Rapport d’analyse - justifications

1. MCD - UML

Compréhension du sujet :

La classe Utilisateur

Le système est destiné à être utilisé par différents acteurs ayant chacun des rôles bien particuliers et définis : les administrateurs, les rédacteurs, les éditeurs, les modérateurs et les lecteurs. Chaque rôle est associé à un ensemble de permissions et d’actions possibles que l’acteur se voit accorder en même temps que le rôle.
Nous avons considéré que ces rôles pouvaient se cumuler (un administrateur pouvant être également modérateur, par exemple). En effet, nous ne voulions pas imposer une telle contrainte aux utilisateurs futurs du système et une telle souplesse dans le système de gestion de droit des utilisateurs (organe vital du système) nous semblait primordiale.
Nous avons donc défini les rôles comme des héritages non exclusifs de la classe Utilisateur.

Les utilisateurs sont des lecteurs

Un utilisateur enregistré dans le système ne devrait pas avoir besoin de l’action d’un administrateur pour être un lecteur : tout utilisateur est donc considéré comme étant un lecteur, ce qui fait d’Utilisateur une classe abstraite. Tout rôle supplémentaire de l’utilisateur se cumulera donc avec celui de lecteur.
Il aurait été possible de supprimer la classe Lecteur et de rattacher toutes ses relations actuelles directement à la classe Utilisateur. Nous aurions néanmoins perdu en lisibilité/sémantique dans notre UML et aurions perdu un peu de cette décomposition des rôles qui est si primordiale.

Gestion des statuts pour les articles et commentaires

Les articles et les commentaires sont les deux seules entités susceptibles d’être gérées par plusieurs utilisateurs mais surtout par plusieurs utilisateurs ayant des rôles différents.

Penchons nous tout d’abord sur les articles : ces derniers sont gérés par leur rédacteur mais également par les éditeurs. Plus précisément, en plus de créer le contenu de l’article, le rédacteur peut associer trois statuts différents à son article : en rédaction, soumis et supprimé. Il est important de noter que ces statuts sont tout à fait mutuellement exclusifs : en effet, un article qui serait à la fois en rédaction et soumis n’aurait aucun sens, surtout dans la mesure où un article soumis n’est plus modifiable par le rédacteur.
À partir du moment où le rédacteur a imposé le statut soumis à un article, l’éditeur peut alors lui associer à son tour un statut : en relecture, rejeté, à réviser, validé et publié. À noter qu’un article ne peut se voir attribuer le statut publié qu’à l’unique condition que son statut précédent soit validé, et qu’un statut rejeté ou à réviser se verra épaulé d’une justification ou d’une préconisation. Le champ texte réservé à cet usage sera null dans le cas d’un autre statut. Encore une fois, ces statuts sont mutuellement exclusifs.

De façon analogue, un commentaire peut se voir imposer ou retirer le statut supprimé par son auteur, et les statuts visible, masqué et mis en exergue par un modérateur. Ces statuts sont également exclusifs (un commentaire masqué n'a aucun intérêt à être mis en exergue, un commentaire en exergue est forcément visible, etc). Nous n'avons néanmoins pas fait la distinction entre un statut Auteur et un statut Modérateur, car le clivage entre les statuts nous semblait plus moindre que pour les différents statuts des articles. De plus, faire cette distinction risquait d'alourdir fortement notre modèle UML.

Historisation des statuts éditeur pour chaque article

On notera néanmoins que, malgré les apparentes similitudes entre les statuts Rédacteurs et les statuts Éditeurs, ces derniers sont exprimés de façon différente dans notre diagramme de classe : alors que le statut rédacteur est considéré comme un attribut de la classe Article, le statut éditeur est une classe d'association. Un article est en effet destiné, au cours de sa vie, à se voir associer de nombreux statuts éditeurs différents (en relecture, puis à réviser, de nouveau en relecture… puis validé, etc.)
Souhaitant conserver, entre autres, une trace des actions des éditeurs sur les articles, nous ne voulons pas qu'un changement de statut éditeur efface le précédent, ce qui nous est possible de faire en utilisant une classe d'association plutôt qu'un attribut.

Composition - Articles & blocs

Un article, afin d'avoir une grande souplesse et possibilité de contenu, est composé de blocs. Ces derniers peuvent être soit des images soit du contenu texte. Dans tous les cas, ils seront dotés d'un titre. Nous avons traduit ceci dans notre diagramme par la classe abstraite Bloc ainsi qu'un héritage exclusif.
Néanmoins cette solution seule était insuffisante pour répondre à nos besoins : en effet, un article est affiché et lu dans un ordre linéaire précis de blocs. Il faut donc ordonner ces derniers pour qu'ils forment un ensemble cohérent et conforme à la volonté du rédacteur, et lisible par un lecteur. Un attribut *ordre,* rajouté à la classe abstraite Bloc, était donc primordial pour le bon fonctionnement du système.
Il sera noté que le sujet ne précise pas que l'article soit doté d'un titre, ce qui pourrait paraître étrange du fait qu'un article de journal, électronique ou non, est traditionnellement doté d'un titre. Nous avons donc rajouté un attribut titre à la classe en tant que clé primaire (on suppose que deux articles ne peuvent avoir le même titre), considérant que c'était implicite et qu'un rédacteur donnera nécessairement un titre à ses articles.

La rubrique à l'honneur

Nous avons choisi de ne pas considérer une instance Catégorie spéciale (qui hériterait de la classe Catégorie), mais d'en faire une entité à part, composée de tous les articles qui seraient mis à l'honneur (on imagine par les éditeurs, responsables de la mise en correspondance d'articles avec des rubriques). Ainsi, l'accent est mis non pas sur la catégorie à l'honneur mais bien sur les articles qui la composent : plutôt que de représenter cette catégorie particulière (qu'on pourrait voir comme un *Singleton* en POO) dans notre MCD, nous nous contentons d'ajouter un attribut booléen à la classe Article.

2. MLD - Relationnel

Traduction des statuts

En ce qui concerne les statuts, le plus simple est de traduire le statut rédacteur, puisque celui-ci est unique pour chaque article, même si amené à évoluer. Par conséquent, nous l'avons représenté *via* un attribut énuméré statusRedacteur. On considère que ce statut vaut par défaut "en rédaction" pour chaque article créé.

Cependant, cette solution ne pouvait fonctionner avec les statuts d'éditeur, puisque plusieurs éditeurs peuvent modifier le même article (ajouté au fait qu'on veut conserver un historique de toutes leurs actions), même si chaque article ne possède à un instant *t* qu'un seul statut éditeur actif. Nous avons donc créé un attribut statusEditeur qui indique le statut éditeur actif en clé étrangère vers une table Statut_editeur_article pouvant être null si l'article considéré n'a pas encore été soumis au moins une fois. La table Statut_editeur_article stocke, pour un article donné et un éditeur donné, le statut que ce dernier lui a donné à une date donnée. Pour des simplicités de références, nous avons donc créé une clé artificielle *id* pour cette table, mais (article, date) reste une clé candidate (on ne peut pas attribuer deux statuts éditeur à un article au même moment). Enfin, des triggers dynamiques s'occuperaient de vérifier la cohérence des données (justifications_preconisations NOT NULL ssi status = rejete OR status = a_reviser par exemple).

Nous avons appliqué le même raisonnement pour les commentaires attribués aux articles : chaque commentaire possède un statut de visibilité (par défaut, visible), mais celui-ci pourra être amené à changer (suppression par son auteur, ou suppression, masquage, mise en exergue par un modérateur). On veut de même garder trace de toutes ces actions, d'où la création de la table Statut_commentaire fonctionnant de manière analogue à la table Statut_editeur_article.

Traduction de l'héritage des blocs

La classe mère bloc étant abstraite et l'héritage exclusif, nous avons choisis un héritage absorbé par les classes filles. La classe mère n'est donc pas représentée. Nous avons considéré que le titre d'un bloc ainsi que son ordre était unique pour chaque article auquel ce dernier appartenait. Nous n'avons pas représenté de contraintes sur le fait qu'un bloc texte et image issus d'un même article pouvaient avoir le même ordre. Nous ne pensons pas, en effet, que ce sera gênant dans l'utilisation future du système : il faudra soit gérer ce cas de figure lors de l'affichage de l'article (en mettant systématique les blocs images devant les blocs de texte ou inversement, par exemple) ou bien lors de l'insertion dans la base de données.

Traduction de l'héritage des utilisateurs

La relation User conserve l'ensemble des informations liés à chaque utilisateur du système, du point de vue de l'identité. La clé étrangère creator permet de savoir quel administrateur s'est occupé de la création du compte. Nous avons choisi un héritage par la classe mère, car il s'agissait pour nous du meilleur choix possible. La classe mère n'est pas abstraite (puisque, par défaut, tout le monde est considéré comme lecteur potentiel), l'héritage non exclusif (les rôles peuvent se cumuler), et loin d'être complet (puisqu'à chaque rôle correspond des associations spécifiques). Le passage par les classes filles aurait créé autant de redondances qu'il y a cumul

des rôles pour chaque utilisateur. On aurait pu faire un passage par référence, mais le même problème de redondance aurait été observé.

Gestion des rôles/droits avec Droit_User

Il nous fallait cependant trouver une solution pour gérer efficacement les droits cumulés pour chacun des utilisateurs (sans trop rentrer dans le détail de l'implémentation). Nous avons créé une table Droit_User qui se comporte légèrement comme la traduction d'une association N:N en UML (ce que nous ne pouvions faire lors du MCD, en effet, créer une association "possède" entre deux classes Utilisateur et Droit aurait empêché toute la sémantique et les associations liées à chaque rôle particulier). C'est une relation toute-clé qui référence User, mais sans référencer une deuxième relation Droit (qui contiendrait des intitulés jugés uniques) : à la place, nous avons seulement pris un attribut énuméré *droit : {administrateur, redacteur, editeur, moderateur}*. Deux raisons à ce choix : premièrement, créer une table Droit avec intitulés uniques aurait été pertinent dans la mesure où l'on voudrait à l'avenir créer d'autres droits spécifiques ; or, nous avons considéré que la situation présentée est stable est que les rôles présentés sont les seuls qui nous intéressent dans ce projet, et dans ce cadre précis (de Fil Étudiant). Deuxièmement, cette énumération ne contient pas "lecteur" puisque nous considérons que par défaut, tout le monde est lecteur : dès lors, si un utilisateur donné ne figure dans aucun tuple de Droit_User, cela signifie qu'il n'a aucun droit particulier et donc qu'il est seulement lecteur.

création d'attributs date : Date pour tenir l'historique des actions + champ user pour l'auteur d'une action ;

Le sujet met l'accent sur l'importance de pouvoir garder une trace des actions effectuées par l'ensemble de l'équipe du Fil. En effet, il est capital pour le bon travail en équipe de savoir qui est le responsable de chaque actions. Nous devions donc concevoir notre système afin que ce dernier puisse retracer l'ensemble des actions acquises par un commentaire, un article ou une catégorie, que ces dernières soient datées et que leur responsable soit identifié.
Nous avons donc rajouté, dans de nombreuses relations, un attribut de type Date (date) et un attribut référençant vers un User (creator/editor/user) afin de conserver ces informations.
On notera néanmoins des exceptions sur certaines actions : il ne nous a pas paru nécessaire de garder une trace des différents statusRedacteur, dans la mesure où ils seront toujours changés par la même personne (l'auteur de l'article) et que ces différents états ne rentrent pas en conflit avec le reste du travail de l'équipe, contrairement aux statuts éditeurs, par exemple.

Traduction de la hiérarchie des rubriques (association "possède")

L'association "possède" en UML représente la hiérarchie des rubriques les unes par rapport aux autres. Traditionnellement, puisqu'il s'agit d'une association N:N, nous aurions pu traduire cela par une nouvelle relation prenant deux clés étrangères référençant la classe Rubrique. Nous avons cependant préféré une solution plus "élégante", inspirée des arbres binaires : chaque rubrique possède un attribut *mother* qui correspond à sa catégorie mère. L'ensemble des informations de parenté est conservé, et permet d'économiser une relation supplémentaire.

La rubrique à l'honneur

Nous avons vu dans la justification du MCD que nous considérions que la rubrique à l'honneur est en effet constituée des articles jugés, au goût des éditeurs, comme étant dignes de se voir

ainsi distingués. Chaque article possède donc un attribut booléen *honor* repris dans le MLD qui détermine si oui ou non l'article est sélectionné.
Remarquons que nous ne perdons pas de sémantique, ni d'efficacité pratique, même en regard d'une possible implémentation (dont nous ne nous occupons pas ici) : d'une part, la rubrique à l'honneur pourrait être transversale, jouir d'une position particulière sur une page web (insistant sur son côté "à part") ; d'autre part, récupérer les articles concernés consisterait en un simple filtrage par leur attribut *honor*.

3. Normalisation

Première Forme Normale

Tous les attributs de nos relations sont atomiques et ces dernières possèdent toutes au moins une clé. Notre modèle logique est donc en 1NF.

Seconde Forme Normale

Nos relations ne possédant qu'une clé primaire sans clés candidates ou étant des relations toutes clés sont, de façon triviale, en 2NF.
Pour les relations possédant des clés candidates, (Statut_editeur_article et statut_commentaire), nous ne considérons pas la clé artificielle dans notre étude de la forme normale, ces relations sont donc comme celles restantes (Rubriques_articles, Bloc_txt, Bloc_img et Note), c'est à dire que leur clé primaire est constituée de plusieurs attributs et ne possédant pas de clés candidates. Les dépendances fonctionnelles sont bien élémentaires dans la mesure où une partie de la clé ne détermine à aucun moment le reste de la relation et où les attributs déterminés ne font pas partie de la clé.

Troisième Forme Normale

Encore une fois, nous ne considérons pas les clés artificielles dans notre étude. Il n'existe pas, dans toutes nos relations, de dépendances fonctionnelles autres que celles issues de nos clés primaires. Notre modèle est donc bien en 3NF.