Gestion de variantes de ressources textuelles

Xavier Crégut <nom@n7.fr>

Mots-clés : Variantes de ressources textuelles, gestion des variantes, évolutions, cohérence.

Dans l'activité d'enseignement, il est fréquent d'utiliser les mêmes exercices dans des contextes différents ou avec des objectifs différents. On aboutit donc à plusieurs versions de l'énoncé et du code correspondant.

Par exemple, en programmation objet, on peut avoir une classe Point indépendante qui ensuite sera une sous-classe de la classe abstraite ObjetGeometrique. Cette classe pourrait fournir ou pas les coordonnées polaires. On peut prévoir des traces dans les constructeurs et le destructeur pour visualiser la création et la destruction de ses objets. Dans certains exercices, on ajoute un méthode dessiner pour visualiser le point sur un écran graphique, etc.

Gérer ces différentes variantes de manière indépendantes n'est pas satisfaisant en particulier quand on veut faire une modification qui doit être répercutée sur toutes les variantes. Par exemple, la correction d'un commentaire de documentation.

Une autre solution est d'avoir une seule version qui inclut les différentes variantes délimitées par des marqueurs. On peut alors demander à extraire la variante (par exemple, code initial donné au début du TP et solution proposée après le TP). La correction du commentaire de documentation se fait alors sur la seule version maître et est récupérée par toutes les variantes.

Dans une telle approche, plusieurs problèmes se posent :

- Conserver une syntaxe facile à utiliser pour l'auteur du code et des exercices.
- Gérer les marqueurs lors de la définition de la version maître (signification, compatibilité entre marqueurs, relations entre marqueurs : implication, exclusion...) et lors de leur sélection pour définir une variante.
- Gérer les variantes (ensembles de marqueurs cohérents) avec possibilité de partager des marqueurs entre différentes variantes (factorisation). Par exemple, pour un ensemble de variantes données, on veut pouvoir les décliner en version générique ou en version non générique.
- Faciliter l'évolution dans la version maître en identifiant les impacts d'une modification, en vérifiant automatiquement (pour le code), qu'il n'y a pas d'erreur de compilation et que les tests passent toujours
- Limiter la redondance et donc gérer des morceaux de texte qui peuvent être placés à des endroits différents suivant les variantes (et potentiellement avec des niveaux d'indentation différents).
- Prévoir des mécanismes pour gérer des variantes pour lesquels les marqueurs ne se prêtent pas bien. Par exemple, pour gérer la généricité en Java, mieux vaut utiliser [[A, B]] pour déclarer des paramètres de généricité qui seront remplacés par <A, B> ou supprimés suivant que l'on veut la variante avec généricité ou non.
- Migrer les ressources existantes vers la nouvelle solution.
- Proposer des paramétrisations de l'éditeur de texte vim/gvim pour gérer les variantes (syntaxe, folding, etc.).

La solution actuellement utilisée s'appuient sur un système simple de marqueurs, l'utilisation de scripts utilisant la commande sed pour sélectionner des variantes, la définition de variante via une

sélection manuelle de marqueurs dans des Makefile sans vérification a priori de leur cohérence, etc. Au total une solution assez lourde et très fragile. La partie sed a été remplacée par un programme en Python minimaliste.

L'objectif du projet est donc d'apporter une solution aux problèmes ci-dessus. Parmi quelques pistes qui pourront être explorées, citons les préprocesseurs (m4, etc.), langages de template (FreeMarker, mako, jinja2, etc.), lignes de produits, outils de résolution de contraintes, etc.

Un premier projet long a permis de définir les bases de ce système. Un stage de La Prépa des INP a ajouté une interface graphique qui permet de voir la variante en fonction des marqueurs activés.

Les contraintes à respecter impérativement sont :

— avoir une solution légère pour gérer les variantes dans le document maître
— développer des programmes avec interface en ligne de commande pour permettre l'automatisation via make (l'interface graphique ou web n'est pas une priorité même si elle peut être utile pour montrer l'outil et faciliter son adoption)
— développer le code sous une licence libre

Pour la partie « état de l'art » il sera certainement intéressant de regarder :

— les préprocesseurs type cpp, m4, etc.
— les langages de template tels FreeMarker, mako, jinja2 etc.
— la notion de ligne de produits
— les langages de contraintes