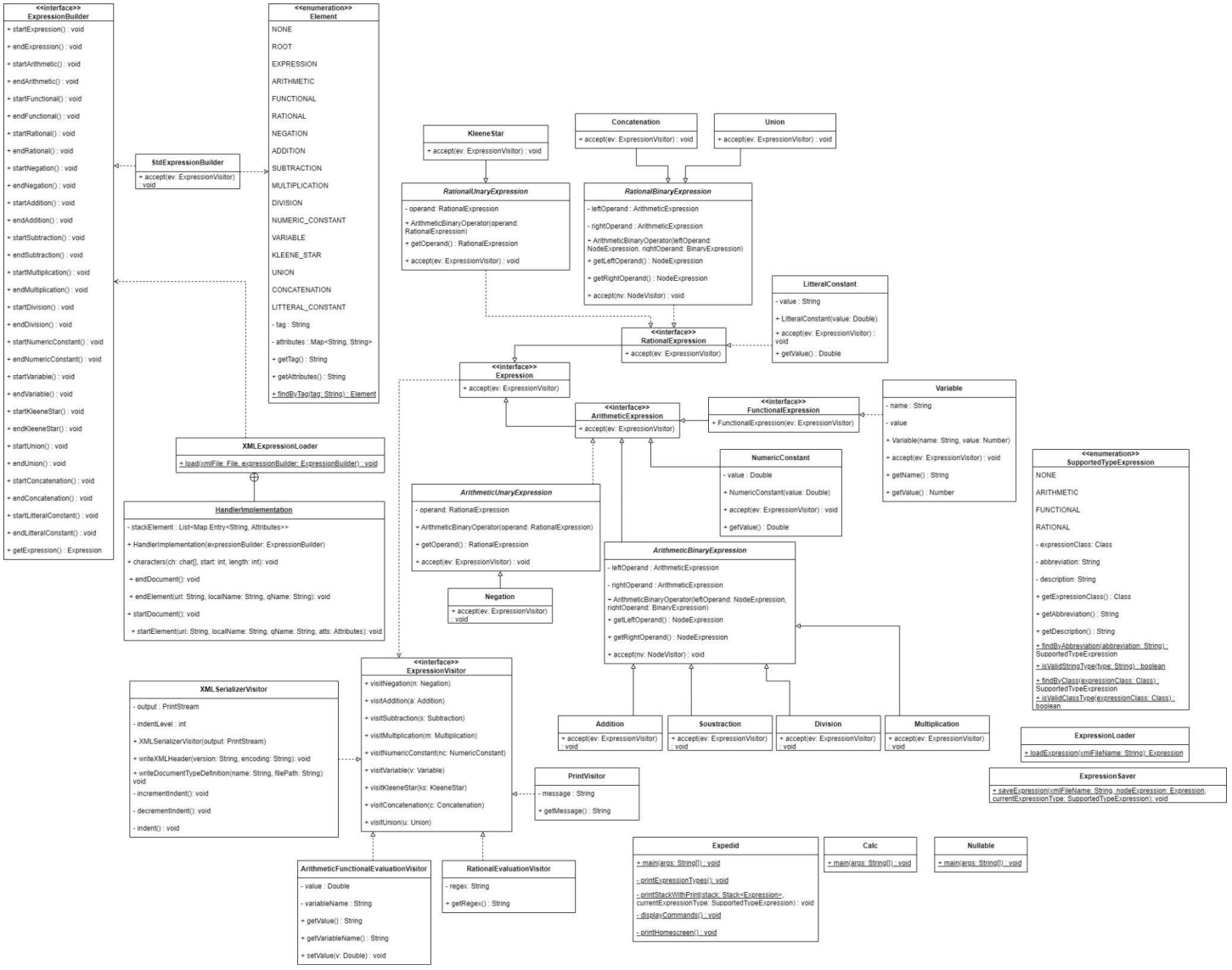
Rapport de projet - Symbolib



Sommaire

Introduction

Afin de mettre en pratique nos compétences en architecture logicielle, nous avons réalisé un projet concernant une bibliothèque de manipulation d'expressions symboliques.

Les patrons de conceptions utilisées

Le patron de conception : Visiteur

Le patron Visitor est un patron de conception comportemental qui permet de séparer l'algorithme de traitement d'un objet de sa structure. En utilisant ce patron, j'ai pu ajouter de nouvelles fonctionnalités à mon projet sans modifier la structure des classes d'expression existantes.

Pour mettre en place le patron Visitor, j'ai commencé par créer une interface appelée "ExpressionVisitor" qui définit les méthodes de visite pour chaque type d'expression que je souhaite traiter. Cette interface contient des méthodes de visite pour les expressions arithmétiques telles que la négation, l'addition, la soustraction, la multiplication et la division, ainsi que pour les expressions rationnelles telles que la concaténation, l'étoile de Kleene et l'union.

Ensuite, j'ai implémenté une classe concrète appelée "CalculationVisitor" qui implémente l'interface "ExpressionVisitor". Cette classe est responsable du calcul d'une expression. Elle contient des méthodes de visite pour chaque type d'expression, et elle effectue les calculs appropriés en fonction du type d'expression visité. Par exemple, pour l'expression de négation, la méthode "visitNegation(Negation n)" effectue le calcul de la négation en appelant la méthode "accept(this)" sur l'opérande de la négation, puis en stockant le résultat dans la variable "value" de la classe "CalculationVisitor".

J'ai également mis en place une autre classe concrète appelée "XMLSerializerVisitor" qui implémente également l'interface "ExpressionVisitor". Cette classe est responsable de la sérialisation des expressions visitées en format XML. Elle utilise une instance de la classe "PrintStream" pour écrire les balises XML dans un fichier de sortie spécifié. La classe "XMLSerializerVisitor" utilise également un système d'indentation pour formater le fichier XML de manière lisible.

Et enfin une classe concrète nommée “PrintVisitor” qui implémente aussi l'interface "ExpressionVisitor". Elle est responsable de l’affichage d’une expression dans le terminal.

Pour utiliser le patron Visitor :

Une fois que j'ai mis en place les classes "ExpressionVisitor", "CalculationVisitor", "PrintVisitor", "XMLSerializerVisitor", j'ai pu les utiliser pour implémenter les fonctionnalités spécifiques que je souhaitais. Par exemple, pour effectuer des calculs sur les expressions arithmétiques, j'ai créé une instance de la

classe "CalculationVisitor" et j'ai appelé la méthode "accept(this)" sur l'expression arithmétique que je voulais traiter. La méthode de visite appropriée dans la classe "CalculationVisitor" a été appelée en fonction du type d'expression visité, et le calcul approprié a été effectué.

Pour sérialiser les expressions en format XML, j'ai créé une instance de la classe "XMLSerializerVisitor" et j'ai appelé la méthode "accept(this)" sur l'expression que je voulais sérialiser. La méthode de visite appropriée dans la classe "XMLSerializerVisitor" a été appelée en fonction du type d'expression visité, et les balises XML appropriées ont été écrites dans le fichier de sortie spécifié.

De même que pour "PrintVisitor" qui reprend le même concept.

Le patron de conception : Builder

Le Builder est un patron de conception de création qui permet de construire des objets complexes étape par étape en séparant la construction de l'objet final de sa représentation. Dans mon cas, j'ai utilisé le Builder pour construire des expressions mathématiques composées d'opérations arithmétiques, fonctionnelles et rationnelles, ainsi que de constantes numériques, de variables et de constantes littérales.

Tout d'abord, j'ai créé une interface ExpressionBuilder qui définit les méthodes nécessaires pour construire les différentes parties d'une expression, telles que les opérations arithmétiques, fonctionnelles et rationnelles, ainsi que les constantes numériques, les variables et les constantes littérales. Ces méthodes sont appelées par le client pour définir les différentes parties de l'expression.

Ensuite, j'ai implémenté cette interface dans ma classe StdExpressionBuilder, qui est la classe concrète du Builder. Cette classe utilise une approche basée sur la pile pour construire l'expression de manière modulaire. Elle utilise également une map pour associer les symboles des opérations aux classes correspondantes d'expressions mathématiques. Par exemple, j'ai associé le symbole de négation "-" à la classe Negation, le symbole d'addition "+" à la classe Addition, etc.

J'ai utilisé des méthodes start*() et end*() dans ma classe StdExpressionBuilder pour définir le début et la fin de chaque partie de l'expression. Par exemple, j'ai utilisé les méthodes startArithmetic() et endArithmetic() pour définir le début et la fin d'une opération arithmétique, les méthodes startFunctional() et endFunctional() pour définir le début et la fin d'une opération fonctionnelle, etc. Entre ces méthodes, j'ai utilisé les méthodes spécifiques aux opérations pour construire les parties de l'expression en utilisant la pile et les classes correspondantes d'expressions.

J'ai également utilisé les méthodes startExpression() et endExpression() pour définir le début et la fin de l'expression dans son ensemble.

Enfin, j'ai utilisé la méthode getExpression() pour obtenir l'expression finale construite par le Builder. Cette méthode retourne l'objet d'expression construit, qui peut être utilisé par le client pour effectuer des opérations supplémentaires sur l'expression.

Les patrons envisagés

Le patron de conception : Composite

Je n’ai pas utilisé le patron Composite car je n'ai pas trouvé cela nécessaire pour représenter les différentes types d'expressions arithmétiques. Par exemple, l'addition peut contenir elle-même d'autres expressions arithmétiques en tant qu'opérandes gauche et droite dans son constructeur. Cela signifie qu'il n'est pas nécessaire de créer une hiérarchie d'objets composites et de feuilles pour représenter les opérations arithmétiques. Cependant, il est préférable de l'utiliser afin d’uniformiser le traitements des objets et cela pour chaque type d’expression (arithmétique, fonctionnelle et rationnelle), donc un composite arithmétique, fonctionnelle et rationnelle afin de distinguer les différentes expressions qui peuvent se contenir elle même.

Le patron de conception : Abstract Factory

Je n'ai pas exploité le patron de conception Abstract Factory, même si j'aurais aimé l'utiliser. En effet, en écrivant le rapport, j'ai réalisé que j'aurais pu utiliser ce patron pour créer une famille d'objets de manière cohérente et sans dépendre de classes concrètes. Par exemple, si j'avais eu besoin de créer des expressions arithmétiques de différentes natures (addition, soustraction, multiplication, etc.), j'aurais pu utiliser l'Abstract Factory pour créer une fabrique d'expressions arithmétiques qui produirait les instances appropriées en fonction de la nature demandée. Cela aurait permis de rendre le code plus modulaire et facilement extensible.

Le principe de base

SOLID

Maintenant, intéressons- nous au principe de base de l’architecture logicielle.
Single Responsibility Principle : Ce principe stipule qu'une classe ne doit avoir qu'une seule raison de changer. En d'autres termes, une classe ne doit être responsable que d'une seule fonctionnalité ou d'un seul aspect du système. Cela permet d'obtenir un code plus modulaire, maintenable et évolutif. Ce principe en veillant à ce que chaque classe ait une seule responsabilité spécifique,notamment, la classe LitteralConstant est responsable que de la manipulation des constantes littérales.
Open/Close Principle : Ce principe stipule que les classes doivent être ouvertes à l'extension mais fermées à la modification. Ce principe est respecté en utilisant l'héritage, le polymorphisme ou les interfaces pour permettre l'extension de fonctionnalités sans modifier les classes existantes. C’est le cas par exemple, avec

l’interface ArithmeticExpression, si il y a d’autres types d’expression arithmétique à ajouter.

Liskov Substitution Principle : Ce principe stipule que les objets d'une classe dérivée doivent pouvoir être substitués à des objets de la classe de base sans altérer la cohérence du programme. En d'autres termes, les classes dérivées doivent respecter le contrat défini par leur classe de base. Un exemple est que la classe Addition respecte le contrat de l’interface ArithmeticBinaryExpression.

Interface Segregation Principle : Ce principe stipule que les interfaces d'une classe doivent être spécifiques aux besoins des clients et ne doivent pas imposer d'implémentations inutiles. Cela évite la surcharge de méthodes inutiles pour les clients qui n'en ont pas besoin. On peut citer l’interface ArithmeticBinaryExpression qui sert à représenter que les expressions arithmétiques binaires..

Dependency Inversion Principle : Ce principe stipule que les modules d'un système doivent dépendre d'abstractions et non de détails. Cela permet de réduire les dépendances entre les classes et de rendre le code plus flexible et facile à maintenir.

Document Type Definition

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT symbolib (expression)>

<!ELEMENT expression (arith | func | rat)>

<!ELEMENT arith (negation | addition | subtraction | multiplication | division | numeric_constant)>
<!ELEMENT func (negation | addition | subtraction | multiplication | division | numeric_constant | variable)>
<!ELEMENT rat (kleene_star | union | concatenation | litteral_constant)>

<!ELEMENT negation (#PCDATA)>
<!ATTLIST negation symbol CDATA #IMPLIED>
<!ELEMENT addition (#PCDATA)>
<!ATTLIST addition symbol CDATA #IMPLIED>
<!ELEMENT subtraction (#PCDATA)>
<!ATTLIST subtraction symbol CDATA #IMPLIED>
<!ELEMENT multiplication (#PCDATA)>
<!ATTLIST multiplication symbol CDATA #IMPLIED>
<!ELEMENT division (#PCDATA)>
<!ATTLIST division symbol CDATA #IMPLIED>
<!ELEMENT numeric_constant (#PCDATA)>
<!ATTLIST numeric_constant value CDATA #IMPLIED>
<!ELEMENT variable EMPTY>
<!ATTLIST variable
        name CDATA #REQUIRED
        value CDATA #IMPLIED
>
<!ELEMENT kleene_star (#PCDATA)>
<!ATTLIST kleene_star symbol CDATA #IMPLIED>
<!ELEMENT union (#PCDATA)>
<!ATTLIST union symbol CDATA #IMPLIED>
<!ELEMENT concatenation (#PCDATA)>
<!ATTLIST concatenation symbol CDATA #IMPLIED>
<!ELEMENT litteral_constant (#PCDATA)>
<!ATTLIST litteral_constant value CDATA #IMPLIED>
```

Installation et exécution des programmes du projet

1. Ouvrez Eclipse et importez le projet Symbolib dans votre espace de travail.
2. Naviguez jusqu'aux classes `Expedid.java`, `Calc.java` et `Nullable.java` dans le dossier src du projet.
3. Cliquez avec le bouton droit sur l'une de ces classes et sélectionnez "Run As" > "Java Application" dans le menu contextuel.
4. Un des programmes sera alors exécuté et vous pourrez interagir avec celui-ci via la console.

Conclusion

Pour conclure, le module d'architecture logicielle m'a fourni les connaissances et les compétences nécessaires pour concevoir et mettre en œuvre des architectures logicielles efficaces et modulaires.
Je tiens à remercier M.Florent Nicart pour ses cours d'architecture logicielle et ses travaux pratiques, ainsi que envers M.Yannick Guesnet pour ses explications en travaux pratiques des concepts abordés en cours.
L'utilisation des patrons de conception Visitor et Builder dans mon projet de bibliothèque de manipulation d'expressions symboliques m'a permis d'améliorer la modularité, la maintenabilité et l'extensibilité de mon code tout en appliquant les principes de bases S.O.L.I.D, ce qui a été une expérience enrichissante et bénéfique pour ma formation en architecture logicielle.