INF356 - Projet de programmation 3: feuille 2

Méthodes secondaires et Flots

Exercice 2.1 *Classes mixin, mémoïsation*
Reprendre la feuille 10 (implémentation des polygones). Faire l'exercice 10.4.

Exercice 2.2

1. Écrire une classe `personne` avec un créneau `date-naissance` initialisé par
   `:initarg :date-naissance`
   et ayant un `reader date-naissance`. La date de naissance devrait être un entier acceptable par `decode-universal-time`. Créer une instance de cette classe, et vérifier la bonne initialisation du créneau (en utilisant par exemple la fonction `describe`). On imagine alors que cette classe va être utilisée dans une application écrite par d'autres personnes, laquelle s'en sert de la même manière, à savoir
   `(make-instance 'personne :date-naissance ...)`
2. Rajouter à la classe `personne`, un créneau `age` avec un accesseur `age`. Définir une méthode `:after` sur la fonction générique `initialize-instance` qui initialise ce créneau (grâce à l'accesseur) à partir du résultat de la fonction générique `date-naissance`. Vérifier qu'il est possible de créer des instances de la classe comme précédemment et vérifier encore une fois la bonne initialisation des créneaux.
3. On souhaite éliminer le créneau `date-naissance` de la classe `personne`. Mais, pour que ce changement soit transparent pour les utilisateurs extérieurs, il faut qu'ils puissent pouvoir continuer à écrire

   `(make-instance 'personne :date-naissance ...)`

   même si le créneau a disparu.

   Modifier la classe `personne` et la méthode `:after` sur `initialize-instance` de manière à éliminer le créneau `date-naissance` (mais il faut toujours pouvoir utiliser la fonction générique correspondante). Pour cela, initialiser le créneau `age` directement à partir de l'initarg `:date-naissance` dans la méthode `:after` de la fonction `initialize-instance`. Calculer la date de naissance de la personne à partir son l'âge et de la date du jour. Vérifier que la création d'instances se fait comme précédemment.

Exercice 2.3
On imagine une arborescence de classes dont les instances doivent être transmises de manière efficace dans un canal de communication. C'est peut-être le cas d'un jeu pour téléphones mobiles pour lequel la bande passante est relativement limitée et où le coût de la communication est relativement élevé.

Comme exemple, on imagine les classes suivantes :

`participant`, un objet participant du jeu. Cette classe contient un seul créneau, le *nom* du participant sous la forme d'une chaîne de caractères.

`personnage`, une sous-classe de `participant`, contentant un créneau supplémentaire, l'âge du personnage, contenant un entier entre 0 et 255.

`heros`, une sous-classe de `personnage` contenant un créneau `force` contenant un entier entre 0 et 255.

Sans se préoccuper du problème général de la transmission de ces instances par un canal de communication, on va considérer le calcul de la *taille* de celles-ci, à savoir combien d'octets sont nécessaires pour la transmission.

Pour simplifier la question, nous allons considérer qu'une chaîne de caractères nécessite autant d'octets que sa taille, et qu'un entier entre 0 et 255 nécessite un octet. Transmettre une instance de la classe `heros`, nécessite donc un nombre d'octets déterminé par la taille de son nom, plus un octet pour l'âge et un octet pour la force.

Au nom de la modularité, on considère que chaque classe figure dans un module séparé. Dans le module où est définie la classe `heros`, on n'a pas d'information concernant la taille des deux autres classes.

1. Écrire une fonction générique `taille` et une méthode spécialisée pour chacune des trois classes ci-dessus. Afin de récupérer la taille de la super-classe, employer `call-next-method` (sauf dans le cas de la classe `participant` où il n'y a pas de méthode plus générale applicable.
2. Lire la documentation de `next-method-p` dans la HyperSpec. Modifier les implémentations de méthodes de `taille` pour éliminer la dépendance sur l'existence ou non de méthodes plus générales.

   (avant de continuer, faire : `(fmakunbound 'taille)`. Expliquer pourquoi ?)
3. Modifier l'approche pour que la fonction générique `taille` utilise la combinaison de méthodes `+`. Comparer les approches.

Exercice 2.4 *Lecture de fichiers textes*

Utiliser la fonction `open` pour ouvrir un fichier Lisp existant et stocker le flot qui en résulte dans une variable spéciale. Essayer les fonctions `stream-external-format`, `read-char`, `peek-char`, `unread-char`, `read` et `read-line` avec ce flot. Expliquer le résultat. Fermer le fichier.

Répéter l'exercice avec le fichier `/net/cremi/rostrand/iso-latin-1.text` (en format `ISO-latin-1`, argument `:latin1` pour le paramètre mot-clé `:external-format`). En cas de problème, choisir le restart `attempt-resync`. Expliquer le résultat.

Changer le format externe dans l'expression d'ouverture du fichier, et donner plutôt comme valeur `:utf-8`, et répéter l'exercice. Expliquer le résultat.

Exercice 2.5 *Conversion de format de fichier*

Écrire une fonction Lisp qui crée une copie d'un fichier texte, mais en changeant le format de `ISO-latin-1` à `UTF-8`. Appliquer la fonction au fichier ci-dessus et créer une version `UTF-8` dans votre répertoire, appelée par exemple `utf-8.text`.

Répéter l'exercice précédent sur ce nouveau fichier, en indiquant le format `:latin-1` à `open`. Expliquer le résultat.

Exercice 2.6 *Lecture de fichiers binaires*
Écrire une fonction Lisp qui crée une copie d'un fichier binaire arbitraire (suite d'octets). Tester votre fonction sur les fichiers `iso-latin-1.text` et `utf-8.text`. Vérifier grâce à vos outils Unix habituels (lesquels ?) que le fichier original et la copie sont identiques.