Collection de logiciels libres pour l'enseignement et le développement

Exploration du concept de module en programmation

CoLOED:NT-2024-01_Module

Luc Lavoie *(luc.lavoie@usherbrooke.ca)*

—

CoLOED/Scriptorum/NT-2024-01_Module, version 0.3.1b, en date du 2025-11-18

— document de travail, ne pas citer à l'externe —

Sommaire

Le concept de module est prévalent en informatique appliquée, particulièrement en programmation et en architecture. Fréquemment associé, voire confondu, avec d'autres structures (notamment celle de classe), il est parfois négligé dans les *curricula* contemporains. La présente note vise à explorer le concept de module et à montrer son importance en regard du génie logiciel en général, de la vérification et de la validation en particulier. Sur cette base, et en nous inspirant des langages Modula-2, Ada et Eiffel, nous proposons une méthode pour concrétiser les modules en SQL. Nous montrons également comment les utiliser comme socle à la gestion des droits d'accès aux bases de données.

Mise en garde

La présente note technique est en cours d'élaboration ; en conséquence, elle est incomplète et peut contenir des erreurs. En particulier, une analyse des propostions est requise en vue d'une reformulation et d'une simplication de façon à en arriver à une proposition unique pouvvant être systématisée et mise en pratique.

Historique

diffusion	resp.	description
2025-11-18	LL	Correction de coquilles.
2024-10-09	LL	Récupération de la proposition 3 de la version du 2023-08-18.
2024-08-08	LL	Revue.
2024-07-22	LL	Adaptation et généralisation dans le cadre de CoLOED.
2022-08-18	LL	Prise en compte des commentaires de la réunion du 17 aout 2022 ; reformulations diverses.
2022-08-10	LL	Ajout de la section 3, intégration des commentaires récoltés lors de la réunion de travail.
2022-07-23	LL	Ébauche initiale.

Table des matières

Introduction

Depuis la fin des années (mille-neuf-cents) soixante (Simula-67, Algol-68, la conférence de Garmisch-Partenkirchen en 1968, etc.), le concept de module (aussi appelé composant ou composante) est utilisé par le génie logiciel pour structurer, voire formaliser, l'architecture, la conception, la vérification, la validation, la revue logicielle, les types de tests, la gestion du code, la gestion des configurations, etc. Le concept fut ensuite développé notamment par Per Brinch Hansen et Niklaus Wirth au cours des années (mille-neuf-cents) soixante-dix et incorporé à de nombreux langages de programmation (Pascal concurrent, Clue, Alphard, Modula-2, Modula-3, Ada, Oberon, Eiffel, etc.). En parallèle, il a été utilisé comme unité atomique pour la description de nombreux processus de génie logiciel et des outils développés pour leur mise en place.

Compte tenu de son importance dans les fondements théoriques du génie logiciel comme dans l'expression de ses règles de pratique, on pourrait croire que le concept est rigoureusement défini et systématiquement enseigné. Trop fréquemment, ce n'est pas le cas.

Le concept de module est souvent :

- confondu avec celui de classe ;
- confondu avec celui de « package » ;
- mal intégré, ou parfois même ignoré, au sein de nombreux langages répandus, dont plusieurs langages axiomatiques (Prolog, SQL, OWL) ;
- rendu invisible dans les processus agiles, mais néanmoins utilisé par les outils les soutenant ;
- négligé, voire disparu, des *curricula* informatiques en conséquence de l'occultation des principes d'architecture, en particulier du principe de couplage minimal pour une cohérence maximale.

Le présent document a pour objectif d'établir une caractérisation pratique des modules (section 1) de façon à déterminer les meilleurs moyens de les mettre en oeuvre grâce au langage SQL[et parfois malgré lui] (section 2) en vue de proposer une solution concrète pouvant être utilisée au sein du CoLOED, des groupes re recherche GRIIS et Μητις, et, plus largement, dans un contexte de formation universitaire (section 3).

Une conclusion et un glossaire complètent le document.

1. De la nature du module

Relativement à un langage de programmation, un module est un regroupement d'entités (types, fonctions, procédures) instrumentant le découplage de la définition des entités et de leur concrétisation. Le module peut comporter un état (composé de variables), ou pas. S'il comporte un état, les entités prises en compte peuvent comprendre les invariants et les automatismes.

Ce découplage est en fait l'occasion de cibler, et d'atteindre, d'autres objectifs, par exemple :

- décomposer un tout en parties plus facilement gérables, évoluables et vérifiables ;
- regrouper des définitions complémentaires soumises aux mêmes règles d'accès ;
- permettre le choix cohérent entre différentes concrétisations (variantes) d'un même groupe de définitions.

Les critères de regroupement (ou de séparation) des entités en modules varient selon les auteurs, les méthodes de développement, voire le type de produit développé. Généralement, il découle de ces critères que les entités regroupées dans un même module concourent à un même objectif et présentent des propriétés similaires en termes de gestion du développement (complémentarité, testabilité, évolutivité présumée, etc.).

Le module est un concept incontournable en programmation, que celle-ci soit opératoire, fonctionnelle ou axiomatique. En conséquence, plusieurs auteurs recommandent d'élargir le domaine d'application du concept de modularité * à la spécification (qui comprend la définition) ; * à l'expression (une représentation

particulière pouvant dépendre de choix de modélisation ou de conception).

Afin de permettre le découplage de la spécification des entités et de leur expression, le module est ainsi divisé en deux parties :

- l'interface (une syntaxe associée aux entités définies par la spécification) et
- la mise en oeuvre (l'encodage associé à l'expression des entités, encodage qui prend différentes formes selon que la programmation est opératoire, fonctionnelle ou axiomatique).

La séparation de l'interface et de la mise en oeuvre offre notamment l'avantage de permettre l'utilisation du module sur la seule base de sa définition (son interface), sans avoir à prendre connaissance de l'expression (sa mise en oeuvre). Il est alors possible de diffuser les seules interfaces aux programmeurs utilisateurs. De même, tant que l'interface n'est pas modifiée, il est possible de faire évoluer la mise en oeuvre du module sans induire de modifications aux modules clients (dans la mesure où la mise en oeuvre modifiée respecte toujours la définition).

Pour cette raison, l'interface est aussi appelée « partie publique » du module et la mise en oeuvre, « partie privée » du module.

Certains langages permettent la compilation séparée de l'interface et de la mise en oeuvre tels Modula et Oberon (ou, de façon moins formelle, C et C++). D'autres langages, tel Eiffel, produisent automatiquement la partie interface à partir de l'expression cohésive du module de sorte que l'interface devient alors consultable séparément. De plus, certains langages permettent d'associer la spécification (ou une partie de celle-ci) à l'interface et ainsi automatiser la vérification (le respect) de la spécification (par exemple, Eiffel).

L'interface est donc porteuse d'une partie de la spécification (voire de sa totalité, le plus souvent sous la forme de commentaires structurés, plus rarement sous la forme d'une expression conforme à un langage formel) et, en particulier, des types et des signatures des opérations mises à disposition. L'interface peut être vérifiée (compilée) et même validée (soumise à la revue) indépendamment de la mise en oeuvre. Il est ainsi possible de l'utiliser (de l'importer) dans d'autres modules afin de les valider en regard de celle-ci, et ce, toujours sans que la mise en oeuvre ne soit disponible ni même conçue. Lors de la compilation de la mise en oeuvre, l'interface est consultée afin d'en confirmer le respect. Attendu une gestion adéquate des sources, on peut ainsi vérifier, en partie, la non-dérogation de la mise en oeuvre et des tests unitaires peuvent être engendrés (parfois même automatiquement) en regard de l'interface.

Ces propriétés sont particulièrement utiles dans la mise en place de processus de génie logiciel axés sur la qualité, la spécification des fonctionnalités et le respect des procédés tels que CII (cascade itérative et incrémentale), V et RUP (*Rational Unified Process*). Elles permettent aussi le développement concurrent de la mise en oeuvre et des tests sur une base solide. Le concept de module permet aussi la division du travail en unités plus facilement abordables et planifiables, tout en facilitant l'élaboration de tests stratifiés (unitaires, d'intégration, de non-régression, etc.).

1.1. Rappels sur les tests

Dans ce qui suit, le terme entité désigne tout élément pouvant être défini au sein de l'interface d'un module (type, fonction, procédure, invariant, automatisme). Les valeurs remarquables (aussi appelée constantes) sont des cas particuliers de fonctions (le plus souvent sans paramètres). La possibilité d'inclure les variables dans l'interface demeure très discutée, d'autant que leur inclusion peut toujours être remplacée par la mise à disposition de fonctions et procédures appropriées.

Test unitaire

Un test unitaire a pour objectif de vérifier les propriétés d'une entité. On distingue :

- les propriétés axiomatisées
  - pour lesquelles le test se limite le plus souvent à une suite d'assertions découlant directement des axiomes,

    - les tests de ce type étant alors qualifiés de tests axiomatiques ;
- les propriétés non axiomatisées
    - pour lesquelles le test se limite le plus souvent à une suite de comparaisons portant sur des cas particuliers remarquables,
    - les tests de ce type étant alors qualifiés de tests non axiomatiques.

Il est ainsi généralement plus facile d'assurer l'évolutivité des tests axiomatiques que des tests non axiomatiques, ces derniers devant souvent mettre en jeu des valeurs précises permettant la comparaison.

On remarque que les tests unitaires portant sur les propriétés structurelles sont fréquemment non axiomatiques.

Test d'intégration

Un test d'intégration a pour objectif de vérifier (l'adéquation de) l'interaction entre certaines entités provenant de modules distincts. Par son caractère partiel, il permet de réduire la portée si un diagnostic est requis (lors de la détection d'une erreur). Le test d'intégration permet également de factoriser certaines vérifications communes à plusieurs modules différents (utilisant les mêmes sous-ensembles d'entités). En combinant plusieurs tests d'intégration, il est ainsi possible de vérifier l'ensemble des interactions (ou un sous-ensemble jugé suffisant) d'un ensemble déterminé de modules.

On remarque que les tests d'intégration utilisent fréquemment des propriétés structurelles à défaut d'une axiomatisation suffisamment poussée de la spécification des modules ou de la formalisation du lien entre l'axiomatisation et la conception.

En pratique, il est recommandé de limiter le nombre de modules distincts pour un même test d'intégration de façon à faciliter le diagnostic et l'évaluation de la couverture (au sein du test).

Test de non-régression

Un test de non-régression a pour objectif de vérifier, lors d'une évolution, la préservation d'un comportement antérieur (on s'assure de ne pas dévier du comportement correct). La plupart des tests de non-régression sont mis en oeuvre à l'aide de tests comparatifs. Tous les tests comparatifs ne sont pas des tests de non-régression… pas plus que l'inverse.

La nécessité des tests de non-régression vient d'au moins deux contextes très différents. Le premier est celui de l'occurrence d'une erreur documentée, corrigée, dont on veut s'assurer qu'elle ne survienne pas à nouveau à la faveur d'une évolution. Le second contexte est celui de la faisabilité d'un test qu'il n'avait pas été possible de faire dans les versions antérieures, car le résultat correct ne pouvait être obtenu que du logiciel lui-même — pour autant qu'il soit exact. Or, ce logiciel ayant été éprouvé dans la réalité et certains de ses résultats ayant été confirmés indépendamment, il devient possible de les utiliser pour instrumenter le test auparavant impraticable (ou à tout le moins non significatif).

!!! À compléter !!!

structurel versus essentiel

- définir : structurel ; essentiel ; (essentiel ≡ non structurel).

boite noire versus boite blanche

- corriger : noire → opaque ; blanche → transparente ?
- développer : …

relation entre structuralité et opacité

- développer : structurel → boite opaque ; essentiel → boite transparente.

1.2. Rappels sur les critères de composition des modules

!!! À compléter !!!

- Ne pas confondre, l'objectif de la modularisation, ses modalités, ou sa structure et la nature du module
- Plusieurs approches proposées ent 1970 et 1990
  - approches formelles
    - afin de fonder le module sur des bases solides, arbitrables
    - afin de permettre le raisonnement et la déduction logique
  - approches pragmatiques (en termes de temps de développement, de revue, etc.)
    - trop petit : amortissement insuffisant des ressources requises par la mise en place
    - trop grand : risque d'augmentation des erreurs, risque de perte de contrôle

TAA (formel)

- structure : semi-groupe, groupe, corps, anneau…
- éléments : constructeurs, manipulateurs, observateurs
- noyaux, axiomatique et équivalence (composition)
- complétude en regard des propriétés
- complétude en regard des tests
- application : beaucoup est modélisable comme TAA, mais pas tout (dont IPM)
- difficulté : formation déficiente, « blocage »

FP, FPP (formel)

- structure : signatures, sous-composants informationnels et relations
- éléments : points de fonction, fonction d'évaluation
- application : requière données empiriques difficiles à obtenir, maintenir et valider
- difficulté : formation déficiente, « blocage »

XX1 (règles de pratique)

- lequel choisir ?

XX2 (règles de pratique)

- lequel choisir ?

2. SQL et les modules

Le langage SQL ne soutient pas directement la définition de modules.

- De la distinction entre base de données et module.
  Comme tout autre service, une base de données peut être vue comme un module dont l'étt est représenté par l'ensemble des variables de relations (tables) de son modèle de données. De la même façon également, il est souhaitable de pouvoir convoquer (et pour cela définir et mettre en oeuvre) d'autres modules qui ne soient pas des services (et donc utilisable par le biais de leur interface propre, « normale »), plutôt que par des transactions inter-services).
- De la distinction entre schéma et module.
  Le schéma SQL est essentiellement un mécanisme de définition de portée (syntaxique) éventuellement utilisable pour faciliter la définition de règles d'accès communes aux entités qu'il comprend. Il ne comporte pas de mécanisme de découplage entre définition et mise en oeuvre des entités, bien que le partage entre entités publiques et privées puisse être spécifié au prix d'un effort conséquent attendu l'absence du mécanisme d'interface.
- De certains principes.
  Si la structure (expression → mise en oeuvre) doit pouvoir évoluer sans induire de changement à la spécification (interface), les tables (qui sont des variables de relation) doivent placées être dans la partie privée ;

- Des palliatifs envisagés.
  - créer deux portées (grâce aux schémas ?) : une portée publique, une privée ;
  - si la signature des routines ne peut être dissociée de leur corps (grâce au mécanisme de redéfinition, CREATE OR REPLACE), la mise en oeuvre sera exposée, mais des règles d'accès devront en interdire la modification aux modules clients ;
  - quand on prend en compte les droits d'accès, il faut séparer les interfaces selon les politiques devant leur être appliquées ;
  - les parties (et ultimement les fichiers résultant de la segmentation de celles-ci) seront plus nombreuses que les schémas afin, notamment, de respecter les dépendances entre les définitions lors de la compilation.
- Quelques structures potentielles de modules, parmi d'autres :
  - Module A - décomposition

    ```
    début partie+ fin
    ```

  - Module B - factorisation - (attendu un mécanisme permettant de choisir la variante désirée)

    ```
    début partie_commune variante+ fin
    ```

  - Module C - partage public-privé simple

    ```
    début (public privé)+ fin
    ```

  - Module D - partage public-privé avec décomposition

    ```
    début (public privé partie*)+ fin
    ```

  - Module E - partage public-privé avec variantes

    ```
    début (public privé variante*)+ fin
    ```

- La partie « fin » est-elle requise ?
  En termes du module lui-même, non. Par contre, il faut prévoir une partie de gestion des modules (qui correspondent aux CREATE et DROP des schémas et de leurs contenus). Dans ce cas, il serait probablement préférable de distinguer (et séparer) la portion de gestion, d'une éventuelle partie partie d'initialisation. En clair :
  - Module A - décomposition

    ```
    définition partie+ élimination
    ```

  - Module B - factorisation

    ```
    définition partie_commune variante+ élimination
    ```

  - Module C - partage public-privé simple

```
définition (initialisation public privé)+ élimination
```

- Est-il utile de prévoir plusieurs sortes de modules ?
  - modules sans variables (donc sans état) ;
  - modules avec variables (donc avec état)
    - instance unique allouée statiquement (singleton),
    - instances multiples allouées statiquement,
    - instances multiples allouées dynamiquement,
    - …
  - A priori, non : une allocation unique (par BD), statique, avec ou sans état (cela ne semble pas faire de différence sur la structure du module) m'apparait bien suffisante (dans un contexte de bases de données). L'usage en décidera.
- L'hypothèse selon laquelle le découpage en modules correspond le plus souvent au découpage requis pour la gestion des droits d'accès est à discuter. L'usage en décidera.
- La gestion des dépendances entre les parties du module est souvent réalisée en imposant un ordre aux définitions. Quelle flexibilité recherchons-nous ? Pour le moment, c'est l'ordre de construction prescrit par le script de construction (sh ou psql) qui en décidera.
- Gestion des droits d'accès
  - Avec ou sans schémas ? Avec, afin d'alléger une spécification déjà lourde.
  - Répartie dans les fichiers ou concentrée dans un fichier dédié (voire plusieurs) ?
    - En pratique il en faut un minimum de deux, puisque certaines commandes sont application à la BD globalement et d'autres aux schémas (voire aux entités elles-mêmes)
    - J'essaierais de nous en tenir à cela pour éviter l'éparpillement des consignes qui, en lui-même, est un enjeu de sécurité.
    - Plus précisément, un fichier pour le SGBD (spa) et un par interface (bpa) attendue que la plupart des modules n'en auront qu'une et que les deux schémas lorsque requis pour séparer la partie privée de la partie publique peuvent être traités dans dans même fichier.

3. Proposition courante (au 2024-08-08)

3.1. Décomposition des modules en fichiers

Un module est divisé en six types de parties

- ***partie définition*** **(def)**, elle comprend les éléments de configuration permettant d'inclure le module dans une base de données, en particulier la définition des schémas requis et de leurs propriétés ;
- ***partie élimination*** **(eli)**, elle permet de retirer un module (ses schémas et leurs entités) de la base de données ;
- ***partie initiale*** **(ini)**, elle comprend les éléments d'initialisation propres à la partie
- ***partie publique*** **(pub)**, elle rassemble l'ensemble des entités offertes, généralement sans les mises en oeuvre correspondantes ;
- ***partie privée*** **(pri)**, elle rassemble l'ensemble des entités publiques (définition et mise en oeuvre) ainsi que les mises en oeuvre des entités publiques (dans le cas CREATE OR RELACE) ;
- ***partie BD de la politique d'accès*** **(bpa)**, cette partie comprend les éléments de configuration dont la portée est l'instance du module dans une BD spécifique, en particulier les droits d'utilisation et d'exécution sur des entités particulières.

Afin de mettre en place la structure modulaire, plus particulièrement lorsque des politiques d'accès sont à mettre en place, une partie propre à la BD elle-même doit être prévue :

- ***partie SGBD de la politique d'accès* (spa)**, cette partie comprend les éléments de configuration dont la portée est le SGBD lui-même, en particulier les rôles qui ne doivent être définis qu'une fois pour toutes les occurrences du module, peu importe la base de données.

Dans la mesure où la mise en oeuvre complète de la partie privée serait incluse dans la partie publique elle-même, la partie privée pourrait être vide. ???

La partie initiale est séparée des parties publiques et privées, car elle est susceptible d'être omise ou remplacée, notamment parce qu'elle comporte souvent des variantes dialectales importantes, voire parce qu'au sein d'un même dialecte les usages peuvent varier considérable d'un site à un autre. ???

Structure 1 *(avec CREATE OR REPLACE)*

En pratique, la structure et l'ordre de compilation devraient être le suivant :

- au moment de la première inclusion du module dans une BD du SGBD ;
  - spa.
- à chaque inclusion du module dans une BD
  - def ;
  - (ini?, pub, pri, bpa?)+ ;
- à chaque retrait du module dans une BD
  - eli.

Structure 2 *(sans CREATE OR REPLACE)*

La seule différence introduite par la structure 2 est que la partie privée peut devenir inutile lorsqu'aucune entité proprement privée n'est requise hors de la partie initiale puisque la mise en oeuvre des entités publiques devra avoir été incluse dans la partie pub.

Recommandations communes

Bien que certaines parties soient facultatives, nous suggérons de toujours définir les parties initiales et privées, même si elles sont vides, afin de faciliter l'évolution des modules et la rédaction des scripts. Ainsi, les trois parties ini, pub, pri seraient en tout temps définies.

Les parties relatives à la politique d'accès (bpa et spa) peuvent être omises.

Lorsqu'un module met à disposition plusieurs interfaces distinctes, seule l'interface commune obligatoire sera représentée dans les parties (ini, pub, pri, bpa). Les autres interfaces détermineront chacune leurs propres parties, par exemple pour une interface facultative A : (ini-A, pub-A, pri-A, bpa-A). Idélament des interfaces devraient être indépendantes et ne dépendre que de l'interface obligatoire commune.

3.2. Mise en oeuvre de la proposition

Masquer ou non les mises en oeuvre avec CREATE OR REPLACE ?

- Avantages
  - documentation et publication facilitées de la spécification du module ;
  - traçabilité facilitée de la portée des changements ;
  - meilleure gestion du contrôle des changements.
- Inconvénients
  - commande absente de la norme ISO 9075:2016 et de plusieurs dialectes, donc enjeu de transportabilité ;
  - les avantages mentionnés peuvent facilement être réduits à néant par des politiques laxistes de gestion de version et d'intégration continue.

Nomenclature de fichiers

Pour faciliter la documentation des modules, l'organisation des répertoires et le développement de scripts communs, il est recommandé d'établir la nomenclature des fichiers associés aux modules sur la base des parties indiquées précédemment.

Une telle proposition a été élaborée dans CoLOED:STD-ENV-01.

Gestion de la construction

xxx

Gestion des tests unitaires

xxx

Gestion des tests d'intégration

xxx

Gestion des tests d'efficience

xxx

4. Propostions antérieures

4.1. Proposition 1

Structure générale

- pub (schéma `<M>`)
  - la partie publique du module nécessaire tant à l'externe qu'à l'interne ;
  - définitions de types (DOMAIN et TYPE) ;
  - définitions de fonctions «pures» (IMMUTABLE FUNCTION).
- pri (schéma `<M>_pri`)
  - la partie privée du module qui comprend la portion de la mise en oeuvre.
- api (schéma `<M>_<X>`)
  - la partie de l'interface programmatique du module `<M>` assujetti à la politique d'accès `<X>` ;
  - elle comprend à la fois la signature des routines et, malheureusement, leur mise en oeuvre.
- meo (schéma `<M>_pri`)
  - la partie privée du module qui comprend la portion de la mise en oeuvre (meo) de l'API.

Règles

- Définir un usager (USER) `<M>` propriétaire des schémas des parties `<M>` et `<M>_pri`.
- Pour chaque politique d'accès `<X>`
  - Définir un usager (USER) `<X>` propriétaire des schémas de la partie `<M>_<X>`
  - Définir chacune des routines en tant que SECURITY DEFINER.

4.2. Proposition 2

Structure générale

- <M>_def (schéma `<M>`)
  - partie définissant le module <M> lui-même ;
  - elle comprend que
    - la définition du schéma (SCHEMA) et du rôle associé (ROLE),
    - l'octroi de droits d'accès (GRANT),

    - elle est aussi appelée définition commune publique.
- <M>_pub (schéma `<M>`)
    - partie de l'interface du module <M> commune à toutes les autres parties (et donc commune à toutes les politiques d'accès) ;
    - elle comprend des entités nécessaires tant à l'interne qu'à l'externe du module ;
    - elle ne peut comprendre que
        - des définitions de types (DOMAIN et TYPE),
        - des définitions de fonctions «pures» (IMMUTABLE FUNCTION),
        - l'octroi de droits d'accès (GRANT),
    - elle est aussi appelée base commune publique.
- <M>_<X>_pub (schéma `<M>_<X>`)
    - partie de l'interface du module `<M>` assujettie à la politique d'accès `<X>` ;
    - elle ne peut comprendre que
        - la définition du schéma (SCHEMA) et du rôle associé (ROLE),
        - l'octroi de droits d'accès (GRANT),
        - les signatures des routines qui la composent (le corps de chaque signature doit être limité à sa plus simple expression syntaxique puisqu'il a pour vocation à être remplacé par un corps effectif dans `<M>_<X>_pri`).
- <M>_pri (schéma `<M>_pri`)
    - partie privée du module qui comprend de la mise en oeuvre commune du module `<M>`.
- <M>_<X>_pri (schéma `<M>_<X>`)
    - partie privée du module qui comprend la mise en oeuvre des routines définies dans `<M>_<X>_pri` sous la forme du remplacement (REPLACE) du corps d'origine par le corps effectif.

4.3. Analyse des propositions 1 et 2

Remarques

- Finalement, les deux propositions sont semblables ; la proposition 2 permet toutefois de publier uniquement `<M>_pub` et les `<M>_<X>_pub` sans dévoiler les mises en oeuvre ; cela reflète mieux la vraie teneur du contrat représenté par l'interface.
- Cela peut faire un grand nombre de fichiers et une certaine redondance des signatures ; par contre, je crois que cela facilitera la gestion des sources et celles de versions dans le sens où la portée des modifications sera plus claire, puisque le découpage des fichiers respecte mieux celui des responsabilités (pas de fichiers hybrides avec interface et mise en oeuvre).
- Si une seule politique d'accès gouverne le module, les parties `<M>_<X>_pub` peuvent être intégrées dans la partie `<M>_pub` et les parties `<M>_<X>_pri` dans la partie `<M>_pri` avec, au passage, les bénéfices que permettent les simplifications corolaires.
- La redondance des signatures peut être palliée par une génération initiale et un contrôle subséquent grâce à des tests structurels facilement automatisables.
- Quant au nombre de fichiers, il est somme toute restreint dans le cas simple (deux) et n'augmente qu'à partir où du moment où plusieurs GPA sont requises (les deux de base, plus deux par GPA : *2*(#GPA + 1)* — on suppose ici que toute partie publique est accompagnée d'une contre-partie privée, ce qui pourrait ne pas être le cas).
- À ces parties (fichiers), on pourrait vouloir ajouter une autre, initiale (`<M>_def`), permettant de factoriser la mise en contexte, la documentation et même des instructions supplémentaires tributaires de l'environnement de développement.

Commentaires

- CK01 2022-07-17 : Rapidement, la proposition 2 me semble plus intuitive.
- CK01 2022-07-17 : Le langage de spécification de structures (proche des types abstraits ?) permettrait-il de générer automatiquement la structure du projet, les signatures et une partie des tests structurels ?
- LL01 2022-07-17 : Le langage de spécification est relatif à l'essence (la spécification) du module, pas à sa structure (expression, mise en oeuvre). Il pourrait effectivement être issu de la théorie des TAA et permettre la génération automatisée (d'une partie) des tests unitaires. Par contre, la structure est décrite à l'aide d'un langage de programmation (opératoire, fonctionnel ou axiomatique). Il est en général plus difficile de générer automatiquement les tests dans ce cas.
  - Si le langage est axiomatique, l'intérêt des tests est limité : illustration en regard de certains cas singuliers (valeurs remarquables, dont les valeurs limites), couverture à l'aide de tests combinatoires, etc.
  - Si le langage est opératoire, on cherchera en plus à vérifier la «justesse» et la mise en oeuvre des algorithmes et des structures de données.
  - Si le langage est fonctionnel, l'importance et la complexité des tests de «justesse» s'en trouveront réduite.

5. Proposition 3

5.1. Description de la proposition 3

Supposons que le créateur et propriétaire (owner) de la base de données (des schémas qu'elle contient et toutes les entités qui y sont définies) soit l'usager (role) Proprio.

En particulier, pour un module *M*, Proprio est propriétaire du schéma associé (`M`) et de tous les éléments définis dans les parties `M` et `M<X>`, qu'elles soient privées ou publiques (`M_pub`, `M_pri`, `M_<X>_pub`, `M_<X>_pri`). À ce titre, Proprio a donc les droits d'accès à toutes les parties (publiques et privées, d'en utiliser les types, d'en exécuter les routines, d'en évaluer les relations et de les modifier.

On désire toutefois accorder à d'autres usagers certains droits d'accès aux éléments publics du module de la façon suivante :

- Les titulaires du rôle **M** peuvent accéder et utiliser tous les éléments de la base commune publique du module *M* (définie dans la partie `M_pub`).
- Les titulaires du rôle **M_<X>** peuvent accéder à tous les éléments de la partie publique de **M_<X>** (définie dans la partie `M_<X>_pub`) et les utiliser.

Rappels

- Il peut y avoir autant de gestion des politiques d'accès X qu'on veut.
- La partie privée associée à une partie publique peut être vide.
- En PostgreSQL, les concepts d'usager et de groupe d'usagers ont été réunis en un seul concept, le rôle. Un usager est simplement un rôle qui possède le droit de connexion.
- Un rôle peut hériter ou se voir accorder les droits d'un autre rôle — le présent document (et donc la présente proposition) est muet quant à la façon de gérer les rôles eux-mêmes. Nous montrons simplement comment les définir dans un contexte de gestion des droits d'accès associés aux modules.
- Pour que les routines des parties publiques `M_pub` et `M_<X>_pub` puissent accéder aux entités définies dans les parties privées lors de leur exécution, l'appelant doit avoir les droits d'accès aux éléments privés requis (ce que nous désirons éviter).
- Si la routine possède la propriété «security definer», plutôt que l'appelant, c'est le propriétaire de la routine qui doit avoir ces droits pour que la routine puisse accéder aux éléments requis.

Définition de la structure

Chaque partie `<P>` (`M` ou `M_<X>`) pourrait se voir associer un fichier `<P>_gpa` devant comprendre les actions suivantes :

- Accorder au rôle **<P>** le droit d'utilisation (usage) de tous les types et tous les domaines définis dans `<P>_pub`.
- Accorder au rôle **<P>** le droit d'exécution (execute) de toutes les routines définies `<P>_pub`.

Il faudra cependant prévoir deux fichiers, un qui est applicable au SGBD dans son ensemble (`<P>_spa`) et l'autre à la seule BD ciblée (`<P>_spa`).

Finalement, nous recommandons de placer l'ensemble des instructions de création du schéma, les notes de mises en oeuvre et les cartouches du module *M* dans un fichier initial spécifique `M_def`. Ceci permettra de l'omettre ou d'y substituer un autre fichier initial selon le contexte (par exemple, si un déploiement doit se faire sans schéma, avec préfixation, etc.).

Gestion des rôles

Les instructions de création des rôles eux-mêmes (**M** et **M_<X>**) sont regroupées dans un script distinct (`M_roles`), car ils ne devront être créés que lors de l'incorporation du module pour la première fois dans le SGBD. Lorsque d'autres BD recourront au module M1, il **ne faut pas** les recréer. Ce script peut également être l'occasion d'ajouter les droits spécifiques du module en regard d'une politique `<X>` à un rôle (`PA_<X>`) regroupant tous les droits conformes à cette politique.

De même, les rôles associés aux politiques d'accès elles-mêmes ne doivent être créés qu'une fois, pour l'ensemble des modules y ayant recours. Typiquement, un script pourra être mis à disposition à cet effet. Il s'agit toutefois d'une question de gestion des rôles et non de structuration des modules.

Notes

- Nous ne sommes pas arrivés à cantonner la gestion de la politique d'accès aux seuls fichiers `_gpa` (même en les divisant en `_spa` et `_bpa`). En effet, dans les fichiers `_pri`, il faut ajouter, lors de la redéfinition des routines, la propriété «security definer».
- La propriété «security definer» pourrait être soumise à un mécanisme de contrôle de variante qui pourrait la retirer lorsqu'aucune gestion des droits d'accès n'est requise.
- Dans l'ancienne solution, nous devions définir 2*(n+1) schémas, autant de rôles et autant de parties.
- Dans la présente solution, il faut un seul schéma, n+1 rôles et 3*(n+1)+2 parties. Par contre, si aucune GPA n'est requise, il suffit de trois parties, un rôle et un seul schéma. Si la seule politique applicable est le partage public-privé, il suffit de cinq parties, deux rôles et un seul schéma. Finalement, s'il s'agit d'un module de fonctions sans GPA ni partie privée (ou, plus exactement, avec une partie publique comprenant toute la mise en oeuvre), deux parties seront suffisantes et aucun rôle.

5.2. Analyse de la proposition 3

Exercice

Compléter l'analyse de la propostion 3

Conclusion

À compléter

- confirmer choix de la solution
- baliser l'expérimentation subséquente

Glossaire

BDC

```
base de données courante (avec laquelle une connexion est couramment établie).
```

feature

```
définition
  désigne tout élément pouvant être défini au sein d'un module
    * type (type de base, sous-type, domaine)
    * valeur (constante)
    * variable (relation, table, vue)
    * routine (opérateur, fonction, procédure, automatisme)
    * role (utilisateur, groupe, profil)
    * droit (permission, privilège)
traduction
  fr
    * objet (au sens propre, en conflit avec le sens OO)
    * entité (au sens propre, en conflit avec les sens EA et ontologique)
    * élément (en conflit avec la théorie des ensembles)
    * chose (trop vague)
    * fonctionnalité (trop restrictif)
    * article (trop de connotations diverses)
    *  ? ? ? ? ?
```

GPA

```
gestion des politiques d'accès.
```

PA

```
politique d'accès.
```

Produit le 2025-11-19 08:22:47 -0500

Collection de logiciels libres pour l'enseignement et le développement