IFT2255 - Génie Logiciel
TP3

Date de remise : 7 janvier 2012

Objectifs

Le travail à réaliser dans ce TP vise à l'ajout de nouvelles fonctionnalités à Violet : la validation de diagrammes de classes et la génération de code Java à partir de modèles validés.

Conditions de réalisation

Le travail est à effectuer individuellement ou en groupe de deux personnes.

Obtention du code source

Une copie de la version la plus récente de Violet a été placée dans le dépôt spécifique à votre équipe sur github. Les instructions pour accéder à ce dépôt sont disponibles sur la page web du cours. Notez que chaque dépôt n'est accessible qu'aux membres de l'équipe ainsi qu'aux instructeurs du cours. V*ous devez obligatoirement utiliser et modifier cette version de Violet.* Vos modifications devront aussi être publiées sur votre dépôt github afin de permettre la correction.

Vous devrez créer une nouvelle branche `tp3` à partir de la branche `menuExporter` (créée pour le TP2) afin d'effectuer vos modifications.

Travail à réaliser

Vous devez ajouter les fonctionnalités suivantes à Violet :

1. Valider un diagramme de classes UML pour s'assurer qu'il ne contient pas d'incohérence.
2. Générer du code Java à partir d'un diagramme de classes.

Élément déclencheur

Les nouvelles fonctionnalités seront disponibles à partir du menu ajouté lors de la réalisation du TP2. Lorsqu'un utilisateur sélectionne l'option "Exporter le code Java", le diagramme en cours sera validé, et le code sera généré si le diagramme est valide. Si le diagramme courant n'est pas un diagramme de classes, un message devra être présenté à l'utilisateur afin de l'informer que seul un diagramme de classes peut servir à la génération de code.

Validation du diagramme

Avant d'exporter le code, le diagramme devra être validé pour s'assurer que du code Java correct pourra être généré. Vous devez vérifier les propriétés suivantes :

- Tous les identifiants (noms de classes, noms d'attributs, noms de méthodes) doivent être conformes aux règles de Java. Un identifiant valide contient au moins un caractère. Le premier caractère doit être une lettre, le caractère souligné (_) ou le symbole de dollar ($). Les caractères suivants peuvent en plus être des chiffres (0-9). Tout autre caractère est invalide. Certains mots réservés ne sont pas des identifiants valides :

  ```
  abstract assert boolean break byte case catch char class const
  continue default do double else enum extends final finally float for
  goto if implements import instanceof int interface long native new
  package private protected public return short static strictfp super
  switch synchronized this throw throws transient try void volatile
  while
  ```

- Chaque attribut ou méthode doit avoir une visibilité spécifiée (`+`, `-`, `#`, ou `~`).
- Chaque type spécifié doit être valide. Un type peut être :
  - Un type primitif : `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, ou `short`
  - Un type d'objet: un identifiant valide (ex: `String`), ou une séquence d'identifiants valides séparés par des points (ex: `java.lang.String`)
- Chaque attribut doit posséder un type valide, introduit après le nom de l'attribut par le caractère ':'.
- Chaque méthode doit avoir une signature valide : un nom formé d'un identifiant valide, une liste de paramètres (possiblement vide) et une valeur de retour. Les paramètres apparaissent entre parenthèses. Chaque paramètre est spécifié avec un identifiant valide suivi d'un type valide, séparés par le caractère ':'. Si plus d'un paramètre est présent, les paramètres sont séparés par des virgules. Le type de retour de la méthode doit aussi être spécifié et être un type valide. Le type de retour d'une méthode peut aussi prendre la valeur spéciale `void`, indiquant qu'une méthode ne retourne pas de résultat. Par exemple, ces signatures sont toutes valides :
  - `sort():void`
  - `pow(n:double):double`
  - `compare(o1:Object, o2:Object):int`
- Toutes les associations, les agrégations et les compositions doivent être étiquetées (propriété "Middle label") et avoir une cardinalité valide (propriété "End label") : un entier positif, le caractère '*', ou une plage de valeurs (ex: 0..*). Les étiquettes de toutes les associations, agrégations et compositions à partir d'une même classe doivent être distinctes.
- La relation d'héritage (sans inclure les interfaces) ne doit pas contenir de cycle.
- Les interfaces et les paquetages ne sont pas supportés, et ne doivent pas apparaître dans le diagramme. Les notes, si elles sont présentes, devront simplement être ignorées. Les types qui ne sont pas mentionnés plus haut devront aussi être ignorés.

Suite à la validation, toutes les classes et tous les arcs qui contiennent des problèmes devront être identifiés en changeant la couleur de leur bordure à rouge. Aucun code ne sera alors généré. Lors de la prochaine tentative de génération de code, la couleur de chacune des entités devra être rétablie avant de procéder à la validation.

Les espaces dans les déclarations de méthodes et d'attributs ne sont pas significatifs, et doivent être ignorés. Par exemple, les déclarations d'attributs suivantes sont équivalentes:

```
+size:int
+ size : int
```

Similairement, les déclarations de méthodes suivantes sont toutes équivalentes :

```
+compare(o1:Object,o2:Object):int
+ compare(o1:Object, o2:Object):int
+ compare(o1 : Object, o2 : Object) : int
+  compare ( o1 : Object , o2 : Object ) : int
```

Génération de code

Une fois le diagramme validé, l'utilisateur devra choisir un dossier pour la génération de code (à l'aide d'une boîte de dialogue). Une fois le dossier choisi, un fichier Java sera généré pour chacune des classes représentées. Chaque fichier devra contenir tous les attributs et les méthodes spécifiés dans le diagramme de classes. La structure d'héritage entre les classes doit correspondre au diagramme.

Pour chacun des attributs, la visibilité, le nom ainsi que le type devront être spécifiés correctement. Les associations, agrégations et compositions devront aussi générer des attributs. Il n'est pas nécessaire d'initialiser les attributs. Le type de l'attribut généré sera déterminé par sa cardinalité dans le cas d'une association, agrégation ou composition. Pour un attribut de type *T*, si la cardinalité est 1 ou 0..1, le type de l'attribut généré sera aussi *T*. Sinon, le type de l'attribut généré sera `Collection<`*T*`>`. Par souci de simplicité, vous pouvez traiter tous les agrégations et compositions comme des relations unidirectionnelles.

Chaque méthode spécifiée dans le diagramme devra apparaître dans le code. Le code généré doit compiler correctement (en supposant que les types spécifiés sont tous disponibles). Il est donc nécessaire de fournir une implémentation minimale pour les méthodes qui définissent un type de retour autre que `void`. Ces méthodes devront retourner une valeur prédéfinie (ex: `0` ou `null`) selon le type déclaré de la valeur de retour.

Si des imports de classes sont nécessaires pour permettre au code d'être compilé, ils doivent être présents dans le code généré. Par souci de simplicité, vous pouvez supposer que tous les types spécifiés dans le diagramme et qui possèdent un paquetage devront être importés, et que les types sans paquetage sont directement accessibles (par exemple, un diagramme de classes devra utiliser `java.util.Collection` et non `Collection`). Vous n'avez pas à vérifier que les types sont bien accessibles (il est donc possible de générer du code qui ne peut pas être compilé si le diagramme de classes contient des types inaccessibles).

Le code généré doit être facilement lisible et modifiable par un développeur. Il est donc important de générer du code qui est indenté proprement.

Tests JUnit

Vous devez utiliser JUnit pour tester votre implémentation. La qualité des tests remis fait partie de l'évaluation du travail. Les tests devront être ajoutés au code existant et faire partie de la remise. Le code suivant permet d'initialiser les composants nécessaires à la création d'un diagramme de classes dans Violet :

```
@BeforeClass
public static void setupOnce() {
    IUserPreferencesDao userPreferencesDao = new DefaultUserPreferencesDao();
     BeanFactory.getFactory().register(IUserPreferencesDao.class,
                                       userPreferencesDao);

    ThemeManager themeManager = new ThemeManager();
    ITheme theme1 = new ClassicMetalTheme();
    ITheme theme2 = new VistaBlueTheme();
    ITheme theme3 = new DarkAmbianceTheme();
    List<ITheme> themeList = new ArrayList<ITheme>();
    themeList.add(theme1);
    themeList.add(theme2);
    themeList.add(theme3);
    themeManager.setInstalledThemes(themeList);
    themeManager.applyPreferedTheme();
    BeanFactory.getFactory().register(ThemeManager.class, themeManager);
}
```

Points bonis

Pour des points bonis (jusqu'à 10% de la note maximale), vous pouvez :

- supporter les interfaces, incluant détection de cycles (4%)
- supporter les paquetages (3%)
- supporter l'initialisation de tous les attributs générés à partir d'une agrégation ou d'une composition à l'aide d'un constructeur unique qui prend en paramètre une valeur pour chaque attribut à initialiser (3%)

Aide et discussions

Vous êtes encouragés à discuter du projet et à poser vos questions en utilisant le forum créé à cette fin sur StudiUM.

Conseils pour la réalisation du travail

- Les expressions régulières peuvent être utiles pour la validation des différents composants du diagramme. La classe `java.util.regex.Pattern` de la bibliothèque standard de Java est un bon point de départ.
- Assurez-vous de tester votre implémentation avec des diagrammes valides ainsi qu'avec des diagrammes invalides.
- Développez les fonctionnalités de façon incrémentale. N'hésitez à effectuer plusieurs "commits" lors du développement, seule la dernière version sera utilisée pour la correction.

Remise

Le travail doit être remis au plus tard lundi 7 janvier **à minuit**. Le code ne sera pas remis par StudiUM, la correction sera effectuée directement à partir du dépôt github de chaque équipe. Vous devez vous assurer que la branche `tp3` est publiée sur github avant la date limite de la remise.

Barème

Ce travail compte pour 15% de la note finale, divisé comme suit :

6 pts	Validation du code
6 pts	Génération de code
2 pts	Tests
1 pt	Qualité du code Java remis (organisation, lisibilité, commentaires, etc.)