a une date/heure d'arrivée et une date/heure de départ.*

*7. Chaque aéroport dessert une ou plusieurs villes.*

Proposez deux solutions alternatives, dont l'une basée sur modèle ayant deux associations, et l'autre basée sur le modèle ayant l'association un-à-plusieurs. Pour mieux affiner vos modèles, recourrez aux diagrammes d'objets pour construire des exemples de vols avec escales.

## Exercice 3.2

Dans cet exercice, vous allez voir qu'une phrase aussi simple que « un pays a une capitale » peut être modélisée en cinq manières différentes. Proposez des diagrammes de classes tout en respectant les instructions suivantes.

1. Première solution, la plus compacte possible : une classe `Pays` avec un simple attribut capitale. C'est suffisant si vous voulez seulement récupérer le nom de la capitale de chaque pays. Vous pouvez par la suite compléter le modèle en ajoutant quelques attributs à la classe `Pays` : nom, langue, monnaie, etc.

2. Comment faire si vous voulez ajouter des propriétés au concept de capitale, tels que habitants, superficie, etc. ? La solution précédente trouve sa limite et vous devez promouvoir `Capitale` au rang de classe.

3. En effet, un pays contient sa capitale. On peut alors se demander si l'association n'est pas une composition. Une capitale appartient bien à un pays et un seul, vérifiant le premier critère de la composition. Cependant, que se passe-t-il en cas de destruction d'un pays ? La capitale n'est pas forcement détruite, mais elle disparait en tant que concept administratif. Il s'agit donc d'une composition.

4. Vous sentez bien au fil des phrases qu'un concept plus général que capitale vous fait défaut : la notion de ville. Supposez qu'un pays soit annexé. Sa capitale n'existe plus en tant qu'entité administrative, mais la ville ne sera pas forcement détruite ! Donc, si vous souhaitez définir un modèle plus général, il est intéressant de modéliser le fait qu'un pays contient des villes, dont une seule joue le rôle de capitale. Pour ce faire, renommez la classe Capitale en `Ville`, désignez la composition par le rôle de capitale et ajouter une deuxième association un-à-plusieurs pour indiquer qu'un pays contient des villes.

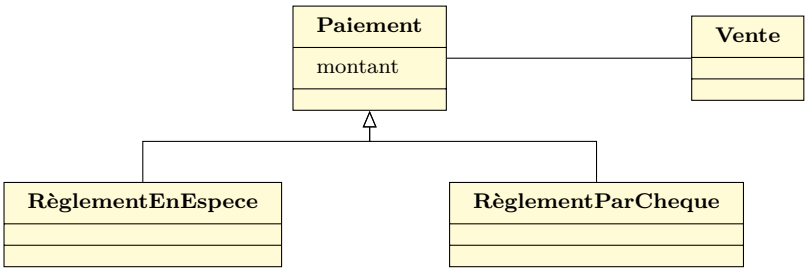
5. Enfin, si vous voulez préciser qu'une capitale est une ville ayant des propriétés spécifiques, il faut réintroduire la classe `Capitale` en tant que sous-classe de `Ville`, et déplacer la composition vers cette nouvelle classe. La classe `Capitale` peut maintenant recevoir des attributs ou associations supplémentaires, si le besoin s'en fait sentir.

Les cinq modèles expriment à leur façon la phrase initiale. Le premier est très compact, simple à implémenter, mais très peu évolutif dans le cas où il faudrait répondre à de nouvelles demandes d'un utilisateur. Le dernier est nettement plus complexe à implémenter, mais très souple. Il résistera longtemps à l'évolution des besoins utilisateurs. Le choix entre les différents solutions doit se faire en fonction du contexte : faut-il privilégier la simplicité, les délais de réalisation, ou au contraire la pérennité et l'involutivité ?

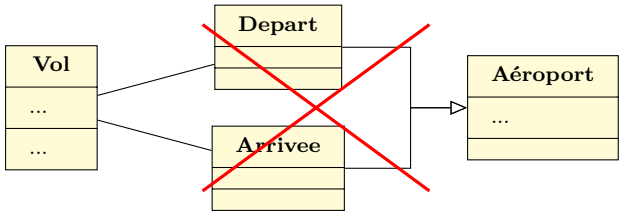
### 3.1.1 Relation de généralisation

La généralisation est un cas particulier d'association non-symétrique. Elle modélise le fait qu'une classe possède les mêmes attributs, associations et opérations d'une autre classe, mais elle peut en spécialiser certains et en rajouter des autres. Cela engendre une relation de type « est-un », car les objets de la classe spécialisée peuvent remplacer ceux de la classe générale.

Par exemple, considérons les concepts de *paiement*, *règlement en espèces* et *règlement par chèque*. Puisque les deux derniers sont des formes spécialisées de payement, ils sont liés par une relation de généralisation (règle « est-un »). Voici un modèle où tout paiement a un montant et il est associé à une vente.




En revanche, reconsidérons les concepts de *vol*, *aéroport de départ* et *aéroport d'arrivée* de la section précédente. Une solution tentante consiste à créer deux sous-classes de `Aeroport`. Pourtant, cette solution est incorrecte ! En effet, tout aéroport est l'origine pour certains vols et la destination pour d'autres. Les sous-classes auraient donc les même instances dupliquées.




*Évitez la généralisation lorsque :*
*(1) la sous-class nullifie des opérations héritées ;*
*(2) les objets ont besoin de changer de classe.*

**Exercice 3.3**

Modélisez les énoncés qui suivent avec les relations appropriées.

1. Une personne conduit une voiture.
2. Une école possède des salles partagées avec d'autres écoles.
3. Une transaction boursière est un achat ou une vente.
4. Un compte peut appartenir à une personne physique ou morale.
5. Deux personnes peuvent être mariées ou pacsées. Deux personnes mariées sont de sexe opposé.
6. Un répertoire peut contenir des répertoires et des fichiers.

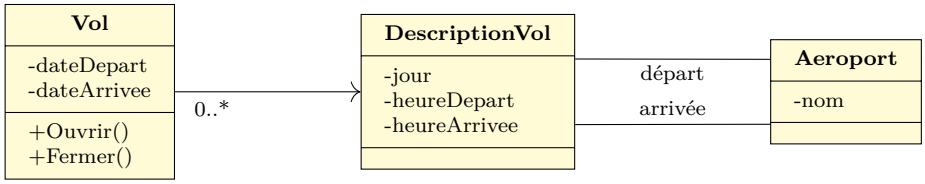
### 3.1.2 Patron de la méta-classe

Quand une classe possède trop de responsabilités (attributs, associations, opérations), dont certaines ne sont pas propres à chaque instance, il est préférable de la factoriser en deux classes séparée au travers d'une association un-à-plusieurs (voir la figure), afin de mieux repartir les responsabilités.


Cette décomposition, nommée *patron de la méta-classe*, permet de :

1. disposer de la description d'un produit ou d'un service (dans une instance de `Description`), indépendamment de l'existence actuelle de l'un de ces produits ou services (instances de `Item`) ;
2. supprimer toutes les instances d'une classe (`Item`) sans entrainer la perte des informations communes (objets `Description`) ;
3. réduire la duplication des informations (attributs de `Description`).

Par exemple, la classe `Vol` de la section précédente possède deux types de responsabilités. Le premier concerne les vols génériques, qui reviennent à l'identique toutes les semaines (il existe un Toulouse-Paris sans escale, tous les lundis à 7h10, proposé par AirFrance). Le second rassemble les vols réels réservés par les clients (le Toulouse-Paris à une date précise). Nous pouvons donc repartir l'ancienne classe `Vol` en deux classes, comme montré ci-dessous.




**Exercice 3.4**

Appliquez le patron de la méta-classe aux domaines suivants.

1. **Articles dans un magasin**. Un article a un numéro de série univoque, ainsi qu'un descriptif et un prix communs à tous ceux du même type.
2. **Livres d'une bibliothèque**. Une bibliothèque possède plusieurs exemplaires du même livre, qui sont empruntés par les adhérents.

**Exercice 3.5**

Intégrez la classe `DescriptionVol` dans le domaine des réservations aériennes. Puis, continuez la modélisation en prenant en compte les phrases suivantes.

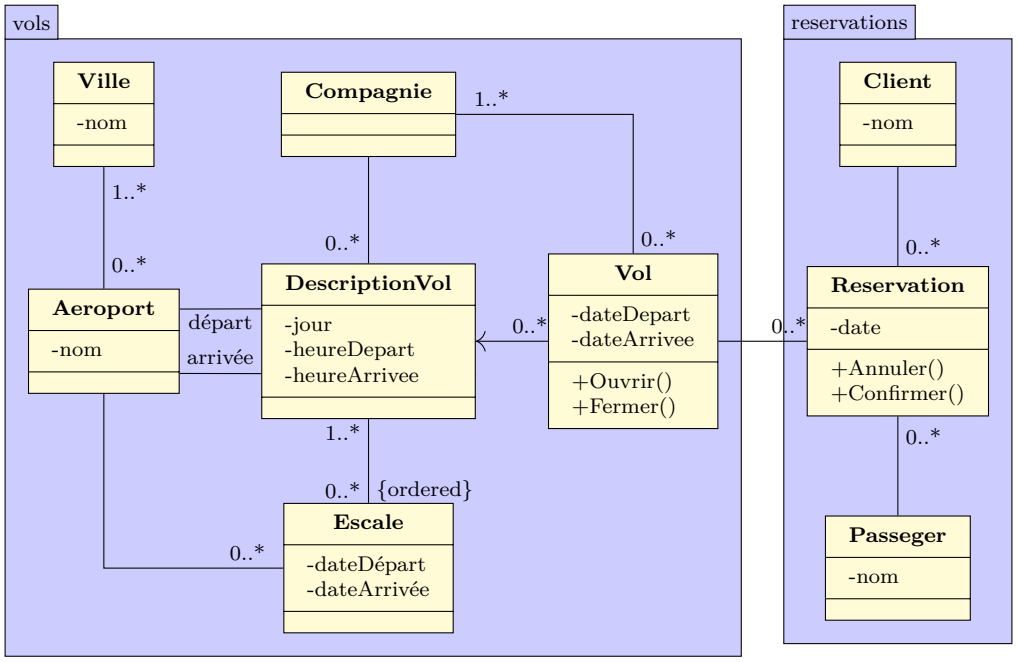
*8. Un client peut réserver un ou plusieurs vols, pour des passegers différents.*

*9. Une réservation concerne un seul vol et un seul passager*

*10. Une réservation peut être annulée ou confirmée.*

### 3.1.3 Structuration en paquets

Dans une application vaste et complexe, le modèle du domaine a tendance à devenir de plus en plus gros. Il devient alors compliqué de comprendre les relations et les interactions entre ses différentes parties. C'est pourquoi il est nécessaire de structurer le modèle en paquets, selon les principes de cohérence et indépendance. Le premier principe consiste à regrouper les classes qui opèrent sur les mêmes données ou qui travaillent de concert pour réaliser une tâche bien définie. Le seconde principe s'efforce de minimiser les relations entre classes de paquets différents. Par exemple, nous pouvons organiser le domaine des réservations ariennes dans les deux paquets suivants.




**Exercice 3.6**

Après tout ce travail sur les réservations de vols, nous souhaitons élargir le modèle en proposant aussi des voyages en bus assurés par des transporteurs.

1. Un voyage en bus a une ville de départ et une ville d'arrivée, avec des dates et heures associées. Le trajet peut comporter des arrêts dans des villes intermédiaires. Un client peut réserver un ou plusieurs voyages, pour un ou plusieurs passagers. Proposez un modèle du domaine.

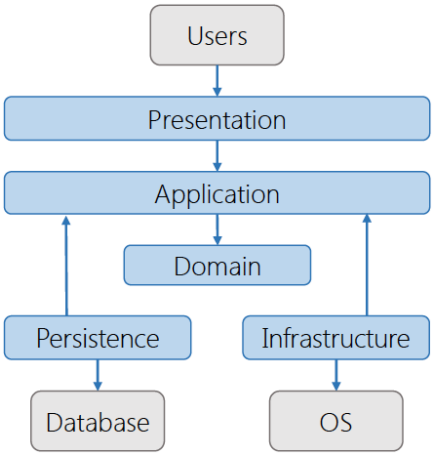
2. Les deux modèles présentent des nombreuses similitudes. Essayez d'isoler les classes communes dans des nouveaux paquets, afin de les réutiliser. Factorisez les propriétés communes dans des classes abstraites.

## 3.2 Transition vers la conception

Dans la conception dirigée par le domaine, la première étape est de créer un modèle du domaine. L'étape suivante est d'implémenter le modèle en code. En effet, tout domaine peut être exprimé à travers de nombreux modèles, et tout modèle peut se traduire de diverses manières dans le code. Lequel choisir ? Celui qui puisse être exprimé convenablement dans du code ! Nous voulons que toute portion du système reflète fidèlement le modèle du domaine. Pour ce faire, nous revisitons le modèle de façon continue, pour qu'il puisse être implémenté plus naturellement dans le logiciel.

Lorsqu'on crée une application logicielle, il peut arriver qu'une grande partie de l'application n'est pas reliée directement au domaine, notamment tout le code lié à l'accès aux données persistantes, aux fichiers, au réseau, aux interfaces utilisateur, etc. Pour cette raison, nous devons partitionner le système en couches. Nous concentrons tout le code lié au modèle du domaine dans une couche, afin de l'isoler des autres partie du système. Les objets du domaine, libérés de la responsabilité de s'afficher eux-mêmes, de se stocker eux-mêmes, de gérer les tâches applicatives, et ainsi de suite, pourront se concentrer sur l'expression du domaine métier. Cela permet à un modèle d'évoluer jusqu'à être assez riche et assez clair pour capturer la connaissance essentielle du métier et la mettre en oeuvre. Une solution courante pour les conceptions dirigées par le domaine est l'architecture en quatre couches.

1. **Domaine**. Cette couche contient les objets du domaine. C'est le coeur du logiciel.
2. **Application**. Cette couche coordonne l'activité des objets du domaine. Elle ne contient pas de logique métier, mais elle peut détenir l'état de la progression d'une tâche applicative.
3. **Présentation**. Cette couche est responsable de l'interface utilisateur, la présentation de l'information et l'interprétation des commandes.
4. **Infrastructure**. C'est le soutien pour toutes les autres couches. Elle fournit la communication entre celles-ci, implémente la persistance des objets, contient les bibliothèques auxiliaires, etc.




Par exemple, dans le logiciel de réservations aériennes, une interaction typique entre les couches présentation, application, domaine et infrastructure pourrait être la suivante. Lorsque l'utilisateur veut réserver un vol, la couche présentation demande à la couche applicative de le faire. Celle-ci récupère les objets du domaine auprès de l'infrastructure et invoque les méthodes adéquates. Une fois que les objets ont effectué toutes les opérations, la couche application persiste les objets en passant par l'infrastructure.

### 3.2.1 Entités et objets-valeurs

Il y a une catégorie l'objets dont l'identité semble rester la même au fil des états du logiciel. Pour ces objets, ce ne sont pas les attributs qui comptent, mais une ligne de continuité et d'identité qui s'étend sur la durée de vie d'un système et peut s'allonger au-delà. Ces objets sont appelés *entités*. Une simple manière de créer une identité unique pour chaque objet consiste à lui affecter un identifiant créé par l'utilisateur ou généré automatiquement (comme c'est le cas pour les clés primaires dans une base de données).

Les entités sont des objets importants d'un modèle du domaine, et elles devraient être examinées dès le début du processus de modélisation. Devons-nous faire de tous les objets des entités ? Il pourrait être tentant de le faire, mais créer et tracer l'identité a un cout. En réalité, il y a des cas où nous avons purement besoin de stocker des attributs d'un élément du domaine. Un objet qui est utilisé pour décrire certains aspects d'un domaine, mais qui n'a pas d'identité, est appelé *objet-valeur*. Il est fortement recommandé que les objets-valeurs soient immuables. Ils sont créés à l'aide d'un constructeur et jamais modifiés au cours de leur vie. Lorsque nous voulons une valeur différente pour l'objet, nous en créons tout simplement un autre.

Par exemple, l'entité *client* a un nom, une rue et une ville. C'est mieux si les informations sur l'adresse sont contenues dans un objet-valeur séparé, plutôt qu'être des attributs de l'entité, car elles forment un tout conceptuel.

### 3.2.2 Services

Quand nous modélisons le domaine, nous découvrons que certaines notions ne se transposent pas facilement en objets, et que certaines opérations ne semblent appartenir à aucun objet. Souvent, ce genre de comportement fonctionne à travers plusieurs objets, potentiellement de classes différentes. Par exemple, transférer de l'argent d'un compte à un autre : cette fonction devrait-elle s'exécuter dans le compte qui envoie ou dans le compte qui reçoit ? L'un ou l'autre semblent tout autant mal placés.

Lorsque nous identifions un comportement qui n'appartient pas naturellement à aucun objet du domaine, il vaut mieux le déclarer en tant que *service*. Ceci est un objet sans état interne qui fournit des opérations utiles aux autres objets du domaine. L'intérêt d'un service ne réside pas dans l'objet en lui-même, mais dans les objets sur ou pour le compte desquels les opérations sont effectuées. Cependant, nous pouvons facilement confondre les services appartenant à la couche domaine et ceux appartenant à la couche application, car ils fournissent tous les deux des opérations reliées aux objets du domaine. Au cours du travail sur le modèle et pendant la phase de conception, nous devons assurer que la couche domaine reste isolée des autres couches.

### 3.2.3 Fabriques

Les relations entre les objets du domaine peuvent parfois être vastes et complexes. Dans ce cas, la construction d'un objet devient un processus laborieux, qui demande une grande connaissance de sa structure interne, des relations avec les autres objets, et des règles qui s'y appliquent. Lorsqu'un objet veut en créer un autre, il doit détenir une connaissance spécifique de l'objet à construire. Cela rompt l'encapsulation des objets du domaine. Si l'objet créateur appartient à la couche application, une partie de la couche domaine est déplacée ailleurs, ce qui détraque toute la conception.

La création d'un objet est une opération importante en soi, dont la responsabilité n'est pas des objets créateurs. C'est pourquoi il est nécessaire d'encapsuler la logique de création dans un objet séparé de type *fabrique*. Celle-ci est nécessaire uniquement pour la création d'objets complexes. Un simple constructeur suffit dans les autres cas, notamment lorsque la création d'un objet n'est pas compliquée, elle n'implique pas la constructions d'autres objets, tous les attributs requis sont passés via le constructeur, et l'objet créateur n'a pas besoin de choisir parmi une liste d'implémentations concrètes.

### 3.2.4 Entrepôts

Les objets ont un cycle de vie qui commence par leur création et se termine par leur suppression ou archivage. Tout l'objectif de créer des objets est de pouvoir les utiliser. Il s'avère qu'un bon nombre d'objets peut être directement récupéré dans un emplacement permanent, comme les entités et les objets-valeurs associés. C'est pourquoi il est nécessaire d'encapsuler la logique de persistance dans un objet séparé de type *entrepôt*. Ce faisant, les objets du domaine n'auront pas à s'occuper de la persistance des entités : ils iront simplement les chercher dans les entrepôts.

Il existe une relation entre les fabriques et les entrepôts. La fabrique crée de nouveaux objets, tandis que l'entrepôt se charge des objets déjà créés. Mais quand l'entrepôt doit reconstruire un objet depuis un lieu de stockage persistant, ou un nouvel objet doit être ajouté à l'entrepôt, l'objet en question devrait d'abord être créé par la fabrique, et ensuite donné à l'entrepôt.

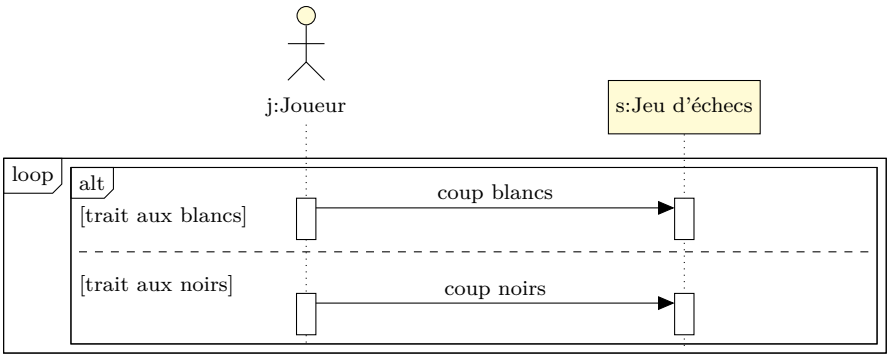
### 3.2.5 Développement piloté par les tests

Les tests unitaires sont incontournables pour valider les opérations métier implémentées dans le modèle du domaine. L'écriture de tests est d'autant plus facilitée par l'architecture en couches et la structuration en paquets, qui permettent d'isoler les éléments testés à l'aide de simulacres et bouchons. Il est à savoir que la conception dirigée par le domaine est parfaitement compatible avec le développement piloté par les tests.

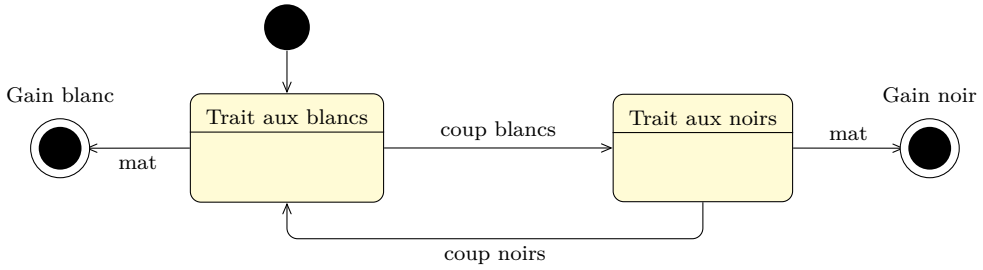
## 3.3 Mini-projet : jeu d'échecs

L'objectif de cette étude est de développer un jeu d'échecs. Le but du jeu est d'opposer deux joueurs sur un échiquier composé de 64 cases blanches et noires. Les joueurs jouent à tour de rôle en déplaçant l'une de leurs 16 pièces. Le but du jeu est d'infliger à son adversaire un échec et mat, une situation dans laquelle le roi d'un joueur est en prise sans possibilité de recours.

Le scénario d'utilisation typique du jeu d'échecs peut être décrit par un diagramme de séquence « au niveau système » qui contient une grande boucle, dans laquelle il arrive soit un coup des blancs soit un coup des noirs.




Les coups sont joués à tour de rôle par les blancs et les noirs. Lorsque le roi d'un joueur est en situation de mat, la partie se termine sur la victoire du joueur qui mate. Cela peut être représenté par un diagramme d'état.




INSTRUCTIONS Pour réaliser le jeu d'échecs, nous vous fournissons une solution Visual Studio organisée en deux dossiers. Le dossier « IHM » contient une simple interface homme-machine, alors que le dossier « Echecs » contient une implémentation partielle qui permet d'avoir une version exécutable du jeu (lancez-le pour tester). L'objectif du TP est de modéliser les règles du jeu d'échecs, afin d'identifier des classes qui serons ensuite utilisée pour implémenter le jeu (dans le dossier « Echecs »).

### 3.3.1 Modélisation du domaine

Modélisez les règles du jeu d'échecs en suivant les instructions ci-dessous. Cela vous permettra d'identifier un ensemble initial de classes que vous penserez à implémenter dans le dossier « Echecs » du projet fourni.

#### Échiquier

Le jeu d'échecs se joue à deux joueurs sur un **échiquier** carré composé de 64 **cases**, alternativement noires et blanches. Chaque case est caractérisée par la *couleur*, et son emplacement sur l'échiquier (*rangée* et *colonne*). Modélisez cette situation avec les relations appropriées, en sachant que les substantifs en gras et en italique sont respectivement des classes et des attributs.

#### Pièces

Chaque **joueur** a une *couleur* et possède initialement huit **pions**, ainsi qu'un **roi**, une **dame**, deux **tours**, deux **fous** et deux **cavaliers**. Par le biais de la promotion des pions en huitième rangée (transformation obligatoire en pièce à choisir sauf un roi), un joueur peut ainsi posséder jusqu'à neuf dames, dix tours, cavaliers ou fous. Modélisez cette situation avec les relations appropriées, en sachant que les substantifs en gras sont des classes.

#### Position des pièces

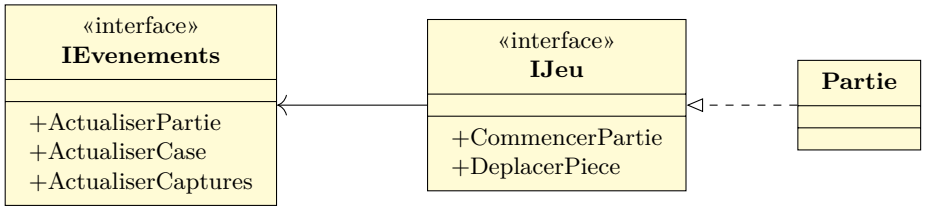
Une case ne peut accueillir qu'une seule **pièce** à la fois, laquelle est soit sur une case soit hors du jeu. Le mot « pièce » exprime que le pion, roi, dame, tour, fou et cavalier ont des choses en communs : ils ont une *couleur* et sont placés sur une seule case au maximum. On se demande si *pièce* est un concept plus général que pion, roi, etc. Respectent-ils la règle « est-un » ? Les sous-classes ont-elles toutes le même attribut (couleur) et la même association (case) qui peut être extraite et factorisée dans la super-classe ? Modifier le modèle précédent en fonction des réponses.

#### Partie

À chaque tour, un joueur déplace une pièce de son camp (blanc ou noir). On ne peut ni passer son tour, ni jouer deux fois de suite, et c'est toujours les blancs qui jouent en premiers. Une **partie** est donc une suite ordonnée de **coups**. Chaque coup comporte une pièce qui bouge de la case de départ à celle d'arrivée. Modifier le modèle précédent en reliant les concepts de joueur et partie (utilisez deux associations pour distinguer les blancs et les noires). Ensuite, reliez les concepts de partie, coup et pièce, puis les concepts de coup et case (utilisez deux associations pour les cases de départ et arrivée).

### 3.3.2 Conception de la couche présentation

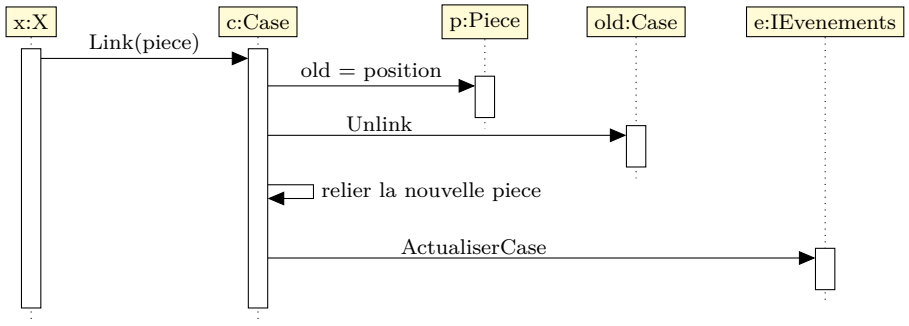
L'interaction entre les joueurs et le jeu est véhiculée par l'interface utilisateur, qui est responsable de visualiser l'état courant du jeu et d'interpréter les actions exécutées par les joueurs. Plus précisément, les couches présentation et domaine communiquent par le biais des interfaces `IJeu` et `IEvenements`. La première déclare les opérations que le joueur peut invoquer durant une partie, telles que `CommencerPartie()` et `DeplacerPiece()`. Elle est implémentée par la classe `Partie`. La deuxième spécifie les opérations pour mettre à jour l'interface utilisateur au fur et à mesure que le jeu avance.




#### Visualiser le contenu d'une case

La classe `Case` possède une association à la classe `Piece` pour indiquer qu'une case peut accueillir (au plus) une pièce. Pour relier correctement une case à une pièce, la classe `Case` fournit les méthodes `Unlink` et `Link`. La méthode `Unlink` annule la référence sur l'objet `Piece` et soulève un événement `ActualiserCase` pour effacer la case correspondante dans l'IHM.

En revanche, la méthode `Link` commence par invoquer `Unlink` sur la case où la pièce d'entrée se trouve, relie celle-ci avec la case courante (notamment l'objet « this »), et enfin soulève un événement `ActualiserCase` pour mettre à jour le contenu de la case correspondante dans l'interface utilisateur.




**instructions** Implémentez les méthodes `Unlink` et `Link` dans la classe `Case`, puis complétez la méthode `CommencerPartie` en suivant les indications en commentaire. Testez en supprimant le code entre TEST et FIN TEST.

### 3.3.3 Conception des opérations métier

Continuez l'implémentation des classes du domaine à travers l'ajout des opérations qui sont spécifiées dans les sections suivantes.

#### Création d'une nouvelle partie

La méthode `CommencerPartie` permet de démarrer une nouvelle partie. En particulier, elle crée les deux joueurs et l'échiquier, elle demande aux joueurs de positionner leurs pièces sur l'échiquier, et enfin elle émet l'événement `ActualiserPartie` pour donner le trait aux blancs.

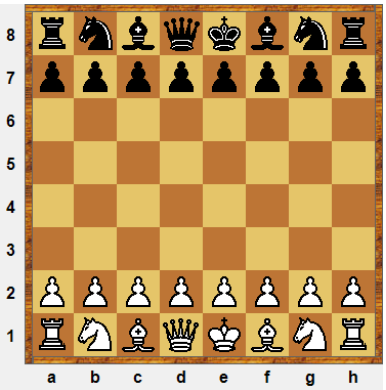



**Figure 3.1** *Configuration initiale de l'échiquier.*

instruction Complétez la méthode `PlacerPieces` dans la classe `Joueur`, afin de positionner les pièces sur l'échiquier comme montré dans la figure ci-dessus. Ensuite, décommentez le contenu de la méthode `DeplacerPiece` et supprimez le code entre TEST et FIN TEST. Vérifier que vous pouvez bouger les pièces sur l'échiquier en alternance entre les blancs et les noirs.

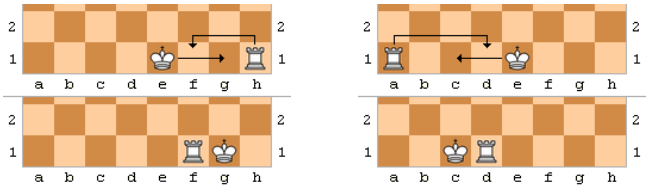
#### Déplacement des pièces

La méthode `DeplacerPiece` s'occupe de bouger une pièce de la case de départ à celle d'arrivée. Cependant, pas tous les déplacements sont valides, car les pièces ont chacune leur mode de déplacement propre. Votre tache est de modifier la méthode `DeplacerPiece`, ainsi que d'implémenter toute autre méthode que vous jugez nécessaires, afin de gérer le déplacement des pièces conformément aux règles des échecs. *Conseil :* L'opération de déplacement est polymorphe, c'est-à-dire chaque instance de la classe `Piece` doit se déplacer en fonction de l'algorithme implémenté au niveau des sous-classes concrètes.

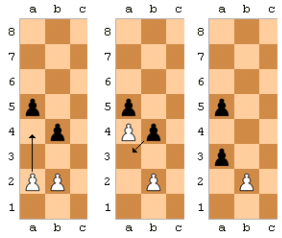
### Autres opérations

Vous êtes encouragés à implémenter un maximum d'autres opérations. Elles peuvent concerner le domaine du jeu d'échecs, telles que :

1. le roque, consistant à déplacer en un seul coup le roi et l'une des tours ;



2. la prise en passant, qui s'applique aux pions déplacés de deux cases ;



3. la détection de l'échec et l'interdiction de déplacer les pièces de telle sorte que le roi est exposé à l'échec ;
4. la promotion des pions qui atteignent la dernière rangée de l'échiquier (la huitième pour les blancs et la première pour les noirs) ;
5. la déclaration d'une partie nulle, qui désigne une position dans laquelle le camp ayant le trait et n'étant pas sous le coup d'un échec, ne peut plus jouer de coup légal sans mettre son propre roi en échec.

Également, elles peuvent concerner la conception d'une couche applicative qui supporte des fonctionnalités avancées de l'interface utilisateur, telles que :

1. l'affichage des pièces perdues ;
2. la possibilité de recommencer une nouvelle partie ;
3. l'affichage du score ;
4. la gestion du « play & pause » ;
5. l'historique des mouvements précédents ;
6. l'affichage des prédictions lors d'un déplacement d'une pièce ;
7. la gestion de `Undo()` et `Redo()`.