Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
UNIVERSITÉ DE NAMUR
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Historique des échanges entre un système d'E.A.O. et les apprenants

Merckx, Olivier

Award date:
2019

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

Historique des échanges entre
un système d'E.A.O. et les apprenants.

Mémoire présenté par
Olivier MERCKX
pour l'obtention du titre de
licencié et maître en informatique.

Promoteur : Claude CHERTON.

Année académique 1985-1986

Je remercie tous ceux qui ont collaboré à
l'élaboration de ce mémoire,
et en particulier, Monsieur Cherton pour son
aide avisée et ses bons conseils,
ainsi que tous ceux qui m'ont appuyé et soutenu
pour atteindre le but fixé.

Table des matières :

Introduction : typologie des systèmes d'E.A.O.

Développements de l'E.A.O. (Enseignement Assisté par Ordinateur).

Jalons & portraits.

Partie 1 : Analyse conceptuelle.

1. Description d'un système d'E.A.O.

1.1. Description générale.

1.2. Description d'un dialogue.

1.3. Détermination des exercices à présenter.

1.4. Exemple de dialogue.

2. Définition & buts de l'historique.

2.1. Importance de l'historique pour le système d'E.A.O.

2.2. Importance de l'historique pour les enseignants.

3. Description des modules d'un système d'E.A.O.

3.1. Modèle de la matière.

3.2. Modèle de l'élève.

3.3. Description des dialogues.

3.4. Modèle d'enseignement-apprentissage.

4. Utilisation du système de gestion de l'historique.

4.1. Utilisation de l'historique par le système d'E.A.O.

4.2. Utilisation de l'historique par l'enseignant.

5. Description des données.

5.1. Schéma Entité-Association.

5.2. Le dialogue.

5.3. Le domaine d'enseignement.

5.4. La synthèse des connaissances.

6. Description des fonctions.

6.1. Mémorisation des dialogues.

6.2. Détermination des exercices.

6.3. Mise à jour de la synthèse.

6.4. Mise à jour des étudiants.

6.5. Mise à jour du domaine.

6.6. Analyse de l'historique.

Partie 2 : Conception logique.

7. Conception logique.

7.1. Schéma des accès possibles.

7.2. Spécification des modules.

Partie 3 : Conception physique.

8. Conception physique.

8.1. Description de Turbo Pascal et de Turbo Access System.

8.2. Schéma des accès en Turbo Pascal.

8.3. Descriptions des constantes, des types et des fichiers.

8.4. Descriptions des fonctions & des modules.

8.5. Utilisation des modules de l'historique.

Conclusion.

Sources bibliographiques.

Partie 4 : Annexes.

A. Schéma des modules de la conception logique.

B. Descriptions des conditions d'erreurs dans les procédures de la conception physique.

C. Description d'un programme utilisant le système de gestion de l'historique.

D. Programme du système de gestion de l'historique.

Développements de l'E.A.O. (Enseignement Assisté par Ordinateur).

L'informatique a tendance à s'étendre de plus en plus à tous les secteurs de la vie économique et sociale de l'homme; l'enseignement et l'éducation ne font pas exception à cette règle. L'introduction générale de micro-ordinateurs dans les lycées français est assez significative à cet égard. De nombreux programmes d'enseignement, très simples, appelés didacticiels, ont vu le jour. Malheureusement, les logiciels disponibles sur le marché sont très différents quant à leur valeur pédagogique. Certains enseignants se sont précipités sur ceux-ci sans discernement et sans réelle information concernant leur usage judicieux. Néanmoins, les progrès accomplis depuis les premiers didacticiels d'enseignement ouvrent de nouveaux horizons.

L'E.A.O. laisse entrevoir d'intéressantes perspectives. En effet, l'enseignement scolaire peut, grâce à lui, acquérir de puissants outils facilitant l'entraînement des étudiants dans certaines matières, mais aussi soulageant le professeur dans sa tâche. L'E.A.O. peut prendre en charge individuellement chaque apprenant et l'aider à surmonter ses déficits ou ses lacunes dans telle ou telle matière. Il peut servir également d'outil de recyclage ou de formation professionnelle, ainsi qu'en apprentissage à domicile. Le champ est vaste et encore peu exploré.

Jalons & portraits.

On peut distinguer deux types de systèmes : ceux qui guident réellement l'élève (directifs) et ceux qui laissent l'élève procéder par essais-erreurs (non directifs), en intervenant si besoin est pour lui apporter de l'aide. Dans le cas directif, le système détermine la matière à présenter et l'étudiant n'a d'autre choix que de répondre à l'exercice proposé. Dans le cas non directif, l'apprenant évolue au sein d'un environnement établi et dispose d'une série d'outils pour parvenir à un objectif. Il décide par lui-même quel outil il emploiera pour atteindre le but fixé (ex : Logo).
Par la suite, je parlerai essentiellement des systèmes d'E.A.O. directifs.

Les didacticiels existant reposent essentiellement sur une méthodologie du type "stimulant/réaction" [cfr 5]. La matière enseignée par le système est présentée sous forme de cours entrecoupés d'exemples et d'exercices que les étudiants sont invités à résoudre. La machine évalue ensuite chaque solution de l'élève en terme de réponse correcte ou fausse et détermine ensuite, en fonction de celle-ci, le chemin que devra suivre l'apprenant au sein du didacticiel. L'énoncé ou le cours suivant présenté au sujet dépend entièrement de la dernière réponse fournie.

Ce type de didacticiel utilise généralement des langages dits "d'auteur". Ce sont des langages-outils mis à la disposition des enseignants pour décrire chaque leçon, chaque exercice et les enchaînements entre ceux-ci en fonction des types de réponse. Leur description est totalement explicite. Chaque écran présenté à l'apprenant est complètement défini par le texte du langage. Pour évaluer les réponses de l'élève, la machine les compare avec une ou plusieurs réponses prédéfinies spécifiées par l'enseignant (les logiciels les plus complexes permettent des analyses par mots-clés). Les cours sont donc décrits sous forme de graphe dans lequel chaque liaison (entre leçons) est figée. Ces didacticiels reposent sur un modèle statique d'enseignement.

De tels systèmes d'E.A.O. présentent beaucoup d'inconvénients : l'investissement important de temps consacré par l'enseignant pour spécifier les leçons, le caractère figé de l'enseignement et le fait de ne tenir compte que d'une seule réponse pour décider du chemin à parcourir. L'enseignant doit déterminer "toutes" les réponses possibles pour chaque exercice ainsi que les liaisons correspondantes avec les autres parties du cours. Généralement, ces systèmes ne donnent aucune explication à l'élève lorsque les réponses de celui-ci ne sont pas correctes. La machine signale simplement si la réponse est exacte ou non. L'apprenant ne peut donc discerner à quel niveau se situe l'erreur.

En effet, ces didacticiels n'ont aucune connaissance relative à la matière enseignée car l'ensemble des cours est matérialisé par une succession d'écrans dont le contenu n'a aucune importance, ni aucune signification pour la machine. L'analyse des réponses se réduit à une simple évaluation syntaxique en fonction de conventions ou de règles établies par l'enseignant. Puisque le système ne tient pas compte de la sémantique des réponses, certaines, erronées mais valides syntaxiquement, seront acceptées tandis que d'autres seront refusées alors qu'elles devraient être considérées comme étant correctes.

Les nombreux inconvénients des didacticiels actuels nécessitent une remise en question de leur valeur pédagogique. Les logiciels doivent posséder une certaine "connaissance" de la matière pour analyser les réponses de l'élève et déterminer les exercices suivants à proposer. Le système lui expliquera généralement la ou les erreurs qu'il a commises. L'enseignant aura la possibilité de décrire ses leçons sous forme synthétique et non plus analytique : les cours sont ainsi plus faciles à spécifier. Cela lui fait donc gagner du temps. Grâce au modèle de la matière, la machine peut interpréter le contenu des leçons à présenter à l'étudiant et analyser les réponses de celui-ci.

De même, les didacticiels doivent tenir compte de l'ensemble des réponses de l'étudiant dans l'élaboration des situations pédagogiques ultérieures. L'enchaînement des exercices n'est plus statique, mais évolue selon les circonstances. Les didacticiels se basent ainsi sur un modèle dynamique d'enseignement.
Contrairement aux systèmes décrits précédemment, on peut considérer que les logiciels de ce type disposent d'un certain modèle de l'élève caractérisant chaque apprenant en fonction des dialogues antérieurs qu'il a eus avec la machine.
L'ensemble des réponses de l'étudiant permet à la machine de déterminer l'évolution de son comportement et de guider celui-ci à travers le didacticiel.

Le système n'utilise pas directement l'ensemble des données brutes disponibles dans l'historique des réponses, mais, par le biais de statistiques et de traitements divers, extrait des renseignements pour constituer une synthèse. Cette synthèse rassemble des informations relatives à la maîtrise par l'apprenant de la matière enseignée. Elle élabore un véritable profil de l'étudiant qui représente le modèle de l'élève au sein de la machine. Le système utilisera ce profil pour adapter son enseignement à l'apprenant. La machine analyse la compréhension de l'étudiant en lui posant des questions spécifiques ou des problèmes plus ou moins difficiles à résoudre. Elle s'efforce ensuite de formuler des hypothèses sur la mauvaise compréhension de l'élève. Les réactions du programme doivent amener l'étudiant à réfléchir sur ce qu'il a mal assimilé, les erreurs sont donc constructives.

Si les théories psycho-pédagogiques adaptables à l'ordinateur se développent et s'améliorent, peut-être pourra-t-on dépasser les cas purement descriptifs vers des modèles de plus en plus "explicatifs". Le profil de l'élève sera plus élaboré car il ne tiendra plus seulement compte de la maîtrise de la matière par l'étudiant, mais aussi des processus cognitifs du sujet. De véritables techniques pédagogiques seront utilisées pour adapter l'enseignement prodigué par la machine aux caractéristiques de l'apprenant communiquant avec celle-ci. Les didacticiels essaieront d'anticiper les réactions de l'étudiant et s'amélioreront en fonction des expériences antérieures.

L'enseignement ne peut être systématisé en pédagogie directive ou non directive. Il serait possible de combiner les deux types d'approches en laissant une plus ou moins grande initiative à l'élève en fonction de ses caractéristiques personnelles. D'autres méthodes pédagogiques pourraient aussi être envisagées. Ainsi, peut-être pourra-t-on concevoir des logiciels d'enseignement intégrant de multiples approches pédagogiques d'une même matière et utilisant l'une ou l'autre méthode selon les besoins ou le choix de l'apprenant.

ANALYSE CONCEPTUELLE

1. Description d'un système d'E.A.O.

La typologie des systèmes d'E.A.O. a souligné les inconvénients majeurs de la plupart des didacticiels actuellement disponibles sur le marché. La gestion d'un historique s'adresse donc principalement aux logiciels qui se basent sur l'ensemble des réponses de l'apprenant pour déterminer le parcours du sujet à travers la matière enseignée. Par conséquent, ces logiciels doivent posséder des connaissances relatives à la matière et à l'élève.

Le système de gestion de l'historique développé dans ce mémoire laissera une grande liberté à l'utilisateur quant aux informations qui seront mémorisées. Celui-ci enverra les données qu'il juge pertinentes et celles-ci seront enregistrées selon une structure spécifique. La gestion de l'historique offrira simplement des facilités d'accès aux informations et leur traitement sera laissé à la charge de l'utilisateur.

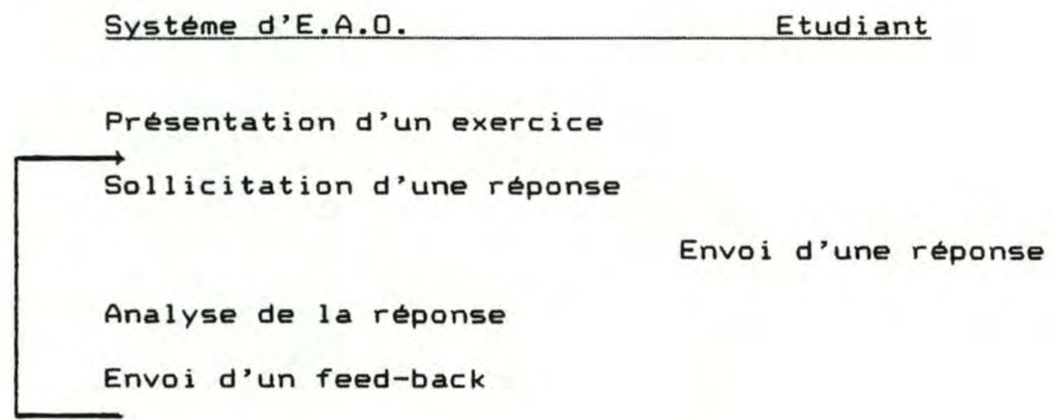
Dans les pages qui suivent, l'enseignement sera vu dans une optique directive.

1.1. Description générale.

Un système d'E.A.O. se caractérise par la présentation d'exercices à différents apprenants. Chaque exercice est une façon de concrétiser les objectifs pédagogiques poursuivis par le système. Les réponses de l'élève sont évaluées par la machine pour détecter les éventuelles erreurs commises. En fonction de celles-ci, la machine renvoie à l'étudiant un feed-back (informations de retour) adapté; cela lui permet de détecter les connaissances et savoir-faire qui lui manquent. Si le système juge que l'étudiant n'a pas répondu correctement, il peut lui demander (un certain nombre de fois) d'expliciter sa réponse, c'est-à-dire de détailler les étapes de son raisonnement pour parvenir à la réponse.

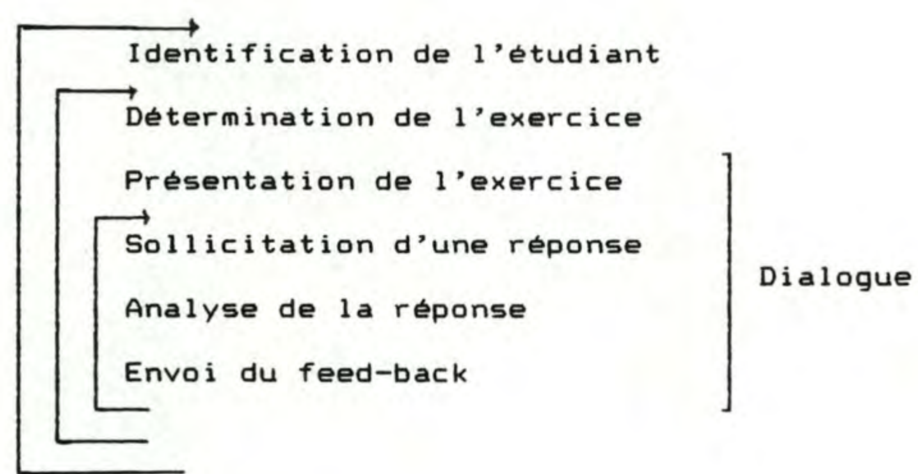
A tout moment, le système peut décider de présenter un nouvel exercice si la réponse est correcte, ou bien si le raffinement de la réponse est parvenu à un stade terminal d'explication (la machine, après avoir dégager les erreurs, ne peut aller plus loin). Cette succession d'actions relatives à un exercice constitue un dialogue, c'est-à-dire une suite d'échanges d'informations entre le système d'E.A.O. et l'apprenant. Un dialogue particulier est propre à un individu et est relatif à un exercice. Après chaque dialogue, le système détermine l'exercice suivant à présenter à l'élève.

Schéma d'un dialogue :



Un système d'E.A.O. est destiné à être utilisé par plusieurs apprenants. Chaque élève peut dialoguer avec le système à différents moments. Les exercices proposés, ainsi que leur nombre, leur type et leur succession varieront en fonction de chaque étudiant.

Schéma de l'utilisation du système d'E.A.O. par les apprenants :



Avant le premier dialogue, la machine peut évaluer rapidement les connaissances de l'étudiant, relatives à l'enseignement prodigué par le système (par exemple, sous forme de QCM) : cela permet à la machine de vérifier si l'apprenant maîtrise les pré-requis indispensables à l'utilisation du système d'E.A.O. En effet, l'élève réussira mieux dans les nouveaux apprentissages s'il maîtrise suffisamment les notions de base de la matière [cfr 1]. Si l'étudiant ne les domine pas assez, on peut soit l'écarter du système, ce qui permet de limiter les stratégies pédagogiques à envisager (surtout si les lacunes sont importantes), soit de lui accorder l'accès si la machine peut envisager les déficits particuliers du sujet.

Avant de présenter des exercices à l'apprenant, il est important de vérifier s'il ne domine pas déjà certaines connaissances que le système veut lui faire acquérir. Dans ce cas, la machine pourra lui faire sauter les exercices correspondants ou simplement rafraîchir ses connaissances. Elle se servira de ce niveau de maîtrise pour élaborer ses décisions de remédiation et pour réguler la progression de l'élève. "Le passage d'une partie du cours à une autre ne pourra se faire que si l'élève possède les connaissances requises minimums capables de lui assurer le succès par la suite" [cfr 1].

En fin d'apprentissage, la machine peut interroger l'apprenant sous forme de bilan, afin de lui montrer ses faiblesses ainsi que les connaissances qu'il a acquises grâce aux dialogues avec le système.

1.2. Description d'un dialogue.

1.2.1. Présentation de l'exercice et sollicitation d'une réponse.

Elles sont toutes deux influencées par les stratégies pédagogiques envisagées par la machine. Celles-ci définissent les modalités de l'enseignement prodigué par le système. Les stratégies pédagogiques induisent le type de raisonnement que l'apprenant devra utiliser pour résoudre les exercices. Par conséquent, pour que le sujet maîtrise bien la matière, il est nécessaire d'adapter l'enseignement à ses caractéristiques (par exemple à son type de raisonnement, etc ...).

Le système peut donc utiliser différentes approches; l'enseignement peut être très prescriptif ou laisser une grande initiative à l'élève, il peut être de type expositif ou par redécouverte, etc ... [cfr 1]. L'adaptation de l'enseignement au sujet facilite l'apprentissage de celui-ci.

La stratégie pédagogique est directement matérialisée par le type de stimulus-réponse (type de présentation, sollicitation et nature de la réponse) utilisé par la machine lors de chaque dialogue. Celui-ci peut aussi être utilisé pour surmonter certaines lacunes. De plus, certains exercices nécessitent, par exemple, une approche de type analytique. Il faut donc amener l'élève dans le mode de raisonnement souhaité par une stratégie pédagogique adéquate. La nature de la réponse (ex : construite ou choisie, guidée ou non guidée) reflétera directement ce type de raisonnement.

1.2.2. Analyse de la réponse.

L'analyse de la réponse a pour but de détecter les erreurs commises par l'apprenant et de déterminer l'endroit où elles se situent. La complexité de l'analyse varie en fonction du type de la réponse. Par exemple, un QCM est plus facile à corriger que des exercices dont les réponses sont élaborées par l'apprenant en langage naturel. Dans le premier cas, les réponses sont de type "simple" et seront donc uniquement bonnes ou mauvaises.

Dans le deuxième cas, le mécanisme d'analyse sera le suivant :

Si la solution de l'étudiant correspond à la réponse correcte (générée ou mémorisée par le système), la machine considère que l'élève a bien répondu et cherchera un autre exercice à lui présenter.

Dans le cas contraire, la machine va essayer de déterminer à quelle étape de son raisonnement l'étudiant aurait pu commettre une erreur. Elle demande alors à l'apprenant d'expliquer sa réponse. Par un jeu de questions-réponses, celui-ci détaillera son raisonnement. Grâce à ses connaissances de la matière, la machine va essayer de le décomposer de plus en plus finement pour en dégager les erreurs jusqu'à, éventuellement, atteindre un stade terminal, niveau au-delà duquel la machine ne peut plus fournir d'explication. Cela peut correspondre par exemple, à un pré-requis non maîtrisé.

Il n'est pas nécessaire de parvenir à un stade terminal; si l'étudiant a commis une erreur de calcul, une faute de frappe, ou encore grâce au feed-back renvoyé par la machine, il parvient de lui-même à rectifier sa réponse, il pourra répondre correctement aux questions suivantes. Le système peut alors lui présenter d'autres exercices.
A chaque étape d'explicitation de la réponse, la machine peut dégager de nouvelles erreurs.

On pourrait également effectuer une décomposition systématique des réponses de l'élève même quand elles sont correctes (en effet, l'étudiant peut malgré tout commettre des erreurs lors des différentes étapes du raisonnement menant à la réponse finale). Cependant, cela peut être fastidieux pour l'apprenant. Aussi cette décomposition systèmatique pourrait-elle dépendre de la maîtrise des concepts par le sujet, certaines séquences pouvant être sautées si l'élève domine les concepts suffisamment.

1.2.3. Envoi d'un feed-back.

Le feed-back constitue un mécanisme de retour d'informations vers l'apprenant basé sur l'analyse de la réponse de celui-ci. "Le feed-back joue un rôle correctif et informatif" [cfr 1]. En effet, il est nécessaire de transmettre des informations adaptées à chaque réponse afin de permettre le redressement des erreurs. Il ne suffit pas de faire remarquer à l'étudiant qu'il s'est trompé, mais il faut lui dire en quoi et où il s'est trompé, les erreurs ayant une importance variable en fonction de la matière enseignée. "Le feed-back permet un guidage ponctuel autorisant une progression efficace de l'élève vers le comportement souhaité grâce à un contrôle précis de l'apprentissage" [cfr 1].

1.3. Détermination des exercices à présenter.

Dans le domaine enseigné par le système d'E.A.O., un exercice n'a de sens que s'il contient des objectifs pédagogiques ou concepts en relation avec les objectifs terminaux du système. La machine peut donc tenter d'évaluer l'évolution du comportement de l'apprenant vers ces objectifs en analysant toutes les réponses de celui-ci. Il vaut mieux se baser sur la réussite concernant un concept plutôt qu'un cas particulier, car la réponse à un seul exercice n'est pas un critère suffisant pour prendre des décisions conduisant parfois l'élève à sauter des parties entières du cours. Il est alors nécessaire de présenter à l'apprenant plusieurs exercices contenant ces mêmes concepts [cfr 2]. Le système ne peut donc se soucier uniquement de la dernière réponse, mais il doit tenir compte de l'ensemble des réponses.

Il est indispensable d'adapter le niveau de difficulté des exercices à chaque apprenant. La machine doit faire progresser leur complexité tout en maintenant à un niveau élevé la motivation des élèves (les exercices ne doivent être ni trop difficiles ni trop faciles). Si l'étudiant ne maîtrise pas certains concepts, le système peut lui présenter des exercices de complexité moindre (de façon à lui permettre de déceler et de comprendre ses erreurs). Dans l'autre cas, la machine présentera des exercices de plus en plus compliqués. Elle s'assurera, avant de passer à une nouvelle partie du cours, que l'étudiant maîtrise suffisamment les connaissances nécessaires à la compréhension des nouvelles matières.

1.4. Exemple de dialogue : décomposition d'une réponse dans le cas d'une résolution de fraction.

Système d'E.A.O.	Apprenant	Feed-back
Ex. 1 : $\frac{9}{12} + \frac{6}{7} = ?$	$\frac{9}{11}$	Cette réponse n'est pas correcte !
Expliquez comment vous trouvez 9/11 à partir de 9/12 + 6/7 :	$\frac{3}{4} + \frac{6}{7}$	Cette réponse est correcte.
Expliquez comment vous trouvez 9/11 à partir de 3/4 + 6/7 :	$\frac{3 + 6}{4 + 7}$	Cette réponse n'est pas correcte : (réduction au même dénom.)

=> Erreur terminale.

A ce stade d'explication de l'erreur, le système pourrait proposer des réductions de fractions au même dénominateur (concept non maîtrisé par l'apprenant) comme exercices suivants, ou bien redemander le même exercice (ou un exercice équivalent) après avoir expliqué l'erreur et donné éventuellement un exemple.

2. Définition et buts de l'historique.

Dans un enseignement adaptatif, l'historique constitue un ensemble d'informations échangées entre le système et les apprenants, pouvant être utiles à la fois aux enseignants et à la machine afin de déterminer les apports pédagogiques du système d'E.A.O., ainsi que la maîtrise et les faiblesses des apprenants dans le domaine enseigné.

2.1. Importance de l'historique pour le système d'E.A.O.

En exploitant les indices révélés par l'apprenant lors des dialogues antérieurs, le système d'E.A.O. peut tenter une évaluation de l'évolution du comportement de l'étudiant vers les objectifs poursuivis. Cette évaluation tiendra compte de l'ensemble des réponses de l'élève pour déterminer la compréhension des concepts présentés à celui-ci (d'où l'intérêt de l'historique). De même, l'ajustement des situations pédagogiques, c'est-à-dire l'adaptation des exercices à chaque apprenant, devrait dépendre de l'ensemble des comportements manifestés par l'élève tout au long des dialogues avec le système.

Il ne suffit pas de mémoriser l'adéquation ou non de la réponse à la question, il faut en plus, dans le second cas, déterminer les concepts sous-jacents non compris. La machine se basera sur l'origine des erreurs qu'elle a décelées lors des dialogues antérieurs afin de tenter une remédiation.

Le système d'E.A.O. devra déterminer quelles situations pédagogiques seront proposées à l'apprenant. La machine trouvera dans l'historique tous les renseignements nécessaires et devra mettre à jour celui-ci régulièrement après chaque dialogue.

2.2. Importance de l'historique pour les enseignants.

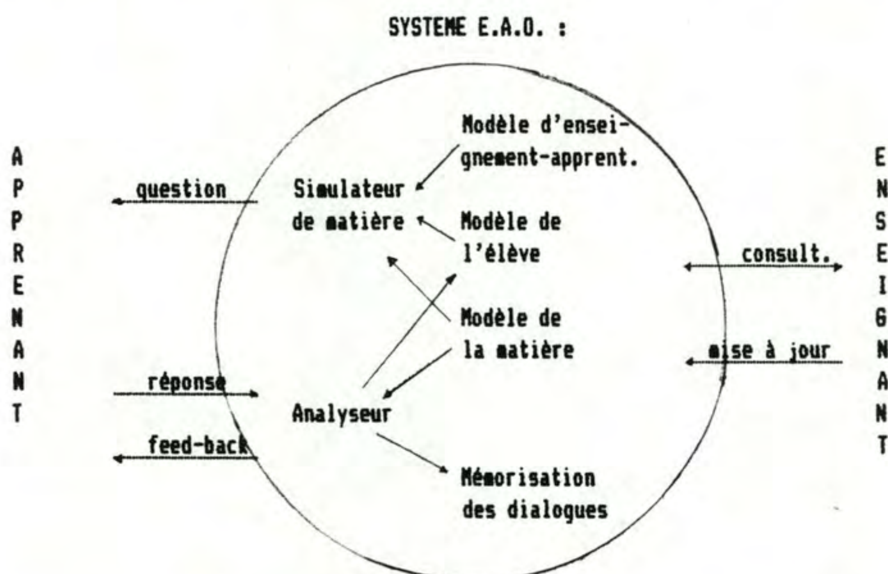
En consultant l'historique des échanges entre le système et les apprenants, l'enseignant peut évaluer les points forts et faibles de chaque élève, relatifs aux différents concepts de base mis en oeuvre par les objectifs du système d'enseignement. Il peut ensuite, lors de son cours, insister sur tel ou tel aspect de la matière. De même, chaque apprenant peut accéder à l'historique afin de recevoir des informations le concernant. La machine offre à l'étudiant la possibilité d'analyser sa propre compréhension de la matière (auto-évaluation) et de savoir comment il est perçu par le système d'E.A.O.

Cette consultation permet aussi une évaluation de l'action de formation en plus de l'évaluation de l'élève. L'enseignant peut alors vérifier l'adéquation ou non des différentes situations pédagogiques présentées aux apprenants. Les stratégies pédagogiques envisagées par le système pouvant aussi être mieux adaptées aux étudiants en fonction des indices révélés par les échanges antérieurs.

L'historique permet une étude des caractéristiques de la population visée, ainsi que des sous-populations qui la composent. L'enseignant doit pouvoir obtenir du système des informations relatives à l'apprentissage global des élèves : par exemple, les erreurs commises, la fréquence et le type des erreurs [cfr 2]. Il doit pouvoir aussi identifier leur origine (mauvaise préreprésentations, connaissances préalables insuffisantes, etc...) dans le but d'y remédier.

3. Descriptions des modules d'un système d'E.A.O.

Les modules d'un système d'E.A.O. sont relativement indépendants et caractérisent chacun une étape spécifique du processus d'enseignement et d'apprentissage. Ces modules sont plus ou moins développés suivant le type du système d'E.A.O.



L'apprenant reçoit une question du système d'E.A.O. et lui fournit une réponse. L'analyseur étudie celle-ci, renvoie un feed-back à l'étudiant en fonction des erreurs qu'il a détectées et met à jour le modèle de l'élève. Le dialogue en cours est mémorisé par la machine. Le simulateur de matière se sert des modèles d'enseignement-apprentissage, de l'élève et de la matière pour élaborer l'exercice suivant ainsi que les stratégies pédagogiques qui seront proposées à l'étudiant.

L'enseignant peut observer le déroulement des dialogues entre la machine et l'étudiant, consulter les différents modèles et éventuellement les mettre à jour (pour mieux adapter le système aux apprenants).

3.1. Modèle de la matière.

Le modèle de la matière représente les connaissances du système d'E.A.O. relatives à l'enseignement prodigué aux apprenants. La machine se base sur ce modèle pour analyser les réponses de l'élève ainsi que pour déterminer les situations pédagogiques suivantes.

- Le domaine :

Le domaine représente la matière que l'on désire enseigner aux apprenants. Cette matière doit être bien délimitée et structurée de façon à pouvoir dégager très facilement les objectifs et/ou les concepts pédagogiques qui doivent être atteints par les apprenants.

- Les exercices :

L'enseignement de cette matière est constitué par un ensemble d'exercices ou situations pédagogiques destinés à être présentés aux élèves. Les exercices forment les éléments qui serviront de base aux objectifs pédagogiques poursuivis. Ils peuvent être spécifiés explicitement par l'enseignant et mémorisés tels quels dans le domaine, ou bien paramétrés et générés automatiquement par le système d'E.A.O.

- Les concepts :

Pour avoir un quelconque intérêt, chaque exercice doit refléter un ou plusieurs concepts sous-jacents au domaine enseigné (certains concepts étant plus difficiles à assimiler par l'apprenant que d'autres). D'aucuns correspondront directement aux objectifs du système d'enseignement. De ce fait, ils constituent un but que chaque élève devra atteindre. D'autres seront associés à des pré-requis dont la maîtrise est indispensable, mais qui ne sont pas, à proprement parler, directement associés à un objectif du système. Les réponses de l'apprenant permettent à la machine d'analyser l'évolution du comportement du sujet vers les objectifs pédagogiques en dégageant les concepts sous-jacents aux exercices plus ou moins bien maîtrisés par l'élève.

- Les classes :

Tous les exercices sont regroupés en classes d'exercices semblables [cfr 2]. Une classe contient tous les exercices présentant les mêmes concepts sous-jacents. Pour évaluer la compréhension d'un apprenant concernant certains concepts, il est nécessaire de lui présenter plusieurs exercices les contenant. La notion de classe facilite la recherche de tels exercices. De plus, pour chacune, on peut élaborer les types d'erreurs possibles afin de déterminer plus facilement les problèmes d'un apprenant pour un ensemble de situations pédagogiques similaires.

- Le niveau de complexité :

La complexité caractérise la difficulté de résolution d'un exercice. Elle dépend des concepts sous-jacents à l'exercice et de la nature de celui-ci (les exercices d'une même classe ayant le même niveau de complexité). La complexité des concepts sous-jacents permet une progression de la difficulté lors de la présentation des exercices aux apprenants; il faut leur donner des exercices ni trop difficiles, ni trop faciles, tout en assurant un niveau de réussite assez élevé. Lorsqu'un élève a réussi les exercices d'une classe et qu'il maîtrise les concepts indispensables, il peut passer à des séquences plus difficiles. Un même exercice peut figurer dans plusieurs classes, tout dépend de l'importance accordée à chaque concept sous-jacent.

3.2. Modèle de l'élève.

Le "modèle" de l'élève représente la description à un moment donné de l'apprenant sur laquelle se base le système d'E.A.O. pour déterminer les stratégies pédagogiques à utiliser et les situations pédagogiques à présenter à l'apprenant. Ce modèle décrit, à tout moment, la maîtrise de chaque étudiant concernant le domaine enseigné. Celle-ci est soumise à une constante réévaluation en fonction des dialogues entre le système et l'élève.

Chaque connaissance de la synthèse précise le degré de maîtrise d'un concept particulier. Ce degré de maîtrise caractérise principalement le niveau d'évolution de l'apprenant vers les objectifs terminaux du système. Il dépend des réponses de l'élève aux exercices proposés, qui contiennent ce concept, mais il sera aussi influencé par la nature et la complexité des exercices. Les exercices les plus récents auront plus d'importance que les anciens pour éviter de tenir compte des "erreurs de jeunesse". Le système d'E.A.O. précisera lui-même, après chaque dialogue, les modifications à apporter aux connaissances relatives aux concepts sous-jacents de l'exercice proposé. Il est facile de déterminer si l'élève maîtrise une classe ou un niveau de complexité, il suffit pour cela de vérifier si tous les concepts de cette classe ou de ce niveau de complexité sont bien maîtrisés.

3.3. Descriptions des dialogues.

Cette partie de l'historique constitue l'enregistrement systématique par ordre chronologique des données relatives aux dialogues entre le système et les apprenants. La mémorisation de toutes les interactions entre la machine et l'élève ne sert que de façon ponctuelle; l'enseignant et le système ne s'y reporteront que pour analyser certains points délicats exprimant les faiblesses de l'étudiant.

- **Les étudiants :**

Ceux-ci doivent être mémorisés dans l'historique, car le système doit savoir qui peut converser avec lui, quels sont les dialogues antérieurs éventuels ainsi que la synthèse des connaissances du sujet.

- **Les échanges :**

Les échanges décrivent toutes les informations échangées entre le système et les apprenants; un échange caractérise un dialogue particulier. Lors de tout dialogue, la machine détermine le ou les concepts non maîtrisés par l'apprenant en analysant les réponses fournies par celui-ci; ces concepts sont associés à l'échange correspondant. Les informations mémorisées à l'occasion d'un échange serviront à la fois au système et à l'enseignant pour observer le déroulement des dialogues, mais aussi pour dégager les aptitudes de l'apprenant, mettre à jour la synthèse des connaissances de celui-ci et déterminer l'adéquation des situations et stratégies pédagogiques. Certaines informations spécifiques aux performances de l'étudiant par rapport aux objectifs du système peuvent être mémorisées dans les échanges (par exemple, le temps de réponse) lorsqu'elles soulignent le comportement de l'élève face à la situation d'apprentissage.

3.4. Modèle d'enseignement-apprentissage.

Le modèle d'enseignement-apprentissage précise les modalités de l'enseignement à prodiguer à chaque apprenant en fonction des caractéristiques du sujet, c'est-à-dire l'adaptation du type d'enseignement au type d'apprentissage de l'élève. Ce modèle contient les règles qui utilisent le degré de maîtrise des concepts, les caractéristiques des exercices (complexité conceptuelle, ...) et les particularités de l'apprenant (motivation, type de raisonnement, ...) pour déterminer les stratégies et les situations pédagogiques adéquates à présenter à l'élève. En effet, un étudiant réussira plus facilement les exercices dont le type de résolution correspond à son raisonnement habituel. Dans d'autres cas, le système voudra améliorer certaines particularités présentant des lacunes [cfr 1].

Cependant, les variables intervenant dans les processus de raisonnement, la motivation, etc... de l'apprenant sont très difficiles à définir et à mettre en oeuvre. De plus, il n'existe pas encore de théorie réellement exploitable pour l'enseignement assisté par ordinateur, quantifiant ces variables afin de définir un modèle d'enseignement-apprentissage utilisable. Dans ce mémoire, l'adaptation aux caractéristiques individuelles sera réduite à l'analyse de la maîtrise des concepts sous-jacents aux exercices présentés à l'élève. On ne pourra donc se baser que sur une manifestation d'une connaissance et non pas sur un comportement.

Néanmoins, il est nécessaire de distinguer connaissance et savoir-faire. Ce n'est pas parce qu'un étudiant peut résoudre correctement des exercices d'un certain type, qu'il possède réellement les connaissances nécessaires à la maîtrise des concepts sous-jacents à ceux-ci.
Par conséquent, le système, tel qu'il est décrit, ne peut donc que se baser sur des présomptions quant aux connaissances de l'apprenant relatives à la matière enseignée. Ces suppositions seront d'autant plus fondées que le nombre d'exercices proposés sera élevé. Cependant la qualité et la complexité des exercices interviennent car la quantité seule ne suffit pas à s'assurer que le sujet maîtrise bien les concepts sous-jacents.

Dans un développement ultérieur, le système de gestion de l'historique pourrait être amélioré de façon à tenir compte de caractéristiques autres que la maîtrise de la matière par l'apprenant. La machine pourra mieux adapter situations et stratégies pédagogiques à chaque élève.

4. Utilisation du système de gestion de l'historique.

La gestion de l'historique est destinée à être utilisée à la fois par le système d'E.A.O. lors des conversations avec les apprenants et ultérieurement par l'enseignant. Aussi ce chapitre sera-t-il divisé en deux parties : l'une décrira les interactions entre la machine et l'élève, l'autre la communication avec l'enseignant. Les traitements décrits ci-après correspondent aux opérations qui devront être effectuées par le système de gestion de l'historique à la demande de l'enseignant ou de la machine.

4.1. Utilisation de l'historique par le système d'E.A.O.

Les traitements suivants sont divisés en trois parties car ils portent chacun sur des données spécifiques différentes de l'historique avec des interactions éventuelles entre eux (voir modèles du chap.3).

- **Enregistrement des dialogues.**

Lors de chaque dialogue avec un apprenant, le système doit enregistrer les informations correspondantes dans l'échange en cours. La machine devra déterminer, par l'analyse des réponses, les concepts sous-jacents à l'exercice non maîtrisés par l'étudiant et qui seront associés à cet échange.

- **Détermination des exercices.**

La machine se sert de la synthèse des connaisances de l'apprenant pour déterminer les exercices suivants à présenter. Chaque connaissance précise le degré de maîtrise d'un concept. La machine choisit les concepts à proposer à l'élève ainsi que les classes les contenant. Ce choix dépend en grande partie du modèle d'enseignement-apprentissage (voir 3.4.). Chaque classe rassemble des exercices de même complexité contenant des concepts identiques. Le système devra proposer plusieurs exercices similaires à l'élève.

Si l'élève rate les exercices d'un certain type, la machine peut lui proposer des exercices de complexité moindre, mais présentant un ou plusieurs concepts qu'il ne maîtrisait pas. Dans l'autre cas, la machine peut, soit lui donner des exercices plus difficiles (contenant des concepts déja vus), soit lui proposer des exercices dont certains concepts sous-jacents ne lui ont jamais été soumis.

- Mise à jour de la synthèse des connaissances de l'apprenant.

En fonction des réponses de l'élève, le système met à jour la synthèse des connaissances de celui-ci en modifiant le degré de maîtrise relatif aux concepts présentés. Lorsqu'il s'agit d'un nouveau concept n'ayant pas été soumis à l'élève auparavant, la machine doit introduire dans l'historique la connaissance correspondante avec son degré de maîtrise. La synthèse est donc en constante évolution et reflète à tout moment l'opinion du système d'E.A.O. sur les connaissances de l'apprenant relatives à l'enseignement prodigué.

4.2. Utilisation de l'historique par l'enseignant.

Ici aussi, on peut distinguer trois types de traitements en fonction des données sur lesquelles l'enseignant travaille et du moment pendant lequel il les utilise. Cette structuration se base principalement sur les modèles spécifiés au chapitre précédent.

- Mise à jour des étudiants.

L'enseignant spécifie lui-même les étudiants qui peuvent converser avec le système. A tout moment, il peut introduire de nouveaux apprenants ou bien en retirer de l'historique, ceux-ci présentant des lacunes trop importantes, ou bien maîtrisant suffisamment les objectifs pédagogiques poursuivis par le système d'E.A.O. Seuls les élèves figurant dans l'historique pourront dialoguer avec le système.

- Mise à jour du domaine.

L'enseignant doit essayer d'adapter l'enseignement du système aux caractéristiques des apprenants qui doivent converser avec celui-ci. Par exemple, le système doit proposer des concepts abordables aux étudiants et les exercices qui leur seront présentés ne doivent pas être trop compliqués pour eux. L'enseignant peut donc, à tout moment, modifier le domaine d'enseignement en ajoutant ou supprimant des exercices, des classes, des concepts ou des niveaux de complexité ou en modifiant les relations entre ceux-ci.

Toutefois, les modifications du domaine peuvent altérer la cohérence des informations; en effet, si, par exemple, un exercice est supprimé, cela rend inepte tout échange durant lequel il aurait été effectué. La suppression visera essentiellement à empêcher toute utilisation ultérieure des informations. Elles doivent cependant rester disponibles afin de conserver la signification des échanges et des connaissances mémorisées antérieurement, en vue d'une consultation de l'historique.

- Analyse de l'historique.

L'analyse de l'historique a pour but de fournir à l'enseignant des renseignements relatifs au fonctionnement général du système et à l'utilisation de celui-ci par les apprenants. On peut donc distinguer deux types d'informations : les données spécifiques à un étudiant particulier et les données propres à l'ensemble des élèves. De la même façon, on distinguera les traitements en fonction des informations qu'ils recherchent.

a. Informations spécifiques à un apprenant.

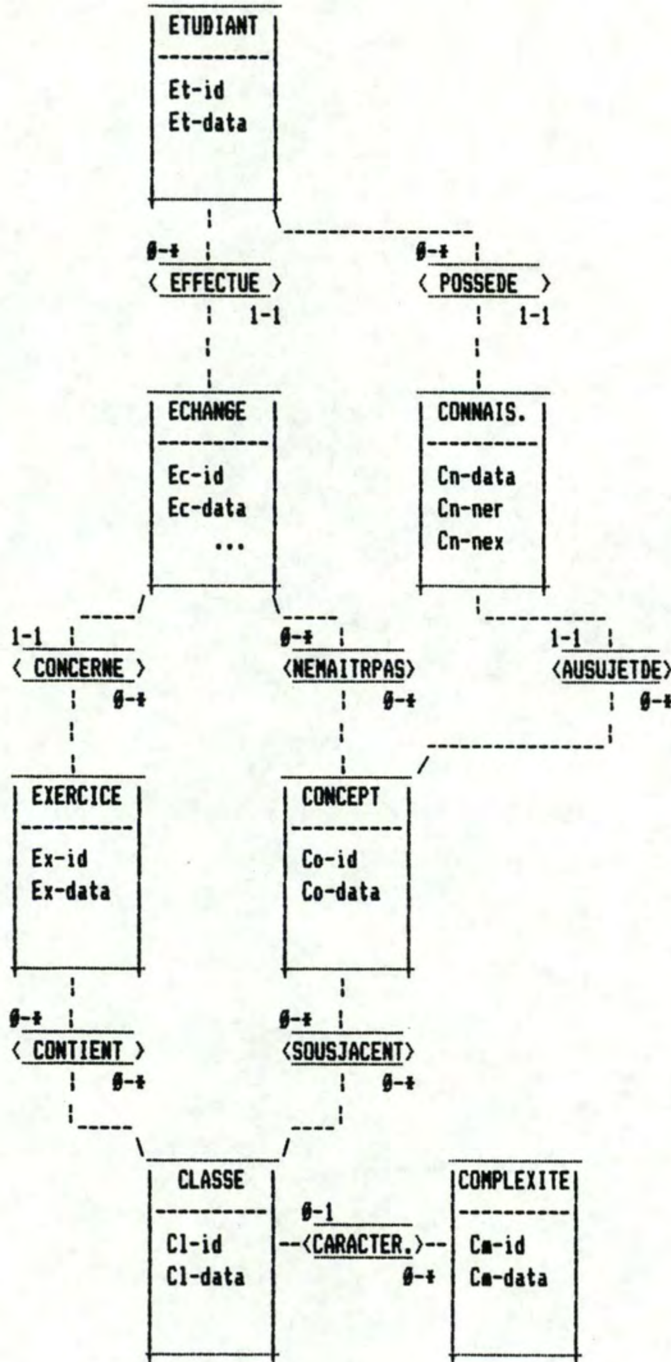
L'enseignant peut utiliser l'historique pour déterminer les connaissances et aptitudes acquises par l'apprenant au sujet de la matière que le système enseigne. Il consultera donc principalement la synthèse. Pour les points jugés délicats, l'enseignant peut analyser de façon plus systèmatique les dialogues de l'élève. Il pourra recevoir ainsi des informations telles que le nombre d'exercices réussis ou non d'une classe spécifique, les erreurs les plus fréquentes de l'élève concernant les concepts présentés, etc...

b. Informations relatives à l'ensemble des apprenants.

Les informations recueillies dans l'historique doivent permettre à l'enseignant d'analyser le système en fonction des apprenants qui dialoguent avec lui. L'enseignant peut par exemple étudier l'adéquation des situations pédagogiques proposées (le taux de réussite des étudiants face aux concepts présentés indique si les exercices sont trop faciles ou trop difficiles). En analysant le déroulement des dialogues, l'enseignant peut vérifier l'impact des stratégies pédagogiques sur l'apprentissage des élèves. Il pourra recevoir des statistiques variées telles que le pourcentage d'étudiants ayant réussi tel type d'exercices, les erreurs les plus fréquentes, etc ... L'analyse de l'historique facilite l'étude des populations des apprenants dialoguant avec le système d'E.A.O.

5. Description des données.

5.1. Schéma Entité-Association.



5.2. Le dialogue.

5.2.1. Type d'entité ETUDIANT.

Définition : tout apprenant autorisé à utiliser le système d'E.A.O. pour dialoguer avec celui-ci dans le but d'améliorer ses connaissances et ses aptitudes dans le domaine enseigné par le système.

Attributs :

a. ET-ID :

Définition : identifiant de l'étudiant caractérisant un apprenant parmi l'ensemble des apprenants habilités à dialoguer avec le système d'E.A.O.

Propriétés : simple, obligatoire.

Type de valeur : alphanumérique.

Remarque : cet attribut pourrait, par exemple, être composé du nom et du prénom de l'étudiant.

b. ET-DATA :

Définition : renseignements divers concernant l'étudiant.

Propriétés : simple, décomposable, facultatif.

Type de valeur : alphanumérique.

Remarques : cet attribut, caractérisant l'apprenant, peut servir de base à l'obtention de statistiques diverses (ex. : classe, section, ...) à des fins de classification des apprenants.

5.2.2. Type d'entité ECHANGE.

Définition : informations décrivant un dialogue entre le système et l'étudiant. L'ensemble des échanges d'un étudiant constitue l'enregistrement systématique par ordre chronologique des données caractérisant les dialogues de celui-ci.

Attributs :

a. EC-ID :

Définition : numéro d'ordre chronologique de l'échange. Ce numéro, avec l'identifiant (ET-ID) d'un étudiant identifie un échange particulier.

Propriétés : simple, élémentaire, obligatoire.

Type de valeur : numérique.

b. EC-DATA :

Définition : renseignements divers décrivant l'échange (voir attributs ci-après).

Propriétés : répétitif, décomposable, obligatoire.

Type de valeur : alphanumérique.

Remarques : cet attribut est en fait un groupe répétitif d'attributs. Chaque groupe caractérise une étape dans le processus de décomposition et d'explicitation des réponses de l'étudiant lors du dialogue correspondant.

EC-DATA se décompose en les attributs suivants:

b1. EC-SOL :

Définition : information exprimant la sollicitation d'une réponse de l'apprenant lors du dialogue en cours.

Propriétés : simple, élémentaire, facultatif.

Type de valeur : alphanumérique.

b2. EC-REP :

Définition : réponse de l'étudiant à la sollicitation du système.

Propriétés : simple, élémentaire, obligatoire.

Type de valeur : alphanumérique.

b3. EC-FEED-BACK :

Définition : informations renvoyées à l'étudiant après évaluation de chaque réponse.

Propriétés : simple, élémentaire, facultatif.

Type de valeur : alphanumérique.

Ces trois attributs décrivent le déroulement de chaque étape d'un dialogue. La sollicitation d'une réponse et le feed-back renvoyé constituent, avec la présentation de l'exercice, le seul interface du système d'E.A.O. visible par l'apprenant. Leur mémorisation dans l'historique permettra à l'enseignant de vérifier l'adéquation des stratégies pédagogiques et des mécanismes d'analyse de réponses employées par la machine. Ces deux derniers attributs sont utiles lorsque l'enseignant ne peut connaître à l'avance les réactions du système.

b4. EC-INF :

Définition : informations relatives aux performances de l'étudiant. Ce sont des données intéressantes à mémoriser car elles caractérisent le comportement de l'individu face à la situation d'apprentissage. Par exemple, le temps de réponse de l'étudiant dans des exercices de calcul mental, etc...

Propriétés : simple, décomposable, facultatif.

Type de valeur : alphanumérique.

5.2.3. Type d'association EFFECTUE.

Définition : relation existant entre un étudiant et les échanges qu'il a effectués.

Connectivité : à un échange particulier correspond un et un seul étudiant ayant effectué cet échange, tandis qu'un étudiant particulier peut avoir effectué un nombre quelconque d'échanges (Ø, 1 ou plusieurs).

T.A. EFFECTUE (ETUDIANT, ECHANGE) = (Ø-*, 1-1)

5.2.4. Type d'association CONCERNE.

Définition : relation existant entre un échange et l'exercice présenté à l'étudiant lors de cet échange.

Connectivité : à un échange particulier correspond un et un seul exercice effectué, tandis qu'à un exercice peut correspondre un nombre quelconque d'échanges (Ø, 1 ou plusieurs).

T.A. CONCERNE (ECHANGE, EXERCICE) = (1-1, Ø-*).

5.2.5. Type d'association NE-MAITRISE-PAS.

Définition : relation existant entre un échange et un concept sous-jacent à l'exercice présenté à l'apprenant lors de cet échange lorsque l'évaluation de la réponse de l'élève permet au système de déterminer une mauvaise compréhension de ce concept.

Contrainte d'intégrité : un concept non maîtrisé ne peut être associé à un échange que s'il est réellement sous-jacent à l'exercice présenté lors de cet échange.

Connectivité : à un échange particulier peut correspondre Ø, 1 ou plusieurs concepts non maîtrisés, tandis qu'un concept peut être non maîtrisé lors d'un nombre quelconque d'échanges.

T.A. NE-MAITRISE-PAS (ECHANGE, CONCEPT) = (Ø-*, Ø-*).

5.3. Le domaine d'enseignement.

5.3.1. Type d'entité EXERCICE.

Définition : tout exercice (ou situation pédagogique) destiné à être présenté à un étudiant qui devra fournir une ou plusieurs réponses. Chaque exercice représente une unité pédagogique autonome en relation avec le domaine enseigné.

Attributs :

a. EX-ID :

<u>Définition</u> : identifiant de l'exercice, caractérisant un exercice dans le domaine d'enseignement.

<u>Propriétés</u> : simple, élémentaire, obligatoire.

<u>Type de valeur</u> : numérique.

b. EX-DATA :

<u>Définition</u> : renseignements divers décrivant l'exercice. Cet attribut peut, par exemple, contenir l'énoncé de l'exercice.

<u>Propriétés</u> : simple, décomposable, facultatif.

<u>Type de valeur</u> : alphanumérique.

5.3.2. Type d'entité CONCEPT.

Définition : objectif conceptuel sous-jacent aux exercices. Chaque concept correspond à un objectif pédagogique du système ou à un pré-requis et sert de base à la détermination des erreurs de compréhension de l'étudiant par la machine.

Attributs :

a. CO-ID :

<u>Définition</u> : identifiant du concept caractérisant, identifiant celui-ci parmi l'ensemble des concepts du domaine d'enseignement.

<u>Propriétés</u> : simple, élémentaire, obligatoire.

<u>Type de valeur</u> : numérique.

b. CO-DATA :

<u>Définition</u> : renseignements divers décrivant le concept.

<u>Propriétés</u> : simple, décomposable, facultatif.

<u>Type de valeur</u> : alphanumérique.

5.3.3. Type d'entité CLASSE.

Définition : catégorie d'exercices présentant les mêmes concepts de base. La notion de classe facilite la recherche d'exercices similaires.

Attributs :

a. CL-ID :

Définition : identifiant d'une classe, caractérisant celle-ci parmi l'ensemble des classes du domaine d'enseignement.

Propriétés : simple, élémentaire, obligatoire.

Type de valeur : numérique.

b. CL-DATA :

Définition : renseignements divers décrivant la classe d'exercices spécifiée.

Propriétés : simple, décomposable, facultatif.

Type de valeur : alphanumérique.

5.3.4. Type d'entité COMPLEXITE.

Définition : complexité d'une classe d'exercices. Elle caractérise le niveau de difficulté de résolution des exercices d'une classe.

Attributs :

a. CM-ID :

Définition : identifiant du niveau de complexité, caractérisant celui-ci dans le domaine d'enseignement.

Propriétés : simple, élémentaire, obligatoire.

Type de valeur : numérique.

b. CM-DATA :

Définition : renseignements divers décrivant le niveau de complexité.

Propriétés : simple, décomposable, facultatif.

Type de valeur : alphanumérique.

5.3.5. Type d'association CONTIENT.

Définition : relation existant entre un exercice et les classes auxquelles il appartient.

Connectivité : un exercice particulier appartient à un nombre quelconque (Ø,1 ou plus) de classes, tandis qu'une classe spécifique peut contenir un nombre quelconque d'exercices.

T.A. CONTIENT (EXERCICE,CLASSE) = (Ø-*, Ø-*).

5.3.6. Type d'association SOUS-JACENT.

Définition : relation existant entre une classe d'exercices et les concepts sous-jacents à ceux-ci : tous les exercices d'une classe possèdent les mêmes concepts de base.

Connectivité : un concept peut figurer dans un nombre quelconque de classes, tandis qu'une classe d'exercices peut contenir Ø, 1 ou plusieurs concepts.

T.A. SOUS-JACENT (CONCEPT, CLASSE) = (Ø-*, Ø-*).

5.3.7. Type d'association CARACTERISE.

Définition : relation existant entre une classe d'exercices et un niveau de complexité exprimant la difficulté de résolution des exercices de cette classe.

Connectivité : à une classe correspond au plus un niveau de complexité tandis qu'à un niveau de complexité correspond un nombre quelconque de classes d'exercices.

T.A. CARACTERISE (CLASSE, COMPLEXITE) = (Ø-1, Ø-*).

5.4. La synthèse des connaissances.

5.4.1. Type d'entité CONNAISSANCE.

Définition : connaissance relative à la maîtrise d'un concept par l'apprenant à un moment déterminé.

Attributs :

a. Une connaissance particulière est identifiée par l'identifiant de l'étudiant et du concept auxquels elle se rapporte.

b. CN-DATA :

Définition : renseignements relatifs à la connaissance spécifiée. Cet attribut contient le degré de maîtrise du concept correspondant.

Propriétés : simple, décomposable, obligatoire.

Type de valeur : alphanumérique.

c. CN-NER :

Définition : nombre d'exercices réussis par l'apprenant et contenant le concept correspondant à la connaissance spécifiée. Cet attribut sera automatiquement mis à jour par le système de gestion de l'historique.

Propriétés : simple,élémentaire,obligatoire.

Type de valeur : numérique.

d. CN-NEX :

Définition : nombre d'exercices donnés à l'étudiant et contenant le concept correspondant à la connaissance spécifiée. Cet attribut permet facilement de calculer le poucentage de réussite d'un étudiant relatif au concept correspondant.

$$\text{Pourcentage de réussite} = \frac{\text{CN-NER}}{\text{CN-NEX}}$$

Nombre d'exercices ratés = CN-NEX-CN-NER.

Propriétés : simple, élémentaire, obligatoire.

Type de valeur : numérique.

5.4.2. Type d'association POSSEDE.

Définition : relation existant entre un étudiant et la connaissance qu'il possède.

Contraintes d'intégrité :
- il ne peut exister au plus qu'une relation "POSSEDE" entre un étudiant et la connaissance d'un concept (via T.A. AU-SUJET-DE).
- un étudiant ne peut être associé à une connaissance que s'il a déja effectué, lors d'un échange, un exercice contenant le concept correspondant à celle-ci.

Connectivité : à une connaissance particulière correspond un et un seul étudiant, tandis qu'à un étudiant particulier correspond un nombre quelconque de connaissances possédées par celui-ci (mais relatives à des concepts différents).

T.A. POSSEDE (ETUDIANT, CONNAISSANCE) = (Ø-*, 1-1).

5.4.3. Type d'association AU-SUJET-DE.

Définition : relation existant entre une connaissance particulière et le concept correspondant.

Contraintes d'intégrité :

- il ne peut exister au plus qu'une relation "AU-SUJET-DE" entre un concept et la connaissance correspondante propre à un étudiant (via T.A. POSSEDE).
- un concept ne peut être associé à la connaissance d'un étudiant que s'il est sous-jacent à un exercice effectué par cet apprenant lors d'un échange.

Connectivité : à une connaissance particulière d'un étudiant correspond un et un seul concept tandis qu'à un concept peut correspondre un nombre quelconque de connaissances (relatives à des étudiants différents).

T.A. AU-SUJET-DE (CONCEPT, CONNAISSANCE) = (Ø-*, 1-1).

6. Description des fonctions.

La gestion de l'historique sera constituée d'un ensemble de fonctions que l'utilisateur (système d'E.A.O. ou enseignant) emploiera selon son gré. Pour faciliter la définition de ces fonctions, celles-ci sont rassemblées en phases selon la découpe vue en 4. Cependant, cela ne présage nullement de l'utilisation ultérieure des fonctions spécifiées. En effet, pour déterminer les exercices suivants à présenter à un apprenant, le système peut se servir de fonctions définies dans l'analyse de l'historique. Autre exemple : si le système génère lui-même les exercices, ceux-ci doivent être mémorisés lors de chaque échange. La fonction de création d'un exercice, définie dans la phase "Mise à jour du domaine", sera utilisée dans la phase "Mémorisation des dialogues".

Ce chapitre apporte une idée de solution dans l'élaboration du projet en donnant les spécifications de chaque traitement ainsi que les règles de transformations des informations d'input en output. On utilise pour décrire les fonctions, le schéma suivant:

<u>Obj.</u> : objectif à réaliser par la fonction.
<u>M.D.</u> : message-donnée = arguments de la fonction.
<u>Cm.</u> : consultation de la mémoire (ici l'historique).
<u>M.R.</u> : message-résultat (output de la fonction).
<u>Act.</u> : actions sur la mémoire du système (création,modification,suppression de l'historique).
<u>Règle</u> : règles de traitement.

6.1. Mémorisation des dialogues.

Définition : cette phase a pour but d'enregistrer systématiquement, par ordre chronologique dans l'historique, toutes les données relatives aux dialogues entre le système d'E.A.O. et l'apprenant. Elle fournit également toutes les données nécessaires à la mise à jour de la synthèse des connaissances de l'élève.

- CREATION-ECHANGE.

Obj. : enregistrer dans l'historique les données relatives au dialogue effectué.
M.D. : étudiant : ET-ID, échange : EC-ID,données : EC-DATA, {concept : CO-ID}, exercice : EX-ID.
Cm. : T.E. ETUDIANT, T.E. EXERCICE, T.E. ECHANGE, T.A. EFFECTUE, T.A. CONCERNE, T.A. CONTIENT, T.A. SOUS-JACENT.
M.R. : -
Act. : ajout d'une occurence du T.A. EFFECTUE, T.A. CONCERNE, T.E. ECHANGE et éventuellement de une ou plusieurs occurences du T.A. NE-MAITRISE-PAS.
Règle : si l'étudiant et l'exercice existent et que l'échange n'existe pas encore, créer un nouvel échange ainsi que les associations de type EFFECTUE et CONCERNE correspondantes. Si chaque concept spécifié existe et est sous-jacent à l'exercice, alors création d'une association de type NE-MAITRISE-PAS entre l'échange et ce concept. La liste des concepts sous-jacents à l'exercice et celle des concepts non-maîtrisés permet la mise à jour automatique de la synthèse (voir ci-après).

6.2. Détermination des exercices.

Définition : cette phase a pour but de rechercher les exercices qui seront proposés à l'apprenant lors des dialogues ultérieurs; elle dépend d'un modèle d'enseignement-apprentissage (voir 3.4.). Pour cette raison, le système d'E.A.O. sélectionnera lui-même les concepts nécessaires à présenter à l'élève, puis les classes et le niveau de complexité, et enfin les exercices selon des critères qui lui sont propres.

- RECHERCHE-CONNAISSANCES-ETUDIANT.

Obj. : fournir l'ensemble des connaissances possédées par un étudiant.
M.D. : étudiant : ET-ID.
Cm. : T.E.ETUDIANT, T.A.POSSEDE, T.E.CONNAISSANCE, T.A.AU-SUJET-DE, T.E.CONCEPT.
Act. : -
M.R. : {(concept,données,ner,nex) / concept : CO-ID, données : CN-DATA, ner : CN-NER, nex : CN-NEX}.
Règle : si l'étudiant existe, rechercher toutes les connaissances qu'il possède et les concepts correspondants.

- RECHERCHE-CLASSES-COMPLEXITE.

  Obj. : fournir l'ensemble des classes d'un niveau de complexité.
  M.D. : complexité : CM-ID.
  Cm. : T.E.COMPLEXITE, T.E.CLASSE, T.A.CARACTERISE.
  Act. : -
  M.R. : {classe : CL-ID}.
  Règle : si le niveau de complexité spécifié existe, rechercher toutes les classes qui lui sont associées.

- RECHERCHE-CLASSES-CONCEPTS.

  Obj. : fournir l'ensemble des classes contenant les concepts spécifiés ainsi que le niveau de complexité de celles-ci.
  M.D. : {concept : CO-ID}.
  Cm. : T.E.CONCEPT, T.A.SOUS-JACENT, T.E.CLASSE.
  Act. : -
  M.R. : {(classe, complexité) / classe : CL-ID, complexité : CM-ID}.
  Règle : si tous les concepts spécifiés existent, alors rechercher les éventuelles classes qui les contiennent ainsi que le niveau de complexité de celles-ci.

- RECHERCHE-EXERCICES-CLASSE.

  Obj. : fournir tous les exercices appartenant à une classe.
  M.D. : classe : CL-ID.
  Cm. : T.E.CLASSE, T.E.EXERCICE, T.A.CONTIENT.
  Act. : -
  M.R. : {exercice : EX-ID}.
  Règle : si la classe spécifiée existe, rechercher tous les exercices de cette classe.

6.3. Mise à jour de la synthèse.

Définition : cette phase a pour but de mettre à jour et de rectifier l'idée que la machine se fait de la maîtrise, par l'étudiant, des objectifs pédagogiques.

- MISE-A-JOUR-CONNAISSANCE.

  Obj. : mettre à jour les données relatives à une connaissance.
  M.D. : étudiant : ET-ID, concept : CO-ID, données : CN-DATA, degré : CN-NER, nbre-ex : CN-NEX.
  Cm. : T.E.ETUDIANT, T.E.CONNAISSANCE, T.E.CONCEPT, T.A.POSSEDE, T.A.AU-SUJET-DE.
  Act. : modification ou création d'une occurence du T.E.CONNAISSANCE ainsi que des T.A.AU-SUJET-DE et POSSEDE.
  M.R. : -
  Règle : si l'étudiant, le concept et la connaissance correspondante existent, alors modifier les données de celle-ci. Par contre, si la connaissance n'existe pas encore, créer celle-ci avec les données spécifiées ainsi que les associations de type POSSEDE et AU-SUJET-DE correspondantes entre la connaissance, l'étudiant et le concept.

6.4. Mise à jour des étudiants.

Définition : cette phase a pour but de spécifier et de mettre à jour les étudiants qui pourront dialoguer avec le système d'E.A.O.

- CREATION-ETUDIANT.

  Obj. : enregistrer un nouvel étudiant dans l'historique.
  M.D. : étudiant : ET-ID, données : ET-DATA.
  Cm. : T.E.ETUDIANT.
  Act. : ajout d'une nouvelle occurence du T.E.ETUDIANT.
  M.R. : -
  Règle : si l'étudiant spécifié n'existe pas encore, alors créer cet étudiant dans l'historique.

- MODIFICATION-ETUDIANT.

  Obj. : mettre à jour les données relatives à un étudiant.
  M.D. : étudiant : ET-ID, données : ET-DATA.
  Cm. : T.E.ETUDIANT.
  Act. : modification d'une occurrence du T.E.ETUDIANT.
  M.R. : -
  Règle : si l'étudiant spécifié existe, alors lui attribuer les nouvelles données.

- SUPPRESSION-ETUDIANT.

Obj. : retirer un étudiant de l'historique.
M.D. : étudiant : ET-ID.
Cm. : T.E.ETUDIANT, T.A.EFFECTUE, T.E.ECHANGE, T.A.CONCERNE, T.A.NE-MAITRISE-PAS, T.A.POSSEDE, T.E.CONNAISSANCE, T.A.AU-SUJET-DE.
Act. : supprimer une occurrence du T.E.ETUDIANT ainsi que toutes les occurrences des échanges, des connaissances et des associations correspondantes relatives à cet étudiant.
M.R. : -
Règle : si l'étudiant spécifié existe, alors supprimer de l'historique toutes les informations le concernant.

6.5. Mise à jour du domaine.

Définition : cette phase a pour but de structurer le domaine d'enseignement selon les spécifications de l'enseignant. La mise à jour du domaine permet à l'enseignant d'adapter la matière aux apprenants. Cependant, cette mise à jour peut provoquer de grands troubles dans la cohérence des informations (voir 4.2).
Le domaine, contrairement aux échanges, aux connaissances et aux étudiants qui évoluent sans cesse, constitue la partie statique de l'historique. Toute modification ou suppression de celui-ci peut altérer la signification des données mémorisées antérieurement. Aussi, ne permettra-t-on une suppression ou une modification effective des informations du domaine que s'il n'y a aucune relation avec des connaissances ou des concepts. Dans le cas contraire, les informations seront "marquées" pour empêcher toute utilisation ultérieure (sauf consultation des données antérieures) à des fins de présentation aux apprenants. Les suppressions réelles pourront être effectuées sur demande explicite de l'enseignant quand les données ne seront plus liées à la synthèse ou aux échanges des apprenants.

- CREATION-EXERCICE :

Obj. : enregistrer un nouvel exercice dans le domaine d'enseignement.
M.D. : exercice : EX-ID, données : EX-DATA.
Cm. : T.E.EXERCICE.
Act. : ajout d'une nouvelle occurrence du T.E.EXERCICE.
M.R. : -
Règle : si l'exercice spécifié n'existe pas encore, créer celui-ci dans le domaine d'enseignement.

- MODIFICATION-EXERCICE :

Obj. : mettre à jour les données relatives à un exercice.
M.D. : exercice : EX-ID, données : EX-DATA.
Cm. : T.E.EXERCICE, T.A.CONCERNE.
Act. : modification d'une occurrence du T.E.EXERCICE.
M.R. : -
Règle : si l'exercice spécifié existe et qu'il n'est associé à aucun échange, lui attribuer les nouvelles données.

- SUPPRESSION-EXERCICE :

Obj. : retirer un exercice de l'historique.
M.D. : exercice : EX-ID.
Cm. : T.E.EXERCICE, T.A.CONCERNE, T.A.CONTIENT.
Act. : suppression d'une occurrence du T.E.EXERCICE et, éventuellement, d'occurrences du T.A.CONTIENT.
M.R. : -
Règle : si l'exercice spécifié existe et qu'il n'est associé à aucun échange, le supprimer ainsi que tous les chemins d'accès éventuels entre lui et les classes auxquelles il appartient.

- CREATION-CONCEPT :

Obj. : enregistrer un nouveau concept dans le domaine d'enseignement.
M.D. : concept : CO-ID, données : CO-DATA.
Cm. : T.E.CONCEPT.
Act. : ajout d'une nouvelle occurrence du T.E.CONCEPT.
M.R. : -
Règle : si le concept identifié par la valeur de CO-ID n'existe pas encore, alors le créer dans le domaine d'enseignement.

- MODIFICATION-CONCEPT :

Obj. : mettre à jour les données relatives à un concept.
M.D. : concept : CO-ID, données : CO-DATA.
Cm. : T.E.CONCEPT, T.A.AU-SUJET-DE.
Act. : modification d'une occurrence du T.E.CONCEPT.
M.R. : -
Règle : s'il existe un concept identifié par la valeur de CO-ID et si celui-ci n'est associé à aucune connaissance lui attribuer les valeurs de CO-DATA. On supposera qu'un concept associé à un échange ou sous-jacent à un exercice impliqué dans un échange, sera nécessairement associé à une connaissance et réciproquement.

- SUPPRESSION-CONCEPT :

Obj. : retirer un concept de l'historique.
M.D. : concept : CO-ID.
Cm. : T.A.AU-SUJET-DE, T.E.CONCEPT, T.A.SOUS-JACENT.
Act. : suppression d'une occurrence du T.E.CONCEPT et éventuellement de une ou plusieurs occurrences du T.A.SOUS-JACENT.
M.R. : -
Règle : si le concept spécifié existe et n'est associé à aucune connaissance (voir remarque précédente), supprimer toutes les relations entre celui-ci, son niveau de complexité et les classes auxquelles il est sous-jacent.

- CREATION-CLASSE :

Obj. : enregistrer une nouvelle classe d'exercices dans le domaine d'enseignement.
M.D. : classe : CL-ID, données : CL-DATA.
Cm. : T.E.CLASSE.
Act. : ajout d'une nouvelle occurrence du T.E.CLASSE.
M.R. : -
Règle : créer la classe identifiée par la valeur de CL-ID si elle n'existe pas encore dans le domaine.

- MODIFICATION-CLASSE :

Obj. : mettre à jour les données relatives à une classe d'exercices.
M.D. : classe : CL-ID, données : CL-DATA.
Cm. : T.E.CLASSE, T.A.CONTIENT, T.E.EXERCICE,T.A.CONCERNE.
Act. : modification d'une occurrence du T.E.CLASSE.
M.R. : -
Règle : si la classe identifiée par la valeur de CL-ID existe et qu'elle ne contient aucun exercice impliqué dans un échange, lui attribuer les valeurs de CL-DATA.

- SUPPRESSION-CLASSE :

<u>Obj.</u> : retirer une classe du domaine d'enseignement.
<u>M.D.</u> : classe : CL-ID.
<u>Cm.</u> : T.E.CLASSE, T.A.CONTIENT, T.A.SOUS-JACENT, T.A.CARACTERISE, T.E.EXERCICE, T.A.CONCERNE.
<u>Act.</u> : suppression d'une occurrence du T.E.CLASSE et, éventuellement de une ou plusieurs occurrences des T.A.CONTIENT, SOUS-JACENT & CARACTERISE.
<u>M.R.</u> : -
<u>Règle</u> : si la classe spécifiée existe et qu'elle ne contient aucun exercice impliqué dans un échange, alors la supprimer de l'historique ainsi que toutes les relations entre elle, les exercices qu'elle contient, les concepts qui lui sont sous-jacents et le niveau de complexité qui la caractérise.

- CREATION-COMPLEXITE :

<u>Obj.</u> : enregistrer un nouveau niveau de complexité dans le domaine d'enseignement.
<u>M.D.</u> : complexité : CM-ID, données : CM-DATA.
<u>Cm.</u> : T.E.COMPLEXITE.
<u>Act.</u> : ajout d'une nouvelle occurrence du T.E.COMPLEXITE.
<u>M.R.</u> : -
<u>Règle</u> : Si le niveau de complexité identifié par la valeur de CM-ID n'existe pas encore, alors le créer dans le domaine.

- MODIFICATION-COMPLEXITE :

<u>Obj.</u> : mettre à jour les données relatives à un niveau de complexité.
<u>M.D.</u> : complexité : CM-ID, données : CM-DATA.
<u>Cm.</u> : T.E.COMPLEXITE, T.A.CARACTERISE, T.E.CLASSE, T.A.CONTIENT, T.E.EXERCICE, T.A.CONCERNE.
<u>Act.</u> : modification d'une occurrence du T.E.COMPLEXITE.
<u>M.R.</u> : -
<u>Règle</u> : si le niveau de complexité identifié par la valeur CM-ID existe et qu'il n'est associé (via T.A.CARACTERISE, T.E.CLASSE, T.A.CONTIENT) à aucun exercice impliqué dans un échange, lui attribuer les valeurs de CM-DATA.

- SUPPRESSION-COMPLEXITE :

<u>Obj.</u> : retirer un niveau de complexité du domaine d'enseignement.
<u>M.D.</u> : complexité : CM-ID.
<u>Cm.</u> : T.E.COMPLEXITE, T.A.CARACTERISE, T.A.CLASSE, T.A.CONTIENT, T.E.EXERCICE, T.A.CONCERNE.
<u>Act.</u> : suppression d'une occurrence du T.E.COMPLEXITE, et, éventuellement, une ou plusieurs occurrences du T.A.CARACTERISE.
<u>M.R.</u> : -
<u>Règle</u> : si le niveau de complexité existe et qu'il n'est associé (via T.A.CARACTERISE, T.A.CLASSE et T.A.CONTIENT) à aucun exercice impliqué dans un échange, alors le supprimer de l'historique ainsi que toutes les associations entre les différentes classes et celui-ci.

- CREATION-CONTIENT :

<u>Obj.</u> : créer une association entre un exercice et une classe.
<u>M.D.</u> : classe : CL-ID, exercice : EX-ID.
<u>Cm.</u> : T.E.EXERCICE, T.A.CONTIENT, T.E.CLASSE.
<u>Act.</u> : ajout d'une nouvelle occurrence du T.A.CONTIENT.
<u>M.R.</u> : -
<u>Règle</u> : créer une relation entre l'exercice identifié par la valeur de EX-ID et la classe identifiée par la valeur de CL-ID, si les deux entités existent et ne sont pas encore reliées entre elles.

- SUPPRESSION-CONTIENT :

<u>Obj.</u> : supprimer une relation entre un exercice et une classe.
<u>M.D.</u> : exercice : EX-ID, classe : CL-ID.
<u>Cm.</u> : T.A.CONTIENT, T.E.EXERCICE, T.E.CLASSE, T.A.CONTIENT.
<u>Act.</u> : suppression d'une occurrence du T.A.CONTIENT.
<u>M.R.</u> : -
<u>Règle</u> : si l'exercice identifié par la valeur de EX-ID, la classe identifiée par la valeur CL-ID et la relation entre ceux-ci existent et que l'exercice n'est impliqué dans aucun échange, supprimer cette relation entre l'exercice et la classe.

- CREATION-SOUS-JACENT :

<u>Obj.</u> : créer une association entre un concept et une classe.
<u>M.D.</u> : concept : CO-ID, classe : CL-ID.
<u>Cm.</u> : T.E.CONCEPT, T.E.CLASSE, T.A.SOUS-JACENT.
<u>Act.</u> : ajout d'une nouvelle occurrence du T.A. SOUS-JACENT.
<u>M.R.</u> : -
<u>Règle</u> : si le concept identifié par la valeur de CO-ID existe ainsi que la classe identifiée par la valeur de CL-ID et que l'association entre ces deux entités n'existe pas encore, créer cette relation entre le concept et la classe.

- SUPPRESSION-SOUS-JACENT :

<u>Obj.</u> : supprimer une relation entre un concept et une classe.
<u>M.D.</u> : concept : CO-ID, classe : CL-ID.
<u>Cm.</u> : T.E.CONCEPT, T.E.CLASSE, T.A.SOUS-JACENT, T.A.AU-SUJET-DE.
<u>Act.</u> : suppression d'une occurence du T.A.SOUS-JACENT.
<u>M.R.</u> : -
<u>Règle</u> : si le concept et la classe spécifiés existent ainsi que la relation entre les deux et que le concept n'est associé à aucune connaissance, alors supprimer cette relation de l'historique.

- CREATION-CARACTERISE :

<u>Obj.</u> : créer une association entre une classe et un niveau de complexité.
<u>M.D.</u> : complexité : CM-ID, classe : CL-ID.
<u>Cm.</u> : T.E.CLASSE, T.E.COMPLEXITE, T.A.CARACTERISE.
<u>Act.</u> : ajout d'une nouvelle occurrence du T.A.CARACTERISE.
<u>M.R.</u> : -
<u>Règle</u> : si la classe et le niveau de complexité spécifiés existent mais que la classe n'est pas encore associée à un niveau de complexité, alors créer la relation entre les deux.

- SUPPRESSION-CARACTERISE :

Obj. : supprimer une association entre une classe et un niveau de complexité.
M.D. : classe : CL-ID, complexité : CM-ID.
Cm. : T.E.CLASSE, T.E.COMPLEXITE, T.A.CARACTERISE, T.A.CONTIENT, T.E.EXERCICE, T.A.CONCERNE.
Act. : supprimer une occurence du T.A.CARACTERISE.
M.R. : -
Règle : si la classe et le niveau de complexité spécifiés existent ainsi que la relation entre les deux et que la classe ne contient aucun exercice impliqué dans un échange, alors supprimer cette relation.

6.6. Analyse de l'historique.

Définition : cette phase a pour but de fournir à l'enseignant des renseignements concernant l'utilisation du système par les apprenants.

Toutes les fonctions suivantes fournissent des informations propres à un étudiant.

- RECHERCHE-ECHANGES-ETUDIANT :

Obj. : fournir tous les échanges effectués par un étudiant.
M.D. : étudiant : ET-ID.
Cm. : T.E.ETUDIANT, T.A.EFFECTUE, T.A.ECHANGE, T.A.CONCERNE, T.E.EXERCICE.
Act. : -
M.R. : {(échange,exercice) / échange : EC-ID, exercice : EX-ID}.
Règle : si l'étudiant spécifié existe, rechercher tous les échanges qu'il a effectués et les exercices correspondants.

- RECHERCHE-CONCEPTS-ECHANGE :

Obj. : fournir l'ensemble des concepts non maîtrisés lors d'un échange.
M.D. : étudiant : ET-ID, échange : EC-ID.
Cm. : T.E.ETUDIANT, T.A.EFFECTUE, T.E.ECHANGE, T.A.NE-MAITRISE-PAS, T.A.CONCEPT.
Act. : -
M.R. : {concept : CO-ID}.
Règle : si l'étudiant et l'échange correspondant existent, rechercher tous les éventuels concepts non maîtrisés lors de celui-ci.

- RECHERCHE-CONNAISSANCES-ETUDIANT :

voir phase "Détermination des exercices" (6.2).

- RECHERCHE-CLASSES-CONCEPTS :

voir phase "Détermination des exercices" (6.2.). Cette fonction permet de déterminer quelles classes d'exercices un apprenant maîtrise ou ne maîtrise pas (en fonction de concepts jugés intéressants).

- RECHERCHE-CLASSES-EXERCICE :

Obj. : fournir l'ensemble des classes auxquelles appartient un exercice.
M.D. : exercice : EX-ID.
Cm. : T.E.EXERCICE, T.A.CONTIENT.
Act. : -
M.R. : {classe : CL-ID}.
Règle : si l'exercice spécifié existe, rechercher toutes les classes correspondantes.

- RECHERCHE-COMPLEXITE-CLASSE :

Obj. : fournir le niveau de complexité d'une classe.
M.D. : classe : CL-ID.
Cm. : T.E.CLASSE, T.A.CARACTERISE, T.E.COMPLEXITE.
Act. : -
M.R. : {complexite : CO-ID}.
Règle : si la classe spécifiée existe, rechercher le niveau de complexité correspondant.

Les fonctions suivantes donnent des renseignements relatifs à l'ensemble des apprenants.

- RECHERCHE-CONNAISSANCES-CONCEPT :

Obj. : fournir l'ensemble des étudiants et des connaissances relatives à un concept spécifié .
M.D. : concept : CO-ID.
Cm. : T.E.CONCEPT, T.A.AU-SUJET-DE, T.E.CONNAISSANCE, T.A.POSSEDE, T.E.ETUDIANT.
Act. : -
M.R. : {(étudiant, concept, données, ner, nex) / etudiant : ET-ID, données : CN-DATA, ner : CN-NER, CN-NEX}.
Règle : si le concept spécifié existe, rechercher tous les étudiants ainsi que les connaissances correspondantes.

Cette fonction permet facilement de déterminer le pourcentage d'étudiants maîtrisant un concept ou une liste de concepts (propres à une classe ou un niveau de complexité, voir fonctions suivantes).

- RECHERCHE-CLASSES-COMPLEXITE :

  voir phase "Détermination des exercices" (6.2).

- RECHERCHE-CONCEPTS-CLASSES :

  Obj. : fournir l'ensemble des concepts sous-jacents à l'ensemble des classes spécifiées.
  M.D. : classe CL-ID.
  Cm. : T.E.CLASSE, T.A.SOUS-JACENT, T.E.CONCEPT.
  Act. : -
  M.R. : {concept : CO-ID}.
  Règle : si les classes spécifiées existent, rechercher l'ensemble des concepts sous-jacents à ces classes. Cette fonction permet de déterminer les concepts appartenant à une ou plusieurs classes.

Les fonctions suivantes fournissent des renseignements généraux décrivant les différentes données contenues dans l'historique.

- LISTE-ETUDIANTS :

  Obj. : fournir l'ensemble des étudiants figurant dans l'historique.
  M.D. : -
  Cm. : T.E.ETUDIANT.
  Act. : -
  M.R. : {etudiant : ET-ID}.
  Règle : la liste renvoyée contient l'identifiant de tous les étudiants se trouvant dans l'historique.

Les fonctions correspondant aux exercices, concepts, classes et niveaux de complexité peuvent facilement être décrites de cette façon. Il n'y a pas de fonctions LISTE-ECHANGES, ni LISTE-CONNAISSANCES car les échanges et les connaissances dépendent d'un étudiant spécifique.

CONCEPTION LOGIQUE

7. Conception logique.

Cette partie a pour but de définir une implémentation des structures de données et des traitements sur une machine abstraite. Ceux-ci seront regroupés en modules. Le but d'une hiérarchisation en niveaux de modules est d'offrir certaines facilités quant à la conception, la validation et la modification d'un logiciel ainsi que de sa maintenance. Chaque module d'un niveau "utilise" des modules des niveaux inférieurs, c'est-à-dire que le bon fonctionnement d'un module dépend de celui des modules des niveaux inférieurs. La découpe suivante sera employée dans le reste de la conception logique.

- Niveau 6 : modules fonctionnels :

  Les modules de ce niveau proviennent directement des traitements issus de l'analyse fonctionnelle. Ils cachent des traitements complexes qui correspondent aux différentes phases.

- Niveau 5 : modules intermédiaires :

  Du point de vue des données, ce niveau correspond à un interface entre le niveau 6 et le niveau 4, c'est-à-dire entre les traitements complexes et les fonctions élémentaires. Il offre une facilité d'utilisation des fonctions du niveau inférieur. Ce niveau sera constitué de trois modules : expansion, fusion et extraction de sous-listes.

- Niveau 4 : modules élémentaires (I/O) :

  Les modules de ce niveau sont soit des modules de données, soit des modules de traitement élémentaire (qui n'utilisent aucun autre module du système). Les modules de données sont définis par la structure de données qu'ils représentent (T.E. et T.A.) sans considérer l'organisation interne et par les fonctions ou opérations réalisables sur cette structure de données (accès par clé, création, modification, suppression, etc...).

- Niveau 3 : modules de contrôle :

  Ces modules gèrent le comportement dynamique du système; on y trouve les modules coordinateurs et synchronisateurs qui assurent le bon déroulement de l'exécution du système. Les modules de ce niveau ne seront pas définis dans ce mémoire. Toutefois, seront spécifiées les indications relatives à l'ordonnancement des modules décrits par la suite.

- Niveau 2 : modules outils :

  Ce sont des modules offerts par l'O.S., le système de gestion des fichiers ou de base de données, etc... Ils existent dans l'environnement et ne doivent pas être spécifiés. Ces modules ne sont visibles que par leur interface (nom de procédure + liste de paramètres + spécification des effets).

- Niveau 1 : module noyau :

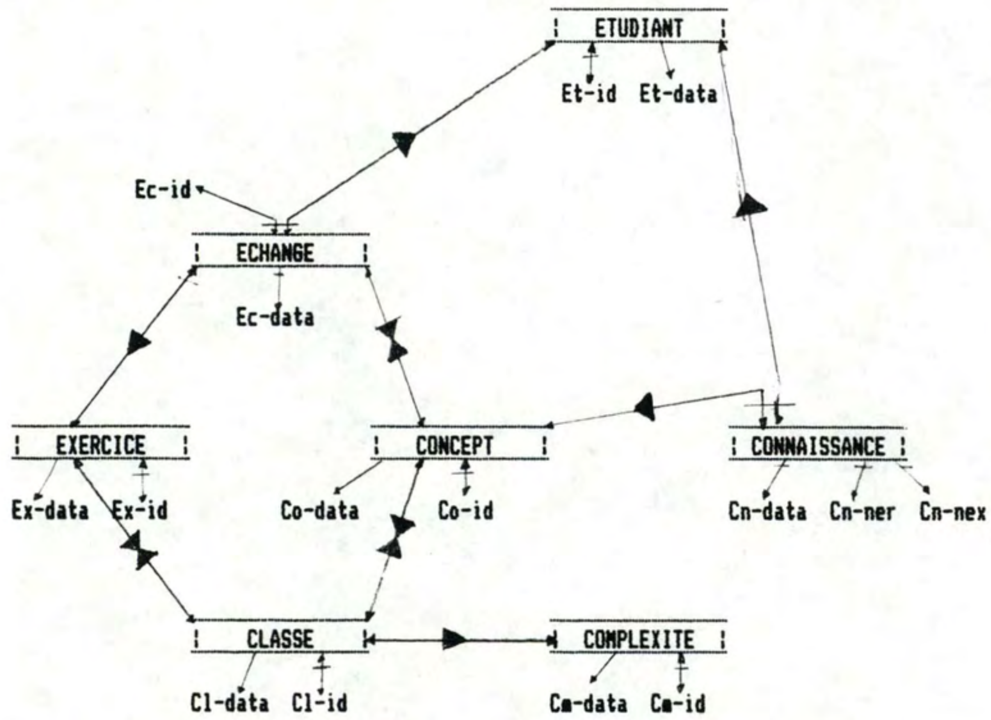
  Ce module correspond à l'O.S. lui-même. Il peut être lui aussi hiérarchisé en niveaux (exemple : hiérarchie de Dijkstra).

- Niveau Ø : matériel :

  Cela correspond directement au niveau hardware de la machine.

L'avantage d'une telle hiérarchisation est de considérer chaque module comme une boîte noire dont on suppose qu'elle respecte ses spécifications. Ainsi, un module qui en utilise un autre ne se préoccupe pas de la façon dont celui-ci effectue ses traitements pour vérifier ses spécifications. Cela permet une simplification de chaque module et celles-ci restent indépendantes de toute solution d'implémentation. De même, en cas de modification du système, les changements (de structure de données par exemple) seront localisés à un petit nombre de modules sans altérer les autres modules.

7.1. Schéma des accès possibles.



7.2. Spécifications des modules.

Avant d'entamer la spécification de chaque module, il est nécessaire de décrire la syntaxe et la sémantique qui seront utilisées.
Les fonctions de chaque module seront spécifiées par :

- Obj. : objectifs à réaliser par la fonction.
- Args : liste des arguments en imput qui devront être fournis à la fonction.
- Pré : pré-conditions qui devront être respectées par les arguments en input pour que la fonction vérifie bien ses spécifications.
- Rés. : résultats fournis par la fonction après sa complète exécution si les pré-conditions sont respectées.
- Post : post-conditions vérifiées par les résultats de la fonction.

On estimera que tous les arguments fournis à chaque fonction seront valides syntaxiquement, c'est-à-dire qu'ils pourront correspondre à une occurrence possible de la structure de données à laquelle ils sont supposés appartenir.
Toutes les fonctions des modules des niveaux 4 & 6 peuvent directement être utilisées par le système d'E.A.O. ou l'enseignant en respectant toutefois les préconditions des arguments de celles-ci. Dans le cas ou le résultat d'une fonction est une liste (ensemble d'éléments de même type), celle-ci peut être vide (= $\emptyset$). Le schéma des relations entre les différents modules et fonctions se trouvent dans l'annexe A.

Toutes les entités du domaine d'enseignement que l'on désire supprimer, mais qui ne peuvent l'être sous peine d'altérer la cohérence des informations mémorisées antérieurement dans l'historique, ne pourront être retirées et seront "marquées" (afin de restreindre les utilisations ultérieures). Le système d'E.A.O. ou l'enseignant décideront eux-mêmes de tenir compte ou non de ces informations.
De même, on ne pourra supprimer des relations ou modifier des données que si cela ne risque pas de provoquer des incohérences.

7.2.1. Spécification des modules de niveau 6.

- MISE-A-JOUR-ETUDIANT :

Obj. : ce module permet la mise à jour des données relatives à un étudiant.
Args : étudiant : ET-ID, données : ET-DATA, op ∈ {création, suppression, modification, consultation}
Pré : -
Rés. : données : ET-DATA <=> op = consultation.
Post : en cas de création, l'étudiant ne peut exister sinon il y a erreur. Dans les autres cas, l'étudiant doit exister sinon erreur.
- op = création : enregistrement de l'étudiant dans l'historique.
- op = modification : les données relatives à l'étudiant sont remplacées par celles spécifiées.
- op = consultation : la fonction renvoie les données propres à l'étudiant.
- op = suppression : l'étudiant est retiré de l'historique ainsi que tous les échanges et les connaissances de celui ainsi que les relations correspondantes.

- CREATION-ECHANGE :

Obj. : ce module a pour but de mémoriser un nouvel échange dans l'historique et de mettre à jour automatiquement la synthèse des connaissances de l'apprenant en fonction du dialogue en cours.
Args : étudiant : ET-ID, exercice : EX-ID, échange : EC-ID, données : EC-DATA, {concept : CO-ID} (liste des concepts non maîtrisés par l'étudiant lors du dialogue).
Pré : -
Rés. : -
Post : si l'étudiant et l'exercice existent et que l'échange n'existe pas encore, enregistrement de celui-ci et de leurs chemins d'accès respectifs, sinon erreur. Pour chaque concept non-maîtrisé existant et sous-jacent à l'exercice, création d'un lien avec l'échange. Les concepts sous-jacents à l'exercice sont recherchés. Si la connaissance de l'étudiant correspondant à chaque concept existe, les données de celle-ci sont mises à jour en fonction de la maîtrise ou non du concept lors de l'échange. Si la connaissance n'existe pas encore, elle est créée avec les données adéquates.

- MISE-A-JOUR-COMPLEXITE :

Obj. : ce module permet la mise à jour des données relatives à un niveau de complexité.
Args : complexité : CM-ID, données : CM-DATA, op ∈ {création, modifications, suppression, consultation}.
Pré : -
Rés. : données : CM-DATA <=> op = consultation.
Post : en cas de création, le niveau de complexité ne peut exister sinon il y a erreur. Dans les autres cas, le niveau de complexité doit exister sinon erreur.
- op = création : enregistrement du niveau de complexité dans l'historique.
- op = modification : les données relatives au niveau de complexité sont remplacées par celles spécifiées à condition que la complexité ne soit pas relié à un exercice impliqué dans un échange.
- op = consultation : la fonction renvoie les données propres au niveau de complexité.
- op = suppression : le niveau de complexité est retiré de l'historique ainsi que tous les chemins d'accès entre celui-ci et des classes éventuelles à condition que la complexité ne soit relié à aucun exercice impliqué dans un échange, sinon erreur.

- MISE-A-JOUR-CONCEPT :

Obj. : ce module permet la mise à jour des données relatives à un concept.
Args : concept : CO-ID, données : CO-DATA, op ∈ {création, modification, suppression, consultation}.
Pré : -
Rés. : données : CO-DATA <=> op = consultation.
Post : en cas de création, le concept ne peut exister sinon il y a erreur. Dans les autres cas, le concept doit exister sinon erreur.
- op = création : enregistrement du concept dans l'historique.
- op = modification : les données relatives au concept sont remplacées par celles spécifiées, à condition que le concept ne soit associé à aucune connaissance.
- op = consultation : la fonction renvoie les données propres au concept.
- op = suppression : le concept est retiré de l'historique s'il n'est associé à aucune connaissance ainsi que les éventuels chemins d'accès entre ce concept et les classes. Dans le cas contraire, le concept est "marqué" et il y a erreur.

- MISE-A-JOUR-CLASSE :

<u>Obj.</u> : ce module permet la mise à jour des données relatives aux classes d'exercices.
<u>Args</u> : classe : CL-ID, données : CL-DATA, op ∈ {création, modification, suppression, consultation}.
<u>Pré</u> : -
<u>Rés.</u> : données : CL-DATA <=> op = consultation.
<u>Post</u> : en cas de création, la classe ne peut exister sinon il y a erreur. Dans les autres cas, la classe doit exister sinon erreur.
- op = création : enregistrement de la classe dans l'historique.
- op = modification : les données relatives à la classe sont remplacées par celles spécifiées à condition que la classe ne contienne aucun exercice impliqué dans un échange, sinon erreur.
- op = consultation : la fonction renvoie les données propres à la classe.
- op = suppression : la classe est retirée de l'historique ainsi que tous les chemins d'accès entre celle-ci, les exercices, les concepts et le niveau de complexité, si la classe ne contient aucun exercice impliqué dans un échange, sinon erreur et marquage.

- MISE-A-JOUR-EXERCICE :

<u>Obj.</u> : ce module permet la mise à jour des données relatives aux exercices.
<u>Args</u> : exercice : EX-ID, données : EX-DATA, op ∈ {création, modification, suppression, consultation}.
<u>Pré</u> : -
<u>Rés.</u> : données : EX-DATA <=> op = consultation.
<u>Post</u> : en cas de création, l'exercice ne peut exister sinon il y a erreur. Dans les autres cas, l'exercice doit exister sinon erreur.
- op = création : enregistrement de l'exercice dans l'historique.
- op = modification : les données relatives à l'exercice sont remplacées par celles spécifiées si l'exercice n'est associé à aucun échange, sinon erreur.
- op = consultation : la fonction renvoie les données propres à l'exercice.
- op = suppression : l'exercice est retiré de l'historique ainsi que les éventuels chemins d'accès entre des classes s'il n'est lié à aucun échange, sinon il y a erreur. Dans ce cas, la marque de tentative de suppression sera positionnée.

- MISE-A-JOUR-SOUS-JACENT.

<u>Obj.</u> : créer ou supprimer un chemin d'accès entre une classe et un concept.
<u>Args</u> : classe : CL-ID, concept : CO-ID, op ∈ {création, suppression}
<u>Pré</u> : -
<u>Rés.</u> : -
<u>Post</u> : il y a erreur si la classe ou le concept n'existent pas. En cas de création, cette relation ne peut pas exister tandis que en cas de suppression elle doit exister sinon il y a erreur.
- op = création : un chemin d'accès est créé entre la classe et le concept.
- op = suppression : le chemin d'accès entre la classe et le concept est supprimé si le concept n'est lié à aucune connaissance, sinon erreur.

- MISE-A-JOUR-CONTIENT :

<u>Obj.</u> : créer ou supprimer un chemin d'accès entre un exercice et une classe.
<u>Args</u> : classe : CL-ID, exercice : EX-ID, op ∈ {création, suppression}.
<u>Pré</u> : -
<u>Rés.</u> : -
<u>Post</u> : il y a erreur si la classe ou l'exercice n'existent pas. En cas de création, cette relation ne peut pas exister tandis que, en cas de suppression, elle doit exister.
- op = création : un chemin d'accès est créé entre la classe et l'exercice.
- op = suppression : le chemin d'accès entre la classe et l'exercice est supprimé si l'exercice n'est lié à aucun échange, sinon erreur.

- MISE-A-JOUR-CARACTERISE :

<u>Obj.</u> : créer ou supprimer un chemin d'accès entre une classe et un niveau de complexité.
<u>Args</u> : classe : CL-ID, complexité : CM-ID, op ∈ {création, suppression}.
<u>Pré.</u> : -
<u>Rés.</u> : -
<u>Post</u> : il y a erreur si le niveau de complexité ou la classe n'existent pas. En cas de suppression, cette relation doit exister, tandis que, en cas de création, il ne peut exister aucune relation entre la classe et un niveau de complexité quelconque, sinon il y a erreur.
- op = création : un chemin d'accès est créé entre le niveau de complexité et la classe.
- op = suppression : le chemin d'accès entre le niveau de complexité et la classe est supprimé si la classe ne contient aucun exercice impliqué dans un échange, sinon erreur.

- RECHERCHE-CLASSES-CONCEPTS :

  Obj. : fournir l'ensemble des classes auxquels les concepts spécifiés sont sous-jacents.
  Args : {concept : CO-ID}.
  Pré. : -
  Rés. : {classe : CL-ID}.
  Post : tous les concepts spécifiés doivent exister sinon il y a erreur. Toutes les classes renvoyées contiennent tous les concepts sous-jacents spécifiés.

- RECHERCHE-CONCEPTS-CLASSES :

  Obj. : fournir les concepts communs à différentes classes d'exercices.
  Args : {classe : CL-ID}.
  Pré. : -
  Rés. : {concept : CO-ID}.
  Post : toutes les classes spécifiées doivent exister sinon il y a erreur. Les concepts renvoyés sont sous-jacents à chacune des classes.

7.2.2. Spécification des modules de niveau 5.

- Module EXTRACTION :

  Obj. : fournir tous les éléments d'une liste vérifiant un critère de sélection.
  Args : liste : LISTE, fonction : FONCTION.
  Pré. : la fonction spécifiée doit renvoyer un résultat booléen.
  Rés. : {elem | elem ∈ liste & fonction(elem) = vrai}.
  Post : la liste renvoyée comme résultat fournit l'ensemble des éléments de la liste argument vérifiant la condition spécifiée.

- Module FUSION :

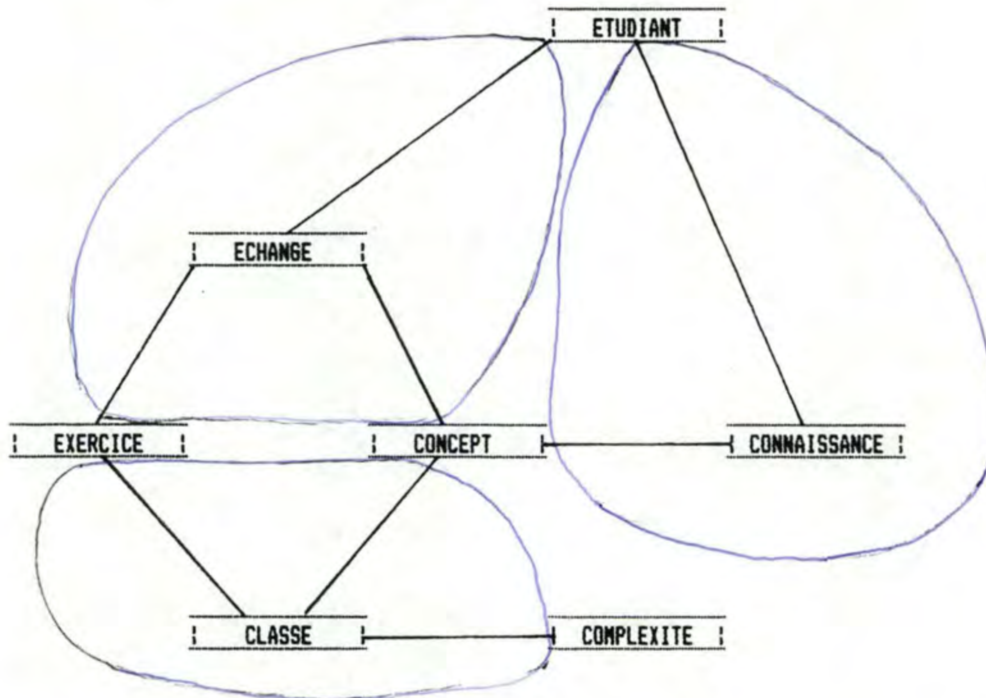
  Obj. : constituer une seule liste à partir de plusieurs listes d'éléments.
  Args : {liste : LISTE}, op ∈ {U,∩,/}.
  Pré. : les éléments de chaque liste doivent être de même type.
  Rés. : {elem | elem ∈ liste}.
  Post : le résultat constitue l'intersection, la réunion ou la différence des listes arguments selon le type d'opération spécifiée.

- Module EXPANSION :

Obj. : appliquer une même fonction à une liste d'arguments.
Args : liste : LISTE, fonction : FONCTION.
Pré. : -
Rés. : {fonction(elem) | elem ∈ liste}.
Post : si un élément de la liste ne constitue pas une occurrence possible d'une entité ou association de l'historique ou si l'application de la fonction sur un des éléments de la liste renvoie un résultat erroné, il y a erreur.

7.2.3. Spécification des modules de niveau 4.

Les modules de ce niveau correspondent directement aux éléments de la structure de données décrits dans les chapitres précédents. Les entités et associations sont groupées en modules selon le schéma suivant :



Il me paraît préférable de regrouper en un seul module chaque entité avec les associations dont elle dépend (exemple : T.E.ECHANGE, T.A.EFFECTUE, T.A.CONCERNE, T.A.NE-MAITRISE-PAS) ainsi que les entités et les associations constituant un noeud de communication avec les autres données (exemple : T.E.CLASSE, T.A.SOUS-JACENT, T.A.CONTIENT, T.A.CARACTERISE).

- Fonctions du module ETUDIANT:

  - Fonction EXIST-ETUDIANT :

    Obj. : vérifier si un étudiant existe ou non dans l'historique.
    Args : étudiant : ET-ID.
    Pré : -
    Rés. : b : BOOLEEN.
    Post : b = vrai si un article de type ETUDIANT identifié par la valeur de ET-ID existe, sinon b = faux.

  - Fonction RECH-ETUDIANT :

    Obj. : fournir les données relatives à un étudiant.
    Args : étudiant : ET-ID.
    Pré : EXIST-ETUDIANT(étudiant).
    Rés. : données : ET-DATA.
    Post : les données sont celles de l'article de type ETUDIANT identifié par la valeur de ET-ID.

  - Fonction LISTE-ETUDIANTS :

    Obj. : fournir tous les étudiants figurant dans l'historique.
    Args : -
    Pré : -
    Rés. : {étudiant : ET-ID}.
    Post : la liste renvoyée contient l'identifiant de chaque article de type étudiant mémorisé dans l'historique.

  - Fonction CREATION-ETUDIANT :

    Obj. : enregistrer un nouvel étudiant dans l'historique.
    Args : étudiant : ET-ID, données : ET-DATA.
    Pré : ¬ EXIST-ETUDIANT(étudiant).
    Rés. :
    Post : ajout dans l'historique d'un nouvel article de type ETUDIANT identifié par la valeur de ET-ID et contenant la valeur de ET-DATA.

  - Fonction MODIF-ETUDIANT :

    Obj. : modifier les données relatives à un étudiant.
    Args : étudiant : ET-ID, données : ET-DATA.
    Pré : EXIST-ETUDIANT(étudiant).
    Rés. : -
    Post : l'article de type ETUDIANT identifié par la valeur de ET-ID contient les nouvelles données spécifiées par la valeur de ET-DATA.

- Fonction SUPPRES-ETUDIANT :

Obj. : supprimer un étudiant de l'historique.
Args : étudiant : ET-ID.
Pré : EXIST-ETUDIANT(étudiant) & RECH-ECHANGES-ETUDIANT(étudiant) = ø & RECH-CONNAISSANCES-ETUDIANT (étudiant) = ø.
Rés. : -
Post : l'article de type ETUDIANT identifié par la valeur de ET-ID est retiré de l'historique.

- Fonctions du module EXERCICE :

- Fonction EXIST-EXERCICE :

Obj. : vérifier si un exercice existe ou non dans le domaine d'enseignement.
Args : exercice : EX-ID.
Pré : -
Rés. : b : BOOLEEN.
Post : b = vrai si un article de type EXERCICE identifié par la valeur de EX-ID existe, sinon b = faux.

- Fonction RECH-EXERCICE :

Obj. : fournir les données relatives à un exercice spécifié.
Args : exercice : EX-ID.
Pré : EXIST-EXERCICE(exercice).
Rés. : données : EX-DATA, m : BOOLEEN.
Post : les données sont celles de l'article de type EXERCICE identifié par la valeur de EX-ID.

- Fonction LISTE-EXERCICES :

Obj. : fournir tous les exercices figurant dans le domaine d'enseignement.
Args : -
Pré : -
Rés. : {exercices : EX-ID}.
Post : la liste renvoyée contient l'identifiant de chaque article de type EXERCICE mémorisé dans l'historique.

- Fonction CREATION-EXERCICE :

  Obj. : enregistrer un nouvel exercice dans le domaine d'enseignement.
  Args : exercice : EX-ID, données : EX-DATA.
  Pré : ¬ EXIST-EXERCICE(exercice).
  Rés. : -
  Post : ajout dans l'historique d'un nouvel article de type EXERCICE identifié par la valeur de EX-ID et contenant la valeur de EX-DATA.

- Fonction MODIF-EXERCICE :

  Obj. : modifier les données relatives à un exercice.
  Args : exercice : EX-ID, données : EX-DATA.
  Pré : EXIST-EXERCICE(exercice) et RECH-ECHANGES-EXERCICE(exercice) = ∅.
  Rés. : -
  Post : l'article de type EXERCICE identifié par la valeur de EX-ID contient les nouvelles données spécifiées par la valeur de EX-DATA.

- Fonction SUPPRES-EXERCICE :

  Obj. : supprimer un exercice du domaine d'enseignement.
  Args : exercice : EX-ID.
  Pré : EXIST-EXERCICE(exercice) & RECH-ECHANGE-EXERCICE(exercice) = ∅ & RECH-CLASSES-EXERCICE(exercice) = ∅
  Rés. : -
  Post : l'article de type EXERCICE identifié par la valeur de EX-ID est retiré de l'historique.

- Fonction du module CONCEPT :

- Fonction EXIST-CONCEPT :

  Obj. : vérifier si un concept existe ou non dans le domaine d'enseignement.
  Args : concept : CO-ID.
  Pré : -
  Rés. : b : BOOLEEN.
  Post : b = vrai si un article de type CONCEPT identifié par la valeur de CO-ID existe, sinon b = faux.

- Fonction RECH-CONCEPT :

<u>Obj.</u> : fournir les données relatives à un concept.
<u>Args</u> : concept : CO-ID, m : BOOLEEN.
<u>Pré</u> : EXIST-CONCEPT(concept).
<u>Rés.</u> : données : CO-DATA.
<u>Post</u> : les données sont celles de l'article de type CONCEPT identifié par la valeur de CO-ID.

- Fonction LISTE-CONCEPTS :

<u>Obj.</u> : fournir tous les concepts figurant dans le domaine d'enseignement.
<u>Args</u> : -
<u>Pré</u> : -
<u>Rés.</u> : {concept : CO-ID}.
<u>Post</u> : la liste renvoyée contient l'identifiant de chaque article de type CONCEPT mémorisé dans l'historique.

- Fonction CREATION-CONCEPT :

<u>Obj.</u> : enregistrer un nouveau concept dans le domaine d'enseignement.
<u>Args</u> : concept : CO-ID, données : CO-DATA.
<u>Pré</u> : ¬ EXIST-CONCEPT(concept).
<u>Rés.</u> : -
<u>Post</u> : ajout dans l'historique d'un nouvel article de type CONCEPT identifié par la valeur de CO-ID et contenant la valeur de CO-DATA.

- Fonction MODIF-CONCEPT :

<u>Obj.</u> : modifier les données relatives à un concept.
<u>Args</u> : concept : CO-ID, données : CO-DATA.
<u>Pré</u> : EXIST-CONCEPT(concept) & RECH-CONNAISSANCES-CONCEPT = ∅.
<u>Rés.</u> : -
<u>Post</u> : l'article de type CONCEPT identifié par la valeur de CO-ID contient les nouvelles données spécifiées par la valeur de CO-DATA.

- Fonction SUPPRES-CONCEPT :

<u>Obj.</u> : supprimer un concept du domaine d'enseignement.
<u>Args</u> : concept : CO-ID.
<u>Pré</u> : EXIST-CONCEPT(concept) & RECH-CONNAISSANCES-CONCEPT(concept) = ∅ & RECH-CLASSES-CONCEPT(concept) = ∅
<u>Rés.</u> : -
<u>Post</u> : l'article de type CONCEPT identifié par la valeur de CO-ID est retiré de l'historique.

- Fonctions du module COMPLEXITE :

  - Fonction EXIST-COMPLEXITE :

    Obj. : vérifier si un niveau de complexité existe ou non dans le domaine d'enseignement.
    Args : complexité : CM-ID.
    Pré : -
    Rés. : b : BOOLEEN.
    Post : b = vrai si un article de type COMPLEXITE identifié par la valeur de CM-ID existe, sinon b = faux.

  - Fonction RECH-COMPLEXITE :

    Obj. : fournir les données relatives à un niveau de complexité spécifié.
    Args : complexité : CM-ID.
    Pré : EXIST-COMPLEXITE(complexité).
    Rés. : données : CM-DATA, m : BOOLEEN.
    Post : les données sont celles de l'article de type COMPLEXITE identifié par la valeur de CM-ID.

  - Fonction LISTE-COMPLEXITES :

    Obj. : fournir tous les niveaux de complexité figurant dans le domaine d'enseignement.
    Args : -
    Pré : -
    Rés. : {complexité : CM-ID}.
    Post : la liste renvoyée contient l'identifiant de chaque article de type COMPLEXITE mémorisé dans l'historique.

  - Fonction CREATION-COMPLEXITE :

    Obj. : enregistrer un niveau de complexité dans le domaine d'enseignement.
    Args : complexité : CM-ID, données : CM-DATA.
    Pré : ¬ EXIST-COMPLEXITE(complexité).
    Rés. : -
    Post : ajout dans l'historique d'un nouvel article de type COMPLEXITE identifié par la valeur de CM-ID et contenant la valeur de CM-DATA.

- Fonction MODIF-COMPLEXITE :

  Obj. : modifier les données relatives à un niveau de complexité.
  Args : complexité : CM-ID, données : CM-DATA.
  Pré : EXIST-COMPLEXITE (complexité) & (∀ classe : CL-ID | classe ∈ RECH-CLASSES-COMPLEXITE(complexité) & ∀ exercice : EX-ID | exercice ∈ RECH-EXERCICES-CLASSE(classe) => RECH-ECHANGES-EXERCICE(exercice) = ∅).
  Rés. : -
  Post : l'article de type COMPLEXITE identifié par la valeur de CM-ID contient les nouvelles données spécifiées par la valeur de CM-DATA.

- Fonction SUPPRES-COMPLEXITE :

  Obj. : supprimer un niveau de complexité du domaine d'enseignement.
  Args : complexité : CM-ID.
  Pré : EXIST-COMPLEXITE(complexité) & RECH-CLASSE-COMPLEXITE(complexite) = ∅.
  Rés. : -
  Post : l'article de type COMPLEXITE identifié par la valeur de CM-ID est retiré de l'historique.

- Fonctions du module CLASSE :

- Fonction EXIST-CLASSE :

  Obj. : vérifier si une classe existe ou non dans le domaine d'enseignement.
  Args : classe : CL-ID.
  Pré : -
  Rés. : b : BOOLEEN.
  Post : b = vrai si un article de type CLASSE identifié par la valeur de CL-ID existe, sinon b = faux.

- Fonction RECH-CLASSE :

  Obj. : fournir les données relatives à une classe.
  Args : classe : CL-ID.
  Pré : EXIST-CLASSE(classe).
  Rés. : données : CL-DATA, m : BOOLEEN.
  Post : les données sont celles de l'article de type CLASSE identifié par la valeur de CL-ID.

- Fonction LISTE-CLASSES :

  Obj. : fournir toutes les classes figurant dans le domaine d'enseignement.
  Args : -
  Pré : -
  Rés. : {classe : CL-ID}.
  Post : la liste renvoyée contient l'identifiant de chaque article de type CLASSE mémorisé dans l'historique.

- Fonction CREATION-CLASSE :

  Obj. : enregistrer une nouvelle classe dans le domaine d'enseignement.
  Args : classe : CL-ID, données : CL-DATA.
  Pré : ¬ EXIST-CLASSE(classe).
  Rés. : -
  Post : ajout dans l'historique d'un nouvel article de type CLASSE identifié par la valeur de CL-ID et contenant la valeur de CL-DATA.

- Fonction MODIF-CLASSE :

  Obj. : modifier les données relatives à une classe.
  Args : classe : CL-ID, données : CL-DATA.
  Pré : EXIST-CLASSE(classe) & ∀ exercice : EX-ID | exercice ∈ RECH-EXERCICES -CLASSE(classe) => RECH-ECHANGES-EXERCICES(exercice) = ∅.
  Rés. : -
  Post : l'article de type CLASSE identifié par la valeur de CL-ID contient les nouvelles données spécifiées par la valeur de CL-DATA.

- Fonction SUPPRES-CLASSE :

  Obj. : supprimer une classe d'exercices du domaine d'enseignement.
  Args : classe : CL-ID.
  Pré : EXIST-CLASSE(classe) & RECH-EXERCICES-CLASSE(classe) = ∅ & RECH-CONCEPTS-CLASSE(classe) = ∅.
  Rés. : -
  Post : l'article de type CLASSE identifié par la valeur de CL-ID est retiré de l'historique.

- Fonction EXIST-CONTIENT :

<u>Obj.</u> : vérifier s'il existe une relation entre une classe et un exercice.
<u>Args</u> : exercice : EX-ID, classe : CL-ID.
<u>Pré</u> : EXIST-EXERCICE(exercice) & EXIST-CLASSE(classe).
<u>Rés.</u> : b : BOOLEEN.
<u>Post</u> : b = vrai s'il existe un chemin d'accès entre l'article de type EXERCICE identifié par la valeur de EX-ID et l'article de type CLASSE identifié par la valeur de CL-ID, sinon b = faux.

- Fonction CREATION-CONTIENT :

<u>Obj.</u> : associer un exercice à une classe.
<u>Args</u> : exercice : EX-ID, classe : CL-ID.
<u>Pré.</u> : EXIST-EXERCICE(exercice) & EXIST-CLASSE(classe) & ¬ EXIST-CONTIENT(exercice, classe).
<u>Rés.</u> : -
<u>Post</u> : création d'un chemin d'accès entre l'article de type EXERCICE identifié par la valeur de EX-ID et l'article de type CLASSE identifié par la valeur de CL-ID.

- Fonction SUPPRES-CONTIENT :

<u>Obj.</u> : supprimer le chemin d'accès entre une classe et un exercice.
<u>Args</u> : exercice : EX-ID, classe : CL-ID.
<u>Pré</u> : EXIST-EXERCICE(exercice) & EXIST-CLASSE(classe) & EXIST-CONTIENT(exercice, classe) & RECH-ECHANGES-EXERCICE(exercice) = ∅.
<u>Rés.</u> : -
<u>Post</u> : suppression du chemin d'accès entre l'article de type EXERCICE identifié par la valeur de EX-ID et l'article de type CLASSE identifié par la valeur de CL-ID.

- Fonction RECH-CONCEPTS-CLASSE :

<u>Obj.</u> : rechercher tous les concepts sous-jacents à une classe.
<u>Args</u> : classe : CL-ID.
<u>Pré.</u> : EXIST-CLASSE(classe).
<u>Rés.</u> : {concept : CO-ID}.
<u>Post</u> : la liste renvoyée contient l'identifiant de chaque article de type CONCEPT sous-jacent à la classe identifiée par la valeur de CL-ID.

- Fonction RECH-CLASSES-CONCEPT :

  Obj. : rechercher toutes les classes auxquelles un concept est sous-jacent.
  Args : concept : CO-ID.
  Pré. : EXIST-CONCEPT(concept).
  Rés. : {classe : CL-ID}.
  Post : la liste renvoyée contient l'identifiant de chaque article de type CLASSE auquel est sous-jacent l'article de type CONCEPT identifié par la valeur de CO-ID.

- Fonction EXIST-SOUS-JACENT :

  Obj. : vérifier s'il existe une relation entre une classe et un concept.
  Args : concept : CO-ID, classe : CL-ID.
  Pré : EXIST-CONCEPT(concept) & EXIST-CLASSE(classe).
  Rés. : b : BOOLEEN.
  Post : b = vrai s'il existe un chemin d'accès entre l'article de type CONCEPT identifié par la valeur de CO-ID et l'article de type CLASSE identifié par la valeur de CL-ID, sinon b = faux.

- Fonction CREATION-SOUS-JACENT :

  Obj. : associer un concept à une classe.
  Args : concept : CO-ID, classe : CL-ID.
  Pré. : EXIST-CONCEPT(concept) & EXIST-CLASSE(classe) & ¬ EXIST-SOUS-JACENT(concept, classe).
  Rés. : -
  Post : création d'un chemin d'accès entre l'article de type CONCEPT identifié par la valeur de CO-ID et l'article de type CLASSE identifié par la valeur de CL-ID.

- Fonction SUPPRES-SOUS-JACENT :

  Obj. : supprimer le chemin d'accès entre une classe et un concept.
  Args : concept : CO-ID, classe : CL-ID.
  Pré. : EXIST-CONCEPT(concept) & EXIST-CLASSE(classe) & EXIST-SOUS-JACENT(concept, classe) & RECH-CONNAISSANCES-CONCEPT(concept) = ∅.
  Rés. : -
  Post : suppression du chemin d'accès entre l'article de type CONCEPT identifié par la valeur de CO-ID et l'article de type CLASSE identifié par la valeur de CL-ID.

- Fonction RECH-EXERCICES-CLASSE :

  Obj. : rechercher tous les exercices d'une classe.
  Args : classe : CL-ID.
  Pré. : EXIST-CLASSE(classe).
  Rés. : {exercice : EX-ID}.
  Post : la liste renvoyée contient l'identifiant de chaque article de type EXERCICE appartenant à la classe identifiée par la valeur de CL-ID.

- Fonction RECH-CLASSES-EXERCICE :

  Obj. : rechercher toutes les classes contenant un exercice.
  Args : exercice : EX-ID.
  Pré. : EXIST-EXERCICE(exercice).
  Rés. : {classe : CL-ID}.
  Post : la liste renvoyée contient l'identifiant de chaque article de type CLASSE contenant l'exercice identifié par la valeur de EX-ID.

- Fonction EXIST-CARACTERISE :

  Obj. : vérifier s'il existe une relation entre une classe et un niveau de complexité.
  Args : complexité : CM-ID, classe : CL-ID.
  Pré : EXIST-COMPLEXITE(complexité) & EXIST-CLASSE(classe).
  Rés. : b : BOOLEEN.
  Post : b = vrai s'il existe un chemin d'accès entre l'article de type CLASSE identifié par la valeur de CL-ID et l'article de type COMPLEXITE identifié par la valeur de CM-ID, sinon b = faux.

- Fonction CREATION-CARACTERISE :

  Obj. : associer une classe à un niveau de complexité.
  Args : complexité : CM-ID, classe : CL-ID.
  Pré : EXIST-COMPLEXITE(complexité) & EXIST-CLASSE(classe) & RECH-COMPLEXITE-CONCEPT(classe) = ∅.
  Rés. : -
  Post : création d'un chemin d'accès entre l'article de type COMPLEXITE identifié par la valeur de CM-ID et l'article de type CLASSE identifié par la valeur de CL-ID.

- Fonction SUPPRES-CARACTERISE :

  Obj. : supprimer la relation entre une classe et un niveau de complexité.
  Args : complexité : CM-ID, classe : CL-ID.
  Pré : EXIST-COMPLEXITE(complexité) & EXIST-CLASSE(classe) & EXIST-CARACTERISE(complexité, classe) & ∀ exercice : EX-ID | exercice ∈ RECH-EXERCICES-CLASSE(classe) => RECH-ECHANGES-EXERCICE = ∅
  Rés. : -
  Post : suppression du chemin d'accès entre l'article de type COMPLEXITE identifié par la valeur de CM-ID et l'article de type CLASSE identifié par la valeur de CL-ID.

- Fonction RECH-COMPLEXITE-CLASSE :

  Obj. : rechercher le niveau de complexité d'une classe.
  Args : classe : CL-ID.
  Pré : EXIST-CLASSE(classe).
  Rés. : {complexité : CM-ID}.
  Post : la liste renvoyée contient l'identifiant de l'article de type COMPLEXITE caractérisant la classe identifiée par la valeur de CL-ID.

- Fonction RECH-CLASSES-COMPLEXITE :

  Obj. : rechercher toutes les classes d'un niveau de complexité.
  Args : complexité : CM-ID.
  Pré : EXIST-COMPLEXITE(complexité).
  Rés. : {classe : CL-ID}.
  Post : la liste renvoyée contient l'identifiant de chaque article de type CLASSE caractérisé par le niveau de complexité identifié par la valeur de CM-ID.

- Fonction du module ECHANGE :

- Fonction EXIST-ECHANGE :

  Obj. : vérifier si un échange existe ou non dans l'historique.
  Args : étudiant : ET-ID, échange : EC-ID.
  Pré. : EXIST-ETUDIANT(étudiant).
  Rés. : b : BOOLEEN.
  Post : b = vrai si un article de type ECHANGE identifié par la valeur de ET-ID et EC-ID existe, sinon b = faux.

- Fonction RECH-ECHANGE :

<u>Obj.</u> : fournir les données relatives à un échange.
<u>Args</u> : étudiant : ET-ID, échange : EC-ID.
<u>Pré.</u> : EXIST-ETUDIANT(étudiant) & EXIST-ECHANGE(étudiant, échange).
<u>Rés.</u> : données : EC-DATA.
<u>Post</u> : les données sont celles de l'article de type ECHANGE identifié par la valeur de ET-ID et EC-ID.

- Fonction RECH-ECHANGES-ETUDIANT :

<u>Obj.</u> : rechercher tous les échanges effectués par un étudiant.
<u>Args</u> : étudiant : ET-ID.
<u>Pré.</u> : EXIST-ETUDIANT(étudiant).
<u>Rés.</u> : {(étudiant, échange, exercice) | échange : EC-ID, exercice : EX-ID}.
<u>Post</u> : la liste renvoyée contient l'identifiant de chaque article de type ECHANGE effectué par l'étudiant identifié par la valeur de ET-ID ainsi que celui de l'exercice effectué lors de l'échange.

- Fonction CREATION-ECHANGE :

<u>Obj.</u> : enregistrement d'un nouvel échange dans l'historique.
<u>Args</u> : étudiant : ET-ID, échange : EC-ID, données : EC-DATA, exercice : EX-ID.
<u>Pré.</u> : EXIST-ETUDIANT(étudiant) & EXIST-EXERCICE(exercice) & ¬ EXIST-ECHANGE(étudiant, échange).
<u>Rés.</u> : -
<u>Post</u> : ajout dans l'historique d'un nouvel article de type ECHANGE identifié par la valeur de ET-ID et EC-ID, contenant la valeur de EC-DATA et associé à un exercice identifié par la valeur de EX-ID et à un étudiant identifié par la valeur de ET-ID.

- Fonction RECH-ECHANGES-EXERCICE :

<u>Obj.</u> : fournir tous les échanges pendant lesquels un exercice a été effectué.
<u>Args</u> : exercice : EX-ID.
<u>Pré.</u> : EXIST-EXERCICE(exercice).
<u>Rés.</u> : {(étudiant, échange) | étudiant : ET-ID & échange : ET-ID}.
<u>Post</u> : la liste renvoyée contient l'identifiant de chaque article de type ECHANGE pendant lequel l'exercice identifié par la valeur de EX-ID a été effectué.

- Fonction SUPPRES-ECHANGE :

  Obj. : supprimer un échange de l'historique.
  Args : étudiant : ET-ID, échange : EC-ID.
  Pré. : EXIST-ETUDIANT(étudiant) & EXIST-ECHANGE(étudiant, échange) & RECH-CONCEPTS-ECHANGE(étudiant, échange) = ∅.
  Rés. : -
  Post : l'article de type ECHANGE identifié par la valeur de ET-ID et EC-ID est retiré de l'historique ainsi que les chemins entre celui-ci, l'étudiant et l'exercice effectué lors de cet échange.

- Fonction EXIST-NE-MAITRISE-PAS :

  Obj. : vérifier s'il existe une relation entre un échange et un concept.
  Args : concept : CO-ID, étudiant : ET-ID, échange : EC-ID.
  Pré. : EXIST-CONCEPT(concept) & EXIST-ETUDIANT(étudiant) & EXIST-ECHANGE(étudiant, échange).
  Rés. : b : BOOLEEN.
  Post : b = vrai s'il existe un chemin d'accès entre l'article de type CONCEPT identifié par la valeur de CO-ID et l'article de type ECHANGE identifié par la valeur de ET-ID et EC-ID, sinon b = faux.

- Fonction RECH-CONCEPTS-ECHANGE :

  Obj. : fournir tous les concepts non maîtrisés lors d'un échange.
  Args : étudiant : ET-ID, échange : EC-ID.
  Pré. : EXIST-ETUDIANT(étudiant) & EXIST-ECHANGE(étudiant, échange).
  Rés. : {concept : CO-ID}.
  Post : la liste renvoyée contient l'identifiant de chaque article de type CONCEPT associé à l'échange identifié par les valeurs de ET-ID et EC-ID.

- Fonction RECH-ECHANGES-CONCEPT :

  Obj. : fournir tous les échanges pendant lesquels un concept n'a pas été maîtrisé.
  Args : concept : CO-ID.
  Pré. : EXIST-CONCEPT(concept).
  Rés. : {(étudiant, échange) | étudiant : ET-ID, échange : EC-ID}.
  Post : la liste renvoyée contient l'identifiant de chaque article de type ECHANGE associé au concept identifié par la valeur de CO-ID.

- Fonction SUPPRES-NE-MAITRISE-PAS :

  Obj. : supprimer la relation entre un échange et un concept.
  Args : concept : CO-ID, étudiant : ET-ID, échange : EC-ID.
  Pré. : EXIST-CONCEPT(concept) & EXIST-ECHANGE(étudiant, échange) & EXIST-ETUDIANT(étudiant) & EXIST-NE-MAITRISE-PAS(concept, étudiant, échange).
  Rés. : -
  Post : suppression du chemin d'accès entre l'article de type CONCEPT identifié par la valeur de CO-ID et l'article de type ECHANGE identifié par la valeur de ET-ID et EC-ID.

- Fonction CREATION-NE-MAITRISE-PAS :

  Obj. : associer un échange à un concept.
  Args : étudiant : ET-ID, échange : EC-ID, concept : CO-ID.
  Pré. : EXIST-ETUDIANT(étudiant) & EXIST-ECHANGE(étudiant, échange) & EXIST-CONCEPT(concept) & ¬ EXIST-NE-MAITRISE-PAS(concept, étudiant, échange).
  Rés. : -
  Post : création d'un chemin d'accès entre l'article de type ECHANGE identifié par la valeur de ET-ID et EC-ID et l'article de type CONCEPT identifié par la valeur de CO-ID.

- Fonctions du module CONNAISSANCE :

  - Fonction EXIST-CONNAISSANCE :

    Obj. : vérifier si une connaissance existe ou non dans l'historique.
    Args : étudiant : ET-ID, concept : CO-ID.
    Pré. : EXIST-ETUDIANT(étudiant) & EXIST-CONCEPT(concept).
    Rés. : b : BOOLEEN.
    Post : b = vrai si un article de type CONNAISSANCE identifiée par la valeur de ET-ID et CO-ID existe, sinon b = faux.

- Fonction CREATION-CONNAISSANCE :

  Obj. : enregistrement d'une nouvelle connaissance dans l'historique.
  Args : étudiant : ET-ID, concept : CO-ID, ner : CN-NER, données : CN-DATA, nex : CN-NEX.
  Pré. : EXIST-ETUDIANT(étudiant) & EXIST-CONCEPT(concept) & ¬ EXIST-CONNAISSANCE(étudiant, concept).
  Rés. : -
  Post : ajout dans l'historique d'un nouvel article de type CONNAISSANCE contenant les valeurs de CN-DATA, CN-NEX et CN-NER et associée à un concept identifié par la valeur de CO-ID et à un étudiant identifié par la valeur de ET-ID.

- Fonction MODIF-CONNAISSANCE :

  Obj. : mettre à jour les valeurs d'une connaissance d'un étudiant.
  Args : étudiant : ET-ID, concept : CO-ID, données : CN-DATA, ner : CN-NER, nex : CN-NEX.
  Pré. : EXIST-ETUDIANT(étudiant) & EXIST-CONCEPT(concept) & EXIST-CONNAISSANCE(étudiant, concept).
  Rés. : -
  Post : l'article de type CONNAISSANCE identifié par la valeur de CO-ID et ET-ID contient les valeurs des données spécifiées.

- Fonction SUPPRES-CONNAISSANCE :

  Obj. : supprimer une connaissance de l'historique.
  Args : étudiant : ET-ID, concept : CO-ID.
  Pré. : EXIST-ETUDIANT(étudiant) & EXIST-CONCEPT(concept) & EXIST-CONNAISSANCE(étudiant, concept).
  Rés. : -
  Post : l'article de type CONNAISSANCE identifié par la valeur de ET-ID et CO-ID est supprimé de l'historique ainsi que les chemins d'accès entre cette connaissance, l'étudiant et le concept correspondants.

- Fonction RECH-CONNAISSANCES-ETUDIANT :

  Obj. : rechercher toutes les connaissances possédées par un étudiant.
  Args : étudiant : ET-ID.
  Pré. : EXIST-ETUDIANT(étudiant).
  Rés. : {(étudiant, concept) | concept : CO-ID}.
  Post : la liste renvoyée contient l'identifiant de chaque article de type CONNAISSANCE possédée par l'étudiant identifié par la valeur de ET-ID.

- Fonction RECH-CONNAISSANCE :

  <u>Obj.</u> : fournir des données relatives à une connaissance.
  <u>Args</u> : étudiant : ET-ID, concept : CO-ID.
  <u>Pré.</u> : EXIST-ETUDIANT(étudiant) & EXIST-CONNAISSANCE(étudiant, concept) & EXIST-CONCEPT(concept).
  <u>Rés.</u> : données : CN-DATA, ner : CN-NER, nex : CN-NEX.
  <u>Post</u> : les données sont celles de l'article de type CONNAISSANCE identifiée par la valeur de ET-ID et CO-ID.

- Fonction RECH-CONNAISSANCES-CONCEPT :

  <u>Obj.</u> : fournir toutes les connaissances relatives à un concept.
  <u>Args</u> : concept : CO-ID.
  <u>Pré.</u> : EXIST-CONCEPT(concept).
  <u>Rés.</u> : {(étudiant, concept) | étudiant : ET-ID}.
  <u>Post</u> : la liste renvoyée contient l'identifiant de chaque article de type CONNAISSANCE relative au concept identifié par la valeur de CO-ID.

CONCEPTION PHYSIQUE

8. Conception physique.

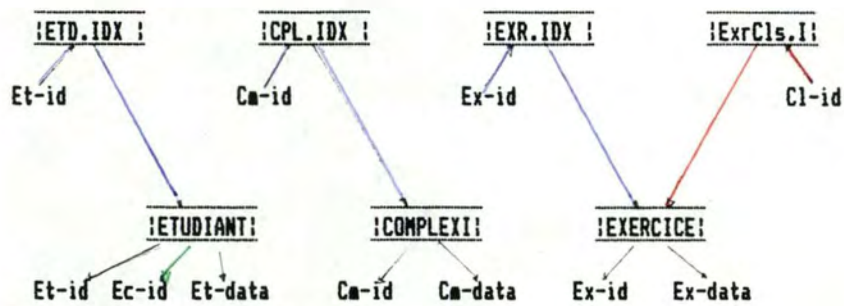
Cette troisième partie traite enfin de l'implémentation de l'historique des échanges sur une machine réelle. A ce stade, il est nécessaire de tenir compte des caractéristiques de l'ordinateur ainsi que des différents outils utilisés.
Ce mémoire étant destiné à être exploité par des enseignants, la machine employée devra donc être accessible à la catégorie des utilisateurs potentiels. Pour cette raison, la gestion de l'historique sera implémentée sur "Personal Computer" en Turbo Pascal.

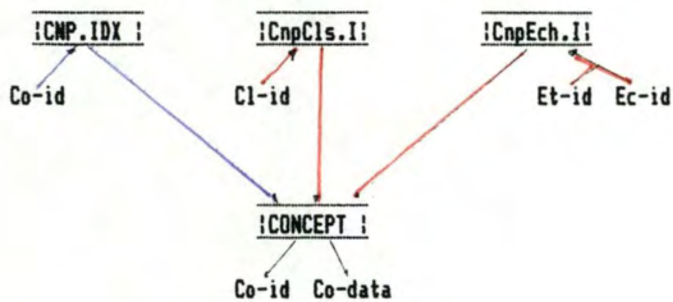
8.1. Description de Turbo Pascal et de Turbo Access System.

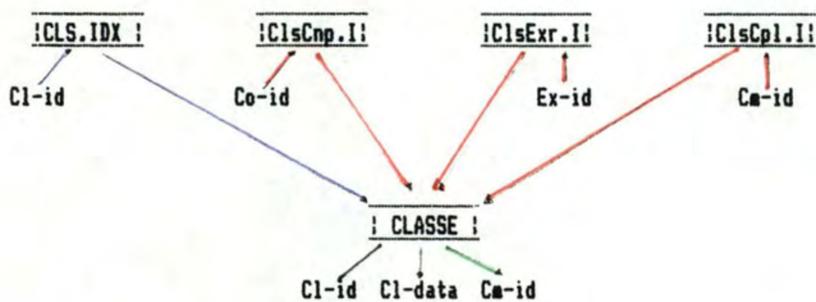
Turbo Pascal est une implémentation du langage Pascal offrant de grandes facilités quant à la gestion des fichiers. Les fichiers (séquentiels et relatifs) définis par Pascal sont disponibles ainsi que les fichiers séquentiels indexés (à accès par clé). Ce dernier type sera utilisé dans l'implémentation étant donné les caractéristiques des accès aux données mémorisées dans l'historique.

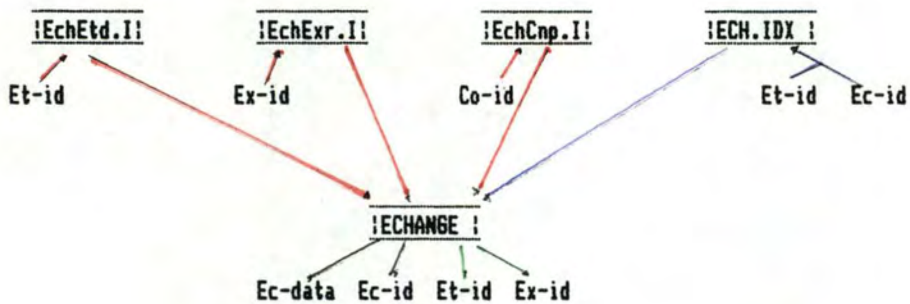
En Turbo Pascal, les fichiers à accès par clé sont constitués de 2 fichiers : celui contenant les données (Data File) et celui contenant la valeurs des clés d'accès et les pointeurs vers les informations associées du fichier des données (Index File). Turbo Access System offre toutes les facilités de lecture par clé des enregistrements, de lecture séquentielle, de mise à jour des données et des clés (création, modification, suppression), etc ... De plus, il est possible non seulement d'avoir des valeurs uniques pour les clés, mais aussi des valeurs identiques pour des clés pointant vers des données différentes (duplicate keys). Cette caractéristique permet la création de chemins d'accès multiples entre les différents types d'informations. Un même fichier de données peut avoir plusieurs fichiers d'index, tandis qu'un fichier d'index dépend d'un seul fichier de données. On peut donc accéder à une donnée grâce à plusieurs clés différentes.

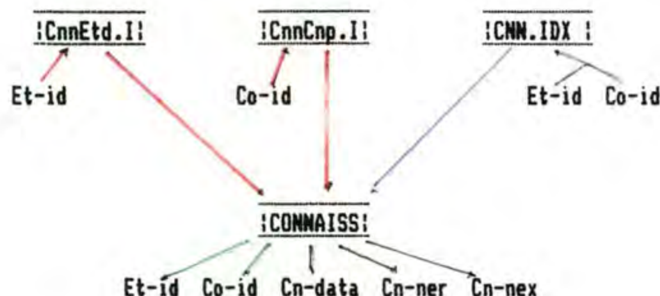
8.2. Schéma des accès en Turbo Pascal.











Ce graphique décrit la transposition du schéma des accès en fonction des caractéristiques des fichiers de Turbo Pascal. Les occurrences de chaque entité sont rassemblées dans un fichier de données avec un ou plusieurs fichiers d'index facilitant l'accès à celles-ci. La valeur d'une clé (dans le fichier d'index) est identique à la valeur de la donnée correspondante du fichier des données.

Toute entité accédée directement par son identifiant sera constituée d'un fichier de donnée et d'un fichier d'index dont la clé unique sera formée de la valeur de l'identifiant (—). Les chemins interarticles (entre fichiers de données différents) seront constitués par des fichiers d'index dont les clés (duplicates) seront formées par l'identifiant des articles origines et la référence associée indiquera un article extrémité (—). Les clés étant duplicates, une même valeur de clé pourra pointer vers plusieurs articles (chemin 1-N). Si le chemin inverse est un chemin 1-N, celui-ci sera aussi constitué d'un fichier d'index pointant vers l'ancienne origine. les chemins 1-1 ou Ø-1 seront constitués par une duplication de l'identifiant de l'article cible dans l'article origine (—).

Malgré le fait que l'accès aux articles se fassent par l'intermédiaire de fichiers d'index dont les clés sont les identifiants de ces articles, ceux-ci contiennent aussi la valeur de leur identifiant. En effet, puisqu'ils peuvent être accédés par des fichiers d'index représentant des chemins interarticles, on doit déterminer quel est l'article cible; l'identifiant figurant dans l'index ne pouvant être atteint à partir de l'article, on doit donc répéter la valeur de celui-ci. De plus, en cas de problème, l'identifiant au sein d'un article permet de reconstituer rapidement un index par une lecture séquentielle du fichier des données (voir manuel Turbo Access).

8.3. Description des constantes, des types & des fichiers.

8.3.1. Descriptions des constantes.

Les constantes représentent des valeurs fixes utilisées par le programme Pascal. Ces valeurs pourront être modifiées pour adapter la gestion de l'historique au système d'E.A.O. qui l'utilise. Les constantes décrites dans ce chapitre doivent être spécifiées dans tout programme qui fait appel à des procédures de l'historique. La valeur de chacune d'entre elles est donnée à titre d'exemple.

Les constantes ci-dessous sont utilisées par Turbo Access et doivent être décrites dans le programme principal. Pour connaître la signification exacte de ces constantes et les calculer, il vaut mieux se référer au manuel de Turbo Access.

- Constantes de Turbo Access :

MaxDataRec = 450
Longueur maximum des records de données (en nombre de bytes) : cette constante représente la longueur du plus grand enregistrement mémorisé dans l'un des fichiers de données de l'historique (voir 8.3.3.). Il s'agira le plus souvent du fichier des échanges. Pour calculer la longueur d'un record, il faut tenir compte du fait qu'un entier occupe 2 bytes, un réel 4, un booléen 1 et une chaîne de caractères (type string) la longueur spécifiée + 1.

MaxKeyLen = 24
Longueur maximum des clés (en nombre de caractères) : cette constante représente la longueur de la plus grande chaîne de caractères, utilisée comme clé d'accès à un des fichiers d'index de l'historique. Il s'agira le plus souvent de la clé du fichier d'index des échanges ou des connaissances.

PageSize = 16
Le maximum de clés permis dans une page d'index.

Order = 8
Le minimum de clés permis dans une page d'index (moitié de la constante précédente).

PageStackSize = 10
Le nombre maximum de pages d'index conservées en mémoire.

MaxHeight = 5
La hauteur maximum de l'Arbre-B.

Les constantes suivantes représentent la longueur des données qui seront mémorisées dans l'historique. Les valeurs sont données comme exemple.

- Longueurs des données :

LenEt_id = 20
Longueur maximum de l'identifiant d'un étudiant.

LenEt_data = 50
Longueur maximum des données d'un étudiant.

LenEc_id = 22
Longueur maximum de l'identifiant d'un échange. Cette constante doit toujours être égale à la longueur de l'identifiant d'un étudiant (LenEt_id) + 2 (voir 8.3.3.).

LenEc_sol = 20
Longueur maximum de la sollicitation d'une réponse.

LenEc_rep = 20
Longueur maximum de la réponse d'un étudiant.

LenEc_feed = 20
Longueur maximum d'un feed-back du système.

LenEc_inf = 20
Longueur maximum des données d'un échange.

LenEx_id = 4
Longueur maximum de l'identifiant d'un exercice.

LenEx_data = 50
Longueur maximum des données d'un exercice.

LenCl_id = 4
Longueur maximum de l'identifiant d'une classe.

LenCl_data = 20
Longueur maximum des données d'une classe.

LenCo_id = 4
Longueur maximum de l'identifiant d'un concept.

LenCo_data = 20
Longueur maximum des données d'un concept.

LenCm_id = 4
Longueur maximum de l'identifiant d'un niveau de complexité.

LenCm_data = 20
Longueur maximum des données d'un niveau de complexité.

LenCn_id = 24
Longueur maximum de l'identifiant d'une connaissance. Cette constante doit toujours être égale à la longueur de l'identifiant d'un étudiant (LenEt_id) + la longueur de l'identifiant d'un concept (LenCo_id).

LenCn_data = 20
Longueur maximum des données d'une connaissance.

Essai = 5
Nombre maximum de décomposition d'une réponse lors d'un échange.

Nargs = 10
Nombre maximum d'éléments dans les listes d'arguments. On utilise des listes d'arguments pour certaines procédures de l'historique recherchant des données relatives à des concepts ou des classes. Aussi, est-il recommandé d'attribuer à cette constante le nombre maximum d'éléments (concept ou classe) qui sera mémorisé dans l'historique.

8.3.2. Descriptions des types de données.

Les types de données qui suivent sont ceux que devront respecter les informations passées comme argument aux procédures de l'historique et que vérifieront les résultats de celles-ci. Les types dépendent en fait des constantes spécifiées précédemment et pourront donc être adaptés à tout système d'E.A.O.

```
Et_id      = string[LenEt_id];       Identifiant étudiant
Et_data    = string[LenEt_data];        données étudiant

Ec_id      = string[LenEc_id];        identifiant échange
Ec_data    = record                      données échange

                 Ec_sol    : array[1..Essai]
                              of string[LenEc_sol];
                                 sollicitation du système
                 Ec_rep    : array[1..Essai]
                              of string[LenEc_rep];
                                     réponse de l'étudiant
                 Ec_feed   : array[1..Essai]
                              of string[LenEc_feed];
                                                feed-back
                 Ec_inf    : array[1..Essai]
                              of string[LenEc_inf];
                                informations de l'échange
             end;

Ex_id      = string[LenEx_id];       identifiant exercice
Ex_data    = string[LenEx_data];        données exercice
```

```
Cl_id       = string[LenCl_id];        identifiant classe
Cl_data     = string[LenCl_data];         données classe

Co_id       = string[LenCo_id];       identifiant concept
Co_data     = string[LenCo_data];        données concept

Cm_id       = string[LenCm_id];  identifiant complexité
Cm_data     = string[LenCm_data];    données complexité

Cn_id       = string[LenCn_id];      identifiant connais.
Cn_data     = string[LenCn_data];  données connaissance
Cn_ner      = integer;          nombre exercices réussis.
Cn_nex      = integer;          nombre total d'exercices.

inter       = Ø .. maxint;    n° de référence dans une
                              liste de résultats.

targs       = Ø .. Nargs;     nombre d'arguments dans une
                              liste.

string2      = string[2];      chaîne de caractères de
                              longueur 2 utilisée pour
                              convertir des entiers en
                              caractères (pour les clés).

tconcepts = array [1 .. Nargs] of co_id;
                              liste de concepts.
tclasses    = array [1 .. Nargs] of cl_id;
                              liste de classes.
```

8.3.3. Description des fichiers de l'historique.

- Description des fichiers de données.

Chaque enregistrement d'un fichier de données contient en première position un entier (Status). Il possède une valeur égale à Ø lorsque le record le contenant est correct et une valeur non nulle lorsque celui-ci a été supprimé par Turbo Access (suppression logique). Cela permet une reconstitution des fichiers en cas de corruption des index (voir manuel).

Description du fichier Etudiant.

```
EtdRec = record
              Status    : integer;
              Etudiant  : et_id;
              EtData    : et_data;
              EtNum     : integer;
         end;
```

Rem. : le fichier contient le numéro du dernier échange effectué (Ø si pas d'échange). Cela facilite l'attribution automatique d'un numéro aux nouveaux échanges

Description du fichier Echange.

```
EchRec = record
              Status      : integer;
              EcEtudiant  : et_id;
              EcNum       : integer;
              EcExercice  : ex_id;
              EcData      : ec_data;
         end;
```

Les records de données du domaine d'enseignement contiendront une variable booléenne (**M**) dont la valeur <u>true</u> signifiera qu'il y a eu une tentative (erronée) de suppression de l'article correspondant. Cela aura pour but de restreindre toute utilisation ultérieure de ces données ou informations.

Description du fichier Exercice.

```
ExrRec = record
              Status     : integer;
              M          : boolean;
              Exercice   : ex_id;
              ExData     : ex_data;
         end;
```

Description du fichier Classe.

```
ClsRec = record
              Status     : integer;
              M          : boolean;
              Classe     : cl_id;
              ClData     : cl_data;
              ClComplexite : cm-id;
         end;
```

Description du fichier Concept.

```
CnpRec = record
            Status     : integer;
            M          : boolean;
            Concept    : co_id;
            CoData     : co_data;
         end;
```

Description du fichier Complexité.

```
CplRec = record
            Status      : integer;
            M           : boolean;
            Complexite  : cm_id;
            CmData      : cm_data;
         end;
```

Description du fichier Connaissance.

```
CnnRec = record
            Status      : integer;
            CnEtudiant  : et_id;
            CnConcept   : co_id;
            CnNer       : cn_ner;
            CnNex       : cn_nex;
            CnData      : cn_data;
         end;
```

<u>- Description des fichiers d'index.</u>

Un fichier d'index contient un ensemble de clés et de références vers un fichier de données. Les clés sont généralement identifiantes excepté celles des fichiers constituant un chemin d'accès entre deux fichiers de données. Les clés doivent être des chaînes de caractères et ne peuvent être numériques. Il faut utiliser les fonctions IntToStr et StrToInt pour convertir des entiers en chaînes de caractères de longueur égale à 2 bytes et réciproquement. Une fois le nombre transformé, il peut être utilisé comme clé d'accès.

L'identifiant d'un échange (numéro de l'échange) possède une valeur numérique. La clé est obtenue en transformant ce numéro en string de longueur 2 qui est ajouté à l'identifiant de l'étudiant correspondant. Cependant, pour des raisons de facilité, cette valeur est mémorisée sous forme numérique dans les fichiers de données. C'est le seul cas où un identifiant est différent de la valeur de la clé correspondante dans le fichier d'index.

8.4. Description des fonctions & des modules.

Les fonctions de chaque module seront représentées par des procédures en Turbo Pascal. Celles-ci mettront à jour la variable globale ETAT qui contiendra le résultat de l'exécution de la procédure. Ce résultat est représenté par un code d'erreur dont les valeurs sont données en annexe. Un code égal à zéro signifie que l'exécution de la procédure s'est correctement déroulée. Les paramètres en input sont passés par valeur (excepté les tableaux) tandis que les paramètres contenant le résultat éventuel des procédures sont passés par adresse. Toutes les procédures définies ci-après sont spécifiées selon le schéma suivant :

Nom de la procédure, nom & type des paramètres

Description des paramètres

Description de la procédure

Description du contenu de la variable ETAT.

Certaines procédures fournissent une liste de résultats (ex: Rech_Echanges_Etudiant). Les résultats sont donnés successivement lors de chaque appel. Ces procédures possèdent un paramètre spécial appelé **nref** qui contient le numéro de séquence du résultat. Ce paramètre doit être initialisé à Ø avant le premier appel pour permettre à la procédure de le distinguer des appels suivants sinon il n'y a aucune garantie des résultats. Au cours du premier appel, la validité des paramètres est testée et les pointeurs vers les fichiers de données sont initialisés. Ensuite, à la fin de chaque appel, la variable nref est incrémentée automatiquement par la procédure jusqu'à ce qu'il n'y ait plus de données à rechercher.

Certaines fonctions, employant les informations du domaine d'enseignement, mettent à jour la marque de suppression (variable m) qui sera positionnée à true, lorsqu'on essaye de supprimer les données qui doivent être conservées (pour sauvegarder la cohérence de l'historique). L'utilisateur tiendra compte ou non, s'il le désire, de ce code lors de la consultation de l'historique.

8.4.1. Module Initialisation.

- Procedure **Erreur;**

Paramètres : -

Description : cette procédure attribue à la variable globale MES_ERR un message d'erreur en fonction du contenu de la variable ETAT. Ce message peut être affiché à l'écran pour signaler à l'utilisateur qu'une situation d'exception s'est produite (voir annexe B).

Contenu ETAT : -

- Procedure **CloseHist;**

Paramètres : -

Description : cette procédure effectue la clôture de tous les fichiers utilisés par la gestion de l'historique. Elle doit être appelée en fin d'utilisation du système d'E.A.O. pour s'assurer que toutes les modifications effectuées sur l'historique sont bien mémorisées.

Contenu ETAT : -

- Procedure **CreatHist;**

Paramètres : -

Description : cette procédure effectue la création et l'initialisation des fichiers utilisés par la gestion de l'historique. Elle doit être appelée uniquement lorsque l'on désire créer un nouvel historique. Au terme de cette procédure, tous les fichiers créés sont fermés. Il est donc nécessaire d'appeler la procédure OpenHist dès la première utilisation du système d'E.A.O. Si on ne parvient pas à créer les fichiers de l'historique, le programme est immédiatement interrompu et un message d'erreur s'affiche.

Contenu ETAT : -

- Procedure **OpenHist**;

Paramètres : -

Description : cette procédure ouvre et initialise tous les fichiers utilisés par la gestion de l'historique. Elle doit être obligatoirement appelée avant la première utilisation du système d'E.A.O. Si l'historique n'existe pas encore, il faut utiliser la procédure CreatHist avant de faire appel à celle-ci. Si on ne parvient pas à ouvrir les fichiers de l'historique, le programme est immédiatement interrompu et un message d'erreur s'affiche.

Contenu ETAT : -

- Function IntToStr(N : integer) : String2;

Parametres :

N : entier ≥ Ø à convertir.

Description : cette fonction effectue la conversion d'un entier positif en chaîne de caractères de longueur égale à 2.

Contenu ETAT : -

- Function StrToInt(S : String2) : integer;

Parametres :

S : chaîne de caractères à convertir.

Description : cette fonction effectue la conversion d'une chaîne de caractères de longueur égale à 2 en entier positif.

Contenu ETAT : -

8.4.2. Module étudiant.

- Procedure **Rech_Etudiant**(etudiant : et_id;
  var num : integer;
  var donnees : et_data);

Paramètres :

etudiant : identifiant d'un étudiant.
num : numéro du dernier échange effectué.
donnees : informations relatives à l'étudiant.

Description : cette procédure recherche et fournit les informations concernant un étudiant ainsi que le numéro du dernier échange qu'il a effectué (= Ø s'il n'y a pas encore eu d'échange).

Contenu ETAT : Ø si le résultat est correct, 5 si l'étudiant n'existe pas.

- Procedure **Creation_Etudiant**(etudiant : et_id;
  donnees : et_data);

Paramètres :

etudiant : identifiant d'un étudiant.
donnees : informations relatives à l'étudiant.

Description : cette procédure effectue la création d'un étudiant dans l'historique.

Contenu ETAT : Ø si l'étudiant est créé, 6 si l'étudiant existe déjà.

- Procedure **Modif_Etudiant**(etudiant : et_id;
  donnees : et_data);

Paramètres :

etudiant : identifiant d'un étudiant.
donnees : informations relatives à l'étudiant.

Description : cette procédure effectue la mise à jour des informations concernant un étudiant.

Contenu ETAT : Ø si la modification est correctement effectuée, 5 si l'étudiant n'existe pas.

- Procedure **Suppres_Etudiant**(etudiant : et_id);

Paramètres :

etudiant : identifiant d'un étudiant.

Description : cette procédure effectue la suppression d'un étudiant de l'historique. Tous les échanges et les connaissances de celui-ci sont également supprimés.

Contenu ETAT : Ø si l'étudiant est supprimé, 5 si l'étudiant n'existe pas.

- Procedure **Liste_Etudiants**(var etudiant : et_id;
  var donnees : et_data;
  var nref : inter);

Paramètres :

etudiant : identifiant de l'étudiant suivant.
donnees : informations relatives à l'étudiant.
nref : n° de référence de l'étudiant.

Description : cette procédure renvoie à chaque appel un nouvel étudiant ainsi que les informations le concernant.

Contenu ETAT : Ø si le résultat est correct, 1 s'il n'y a plus d'étudiant à consulter.

8.4.3. Module Exercice.

- Procedure **Rech_Exercice**(exercice : ex_id;
  var m : boolean;
  var donnees : ex_data);

Paramètres :

exercice : identifiant d'un exercice.
m : marque de suppression de l'exercice.
donnees : informations relatives à l'exercice.

Description : cette procédure recherche et fournit les informations concernant un exercice.

Contenu ETAT : Ø si le résultat est correct, 5 si l'exercice n'existe pas.

- Procedure **Creation_Exercice**(exercice : ex_id;
  donnees : ex_data);

  Paramètres :

  exercice : identifiant d'un exercice.
  donnees : informations relatives à l'exercice.

  Description : cette procédure effectue la création d'un exercice dans l'historique.

  Contenu ETAT : Ø si l'exercice est créé, 6 si l'exercice existe déjà.

- Procedure **Modif_Exercice**(exercice : ex_id;
  donnees : ex_data);

  Paramètres :

  exercice : identifiant d'un exercice.
  donnees : informations relatives à l'exercice.

  Description : cette procédure effectue la mise à jour des informations concernant un exercice.

  Contenu ETAT : Ø si la modification est correctement effectuée, 5 si l'exercice n'existe pas, 12 si l'exercice ne peut être modifié.

- Procedure **Liste_Exercices**(var exercice : ex_id;
  m : boolean;
  var donnees : ex_data;
  var nref : inter);

  Paramètres :

  exercice : identifiant de l'exercice suivant.
  m : marque de suppression de l'exercice.
  donnees : informations relatives à l'exercice.
  nref : n° de référence de l'exercice.

  Description : cette procédure renvoie à chaque appel un nouvel exercice ainsi que les informations le concernant.

  Contenu ETAT : Ø si le résultat est correct, 1 s'il n'y a plus d'exercice à consulter.

- Procedure **Suppres_Exercice**(exercice : ex_id);

  Paramètres :

  exercice : identifiant d'un exercice.

  Description : cette procédure effectue la suppression d'un exercice de l'historique ainsi que les éventuelles relations entre l'exercice et des classes Un exercice ne peut être retiré lorsqu'il est associé à au moins un échange. Dans ce cas, la marque de tentative de suppression sera positionnée.

  Contenu ETAT : Ø si l'exercice est supprimé, 5 si l'exercice n'existe pas, 2 si l'exercice ne peut être supprimé.

8.4.4. Module Concept.

- Procedure **Rech_Concept**(concept : co_id;
  var m : boolean
  var donnees : co_data);

  Paramètres :

  concept : identifiant d'un concept.
  m : marque de suppression de ce concept.
  donnees : informations relatives au concept.

  Description : cette procédure recherche et fournit les informations concernant un concept.

  Contenu ETAT : Ø si le résultat est correct, 5 si le concept n'existe pas.

- Procedure **Creation_Concept**(concept : co_id;
  donnees : co_data);

  Paramètres :

  concept : identifiant d'un concept.
  donnees : informations relatives au concept.

  Description : cette procédure effectue la création d'un concept dans l'historique.

  Contenu ETAT : Ø si le résultat est correct, 6 si le concept existe déja.

- Procedure **Modif_Concept**(concept : co_id;
  donnees : co_data);

  Paramètres :

  concept : identifiant d'un concept.
  donnees : informations relatives au concept.

  Description : cette procédure effectue la mise à jour des informations concernant un un concept.

  Contenu ETAT : Ø si la modification est correctement effectuée, 5 si le concept n'existe pas.

- Procedure **Liste_Concepts**(var concept : co_id;
  var donnees : co_data;
  var nref : inter);

  Paramètres :

  concept : identifiant du concept suivant.
  donnees : informations relatives au concept.
  nref : n° de référence du concept.

  Description : cette procédure renvoie à chaque appel un nouveau concept ainsi que les informations le concernant.

  Contenu ETAT : Ø si le résultat est correct, 1 s'il n'y a plus de concept à consulter.

- Procedure **Suppres_Concept**(concept : co_id);

  Paramètres :

  concept : identifiant d'un concept.

  Description : cette procédure effectue la suppression d'un concept de l'historique. Si le concept est associé à au moins un échange où une connaissance, il ne peut être supprimé. Cependant, toutes les relations éventuelles entre le concept, un niveau de complexité et/ou des classes sont quand même retirées de l'historique.

  Contenu ETAT : Ø si le concept est supprimé, 5 si le concept n'existe pas, 2 si le concept ne peut être supprimé.

8.4.5. Module Classe.

- Procedure **Rech_Classe**(classe : cl_id;
  var m : boolean;
  var donnees : cl_data);

  Paramètres :

  classe : identifiant d'une classe.
  m : marque de suppression de la classe.
  donnees : informations relatives à la classe.

  Description : cette procédure recherche et fournit les informations concernant une classe.

  Contenu ETAT : Ø si le résultat est correct, 5 si la classe n'existe pas.

- Procedure **Creation_Classe**(classe : cl_id;
  donnees : cl_data);

  Paramètres :

  classe : identifiant d'une classe.
  donnees : informations relatives à la classe.

  Description : cette procédure effectue la création d'une classe dans l'historique.

  Contenu ETAT : Ø si la classe est créée, 6 si la classe existe déja.

- Procedure **Modif_Classe**(classe : cl_id;
  donnees : cl_data);

  Paramètres :

  classe : identifiant d'une classe.
  donnees : informations relatives à la classe.

  Description : cette procédure effectue la mise à jour des informations concernant une classe.

  Contenu ETAT : Ø si la modification est correctement effectuée, 5 si la classe n'existe pas, 12 si la classe ne peut être modifiée.

```
- Procedure Liste_Classes(var classe  : cl_id;
                          var m       : boolean;
                          var donnees : cl_data;
                          var nref    : inter);
```

Paramètres :

```
classe  : identifiant de la classe suivante.
m       : marque de suppression de la classe.
donnees : informations relatives à la classe.
nref    : n° de référence de la classe.
```

Description : cette procédure renvoie à chaque appel une nouvelle classe ainsi que les informations la concernant.

Contenu ETAT : Ø si le résultat est correct, 1 s'il n'y a plus de classe à consulter.

```
- Procedure Suppres_Classe(classe : cl_id);
```

Paramètres :

```
classe : identifiant d'une classe.
```

Description : cette procédure effectue la suppression d'une classe de l'historique. Les éventuelles relations entre celle-ci, des concepts et des exercices sont également supprimées.

Contenu ETAT : Ø si la classe est supprimée, 5 si la classe n'existe pas, 2 si la classe ne peut être supprimée. Dans ce cas, la marque de tentative de suppression sera positionnée..

```
- Procedure Creation_Contient(exercice : ex_id;
                              classe   : cl_id);
```

Paramètres :

```
exercice : identifiant d'un exercice.
classe   : identifiant d'une classe.
```

Description : cette procédure effectue la création d'une relation entre une classe et un exercice.

Contenu ETAT : Ø si la création est effectuée, 5 si la classe ou l'exercice n'existent pas, 7 si la relation existe déjà.

- Procedure **Suppres-Contient**(exercice : ex_id;
                                   classe   : cl_id);

  Paramètres :

  exercice : identifiant d'un exerice.
  classe   : identifiant d'une classe.

  Description : cette procédure effectue la suppression d'une relation existant entre une classe et un exercice.

  Contenu ETAT : Ø si la suppression est correctement effectuée, 5 si l'exercice ou la classe n'existent pas, 8 si la relation n'existe pas, 2 si la relation ne peut être supprimée.

- Procedure **Creation_Sous_Jacent**(concept : co_id;
                                       classe : cl_id);

  Paramètres :

  concept : identifiant d'un concept.
  classe : identifiant d'une classe.

  Description : cette procédure effectue la création d'une relation entre un concept et une classe.

  Contenu ETAT : Ø si la création est effectuée, 5 si le concept ou la classe n'existent pas, 7 si la relation existe déjà.

- Procedure **Suppres_Sous_Jacent**(concept : ex_id;
                                      classe : cl_id);

  Paramètres :

  concept : identifiant d'un concept.
  classe : identifiant d'une classe.

  Description : cette procédure effectue la suppression d'une relation existant entre un concept et une classe.

  Contenu ETAT : Ø si la suppression est correctement effectuée, 5 si le concept ou la classe n'existent pas, 8 si la relation n'existe pas, 2 si la relation ne peut être modifiée.

- Procedure **Rech_Exercices_Classe**(classe : cl_id;
var m : boolean;
var exercice : ex_id;
var donnees : ex-data;
var nref : inter);

Paramètres :

classe : identifiant d'une classe.
exercice : identifiant d'un exercice.
m : marque de suppression de l'exercice.
donnees : informations relatives à l'exercice.
nref : n° de référence de l'exercice.

Description : cette procédure recherche et fournit à chaque appel un exercice d'une classe spécifiée ainsi que les informations le concernant.

Contenu ETAT : Ø si le résultat est correct, 5 si la classe n'existe pas, 1 s'il n'y a plus d'exercice.

- Procedure **Rech_Classes_Exercice**(exercice : ex_id;
var classe : cl_id;
var m : boolean;
var donnees : cl_data;
var nref : inter);

Paramètres :

exercice : identifiant d'un exercice.
classe : identifiant d'une classe.
m : marque de suppression de la classe.
donnees : informations relatives à la classe.
nref : n° de référence de la classe.

Description : cette procédure recherche et fournit à chaque appel une classe contenant un exercice spécifié ainsi que les informations la concernant.

Contenu ETAT : Ø si le résultat est correct, 5 si l'exercice n'existe pas, 1 s'il n'y a plus de classe contenant l'exercice.

- Procedure **Rech_Concepts_Classe**(classe : cl_id; var concept : co_id; var m : boolean; var donnees : co_data; var nref : inter);

Paramètres :

classe : identifiant d'une classe.
concept : identifiant d'un concept.
m : marque de suppression d'un concept.
donnees : informations relatives au concept.
nref : n° de référence du concept.

Description : cette procédure recherche et fournit à chaque appel un concept sous-jacent à une classe spécifiée ainsi que les informations correspondantes.

Contenu ETAT : Ø si le résultat est correct, 5 si la classe n'existe pas, 1 s'il n'y a plus de concept sous-jacent à la classe.

- Procedure **Rech_Classes_Concept**(concept : co_id; var classe : cl_id; var m : boolean; var donnees : cl_data; var nref : inter);

Paramètres :

concept : identifiant d'un concept.
classe : identifiant d'une classe.
m : marque de suppression de la classe.
donnees : informations relatives à la classe.
nref : n° de référence de la classe.

Description : cette procédure recherche et fournit à chaque appel une classe à laquelle un concept spécifié est sous-jacent ainsi que les informations correspondantes.

Contenu ETAT : Ø si le résultat est correct, 5 si le concept n'existe pas, 1 s'il n'y a plus de classe à laquelle le concept est sous-jacent.

- Procedure **Rech_Concepts_Classes**

```
(args         : targs;
 var classes  : tclasses;
 var concept  : co_id;
 var m        : boolean;
 var donnees  : co_data;
 var nref     : inter);
```

Paramètres :

args : nombre d'arguments dans la liste des classes.
classes : liste des identifiants de classe.
concept : identifiant d'un concept.
m : marque de suppression du concept.
donnees : informations relatives au concept.
nref : n° de référence du concept.

Description : cette procédure recherche et fournit à chaque appel un concept sous-jacent à un ensemble de classes spécifiées ainsi que les informations correspondantes.

Contenu ETAT : Ø si le résultat est correct, 3 si le nombre de classes spécifiées est égal à Ø, 5 si au moins une des classes n'existe pas, 1 s'il n'y a plus de concept sous-jacent aux classes.

- Procedure **Rech_Classes_Concepts**

```
(args          : targs;
 var concepts  : tconcepts;
 var classe    : co_id;
 var m         : boolean;
 var donnees   : co_data;
 var nref      : inter);
```

Paramètres :

args : nombre d'arguments dans la liste des concepts.
concepts : liste des identifiants de concepts.
classe : identifiant d'une classe.
m : marque de suppression d'une classe.
donnees : informations relatives à la classe.
nref : n° de référence de la classe.

Description : cette procédure recherche et fournit à chaque appel une classe contenant un ensemble de concepts sous-jacents spécifiés ainsi que les informations correspondantes.

Contenu ETAT : Ø si le résultat est correct, 3 si le nombre de concepts spécifiés est égal à Ø, 5 si au moins un des concepts n'existe pas, 1 s'il n'y a plus de classe contenant les concepts sous-jacents.

- Procedure **Rech_Complexite_Classe**

```
(classe          : cl_id;
 var complexite  : cm_id;
 var m           : boolean;
 var donnees     : cm_data);
```

Paramètres :

classe	: identifiant d'une classe.
complexite	: identifiant d'un niveau de complexité.
m	: marque de suppression du niveau de complexité.
donnees	: informations relatives au niveau de complexité.

Description : cette procédure recherche le niveau de complexité éventuel d'une classe ainsi que les informations le concernant.

Contenu ETAT : Ø si le résultat est correct, 5 si la classe n'existe pas, 1 s'il n'existe pas de niveau de complexité.

- Procedure **Rech_Classes_Complexite**

```
(complexite   : cm_id;
 var classe   : cl_id;
 var m        : boolean;
 var donnees  : co_data;
 var nref     : inter);
```

Paramètres :

complexite	: identifiant d'un niveau de complexité.
classe	: identifiant d'une classe.
m	: marque de suppression de la classe.
donnees	: informations relatives à la classe.
nref	: n° de référence du concept.

Description : à chaque appel successif, cette procédure recherche et fournit une classe associée au niveau de complexité spécifié ainsi que les informations la concernant.

Contenu ETAT : Ø si le résultat est correct, 5 si le niveau de complexité n'existe pas, 1 s'il n'y plus de classe.

- Procedure **Creation_Caracterise**(classe : cl_id;
complexite : cm_id);

Paramètres :

classe : identifiant d'une classe.
complexite : identifiant d'un niveau de complexité.

Description : cette procédure effectue la création d'une relation entre une classe et un niveau de complexité.

Contenu ETAT : Ø si la relation est créée, 5 si la classe ou le niveau de complexité n'existent pas, 7 si la classe est déjà associée à un niveau de complexité.

- Procedure **Suppres_Caracterise**(classe : cl_id;
complexite : cm_id);

Paramètres :

classe : identifiant d'un classe.
complexite : identifiant d'un niveau de complexité.

Description : cette procédure effectue la suppression d'une relation existant entre une classe et un niveau de complexité.

Contenu ETAT : Ø si la relation est supprimée, 5 si la classe ou le niveau de complexité n'existent pas, 8 si la relation n'existe pas, 2 si elle ne peut être supprimée.

3.4.6. Module Complexite.

- Procedure **Rech_Complexite**(complexite : cm_id;
  var m : boolean.
  var donnees : cm_data);

  Paramètres :

  complexite : identifiant d'un niveau de complexité.
  m : marque de suppression d'un niveau de complexité.
  donnees : informations relatives au niveau de complexité.

  Description : cette procédure recherche et fournit les informations concernant un niveau de complexité.

  Contenu ETAT : Ø si le résultat est correct, 5 si le niveau de complexité n'existe pas.

- Procedure **Creation_Complexite**(complexite : cm_id;
  donnees : cm_data);

  Paramètres :

  complexite : identifiant d'un niveau de complexité.
  donnees : informations relatives au niveau de complexité.

  Description : cette procédure effectue la création d'un niveau de complexité dans l'historique.

  Contenu ETAT : Ø si le niveau de complexité est créé, 6 si le niveau de complexité n'existe pas.

- Procedure **Modif_Complexite**(complexite : cm_id;
  donnees : cm_data);

  Paramètres :

  complexite : identifiant d'un niveau de complexité.
  donnees : informations relatives au niveau de complexité.

  Description : cette procédure effectue la mise à jour des informations concernant un niveau de complexité.

  Contenu ETAT : Ø si la modification est effectuée, 5 si le niveau de complexité n'existe pas, 12 si le niveau de complexité ne peut être modifié.

- Procedure **Liste_Complexites**(var complexite : cm_id;
  var m : boolean;
  var donnees : cm_data;
  var nref : inter);

  Paramètres :

  complexite : identifiant du niveau de complexité suivant.
  donnees : informations relatives à la complexité.
  m : marque de suppression de la complexité.
  nref : n° de référence du niveau de complexité.

  Description : cette procédure renvoie à chaque appel un niveau de complexité ainsi que les informations le concernant.

  Contenu ETAT : 0 si le résultat est correct, 1 s'il n'y a plus de niveau de complexité à consulter.

- Procedure **Suppres_Complexite**(complexite : cm_id);

  Paramètres :

  complexite : identifiant d'un niveau de complexité.

  Description : cette procédure effectue la suppression d'un niveau de complexité de l'historique. Toutes les relations éventuelles entre celui-ci et des concepts sont également supprimées.

  Contenu ETAT : 0 si la suppression a été effectuée, 5 si la complexité n'existe pas, 2 si la complexité ne peut être supprimée.

8.4.7. Module Echange.

- Procedure **Creation_Echange**(etudiant : et_id;
  exercice : ex_id;
  donnees : ec_data;
  args : targs;
  var concepts : tconcepts;

Paramètres :

etudiant : identifiant d'un étudiant.
exercice : identifiant d'un exercice.
donnees : informations relatives à un échange.
args : nombre de concepts non maîtrisé lors de l'échange (nombre d'arguments ≥ Ø).
concepts : liste des concepts non maîtrisés.

Description : cette procédure effectue la création dans l'historique d'un échange et met à jour automatiquement les connaissances (CN-NEX & CN-NEX) de l'étudiant relatives aux concepts non maîtrisés lors de l'échange ainsi que les concepts sous-jacents à l'exercice. Les connaissances sont créées si elles n'existaient pas encore et les informations correspondantes (CN-DATA) sont mises à blanc. La procédure recherche dans le fichier des étudiants le numéro suivant à attribuer à l'échange créé.

Contenu ETAT : Ø si la création s'est effectuée correctement, 5 si l'exercice ou l'étudiant n'existe pas, 4 si au moins un des concepts n'existe pas, 13 si un des concepts n'est pas sous-jacent à l'exercice, 9 si on a dépassé le nombre limite d'arguments (= Nargs) (dans la recherche des concepts sous-jacents à l'exercice). Dans ces trois derniers cas, la création de l'échange est quand même effectuée.

- Procedure **Rech_Echange**(etudiant : et_id;
  num : integer;
  var exercice : ex_id;
  var donnees : ec_data);

Paramètres :

etudiant : identifiant d'un étudiant.
num : numéro d'un échange.
exercice : identifiant d'un exercice.
donnees : informations relatives à l'échange.

Description : cette procédure recherche et fournit les informations concernant un échange.

Contenu ETAT : Ø si le résultat est correct, 5 si l'étudiant ou l'échange n'existent pas.

- Procedure **Rech_Echanges_Etudiant**

```
(etudiant     : et_id;
 var num      : integer;
 var exercice : ex_id;
 var donnees  : ec_data;
 var nref     : inter);
```

Paramètres :

```
etudiant : identifiant d'un étudiant.
num      : numéro de l'échange suivant.
exercice : identifiant d'un exercice.
donnees  : informations relatives à l'échange.
nref     : n° de référence de l'échange.
```

Description : cette procédure recherche et fournit successivement tous les échanges d'un étudiant.

Contenu ETAT : Ø si le résultat est correct, 5 si l'étudiant n'existe pas, 1 s'il n'y a plus d'échange.

- Procedure **Rech_Echanges_Exercice**

```
(exercice     : ec_id;
 var etudiant : et_id;
 var num      : integer;
 var donnees  : ec_data;
 var nref     : inter);
```

Paramètres :

```
exercice : identifiant d'un exercice.
étudiant : identifiant d'un étudiant.
num      : numéro de l'échange suivant.
donnees  : informations relatives à l'échange.
nref     : n° de référence de l'échange.
```

Description : cette procédure recherche et fournit successivement tous les échanges d'un exercice.

Contenu ETAT : Ø si le résultat est correct, 5 si l'exercice n'existe pas, 1 s'il n'y a plus d'échange.

```
- Procedure Rech_Echanges_Concept(concept       : co_id;
                                  var etudiant : et_id;
                                  var num      : integer;
                                  var exercice : ex_id;
                                  var donnees  : ec_data;
                                  var nref     : inter);
```

Paramètres :

```
concept   : identifiant d'un concept.
etudiant  : identifiant d'un étudiant.
num       : numéro de l'échange suivant.
exercice  : identifiant d'un exercice.
donnees   : informations relatives à l'échange.
nref      : n° de référence de l'échange.
```

Description : cette procédure recherche tous les échanges durant lesquels un concept spécifié n'a pas été maîtrisé.

Contenu ETAT : Ø si le résultat est correct, 5 si le concept n'existe pas, 1 s'il n'y a plus d'échange.

```
- Procedure Rech_Concepts_Echange(etudiant     : et_id;
                                  num         : integer;
                                  var concept : co_id;
                                  var m       : boolean;
                                  var donnees : co_data;
                                  var nref    : inter);
```

Paramètres :

```
etudiant  : identifiant d'un étudiant.
num       : numéro d'un échange.
concept   : identifiant du concept suivant.
m         : marque de suppression du concept.
donnees   : informations relatives au concept.
nref      : n° de référence du concept.
```

Description : cette procédure recherche tous les concepts non maîtrisés durant un échange.

Contenu ETAT : Ø si le résultat est correct, 5 si l'étudiant ou l'échange n'existent pas, 1 s'il n'y a plus de concept.

8.4.8. Module Connaissance.

- Procedure **Rech_Connaissance**(etudiant : et_id;
concept : co_id;
var ner : cn_ner;
var nex : cn_nex;
var donnees : cn_data);

Paramètres :

etudiant : identifiant d'un étudiant.
concept : identifiant d'un concept.
ner : nombre d'exercices réussis.
nex : nombre total d'exercices présentés à l'apprenant.
donnees : informations relatives à la connaissance.

Description : cette procédure recherche et fournit les informations relatives à la connaissance d'un concept par un étudiant.

Contenu ETAT : Ø si le résultat est correct, 5 si le concept, l'étudiant ou la connaissance correspondante n'existent pas.

- Procedure **Modif_connaissance**(etudiant : et_id;
concept : co_id;
donnees : cn_data;
ner : cn_ner;
nex : cn_nex);

Parametres :

etudiant : identifiant d'un étudiant.
concept : identifiant d'un concept.
donnees : informations relatives à la connaissance.
ner : nombre d'exercices réussis.
nex : nombre total d'exercices présentés à l'apprenant.

Description : cette procédure effectue la mise à jour des informations concernant une connaissance.

Contenu ETAT : Ø si la modification a été correctement effectuée, 5 si l'étudiant, le concept ou la connaissance n'existent pas.

- Procedure **Rech_Connaissances_Etudiant**

```
(etudiant     : et_id;
 var concept : co_id;
 var ner     : cn_ner;
 var nex     : cn_nex;
 var donnees : cn_data;
 var nref    : inter);
```

Paramètres :

etudiant : identifiant d'un étudiant.
concept : identifiant d'un concept.
ner : nombre d'exercices réussis.
nex : nombre total d'exercices.
donnees : informations relative à la connaissance.
nref : n° de référence de la connaissance.

Description : cette procédure recherche et fournit successivement toutes les connaissances d'un étudiant.

Contenu ETAT : Ø si le résultat est correct, 5 si l'étudiant n'existe pas, 1 s'il n'y a plus de connaissance.

- Procedure **Rech_Connaissances_Concept**

```
(concept      : co_id;
 var etudiant : et_id;
 var ner      : cn_ner;
 var nex      : cn_nex;
 var donnees  : cn_data;
 var nref     : inter);
```

Paramètres :

concept : identifiant d'un concept.
etudiant : identifiant d'un étudiant.
ner : nombre d'exercices réussis.
nex : nombre total d'exercices.
donnees : informations relative à la connaissance.
nref : n° de référence de la connaissance.

Description : cette procédure recherche et fournit successivement toutes les connaissances relatives à un concept.

Contenu ETAT : Ø si le résultat est correct, 5 si le concept n'existe pas, 1 s'il n'y a plus de connaissance.

8.5. Utilisation des modules de l'historique.

Tout programme qui souhaite utiliser les procédures de la gestion de l'historique doit respecter certaines conventions. Il est nécessaire avant tout de spécifier correctement au début du programme les constantes décrites précédemment, avant d'y inclure (directive de compilation Turbo Pascal = {$I nom de fichier}) les modules de Turbo Access System et d'y inclure ou de recopier le ou les modules désirés de la gestion de l'historique (voir annexe C).

De même, il est obligatoire que le module d'Initialisation soit présent en tête des autres modules car il contient la définition des types de données, les procédures de création et d'ouverture de l'historique ainsi que des procédures spécifiques à la gestion de l'historique (leur nom commence par **Hst**) qui sont utilisés par les autres modules.

Le concepteur du système d'E.A.O. pourra recopier uniquement les procédures qui lui sont nécessaires. Le système d'E.A.O. peut être composé de plusieurs programmes distincts (par exemple, un programme de dialogue avec l'apprenant, un programme de mise à jour des données par l'enseignant,...) qui utilisent des procédures spécifiques de la gestion de l'historique.

Comme il a déjà été remarqué, les procédures qui fournissent de nouveaux résultats à chaque appel successif testent la validité des paramètres et initialisent les pointeurs des fichiers d'index lors du premier appel (lorsque le paramètre nref a une valeur nulle). En fait, il est possible d'intercaler des appels à d'autres procédures entre plusieurs appels successifs tant que celles-ci n'effectuent aucun accès aux fichiers d'index utilisés par la première procédure. Sinon, les pointeurs internes sont modifiés et les résultats ne sont plus garantis comme étant corrects. Il faudra alors réinitialiser nref à 0 avant le prochain appel de la procédure initiale. Si nref n'est pas initialisée à 0, la recherche s'effectuera à partir de l'endroit où est positionné le pointeur interne du fichier d'index.

Mise à part l'intégration de la variable Status dans les enregistrements des fichiers de données, ce qui facilite la reconstitution de ceux-ci ainsi que des fichiers d'index par une lecture séquentielle (voir manuel Turbo Access), aucune mesure de protection et de récupération des fichiers n'a été prévue. La corruption de l'historique se produit lorsqu'une panne de courant survient lors d'une mise à jour de fichiers (par exemple, entre l'enregistrement des données dans le Data File et la création de la clé dans le fichier d'index). Toutes les procédures considèrent la base de données comme étant cohérente.

Conclusions.

Le système de gestion de l'historique développé dans ce mémoire s'adresse essentiellement à un système d'E.A.O. très directif. Cependant, on pourrait tenter de l'adapter à une méthode pédagogique non directive. Il faut pour cela que la matière concernée soit très bien structurée. Le problème essentiel réside dans le fait que la notion d'exercice est relativement différente dans cette nouvelle approche. Cependant, la machine doit pouvoir enregistrer les comportements de l'apprenant, analyser les choix de celui-ci pour lui procurer l'aide nécessaire (en effet, un élève abandonné à son propre sort risque rapidement d'être submergé par l'ampleur ou la difficulté de la tâche); cela suppose qu'elle dispose des connaissances et des moyens adéquats. Les résultats de ces différentes interactions pourront être consultés et analysés par l'enseignant, le système d'E.A.O. et l'étudiant.

La notion de complexité correspond au degré de difficulté de résolution des exercices appartenant à une classe. Elle permet une adaptation des exercices présentés aux apprenants afin de les faire progresser vers les objectifs du système. Une autre optique pourrait être envisagée en considérant la complexité de chaque concept. En effet, certains concepts sont plus difficiles à appréhender que d'autres. On peut ainsi établir une véritable hiérarchie, certains concepts intervenant dans d'autres plus élaborés (par exemple, les pré-requis par rapport aux objectifs du système). La complexité des exercices dépend non seulement de celle des concepts sous-jacents, mais aussi de leur qualité propre. Il sera nécessaire d'en tenir compte lors de leur élaboration; des exercices présentant les mêmes concepts peuvent être plus difficiles les uns que les autres. Pour intégrer cette notion, l'enseignant pourrait spécifier explicitement dans le champ CO-DATA de chaque article de type CONCEPT le niveau de complexité conceptuelle ou bien il faudrait définir une relation entre concepts. Le système d'E.A.O. s'en servirait pour analyser plus facilement l'évolution du comportement du sujet vers les objectifs.

La consultation de l'historique doit permettre l'obtention de statistiques intéressantes. Cet aspect a déja été partiellement abordé (voir 2.2. & 4.2.). Par exemple, le nombre total d'exercices et le nombre d'exercices réussis relatifs à un concept particulier, le degré de maîtrise des concepts, etc... serviront à établir un véritable diagnostic des connaissances de chaque étudiant (points forts et faibles de celui-ci), mais aussi des qualités pédagogiques du système d'E.A.O. lui-même à fin de l'améliorer continuellement.

Comme il a été vu précédemment, certaines modifications de l'historique peuvent altérer la signification des données mémorisées antérieurement. Pour éviter cela, les informations sont, dans certains cas, "marquées" et le soin est laissé à l'utilisateur d'en tenir compte ultérieurement ou non. Cependant, cela a comme grave inconvénient d'empêcher toute suppression d'informations erronées (optique de correction d'erreurs). Il sera alors nécessaire de prévoir des fonctions de correction qui effectueront des mises à jour en chaîne. De plus, la plupart des procédures décrites dans ce mémoire étant atomiques, c'est-à-dire ne pouvant être interrompues, si une panne survient, la base de données risque d'être laissée dans un état d'incohérence. Pour remédier à ce genre de situation, on pourrait utiliser des fonctions de protection et de récupération des erreurs, employées sur la plupart des bases de données (ex : backup, commit, rollback).

Le système de gestion de l'historique se base sur un modèle de l'élève très rudimentaire. En effet, il tient compte uniquement de la manifestation des connaissances de l'apprenant et non de ses processus cognitifs, de sa motivation, de ses caractéristiques psychologiques, etc ... Cela constitue, à mon sens, la limitation la plus complexe à surmonter. Le système, tel qu'il est décrit actuellement pourrait servir de base pour des développements futurs de plus en plus sophistiqués. Pour cela, il faut que les théories psycho-pédagogiques soient approfondies en vue d'une adaptation à l'E.A.O. Je souhaite que le système de gestion de l'historique puisse être utilisé dans une grande variété de didacticiels afin d'explorer de nouvelles perspectives visant à élargir les apports pédagogiques de l'E.A.O.

Sources bibliographiques.

1. "Contribution à un cadre conceptuel pour un enseignement adaptatif médiatisé par ordinateur : mise au point et expérimentation de deux dispositifs d'évaluation formative extemporanée. Vol.1".
Thèse de doctorat en Sciences Psycho-Pédagogiques présentée par Christian Depover [Université de l'Etat Mons, Facultés de Sciences Psycho-Pédagogiques (Année académique 1984-1985)].

2. "Domain-Referenced Curriculum evaluation : A technical handbook and a case study from the Minnemast Project."
[CSE Monograph Series in Evaluation 1
- Center for the study of evaluation University of California. Los Angeles (1973)].

3. "Pour apprécier le travail des élèves."
Jean Cardinet
[Institut Romand de Recherches et de documentation pédagogiques
- Service de la Recherche - IRDP/R 84.12 (août 1984)].

4. "Traité de docimologie - L'évaluation des élèves dans la pratique de la classe."
Lowell A. Schoer
[Presses universitaires de France (1975)].

5. "Informatique et enseignement : A la recherche des systèmes E.A.O. intelligents."
Dick Vervenne
[DATA News + data careers : l'actualité informatique n°14 avril 87].

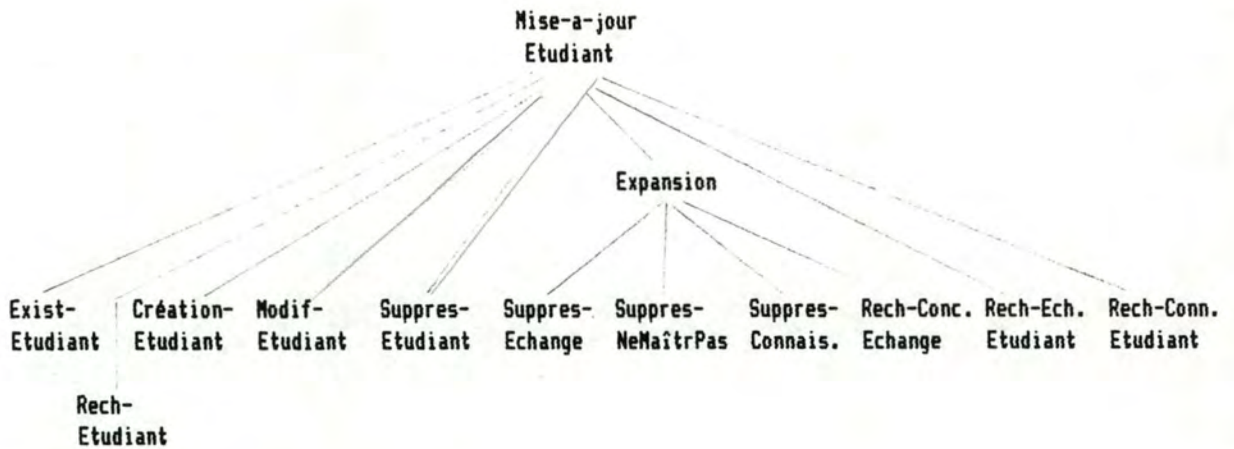
6. "The Lisp Tutor."
John R. Anderson and Brian J. Reiser
[Byte : avril 1985].

7. "L'apport d'un tutorat informatisé."
E. Leclercq-Boxus et P. Delfosse
[REVUE de la direction générale de l'organisation des études : 22e année n° 3 Mars 1987].

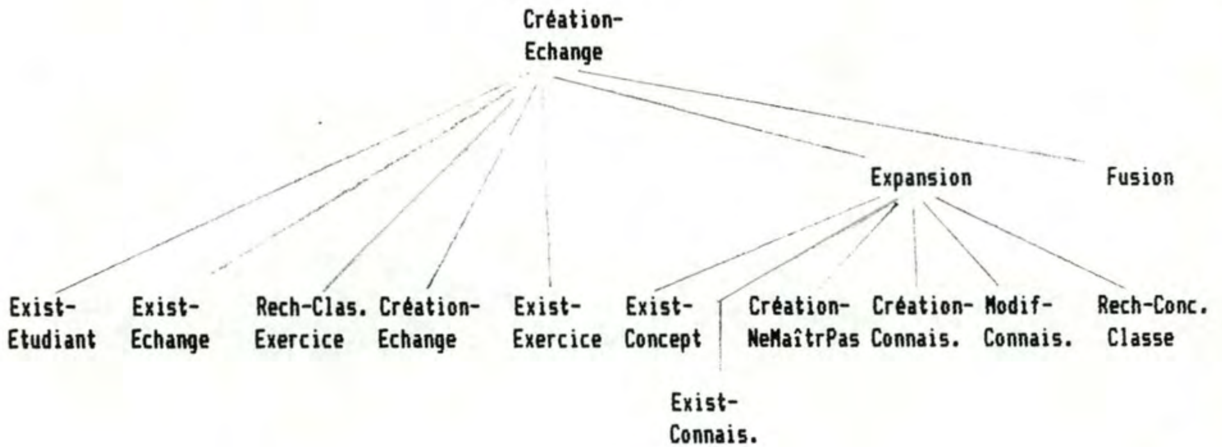
ANNEXES

Annexe A : schémas des modules de la conception logique.

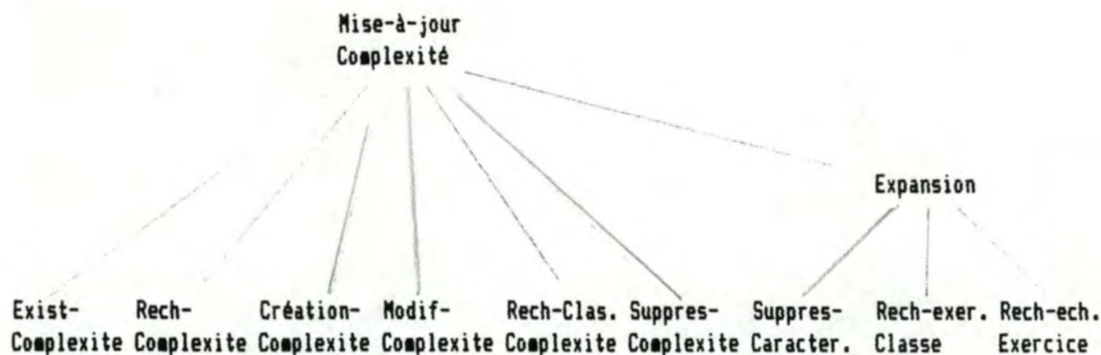
A.1. MISE-A-JOUR-ETUDIANT.



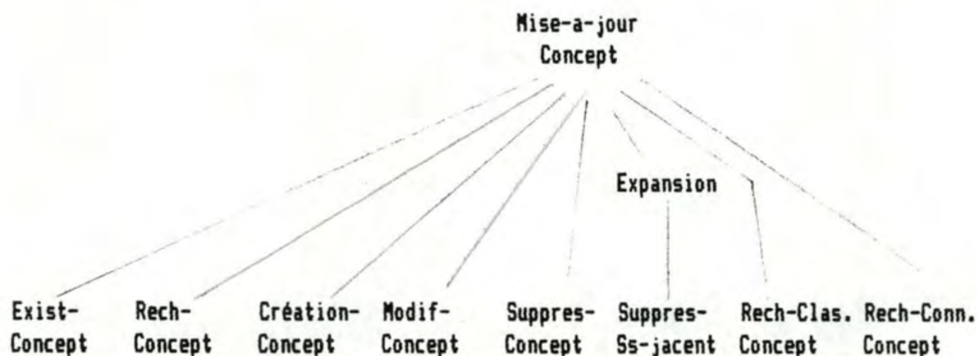
A.2. CREATION-ECHANGE.



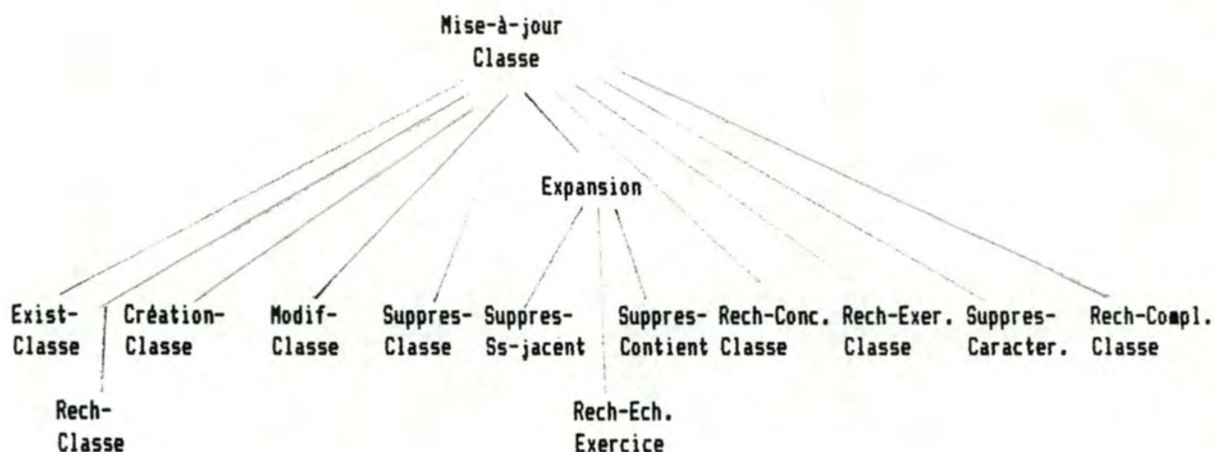
A.3. MISE-A-JOUR-COMPLEXITE.



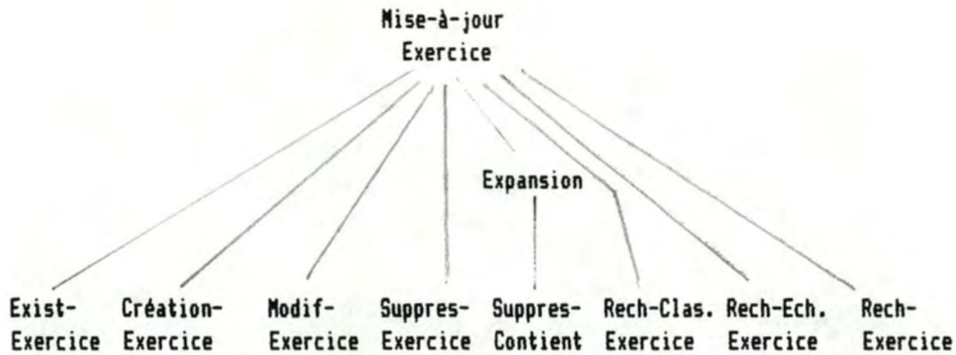
A.4. MISE-A-JOUR-CONCEPT.



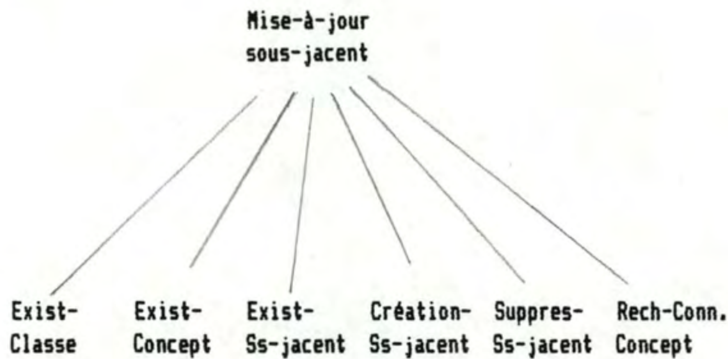
A.5. MISE-A-JOUR-CLASSE.



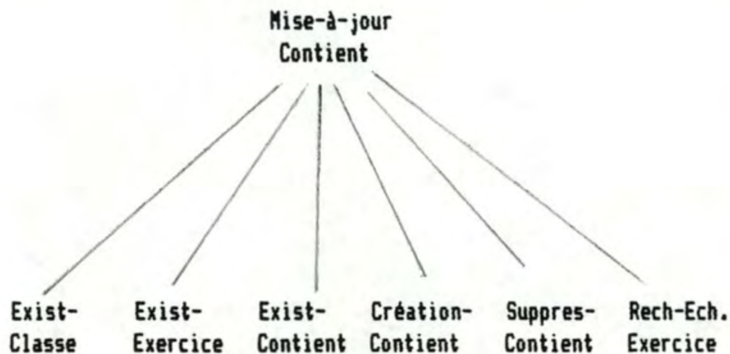
A.6. MISE-A-JOUR-EXERCICE.



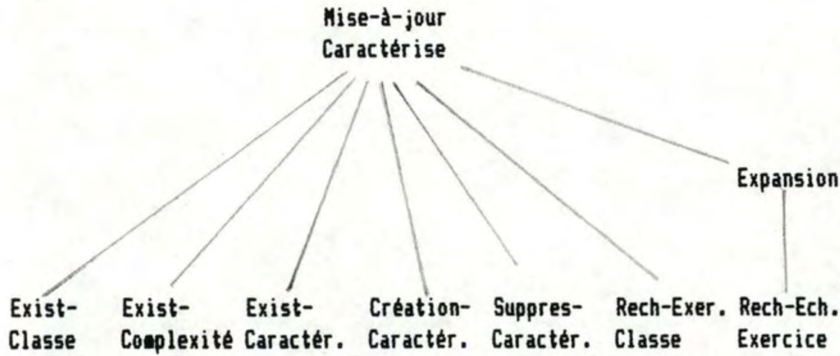
A.7. MISE-A-JOUR-SOUS-JACENT.



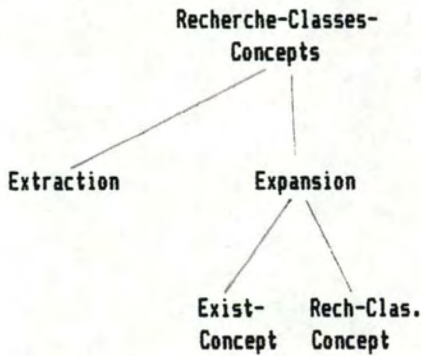
A.8. MISE-A-JOUR-CONTIENT.



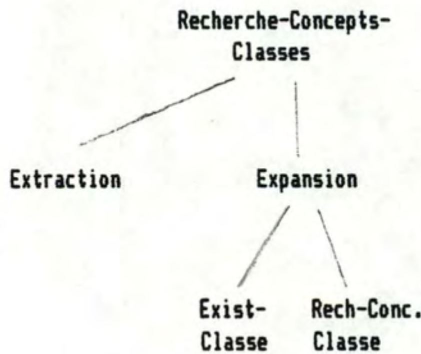
A.9. MISE-A-JOUR-CARACTERISE.



A.10. RECHERCHE-CLASSES-CONCEPTS.



A.11. RECHERCHE-CONCEPTS-CLASSES.



Annexe B : descriptions des conditions d'erreurs dans les procédures de la conception physique.

Le tableau ci-dessous décrit le contenu possible des variables ETAT et MES_ERR ainsi que les conditions d'erreurs associées survenant lors de l'exécution des procédures de la gestion de l'historique. XXXX désigne un nom variable en fonction de la procédure exécutée; il peut correspondre à une entité (ex : ETUDIANT), une relation (ex : CARACTERISE), un fichier (ex : COMPLEXI.IDX) ou une liste d'éléments (ex : CONCEPT).

Etat	Mes_err	Description
0	Résultat correct	- condition normale d'exécution.
1	Fin de la liste des XXXX	- lorsqu'il n'y a plus d'élément suivant pour les procédures fournissant des données successives.
2	XXXX : suppression impossible	- lorsque l'élément à supprimer est relié à un échange ou une connaissance.
3	XXXX : pas d'argument dans la liste	- lorsqu'il n'y a pas d'élément (args = 0), de type concept ou classe, dans une liste de paramètres d'une procédure.
4	XXXX : argument inexistant	- lorsqu'un des éléments, de type concept ou classe, d'une liste de paramètres n'existe pas dans l'historique.
5	XXXX inexistant	- lorsqu'un élément recherché (pour une modification, une suppression ou une consultation) n'existe pas dans l'historique.
6	XXXX déjà existant	- lorsqu'un élément que l'on veut créer existe déjà.
7	XXXX : relation déjà existante	- lorsqu'on veut créer une relation entre 2 entités alors qu'elle existe déjà.
8	XXXX : relation inexistante	- lorsqu'on souhaite supprimer une relation entre 2 entités alors qu'elle n'existe pas.
9	XXXX : nombre limite d'arguments	- lorsqu'on dépasse le nombre maximum d'éléments (Nargs) dans une liste d'arguments : ce cas se produit pour les concepts sous-jacents dans la création des échanges.
10	XXXX : création impossible	- création d'un des fichiers de l'historique impossible par manque de place. Le programme s'arrête automatiquement.
11	XXXX : ouverture impossible	- ouverture d'un des fichiers de l'historique (fichier inexistant). Le programme s'arrête automatiquement.
12	XXXX : modification impossible	- lorsqu'on veut modifier des données du domaine d'enseignement associées à un échange ou une connaissance.
13	XXXX : non sous-jacent ou inexistant	- lorsque le concept non maîtrisé n'est pas sous-jacent à l'exercice ou inexistant.

Annexe C : description d'un programme utilisant le système de gestion de l'historique.

Dans l'extrait de programme qui suit, se trouvent toutes les directives nécessaires pour utiliser correctement le système de gestion de l'historique au sein d'un programme Pascal. Les commentaires situés entre (* et *) sont facultatifs, mais expliquent les instructions correspondantes.

```
{$F40}    (* directive de compilation : nombre de fichiers
             utilisés *)

     (* Constantes utilisées par Turbo Access. Les longueurs
sont données à titre d'exemple. *)

const

     MaxDataRecSize = 450;      (* Long. max des records *)
     MaxKeyLen      = 24;       (* Long. max des clés *)
     PageSize       = 16;       (* Taille max des pages *)
     Order          =  8;       (* Ordre Arbre-B *)
     PageStackSize  =  5;       (* Taille du buffer de page *)
     MaxHeight      =  5;       (* Hauteur max Arbre-B *)

     (* Introduction des modules de Turbo Access
          en spécifiant éventuellement un nom de drive *)

{$I ACCESS3.BOX}
{$I ADDKEY.BOX}
{$I DELKEY.BOX}
{$I GETKEY.BOX}

     (* Descriptions des constantes de l'historique. Les
longueurs sont données à titre d'exemple. *)

const

     LenEt_id    =     20;      (* long. identifiant étudiant *)
     LenEt_data  =     50;      (* long. données étudiant *)
     LenEc_id    =     22;      (* long. identifiant échange *)
     LenEc_sol   =     20;      (* long. sollicit. réponse *)
     LenEc_rep   =     20;      (* long. réponse étudiant *)
     LenEc_feed  =     20;      (* long. feed-back du système *)
     LenEc_inf   =     20;      (* long. informations échange *)
     LenEx_id    =      4;      (* long. identifiant exercice *)
     LenEx_data  =     50;      (* long. données exercice *)
     LenCl_id    =      4;      (* long. identifiant classe *)
     LenCl_data  =     20;      (* long. données classe *)
     LenCo_id    =      4;      (* long. identifiant concept *)
```

```
LenCo_data =    2Ø;     (* long. données concept *)
LenCm_id   =     4;     (* long. identifiant complexi.*)
LenCm_data =    2Ø;     (* long. données complexité *)
LenCn_id   =    24;     (* long. identifiant connais. *)
LenCn_data =    2Ø;     (* long. données connaissance *)

Essai      =     5;     (* nombre max. de décomposition
                           des échanges *)
Nargs      =    1Ø;     (* nombre max. d'arguments par
                           liste *)


     (* Introduction des modules de l'historique en spécifiant
éventuellement un nom de drive. Seul le module init.hst est
obligatoire. Pour les autres, l'utilisateur est libre de les
inclure ou non dans le programme, ou bien de spécifier lui-
même les procédures qui l'intéressent. *)

{$I init.hst}
{$I etudiant.hst}
{$I echange.hst}
{$I exercice.hst}
{$I classe.hst}
{$I concept.hst}
{$I complexi.hst}
{$I connaiss.hst}

(* ------------------------------------------------- *)

(* Spécification du programme de l'utilisateur *)

(* ------------------------------------------------- *)
```

Annexe D : Programme du système de gestion de l'historique.

```
(*******************************************************)
(*                                                     *)
(*            GESTION DE L'HISTORIQUE                  *)
(*                                                     *)
(*                Module INIT.HST                      *)
(*                                                     *)
(*           Auteur : Merckx O. (1987)                 *)
(*                                                     *)
(*******************************************************)


type

     (* Description des donnees *)

     et_id    =    string[LenEt_id];
     et_data  =    string[LenEt_data];

     ec_id    =    string[LenEc_id];
     ec_data  =    record
                         ec_sol   : array[1..Essai] of string[LenEc_sol];
                         ec_rep   : array[1..Essai] of string[LenEc_rep];
                         ec_feed  : array[1..Essai] of string[LenEc_Feed];
                         ec_inf   : array[1..Essai] of string[LenEc_inf];
                   end;

     ex_id    =    string[LenEx_id];
     ex_data  =    string[LenEx_data];

     cl_id    =    string[LenCl_id];
     cl_data  =    string[LenCl_data];

     co_id    =    string[LenCo_id];
     co_data  =    string[LenCo_data];

     cm_id    =    string[LenCm_id];
     cm_data  =    string[LenCm_data];

     cn_id    =    string[LenCn_id];
     cn_data  =    string[LenCn_data];
     cn_ner   =    integer;
     cn_nex   =    integer;


     (* Description des records *)

     EtdRec = record     (* Record fichier des étudiants *)
```

```
            Status      :    integer;
            Etudiant    :    et_id;
            EtNum       :    integer;
            EtData      :    et_data
        end;

EchRec = record     (* Record fichier des échanges *)

            Status      :    integer;
            EcEtudiant  :    et_id;
            EcNum       :    integer;
            EcExercice  :    ex_id;
            EcData      :    ec_data
        end;

ExrRec = record     (* Record fichier des exercices *)

            Status      :    integer;
            M           :    boolean;
            Exercice    :    ex_id;
            ExData      :    ex_data
        end;

ClsRec = record     (* Record fichier des classes *)

            Status      :    integer;
            M           :    boolean;
            Classe      :    cl_id;
            ClData      :    cl_data;
            ClComplexite :   cm_id
        end;

CnpRec = record     (* Record fichier des concepts *)

            Status      :    integer;
            Concept     :    co_id;
            M           :    boolean;
            CoData      :    co_data
        end;

CplRec = record     (* Record fichier des niveaux de complexité *)

            Status      :    integer;
            M           :    boolean;
            Complexite  :    cm_id;
            CmData      :    cm_data
        end;

CnnRec = record     (* Record fichier des connaissances *)

            Status      :    integer;
            CnEtudiant  :    et_id;
            CnConcept   :    co_id;
            CnNer       :    cn_ner;
            CnNex       :    cn_nex;
            CnData      :    cn_data
        end;

inter  = 0 .. maxint;             (* n° de reference dans une liste *)
```

```
    targs  = 0 .. Nargs;              (* nbre d'arguments dans une liste *)

    tconcepts = array [1 .. Nargs] of co_id; (* liste de concepts *)
    tclasses  = array [1 .. Nargs] of cl_id; (* liste de classes *)

    string2   = string[2];

var

    (* Description des fichiers *)

    EtdDat, EchDat, ExrDat, ClsDat : DataFile;
    CnpDat, CplDat, CnnDat : DataFile;

    EtdIdx, ExrIdx, ClsIdx, CnpIdx : IndexFile;
    CplIdx, EchIdx, CnnIdx : IndexFile;

    ExrClsIdx, ClsCnpIdx, ClsExrIdx : IndexFile;
    CnpClsIdx, CnpEchIdx, EchEtdIdx, EchExrIdx : IndexFile;
    EchCnpIdx, CnnEtdIdx, CnnCnpIdx, ClsCplIdx : IndexFile;

    etat : integer;         (* Variable globale contenant le resultat
                             des traitements sur l'historique *)

    mes_err : string[50];  (* Variable contenant les messages résultat
                             de l'exécution des procédures *)


{ DESCRIPTION DES PROCEDURES }

(* Initialisation Historique *)

procedure erreur;
begin
    case etat of 0  : mes_err := 'Résultat correct';
                 1  : mes_err := 'Fin de liste des ' + mes_err;
                 2  : mes_err := mes_err + ' : suppression impossible';
                 3  : mes_err := ' : pas d''argument dans la liste';
                 4  : mes_err := ' : argument inexistant';
                 5  : mes_err := mes_err + ' inexistant';
                 6  : mes_err := mes_err + ' déjà existant';
                 7  : mes_err := mes_err + ' : relation déjà existante';
                 8  : mes_err := mes_err + ' : relation inexistante';
                 9  : mes_err := mes_err + ' : nombre limite d''arguments';
                 10 : mes_err := mes_err + ' : création impossible';
                 11 : mes_err := mes_err + ' : ouverture impossible';
                 12 : mes_err := mes_err + ' : modification impossible';
                 13 : mes_err := mes_err + ' non sous-jacent'
    end
end;

procedure CloseHist;
begin
    (* Fermeture des fichiers de donnees *)

    CloseFile(EtdDat);
    CloseFile(EchDat);
    CloseFile(ExrDat);
    CloseFile(ClsDat);
```

```
     CloseFile(CnpDat);
     CloseFile(CplDat);
     CloseFile(CnnDat);

     (* Fermeture des fichiers d'index *)

     CloseIndex(EtdIdx);
     CloseIndex(ExrIdx);
     CloseIndex(ClsIdx);
     CloseIndex(CnpIdx);
     CloseIndex(CplIdx);
     CloseIndex(EchIdx);
     CloseIndex(CnnIdx);

     (* Fermeture des fichiers chemin d'acces *)

     CloseIndex(ExrClsIdx);
     CloseIndex(ClsExrIdx);
     CloseIndex(ClsCnpIdx);
     CloseIndex(CnpClsIdx);
     CloseIndex(CnpEchIdx);
     CloseIndex(EchCnpIdx);
     CloseIndex(EchEtdIdx);
     CloseIndex(EchExrIdx);
     CloseIndex(CnnEtdIdx);
     CloseIndex(CnnCnpIdx);
     CloseIndex(ClsCplIdx);
end;

procedure CreatHist;
label err;
begin
     (* Création des fichiers de données *)

     MakeFile(EtdDat,'ETUDIANT.DAT',SizeOf(EtdRec));
     if not Ok then begin mes_err := 'ETUDIANT.DAT'; goto err end;
     MakeFile(EchDat,'ECHANGE.DAT',SizeOf(EchRec));
     if not Ok then begin mes_err := 'ECHANGE.DAT'; goto err end;
     MakeFile(ExrDat,'EXERCICE.DAT',SizeOf(ExrRec));
     if not Ok then begin mes_err := 'EXERCICE.DAT'; goto err end;
     MakeFile(ClsDat,'CLASSE.DAT',SizeOf(ClsRec));
     if not Ok then begin mes_err := 'CLASSE.DAT'; goto err end;
     MakeFile(CnpDat,'CONCEPT.DAT',SizeOf(CnpRec));
     if not Ok then begin mes_err := 'CONCEPT.DAT'; goto err end;
     MakeFile(CplDat,'COMPLEXI.DAT',SizeOf(CplRec));
     if not Ok then begin mes_err := 'COMPLEXI.DAT'; goto err end;
     MakeFile(CnnDat,'CONNAISS.DAT',SizeOf(CnnRec));
     if not Ok then begin mes_err := 'CONNAISS.DAT'; goto err end;

     (* Création des fichiers d'index *)

     MakeIndex(EtdIdx,'ETUDIANT.IDX',LenEt_id,0);
     if not Ok then begin mes_err := 'ETUDIANT.IDX'; goto err end;
     MakeIndex(ExrIdx,'EXERCICE.IDX',LenEx_id,0);
     if not Ok then begin mes_err := 'EXERCICE.IDX'; goto err end;
     MakeIndex(ClsIdx,'CLASSE.IDX',LenCl_id,0);
     if not Ok then begin mes_err := 'CLASSE.IDX'; goto err end;
     MakeIndex(CnpIdx,'CONCEPT.IDX',LenCo_id,0);
     if not Ok then begin mes_err := 'CONCEPT.IDX'; goto err end;
```

```
     MakeIndex(CplIdx,'COMPLEXI.IDX',LenCm_id,0);
     if not Ok then begin mes_err := 'COMPLEXI.IDX'; goto err end;
     MakeIndex(EchIdx,'ECHANGE.IDX',LenEt_id + LenEc_id,0);
     if not Ok then begin mes_err := 'ECHANGE.IDX'; goto err end;
     MakeIndex(CnnIdx,'CONNAISS.IDX',LenEt_id + LenCo_id,0);
     if not Ok then begin mes_err := 'CONNAISS.IDX'; goto err end;

     (* Création des fichiers chemin d'accès *)

     MakeIndex(ExrClsIdx,'EXRCLS.IDX',LenCl_id,1);
     if not Ok then begin mes_err := 'EXRCLS.IDX'; goto err end;
     MakeIndex(ClsExrIdx,'CLSEXR.IDX',LenEx_id,1);
     if not Ok then begin mes_err := 'CLSEXR.IDX'; goto err end;
     MakeIndex(ClsCnpIdx,'CLSCNP.IDX',LenCo_id,1);
     if not Ok then begin mes_err := 'CLSCNP.IDX'; goto err end;
     MakeIndex(CnpClsIdx,'CNPCLS.IDX',LenCl_id,1);
     if not Ok then begin mes_err := 'CNPCLS.IDX'; goto err end;
     MakeIndex(CnpEchIdx,'CNPECH.IDX',LenEt_id + LenEc_id,1);
     if not Ok then begin mes_err := 'CNPECH.IDX'; goto err end;
     MakeIndex(EchCnpIdx,'ECHCNP.IDX',LenCo_id,1);
     if not Ok then begin mes_err := 'ECHCNP.IDX'; goto err end;
     MakeIndex(EchEtdIdx,'ECHETD.IDX',LenEt_id,1);
     if not Ok then begin mes_err := 'ECHETD.IDX'; goto err end;
     MakeIndex(EchExrIdx,'ECHEXR.IDX',LenEx_id,1);
     if not Ok then begin mes_err := 'ECHEXR.IDX'; goto err end;
     MakeIndex(CnnEtdIdx,'CNNETD.IDX',LenEt_id,1);
     if not Ok then begin mes_err := 'CNNETD.IDX'; goto err end;
     MakeIndex(CnnCnpIdx,'CNNCNP.IDX',LenCo_id,1);
     if not Ok then begin mes_err := 'CNNCNP.IDX'; goto err end;
     MakeIndex(ClsCplIdx,'CLSCPL.IDX',LenCm_id,1);
     if not Ok then begin mes_err := 'CLSCPL.IDX'; goto err end;

err: if Ok then begin CloseHist; etat := 0 end
          else begin
                 etat := 10; erreur;
                 writeln(mes_err); writeln('Programme interrompu !');
                 halt
               end
end;

procedure OpenHist;
label err;
begin
     (* Ouverture des fichiers de données *)

     OpenFile(EtdDat,'ETUDIANT.DAT',SizeOf(EtdRec));
     if not Ok then begin mes_err := 'ETUDIANT.DAT'; goto err end;
     OpenFile(EchDat,'ECHANGE.DAT',SizeOf(EchRec));
     if not Ok then begin mes_err := 'ECHANGE.DAT'; goto err end;
     OpenFile(ExrDat,'EXERCICE.DAT',SizeOf(ExrRec));
     if not Ok then begin mes_err := 'EXERCICE.DAT'; goto err end;
     OpenFile(ClsDat,'CLASSE.DAT',SizeOf(ClsRec));
     if not Ok then begin mes_err := 'CLASSE.DAT';; goto err end;
     OpenFile(CnpDat,'CONCEPT.DAT',SizeOf(CnpRec));
     if not Ok then begin mes_err := 'CONCEPT.DAT'; goto err end;
     OpenFile(CplDat,'COMPLEXI.DAT',SizeOf(CplRec));
     if not Ok then begin mes_err := 'COMPLEXI.DAT'; goto err end;
     OpenFile(CnnDat,'CONNAISS.DAT',SizeOf(CnnRec));
     if not Ok then begin mes_err := 'CONNAISS.DAT'; goto err end;
```

```
    (* Ouverture des fichiers d'index *)

    InitIndex;

    OpenIndex(EtdIdx,'ETUDIANT.IDX',LenEt_id,0);
    if not Ok then begin mes_err := 'ETUDIANT.IDX'; goto err end;
    OpenIndex(ExrIdx,'EXERCICE.IDX',LenEx_id,0);
    if not Ok then begin mes_err := 'EXERCICE.IDX'; goto err end;
    OpenIndex(ClsIdx,'CLASSE.IDX',LenCl_id,0);
    if not Ok then begin mes_err := 'CLASSE.IDX'; goto err end;
    OpenIndex(CnpIdx,'CONCEPT.IDX',LenCo_id,0);
    if not Ok then begin mes_err := 'CONCEPT.IDX'; goto err end;
    OpenIndex(CplIdx,'COMPLEXI.IDX',LenCm_id,0);
    if not Ok then begin mes_err := 'COMPLEXI.IDX'; goto err end;
    OpenIndex(EchIdx,'ECHANGE.IDX',LenEt_id + LenEc_id,0);
    if not Ok then begin mes_err := 'ECHANGE.IDX'; goto err end;
    OpenIndex(CnnIdx,'CONNAISS.IDX',LenEt_id + LenCo_id,0);
    if not Ok then begin mes_err := 'CONNAISS.IDX'; goto err end;

    (* Ouverture des fichiers chemin d'accès *)

    OpenIndex(ExrClsIdx,'EXRCLS.IDX',LenCl_id,1);
    if not Ok then begin mes_err := 'EXRCLS.IDX'; goto err end;
    OpenIndex(ClsExrIdx,'CLSEXR.IDX',LenEx_id,1);
    if not Ok then begin mes_err := 'CLSEXR.IDX'; goto err end;
    OpenIndex(ClsCnpIdx,'CLSCNP.IDX',LenCo_id,1);
    if not Ok then begin mes_err := 'CLSCNP.IDX'; goto err end;
    OpenIndex(CnpClsIdx,'CNPCLS.IDX',LenCl_id,1);
    if not Ok then begin mes_err := 'CNPCLS.IDX'; goto err end;
    OpenIndex(CnpEchIdx,'CNPECH.IDX',LenEt_id + LenEc_id,1);
    if not Ok then begin mes_err := 'CNPECH.IDX'; goto err end;
    OpenIndex(EchCnpIdx,'ECHCNP.IDX',LenCo_id,1);
    if not Ok then begin mes_err := 'ECHCNP.IDX'; goto err end;
    OpenIndex(EchEtdIdx,'ECHETD.IDX',LenEt_id,1);
    if not Ok then begin mes_err := 'ECHETD.IDX'; goto err end;
    OpenIndex(EchExrIdx,'ECHEXR.IDX',LenEx_id,1);
    if not Ok then begin mes_err := 'ECHEXR.IDX'; goto err end;
    OpenIndex(CnnEtdIdx,'CNNETD.IDX',LenEt_id,1);
    if not Ok then begin mes_err := 'CNNETD.IDX'; goto err end;
    OpenIndex(CnnCnpIdx,'CNNCNP.IDX',LenCo_id,1);
    if not Ok then begin mes_err := 'CNNCNP.IDX'; goto err end;
    OpenIndex(ClsCplIdx,'CLSCPL.IDX',LenCm_id,1);
    if not Ok then begin mes_err := 'CLSCPL.IDX'; goto err end;

err: if Ok then etat := 0
          else begin
                  etat := 11; erreur;
                  writeln(mes_err); writeln('Programme interrompu');
                  halt
                end
end;

function IntToStr(N : Integer):string2;
begin
    IntToStr:=Chr(Hi(N))+Chr(Lo(N))
end;

function StrToInt(S : string2):Integer;
```

```
begin
     StrToInt:=Swap(Ord(S[1])+Ord(S[2]))
end;


(* Procédure de positionnement dans un fichier chemin d'accès *)

procedure HstPosKey(var F   : IndexFile;
                    ref     : integer;
                    var Key          );
var
     ref1 : integer;
     Key1 : TaKeyStr absolute Key;
     Key2 : TaKeyStr;
begin
     Key2 := Key1;
     FindKey(F,ref1,Key1);
     while Ok and (ref1 <> ref) and (Key2 = Key1) do
       NextKey(F,ref1,Key2);
     if Key2 <> Key1 then Ok := false
end;


(* Fonction vérifiant si une classe est reliée à un échange *)

function HstAutori(ref : integer) : boolean;
var
   buf  : ClsRec;
   buf1 : ExrRec;
begin
     GetRec(ClsDat,ref,buf);
     FindKey(ExrClsIdx,ref,buf.Classe);
     while Ok do
       begin
         GetRec(ExrDat,ref,buf1);
         FindKey(EchExrIdx,ref,buf1.Exercice);
         if Ok then begin HstAutori := false; exit end;
         FindKey(ExrClsIdx,ref,buf.Classe)
       end;
     HstAutori := true
end;
```

```
(*****************************************************)
(*                                                   *)
(*            GESTION DE L'HISTORIQUE                *)
(*                                                   *)
(*              Module ETUDIANT.HST                  *)
(*                                                   *)
(*           Auteur : Merckx O. (1987)               *)
(*                                                   *)
(*****************************************************)


procedure Rech_Etudiant(etudiant    : et_id;
                        var num     : integer;
                        var donnees : et_data);
var
     ref : integer;
     buf : EtdRec;
begin
     FindKey(EtdIdx,ref,etudiant);
     if not Ok then begin etat := 5; mes_err := 'Etudiant' end
               else begin
                      GetRec(EtdDat,ref,buf);
                      num := buf.EtNum; donnees := buf.EtData;
                      etat := 0
                    end
end;

procedure Creation_Etudiant(etudiant : et_id;
                            donnees  : et_data);
var
     ref : integer;
     buf : EtdRec;
begin
     FindKey(EtdIdx,ref,etudiant);
     if Ok then begin etat := 6; mes_err := 'Etudiant' end
          else begin
                  buf.Status := 0; buf.Etudiant := etudiant;
                  buf.EtData := donnees; buf.EtNum := 0;
                  AddRec(EtdDat,ref,buf);
                  AddKey(EtdIdx,ref,etudiant);
                  etat := 0
               end
end;

procedure Modif_Etudiant(etudiant : et_id;
                         donnees  : et_data);
var
     ref : integer;
     buf : EtdRec;
begin
     FindKey(EtdIdx,ref,etudiant);
     if not Ok then begin etat := 5; mes_err := 'Etudiant' end
               else begin
                      GetRec(EtdDat,ref,buf);
                      buf.EtData := donnees;
                      PutRec(EtdDat,ref,buf);
                      etat := 0
                    end
end;
```

```
procedure Suppres_Etudiant(etudiant : et_id);
var
    ref : integer;
begin
    FindKey(EtdIdx,ref,etudiant);
    if not Ok then begin etat := 5; mes_err := 'Etudiant'; exit end;
    HstSuppres_Echanges_Etudiant(etudiant);
    HstSuppres_Connaissances_Etudiant(etudiant);
    DeleteKey(EtdIdx,ref,etudiant);
    DeleteRec(EtdDat,ref);
    etat := 0
end;

procedure Liste_Etudiants(var etudiant : et_id;
                          var donnees  : et_data;
                          var nref     : inter);
var
    ref  : integer;
    buf  : EtdRec;
begin
    if nref = 0 then ClearKey(EtdIdx);
    NextKey(EtdIdx,ref,etudiant);
    if not Ok then begin etat := 1; mes_err := 'étudiants' end
              else begin
                     GetRec(EtdDat,ref,buf);
                     donnees := buf.EtData;
                     nref := nref + 1; etat := 0;
                   end
end;
```

```
(*******************************************************)
(*                                                     *)
(*              GESTION DE L'HISTORIQUE                *)
(*                                                     *)
(*                Module ECHANGE.HST                   *)
(*                                                     *)
(*             Auteur : Merckx O. (1987)               *)
(*                                                     *)
(*******************************************************)


procedure HstCreation_Maitrise_Pas(echange  : ec_id;
                                   ref      : integer;
                                   concept  : co_id;
                                   ref1     : integer);
begin
     AddKey(EchCnpIdx,ref,concept);
     AddKey(CnpEchIdx,ref1,echange);
end;

procedure Creation_Echange(etudiant     : et_id;
                           exercice     : ex_id;
                           donnees      : ec_data;
                           args         : targs;
                           var concepts : tconcepts);
var
     num,
     i,
     j,
     ref,
     ref1,
     ref2      : integer;
     echange   : ec_id;
     exer      : ex_id;
     clas      : cl_id;
     buf       : EtdRec;
     buf1      : EchRec;
     buf2      : ClsRec;
     buf3      : CnpRec;
     args2     : targs;
     concepts2 : tconcepts;

function HstRech_Table(args         : integer;
                       concept      : co_id;
                       var concepts : tconcepts) : boolean;
var
     i : integer;
begin
     for i := 1 to args do if concept = concepts[i]
                              then begin HstRech_Table := true; exit end;
     HstRech_Table := false
end;

begin

     (* vérification *)

     FindKey(EtdIdx,ref,etudiant);
     if not Ok then begin etat := 5; mes_err := 'Etudiant'; exit end;
```

```
GetRec(EtdDat,ref,buf);
num := buf.EtNum+1;
FindKey(ExrIdx,ref1,exercice);
if not Ok then begin etat := 5; mes_err := 'Exercice'; exit end;
echange := etudiant + IntToStr(num);

(* création échange *)

buf1.Status := 0; buf1.EcEtudiant := etudiant;
buf1.EcNum := num; buf1.EcExercice := exercice;
buf1.EcData := donnees;
AddRec(EchDat,ref2,buf1);
AddKey(EchIdx,ref2,echange);
AddKey(EchEtdIdx,ref2,etudiant);
AddKey(EchExrIdx,ref2,exercice);
buf.EtNum := num;
PutRec(EtdDat,ref,buf);
etat := 0;

(* Recherche des concepts sous-jacents &
   m-a-j des connaissances correspondantes *)

i := 0;
exer := exercice;
FindKey(ClsExrIdx,ref,exercice);
while OK and (exer = exercice) do
  begin
    GetRec(ClsDat,ref,buf2);
    clas := buf2.Classe;
    FindKey(CnpClsIdx,ref1,clas);
    while Ok and (clas = buf2.Classe) do
      begin
        GetRec(CnpDat,ref1,buf3);
        if not HstRech_Table(i,buf3.Concept,concepts2)
           then begin
                   if not HstRech_Table(args,buf3.Concept,concepts)
                     then HstMaj_Connaissance(etudiant,buf3.Concept,true)
                     else begin
                          HstCreation_Maitrise_Pas(echange,ref2,buf3.Concept,ref1);
                          HstMaj_Connaissance(etudiant,buf3.Concept,false);
                          for j := 1 to args do
                            if buf3.Concept = concepts[j] then concepts[j] := ''
                          end;
                   i := i + 1;
                   if i > Nargs then
                     begin etat := 9; mes_err := 'Concepts'; exit end;
                   concepts2[i] := buf3.Concept;
                 end;
        NextKey(CnpClsIdx,ref1,clas)
      end;
    NextKey(ClsExrIdx,ref,exer)
  end;

(* Création des concepts non maîtrisés &
   m-a-j des connaissances correspondantes *)

if args > 0 then
   for i := 1 to args do
    if concepts[i] <> '' then begin etat := 13; mes_err := 'Concept'; exit end
```

```
end;

procedure Rech_Echange(etudiant     : et_id;
                       num          : integer;
                       var exercice : ex_id;
                       var donnees  : ec_data);
var
     ref     : integer;
     buf     : EchRec;
     echange : ec_id;
begin
     FindKey(EtdIdx,ref,etudiant);
     if not Ok then begin etat := 5; mes_err := 'Etudiant'; exit end;
     echange := etudiant + IntToStr(num);
     FindKey(EchIdx,ref,echange);
     if not Ok then begin etat := 5; mes_err := 'Echange' end
            else begin
                   GetRec(EchDat,ref,buf);
                   donnees := buf.EcData; exercice := buf.EcExercice;
                   etat := 0
                 end
end;

procedure Rech_Echanges_Etudiant(etudiant     : et_id;
                                 var num      : integer;
                                 var exercice : ex_id;
                                 var donnees  : ec_data;
                                 var nref     : inter);
var
     ref  : integer;
     buf  : EchRec;
     etud : et_id;
begin
     if nref = 0 then begin
                        FindKey(EtdIdx,ref,etudiant);
                        if not Ok then
                             begin etat := 5; mes_err := 'Etudiant'; exit end;
                        etud := etudiant;
                        FindKey(EchEtdIdx,ref,etudiant)
                      end
                 else NextKey(EchEtdIdx,ref,etud);
     if not Ok or (etud <> etudiant)
         then begin etat := 1; mes_err := 'échanges d''un étudiant'; exit end;
     GetRec(EchDat,ref,buf);
     num := buf.EcNum; exercice := buf.EcExercice;
     donnees := buf.EcData; nref := nref + 1;
     etat := 0
end;

procedure Rech_Echanges_Exercice(exercice     : ex_id;
                                 var etudiant : et_id;
                                 var num      : integer;
                                 var donnees  : ec_data;
                                 var nref     : inter);
var
     ref  : integer;
     buf  : EchRec;
     exer : ex_id;
```

```
begin
    if nref = 0 then begin
                        FindKey(ExrIdx,ref,exercice);
                        if not Ok then
                              begin etat := 5; mes_err := 'Exercice'; exit end;
                        exer := exercice;
                        FindKey(EchExrIdx,ref,exercice)
                      end
                 else NextKey(EchExrIdx,ref,exer);
    if not Ok or (exer <> exercice)
       then begin etat := 1; mes_err := 'échanges d''un exercice'; exit end;
    GetRec(EchDat,ref,buf);
    num := buf.EcNum; etudiant := buf.EcEtudiant;
    donnees := buf.EcData; nref := nref + 1;
    etat := 0
end;

procedure Rech_Concepts_Echange(etudiant    : et_id;
                                num         : integer;
                                var concept : co_id;
                                var m       : boolean;
                                var donnees : co_data;
                                var nref    : inter);
var
    ref     : integer;
    buf     : CnpRec;
    echange,
    echa    : ec_id;
begin
    echange := etudiant + IntToStr(num);
    if nref = 0 then begin
                        FindKey(EtdIdx,ref,etudiant);
                        if not Ok then begin etat := 5; mes_err := 'Etudiant'; exit end;
                        FindKey(EchIdx,ref,echange);
                        if not Ok then begin etat := 5; mes_err := 'Echange'; exit end;
                        echa := echange;
                        FindKey(CnpEchIdx,ref,echange)
                      end
                 else NextKey(CnpEchIdx,ref,echa);
    if not Ok or (echa <> echange)
        then begin etat := 1; mes_err := 'concepts d''un échange' end
        else begin
                GetRec(CnpDat,ref,buf);
                concept := buf.Concept;
                donnees := buf.CoData; m := buf.M;
                nref := nref + 1; etat := 0
             end
end;

procedure Rech_Echanges_Concept(concept      : co_id;
                                var etudiant : et_id;
                                var num      : integer;
                                var exercice : ex_id;
                                var donnees  : ec_data;
                                var nref     : inter);
var
    ref     : integer;
    buf     : EchRec;
    conc    : co_id;
```

```
begin
    if nref = 0 then begin
                    FindKey(CnpIdx,ref,concept);
                    if not Ok then begin etat := 5; mes_err := 'Concept'; exit end;
                    conc := concept;
                    FindKey(EchCnpIdx,ref,concept)
                  end
              else NextKey(EchCnpIdx,ref,conc);
    if not Ok or (conc <> concept)
        then begin etat := 1; mes_err := 'échanges d''un concept' end
        else begin
              GetRec(EchDat,ref,buf);
              etudiant := buf.EcEtudiant; exercice := buf.EcExercice;
              num := buf.EcNum; donnees := buf.EcData;
              nref := nref + 1; etat := 0
            end
end;


procedure HstSuppres_Maitrise(etudiant : et_id;
                             echange  : ec_id;
                             ref      : integer);
var
    ref1 : integer;
    buf  : CnpRec;
begin
    FindKey(CnpEchIdx,ref1,echange);
    while Ok do
      begin
        GetRec(CnpDat,ref1,buf);
        DeleteKey(EchCnpIdx,ref,buf.Concept);
        DeleteKey(CnpEchIdx,ref1,echange);
        FindKey(CnpEchIdx,ref1,echange)
      end
end;

procedure HstSuppres_Echanges_Etudiant(etudiant : et_id);
var
    ref     : integer;
    buf     : EchRec;
    echange : ec_id;
    echa    : ec_id;
begin
    FindKey(EchEtdIdx,ref,etudiant);
    while Ok do
      begin
        GetRec(EchDat,ref,buf);
        echange := etudiant + IntToStr(buf.EcNum);
        HstSuppres_Maitrise(etudiant,echange,ref);
        DeleteKey(EchExrIdx,ref,buf.EcExercice);
        DeleteKey(EchEtdIdx,ref,etudiant);
        DeleteKey(EchIdx,ref,echange);
        DeleteRec(EchDat,ref);
        FindKey(EchEtdIdx,ref,etudiant)
      end
end;
```

```
(*******************************************************)
(*                                                     *)
(*              GESTION DE L'HISTORIQUE                *)
(*                                                     *)
(*                Module EXERCICE.HST                  *)
(*                                                     *)
(*             Auteur : Merckx O. (1987)               *)
(*                                                     *)
(*******************************************************)


procedure Rech_Exercice(exercice    : ex_id;
                        var m       : boolean;
                        var donnees : ex_data);
var
     ref : integer;
     buf : ExrRec;
begin
     FindKey(ExrIdx,ref,exercice);
     if not Ok then begin etat := 5; mes_err := 'Exercice' end
             else begin
                     GetRec(ExrDat,ref,buf);
                     donnees := buf.ExData; m := buf.M;
                     etat := 0
                   end
end;

procedure Creation_Exercice(exercice : ex_id;
                            donnees  : ex_data);
var
     ref : integer;
     buf : ExrRec;
begin
     FindKey(ExrIdx,ref,exercice);
     if Ok then begin etat := 6; mes_err := 'Exercice' end
          else begin
                  buf.Status := 0; buf.Exercice := exercice;
                  buf.ExData := donnees; buf.M := false;
                  AddRec(ExrDat,ref,buf);
                  AddKey(ExrIdx,ref,exercice);
                  etat := 0
                end
end;

procedure Modif_Exercice(exercice : ex_id;
                         donnees  : ex_data);
var
     ref,
     ref1 : integer;
     buf  : ExrRec;
begin
     FindKey(ExrIdx,ref,exercice);
     if not Ok then begin etat := 5; mes_err := 'Exercice'; exit end;
     FindKey(EchExrIdx,ref1,exercice);
     if Ok then begin etat := 12; mes_err := 'Exercice' end
             else begin
                     GetRec(ExrDat,ref,buf);
                     buf.ExData := donnees;
                     PutRec(ExrDat,ref,buf);
```

```
                    etat := 0
                  end
end;

procedure Liste_Exercices(var exercice : ex_id;
                          var m        : boolean;
                          var donnees  : ex_data;
                          var nref     : inter);
var
     ref  : integer;
     buf  : ExrRec;
begin
     if nref = 0 then ClearKey(ExrIdx);
     NextKey(ExrIdx,ref,exercice);
     if not Ok then begin etat := 1; mes_err := 'exercices' end
               else begin
                      GetRec(ExrDat,ref,buf);
                      donnees := buf.ExData; m := buf.M;
                      nref := nref + 1; etat := 0
                    end
end;

procedure Suppres_Exercice(exercice : ex_id);
var
     ref,
     ref1 : integer;

procedure HstMrkExr(ref : integer);
var
     buf : ExrRec;
begin
     GetRec(ExrDat,ref,buf);
     buf.M := true;
     PutRec(ExrDat,ref,buf)
end;

begin
     FindKey(ExrIdx,ref,exercice);
     if not Ok then begin etat := 5; mes_err := 'Exercice'; exit end;
     FindKey(EchExrIdx,ref1,exercice);
     if Ok then begin HstMrkExr(ref); etat := 2; mes_err := 'Exercice' end
          else begin
                 HstSuppres_Contient_Exercice(exercice,ref);
                 DeleteKey(ExrIdx,ref,exercice);
                 DeleteRec(ExrDat,ref);
                 etat := 0
               end
end;
```

```
(*******************************************************)
(*                                                     *)
(*             GESTION DE L'HISTORIQUE                 *)
(*                                                     *)
(*                Module CLASSE.HST                    *)
(*                                                     *)
(*             Auteur : Merckx O. (1987)               *)
(*                                                     *)
(*******************************************************)


procedure Rech_Classe(classe      : cl_id;
                      var m       : boolean;
                      var donnees : cl_data);
var
     ref : integer;
     buf : ClsRec;
begin
     FindKey(ClsIdx,ref,classe);
     if not Ok then begin etat := 5; mes_err := 'Classe' end
              else begin
                      GetRec(ClsDat,ref,buf);
                      donnees := buf.ClData; m := buf.M;
                      etat := 0
                    end
end;

procedure Creation_Classe(classe   : cl_id;
                          donnees  : cl_data);
var
     ref : integer;
     buf : ClsRec;
begin
     FindKey(ClsIdx,ref,classe);
     if Ok then begin etat := 6; mes_err := 'Classe' end
          else begin
                  buf.Status := 0; buf.Classe := classe;
                  buf.ClData := donnees; buf.M := false;
                  buf.ClComplexite := '';
                  AddRec(ClsDat,ref,buf);
                  AddKey(ClsIdx,ref,classe);
                  etat := 0
                end
end;

procedure Modif_Classe(classe  : cl_id;
                       donnees : cl_data);
var
     ref : integer;
     buf : ClsRec;
begin
     FindKey(ClsIdx,ref,classe);
     if not Ok then begin etat := 5; mes_err := 'Classe'; exit end;
     if not HstAutori(ref)
              then begin etat := 12; mes_err := 'Classe' end
              else begin
                      GetRec(ClsDat,ref,buf);
                      buf.ClData := donnees;
                      PutRec(ClsDat,ref,buf);
```

```
                  etat := 0
                end
end;

procedure Liste_Classes(var classe  : cl_id;
                        var ■       : boolean;
                        var donnees : cl_data;
                        var nref    : inter);
var
     ref  : integer;
     buf  : ClsRec;
begin
     if nref = 0 then ClearKey(ClsIdx);
     NextKey(ClsIdx,ref,classe);
     if not Ok then begin etat := 1; ■es_err := 'classes' end
               else begin
                     GetRec(ClsDat,ref,buf);
                     donnees := buf.ClData; ■ := buf.M;
                     nref := nref + 1; etat := 0
                   end
end;

procedure HstSuppres_SS_Jacent_Classe(classe : cl_id;
                                      ref    : integer);
var
     ref1 : integer;
     buf  : CnpRec;
begin
     FindKey(CnpClsIdx,ref1,classe);
     while not Ok do
       begin
         GetRec(CnpDat,ref1,buf);
         DeleteKey(ClsCnpIdx,ref,buf.Concept);
         DeleteKey(CnpClsIdx,ref1,classe);
         FindKey(CnpClsIdx,ref1,classe)
       end;
end;

procedure HstSuppres_Contient_Classe(classe : cl_id;
                                     ref    : integer);
var
     ref1 : integer;
     buf  : ExrRec;
begin
     FindKey(ExrClsIdx,ref1,classe);
     while not Ok do
       begin
         GetRec(ExrDat,ref1,buf);
         DeleteKey(ClsExrIdx,ref,buf.Exercice);
         DeleteKey(ExrClsIdx,ref1,classe);
         FindKey(ExrClsIdx,ref1,classe)
       end;
end;

procedure Suppres_Classe(classe : cl_id);
var
     ref  : integer;
     buf  : ClsRec;
```

```
procedure HstMrkCls(ref : integer);
var
     buf : ClsRec;
begin
     GetRec(ClsDat,ref,buf);
     buf.M := true;
     PutRec(ClsDat,ref,buf)
end;

begin
     FindKey(ClsIdx,ref,classe);
     if not Ok then begin etat := 5; mes_err := 'Classe'; exit end;
     if not HstAutori(ref) then begin HstMrkCls(ref); etat := 2; mes_err := 'Classe'; exit end;
     GetRec(ClsDat,ref,buf);
     if buf.ClComplexite <> '' then DeleteKey(ClsCplIdx,ref,buf.ClComplexite);
     HstSuppres_SS_Jacent_Classe(classe,ref);
     HstSuppres_Contient_Classe(classe,ref);
     DeleteKey(ClsIdx,ref,classe);
     DeleteRec(ClsDat,ref);
     etat := 0
end;

procedure Creation_Contient(exercice : ex_id;
                            classe   : cl_id);
var
     ref,
     ref1 : integer;
begin
     FindKey(ExrIdx,ref,exercice);
     if not Ok then begin etat := 5; mes_err := 'Exercice'; exit end;
     FindKey(ClsIdx,ref1,classe);
     if not Ok then begin etat := 5; mes_err := 'Classe'; exit end;
     HstPosKey(ClsExrIdx,ref1,exercice);
     if Ok then begin etat := 7; mes_err := 'Contient'; exit end;
     AddKey(ClsExrIdx,ref1,exercice);
     AddKey(ExrClsIdx,ref,classe);
     etat := 0
end;

procedure Suppres_Contient(exercice : ex_id;
                           classe   : cl_id);
var
     ref,
     ref1 : integer;
begin
     FindKey(ExrIdx,ref,exercice);
     if not Ok then begin etat := 5; mes_err := 'Exercice'; exit end;
     FindKey(ClsIdx,ref1,classe);
     if not Ok then begin etat := 5; mes_err := 'Classe'; exit end;
     if not HstAutori(ref1) then begin etat := 2; mes_err := 'Contient'; exit end;
     DeleteKey(ClsExrIdx,ref1,exercice);
     if Ok then DeleteKey(ExrClsIdx,ref,classe);
     if not Ok then begin etat := 8; mes_err := 'Contient' end
             else etat := 0
end;

procedure Creation_Sous_Jacent(concept : co_id;
                               classe  : cl_id);
var
```

```
     ref,
     ref1 : integer;
begin
     FindKey(CnpIdx,ref,concept);
     if not Ok then begin etat := 5; mes_err := 'Concept'; exit end;
     FindKey(ClsIdx,ref1,classe);
     if not Ok then begin etat := 5; mes_err := 'Classe'; exit end;
     HstPosKey(ClsCnpIdx,ref1,concept);
     if Ok then begin etat := 7; mes_err := 'Sous-Jacent'; exit end;
     AddKey(ClsCnpIdx,ref1,concept);
     AddKey(CnpClsIdx,ref,classe);
     etat := 0
end;

procedure Suppres_Sous_Jacent(concept : ex_id;
                              classe  : cl_id);
var
     ref,
     ref1,
     ref2  : integer;
begin
     FindKey(CnpIdx,ref,concept);
     if not Ok then begin etat := 5; mes_err := 'Concept'; exit end;
     FindKey(ClsIdx,ref1,classe);
     if not Ok then begin etat := 5; mes_err := 'Classe'; exit end;
     FindKey(CnnCnpIdx,ref2,concept);
     if Ok then begin etat := 2; mes_err := 'Sous-Jacent'; exit end;
     DeleteKey(ClsCnpIdx,ref1,concept);
     if Ok then DeleteKey(CnpClsIdx,ref,classe);
     if not Ok then begin etat := 8; mes_err := 'Sous-Jacent' end
             else etat := 0
end;

procedure Rech_Exercices_Classe(classe       : cl_id;
                                var exercice : ex_id;
                                var m        : boolean;
                                var donnees  : ex_data;
                                var nref     : inter);
var
     ref  : integer;
     buf  : ExrRec;
     clas : cl_id;
begin
     if nref = 0 then begin
                      FindKey(ClsIdx,ref,classe);
                      if not Ok then begin etat := 5; mes_err := 'Classe'; exit end;
                      clas := classe;
                      FindKey(ExrClsIdx,ref,classe)
                    end
               else NextKey(ExrClsIdx,ref,clas);
     if not Ok or (clas <> classe)
               then begin etat := 1; mes_err := 'exercices d''une classe' end
               else begin
                      GetRec(ExrDat,ref,buf);
                      exercice := buf.Exercice;
                      donnees := buf.ExData; m := buf.M;
                      nref := nref + 1; etat := 0
                    end
end;
```

```
procedure Rech_Classes_Exercice(exercice    : ex_id;
                                var classe  : cl_id;
                                var m       : boolean;
                                var donnees : cl_data;
                                var nref    : inter);
var
    ref  : integer;
    buf  : ClsRec;
    exer : ex_id;
begin
    if nref = 0 then begin
                       FindKey(ExrIdx,ref,exercice);
                       if not Ok then begin etat := 5; mes_err := 'Exercice'; exit end;
                       exer := exercice;
                       FindKey(ClsExrIdx,ref,exercice)
                     end
                else NextKey(ClsExrIdx,ref,exer);
    if not Ok or (exer <> exercice)
                then begin etat := 1; mes_err := 'classes d''un exercice' end
                else begin
                       GetRec(ClsDat,ref,buf);
                       classe := buf.Classe;
                       donnees := buf.ClData; m := buf.M;
                       nref := nref + 1; etat := 0
                     end
end;

procedure Rech_Concepts_Classe(classe      : cl_id;
                               var concept : co_id;
                               var m       : boolean;
                               var donnees : co_data;
                               var nref    : inter);
var
    ref  : integer;
    buf  : CnpRec;
    clas : cl_id;
begin
    if nref = 0 then begin
                       FindKey(ClsIdx,ref,classe);
                       if not Ok then begin etat := 5; mes_err := 'Classe'; exit end;
                       clas := classe;
                       FindKey(CnpClsIdx,ref,classe)
                     end
                else NextKey(CnpClsIdx,ref,clas);
    if not Ok or (clas <> classe)
                then begin etat := 1; mes_err := 'concepts d''une classe' end
                else begin
                       GetRec(CnpDat,ref,buf);
                       concept := buf.Concept;
                       donnees := buf.CoData; m := buf.M;
                       nref := nref + 1; etat := 0
                     end
end;

procedure Rech_Classes_Concept(concept     : co_id;
                               var classe  : cl_id;
                               var m       : boolean;
                               var donnees : cl_data;
```

```
                                 var nref    : inter);
var
    ref  : integer;
    buf  : ClsRec;
    conc : co_id;
begin
    if nref = Ø then begin
                        FindKey(CnpIdx,ref,concept);
                        if not Ok then begin etat := 5; mes_err := 'Concept'; exit end;
                        conc := concept;
                        FindKey(ClsCnpIdx,ref,concept)
                     end
                else NextKey(ClsCnpIdx,ref,conc);
    if not Ok or (conc <> concept)
                then begin etat := 1; mes_err := 'classes d''un concept' end
                else begin
                        GetRec(ClsDat,ref,buf);
                        classe := buf.Classe;
                        donnees := buf.ClData; m := buf.M;
                        nref := nref + 1; etat := Ø
                     end
end;

procedure Rech_Classes_Concepts(args         : targs;
                                var concepts : tconcepts;
                                var classe   : cl_id;
                                var m        : boolean;
                                var donnees  : cl_data;
                                var nref     : inter);
var
    i,
    ref  : integer;
    conc : co_id;
    buf  : ClsRec;
begin
    if args = Ø then begin etat := 3; mes_err := 'Concepts'; exit end;
    if nref = Ø then
       begin
         For i := 1 to args do
           begin
             FindKey(CnpIdx,ref,concepts[i]);
             if not Ok then begin etat := 5; mes_err := 'Concept'; exit end;
           end;
         conc := concepts[1];
         FindKey(ClsCnpIdx,ref,conc)
       end
       else NextKey(ClsCnpIdx,ref,conc);
    while Ok and (conc = concepts[1]) do
      begin
        i := 2;
        while (i <= args) and Ok do begin HstPosKey(ClsCnpIdx,ref,concepts[i]); i := i + 1 end;
        if Ok then begin
                     GetRec(ClsDat,ref,buf);
                     classe := buf.Classe;
                     donnees := buf.ClData; m := buf.M;
                     HstPosKey(ClsCnpIdx,ref,concepts[1]);
                     nref := nref + 1; etat := Ø; exit
                   end
              else begin
```

```
                    HstPosKey(ClsCnpIdx,ref,concepts[1]);
                    NextKey(ClsCnpIdx,ref,conc)
                  end
       end;
     etat := 1; mes_err := 'classes de concepts'
end;

procedure Rech_Concepts_Classes(args        : targs;
                                var classes : tclasses;
                                var concept : co_id;
                                var m       : boolean;
                                var donnees : co_data;
                                var nref    : inter);
var
     i,
     ref  : integer;
     clas : cl_id;
     buf  : CnpRec;
begin
     if args = 0 then begin etat := 3; mes_err := 'Classes'; exit end;
     if nref = 0 then
         begin
           For i := 1 to args do
             begin
               FindKey(ClsIdx,ref,classes[i]);
               if not Ok then begin etat := 5; mes_err := 'Classe'; exit end;
             end;
           clas := classes[1];
           FindKey(CnpClsIdx,ref,clas)
         end
         else NextKey(CnpClsIdx,ref,clas);
     while Ok and (clas = classes[1]) do
       begin
         i := 2;
         while (i <= args) and Ok do begin HstPosKey(CnpClsIdx,ref,classes[i]); i := i + 1 end;
         if Ok then begin
                      GetRec(CnpDat,ref,buf);
                      concept := buf.Concept;
                      donnees := buf.CoData; m := buf.M;
                      HstPosKey(CnpClsIdx,ref,classes[1]);
                      nref := nref + 1; etat := 0; exit
                    end
               else begin
                      HstPosKey(CnpClsIdx,ref,classes[1]);
                      NextKey(CnpClsIdx,ref,clas)
                    end
       end;
     etat := 1; mes_err := 'concepts de classes'
end;

procedure HstSuppres_Contient_Exercice(exercice : ex_id;
                                       ref      : integer);
var
     ref1 : integer;
     buf  : ClsRec;
begin
     FindKey(ClsExrIdx,ref1,exercice);
     while Ok do
       begin
```

```
        GetRec(ClsDat,ref1,buf);
        DeleteKey(ExrClsIdx,ref,buf.Classe);
        DeleteKey(ClsExrIdx,ref1,exercice);
        FindKey(ClsExrIdx,ref1,exercice)
      end;
end;

procedure HstSuppres_SS_Jacent_Concept(concept : co_id;
                                        ref     : integer);
var
     ref1 : integer;
     buf : ClsRec;
begin
     FindKey(ClsCnpIdx,ref1,concept);
     while Ok do
       begin
         GetRec(ClsDat,ref1,buf);
         DeleteKey(CnpClsIdx,ref,buf.Classe);
         DeleteKey(ClsCnpIdx,ref1,concept);
         FindKey(ClsCnpIdx,ref1,concept)
       end;
end;

procedure Rech_Complexite_Classe(classe         : cl_id;
                             var complexite : cm_id;
                             var m          : boolean;
                             var donnees    : cm_data);
var
     ref  : integer;
     buf  : ClsRec;
     buf1 : CplRec;
begin
     FindKey(ClsIdx,ref,classe);
     if not Ok then begin etat := 5; mes_err := 'Classe'; exit end;
     GetRec(ClsDat,ref,buf);
     complexite := buf.ClComplexite;
     if complexite = '' then begin mes_err := 'complexités'; etat := 1; exit end;
     FindKey(CplIdx,ref,complexite);
     GetRec(CplDat,ref,buf1);
     donnees := buf1.CmData; m := buf1.M;
     etat := 0
end;

procedure Rech_Classes_Complexite(complexite  : cm_id;
                              var classe  : cl_id;
                              var m       : boolean;
                              var donnees : cl_data;
                              var nref    : inter);
var
     ref  : integer;
     buf  : ClsRec;
     comp : cm_id;
begin
     if nref = 0 then begin
                        FindKey(CplIdx,ref,complexite);
                        if not Ok then begin etat := 5; mes_err := 'Complexité'; exit end;
                        comp := complexite;
                        FindKey(ClsCplIdx,ref,complexite)
                      end
```

```
                else NextKey(ClsCplIdx,ref,comp);
     if not Ok or (comp <> complexite)
               then begin etat := 1; mes_err := 'classes d''une complexité' end
               else begin
                      GetRec(ClsDat,ref,buf);
                      classe := buf.Classe;
                      donnees := buf.ClData; m := buf.M;
                      nref := nref + 1; etat := 0
                   end
end;

procedure Creation_Caracterise(classe     : cl_id;
                               complexite : cm_id);
var
     ref   : integer;
     buf   : ClsRec;
begin
     FindKey(CplIdx,ref,complexite);
     if not Ok then begin etat := 5; mes_err := 'Complexite'; exit end;
     FindKey(ClsIdx,ref,classe);
     if not Ok then begin etat := 5; mes_err := 'Classe'; exit end;
     GetRec(ClsDat,ref,buf);
     if buf.ClComplexite <> ''
        then begin etat := 7; mes_err := 'Caractérise'; exit end;
     buf.ClComplexite := complexite;
     PutRec(ClsDat,ref,buf);
     AddKey(ClsCplIdx,ref,complexite);
     etat := 0
end;

procedure Suppres_Caracterise(classe     : cl_id;
                              complexite : cm_id);
var
     ref : integer;
     buf : ClsRec;
begin
     FindKey(CplIdx,ref,complexite);
     if not Ok then begin etat := 5; mes_err := 'Complexité'; exit end;
     FindKey(ClsIdx,ref,classe);
     if not Ok then begin etat := 5; mes_err := 'Classe'; exit end;
     if not HstAutori(ref) then begin etat := 2; mes_err := 'Caractérise'; exit end;
     GetRec(ClsDat,ref,buf);
     if buf.ClComplexite <> complexite
         then begin etat := 8; mes_err := 'Caractérise'; exit end;
     buf.ClComplexite := '';
     PutRec(ClsDat,ref,buf);
     DeleteKey(ClsCplIdx,ref,complexite);
     etat := 0
end;

procedure HstSuppres_Caracterise_Complexite(complexite : cm_id);
var
     ref : integer;
     buf : ClsRec;
begin
     FindKey(ClsCplIdx,ref,complexite);
     while Ok do
       begin
         GetRec(ClsDat,ref,buf);
```

```
          buf.ClComplexite := '';
          PutRec(ClsDat,ref,buf);
          DeleteKey(ClsCplIdx,ref,complexite);
          FindKey(ClsCplIdx,ref,complexite);
        end;
end;
```

```
(*****************************************************)
(*                                                   *)
(*            GESTION DE L'HISTORIQUE                *)
(*                                                   *)
(*              Module CONCEPT.HST                   *)
(*                                                   *)
(*           Auteur : Merckx O. (1987)               *)
(*                                                   *)
(*****************************************************)


procedure Rech_Concept(concept     : co_id;
                       var m       : boolean;
                       var donnees : co_data);
var
    ref : integer;
    buf : CnpRec;
begin
    FindKey(CnpIdx,ref,concept);
    if not Ok then begin etat := 5; mes_err := 'Concept' end
            else begin
                   GetRec(CnpDat,ref,buf);
                   donnees := buf.CoData; m := buf.M;
                   etat := 0
                 end
end;

procedure Creation_Concept(concept : co_id;
                           donnees : co_data);
var
    ref : integer;
    buf : CnpRec;
begin
    FindKey(CnpIdx,ref,concept);
    if Ok then begin etat := 6; mes_err := 'Concept' end
        else begin
               buf.Status := 0; buf.Concept := concept;
               buf.CoData := donnees; buf.M := false;
               AddRec(CnpDat,ref,buf);
               AddKey(CnpIdx,ref,concept);
               etat := 0
             end
end;

procedure Modif_Concept(concept : co_id;
                        donnees : co_data);
var
    ref,
    ref1 : integer;
    buf  : CnpRec;
begin
    FindKey(CnpIdx,ref,concept);
    if not Ok then begin etat := 5; mes_err := 'Concept'; exit end;
    FindKey(CnnCnpIdx,ref1,concept);
    if Ok then begin etat := 12; mes_err := 'Concept' end
        else begin
               GetRec(CnpDat,ref,buf);
               buf.CoData := donnees;
               PutRec(CnpDat,ref,buf);
```

```
                etat := 0
              end
end;

procedure Liste_Concepts(var concept : co_id;
                         var m       : boolean;
                         var donnees : co_data;
                         var nref    : inter);
var
    ref  : integer;
    buf  : CnpRec;
begin
    if nref = 0 then ClearKey(CnpIdx);
    NextKey(CnpIdx,ref,concept);
    if not Ok then begin etat := 1; mes_err := 'concepts' end
            else begin
                 GetRec(CnpDat,ref,buf);
                 donnees := buf.CoData; m := buf.M;
                 nref := nref + 1; etat := 0
               end
end;

procedure Suppres_Concept(concept : co_id);
var
    ref,
    ref1 : integer;

procedure HstMrkCnp(ref : integer);
var
    buf : CnpRec;
begin
    GetRec(CnpDat,ref,buf);
    buf.M := true;
    PutRec(CnpDat,ref,buf)
end;

begin
    FindKey(CnpIdx,ref,concept);
    if not Ok then begin etat := 5; mes_err := 'Concept'; exit end;
    FindKey(CnnCnpIdx,ref1,concept);
    if Ok then begin etat := 2; mes_err := 'Concept'; exit end;
    HstSuppres_SS_Jacent_Concept(concept,ref);
    DeleteKey(CnpIdx,ref,concept);
    DeleteRec(CnpDat,ref);
    etat := 0
end;
```

```
(*****************************************************)
(*                                                   *)
(*             GESTION DE L'HISTORIQUE               *)
(*                                                   *)
(*               Module COMPLEXI.HST                 *)
(*                                                   *)
(*             Auteur : Merckx O. (1987)             *)
(*                                                   *)
(*****************************************************)


procedure Rech_Complexite(complexite  : cm_id;
                          var m       : boolean;
                          var donnees : cm_data);
var
    ref : integer;
    buf : CplRec;
begin
    FindKey(CplIdx,ref,complexite);
    if not Ok then begin etat := 5; mes_err := 'Complexité' end
           else begin
                  GetRec(CplDat,ref,buf);
                  donnees := buf.CmData; m := buf.M;
                  etat := 0
                end
end;

procedure Creation_Complexite(complexite : cm_id;
                              donnees    : cm_data);
var
    ref : integer;
    buf : CplRec;
begin
    FindKey(CplIdx,ref,complexite);
    if Ok then begin etat := 6; mes_err := 'Complexité' end
        else begin
               buf.Status := 0; buf.Complexite := complexite;
               buf.CmData := donnees; buf.M := false;
               AddRec(CplDat,ref,buf);
               AddKey(CplIdx,ref,complexite);
               etat := 0
             end
end;

procedure Modif_Complexite(complexite : cm_id;
                           donnees    : cm_data);
var
    ref,
    ref1 : integer;
    buf  : CplRec;
begin
    FindKey(CplIdx,ref,complexite);
    if not Ok then begin etat := 5; mes_err := 'Complexité' end;
    FindKey(ClsCplIdx,ref1,complexite);
    while Ok do
      begin
        if not HstAutori(ref1) then begin etat := 12; mes_err := 'Complexité'; exit end;
        FindKey(ClsCplIdx,ref1,complexite)
      end;
```

```
     GetRec(CplDat,ref,buf);
     buf.CmData := donnees;
     PutRec(CplDat,ref,buf);
     etat := 0
end;

procedure Liste_Complexites(var complexite : cm_id;
                            var m          : boolean;
                            var donnees    : cm_data;
                            var nref       : inter);
var
     ref  : integer;
     buf  : CplRec;
begin
     if nref = 0 then ClearKey(CplIdx);
     NextKey(CplIdx,ref,complexite);
     if not Ok then begin etat := 1; mes_err := 'complexités' end
               else begin
                      GetRec(CplDat,ref,buf);
                      donnees := buf.CmData; m := buf.M;
                      nref := nref + 1; etat := 0
                    end
end;

procedure Suppres_Complexite(complexite : cm_id);
var
     ref,
     ref1 : integer;

procedure HstMrkCpl(ref : integer);
var
     buf : CplRec;
begin
     GetRec(CplDat,ref,buf);
     buf.M := true;
     PutRec(CplDat,ref,buf)
end;

begin
     FindKey(CplIdx,ref,complexite);
     if not Ok then begin etat := 5; mes_err := 'Complexité'; exit end;
     FindKey(ClsCplIdx,ref1,complexite);
     while Ok do
       begin
         if not HstAutori(ref1)
             then begin HstMrkCpl(ref); etat := 2; mes_err := 'Complexité'; exit end;
         FindKey(ClsCplIdx,ref1,complexite)
       end;
     HstSuppres_Caracterise_Complexite(complexite);
     DeleteKey(CplIdx,ref,complexite);
     DeleteRec(CplDat,ref);
     etat := 0
end;
```

```
(*******************************************************)
(*                                                     *)
(*              GESTION DE L'HISTORIQUE                *)
(*                                                     *)
(*                Module CONNAISS.HST                  *)
(*                                                     *)
(*             Auteur : Merckx O. (1987)               *)
(*                                                     *)
(*******************************************************)


procedure Rech_Connaissance(etudiant    : et_id;
                            concept     : co_id;
                            var ner     : cn_ner;
                            var nex     : cn_nex;
                            var donnees : cn_data);
var
    ref  : integer;
    buf  : CnnRec;
    conn : cn_id;
begin
    FindKey(EtdIdx,ref,etudiant);
    if not Ok then begin etat := 5; mes_err := 'Etudiant'; exit end;
    FindKey(CnpIdx,ref,concept);
    if not Ok then begin etat := 5; mes_err := 'Concept'; exit end;
    conn := etudiant + concept;
    FindKey(CnnIdx,ref,conn);
    if not Ok then begin etat := 5; mes_err := 'Connaissance' end
              else begin
                     GetRec(CnnDat,ref,buf);
                     ner := buf.CnNer; nex := buf.CnNex;
                     donnees := buf.CnData; etat := 0
                   end
end;

procedure Modif_Connaissance(etudiant : et_id;
                             concept  : co_id;
                             donnees  : cn_data;
                             ner      : cn_ner;
                             nex      : cn_nex);
var
    ref  : integer;
    buf  : CnnRec;
    conn : cn_id;
begin
    FindKey(EtdIdx,ref,etudiant);
    if not Ok then begin etat := 5; mes_err := 'Etudiant'; exit end;
    FindKey(CnpIdx,ref,concept);
    if not Ok then begin etat := 5; mes_err := 'Concept'; exit end;
    conn := etudiant + concept;
    FindKey(CnnIdx,ref,conn);
    if not Ok then begin etat := 5; mes_err := 'Connaissance' end
              else begin
                     GetRec(CnnDat,ref,buf);
                     buf.CnData := donnees;
                     buf.CnNer := ner; buf.CnNex := nex;
                     PutRec(CnnDat,ref,buf);
                     etat := 0
                   end
```

```
end;

procedure Rech_Connaissances_Etudiant(etudiant    : et_id;
                                      var concept : co_id;
                                      var ner     : cn_ner;
                                      var nex     : cn_nex;
                                      var donnees : cn_data;
                                      var nref    : inter);
var
    ref  : integer;
    buf  : CnnRec;
    etud : et_id;
begin
    if nref = 0 then begin
                       FindKey(EtdIdx,ref,etudiant);
                       if not Ok then
                         begin etat := 5; mes_err := 'Etudiant'; exit; end;
                       etud := etudiant;
                       FindKey(CnnEtdIdx,ref,etudiant)
                     end
                else NextKey(CnnEtdIdx,ref,etud);
    if not Ok or (etud <> etudiant)
       then begin etat := 1; mes_err := 'connaissances d''un étudiant' end
       else begin
              GetRec(CnnDat,ref,buf);
              concept := buf.CnConcept; ner := buf.CnNer;
              nex := buf.CnNex; donnees := buf.CnData;
              nref := nref + 1; etat := 0
            end
end;

procedure Rech_Connaissances_Concept(concept      : co_id;
                                     var etudiant : et_id;
                                     var ner      : cn_ner;
                                     var nex      : cn_nex;
                                     var donnees  : cn_data;
                                     var nref     : inter);
var
    ref  : integer;
    buf  : CnnRec;
    conc : co_id;
begin
    if nref = 0 then begin
                       FindKey(CnpIdx,ref,concept);
                       if not Ok then
                          begin etat := 5; mes_err := 'Concept'; exit; end;
                       conc := concept;
                       FindKey(CnnCnpIdx,ref,concept)
                     end
                else NextKey(CnnCnpIdx,ref,conc);
    if not Ok or (conc <> concept)
      then begin etat := 1; mes_err := 'connaissances d''un concept' end
      else begin
             GetRec(CnnDat,ref,buf);
             etudiant := buf.CnEtudiant; ner := buf.CnNer;
             nex := buf.CnNex; donnees := buf.CnData;
             nref := nref + 1; etat := 0
           end
end;
```

```
procedure HstSuppres_Connaissances_Etudiant(etudiant : et_id);
var
     ref  : integer;
     buf  : CnnRec;
     conn : cn_id;
begin
     FindKey(CnnEtdIdx,ref,etudiant);
     while Ok do
       begin
         GetRec(CnnDat,ref,buf);
         conn := etudiant + buf.CnConcept;
         DeleteKey(CnnCnpIdx,ref,buf.CnConcept);
         DeleteKey(CnnEtdIdx,ref,etudiant);
         DeleteKey(CnnIdx,ref,conn);
         DeleteRec(CnnDat,ref);
         FindKey(CnnEtdIdx,ref,etudiant)
       end
end;

procedure HstMaj_Connaissance(etudiant : et_id;
                              concept  : co_id;
                              b        : boolean);
var
     ref  : integer;
     buf  : CnnRec;
     conn : cn_id;
begin
     conn := etudiant + concept;
     FindKey(CnnIdx,ref,conn);
     with buf do
       if Ok then begin (* Modification *)
                    GetRec(CnnDat,ref,buf);
                    CnNex := CnNex + 1; if b then CnNer := CnNer + 1;
                    PutRec(CnnDat,ref,buf)
                  end
             else begin (* Création *)
                    Status := Ø; CnEtudiant := etudiant;
                    CnConcept := concept; CnData := '';
                    CnNex := 1; if b then CnNer := 1 else CnNer := Ø;
                    AddRec(CnnDat,ref,buf);
                    AddKey(CnnIdx,ref,conn);
                    AddKey(CnnEtdIdx,ref,etudiant);
                    AddKey(CnnCnpIdx,ref,concept)
                  end
end;
```