INSA RENNES

Quatrième année - Informatique

Projet 4ème année INFO

Rapport de conception

Réalisé par :
Nour Romdhane
Liantsoa Rasata
Mathilde Leparquier
Ibrahim Benali
Othmane Kabir

Sous la direction de :
Jean-Louis Pazat

Table des matières

Résumé

Ce rapport décrit la phase de conception de notre projet, intitulé « SYNCJ », effectué dans le cadre de notre scolarité à l'INSA de Rennes.L'objectif de notre projet est de faciliter l'apprentissage des outils de synchronisation Java pour les étudiants en 4éme année au département informatique qui suivent le module de parallèlisme.

Dans un premier temps, nous rappelons le contexte de notre projet ainsi que la technique de réalisation.

Dans un second temps, nous présentons l'architecture logicielle interne puis nous détaillons les différentes stratégies permettant le développement de notre projet.

1 Introduction

Notre travail consiste à traduire un code java utilisant des conditions d'exécution en un code utilsant les outils de synchronisation Java. Pour se faire, nous allons mettre en œuvre les compteurs de synchronisation en Java, ces derniers n'étant pas nativement implémentés.

Pour l'implémentation de ce mécanisme de synchronisation, nous devons :

— déclarer les compteurs de synchronisation et les mettre à jour.
— utiliser à bon escient les outils de synchronisation Java.

Pour le développement, nous avons opté pour la technique de compilation.

1.1 Mécanisme des compteurs de synchronisation

Le processus d'exécution commence lors de l'invocation d'une méthode m qui est mise à jour via plusieurs compteurs qui peuvent être résumés par le tableau suivant :

Compteurs de synchronisation d'une méthode m	Explication
m.req	Nombre d'invocations de m
m.aut	Nombre d'autorisations d'accès à m
m.term	Nombre d'exécution terminée de m
m.att = m.aut – m.req	Nombre de threads en attente d'autorisation d'accès à m
m.act = m.term – m.aut	Nombre de threads en cours d'exécution de m

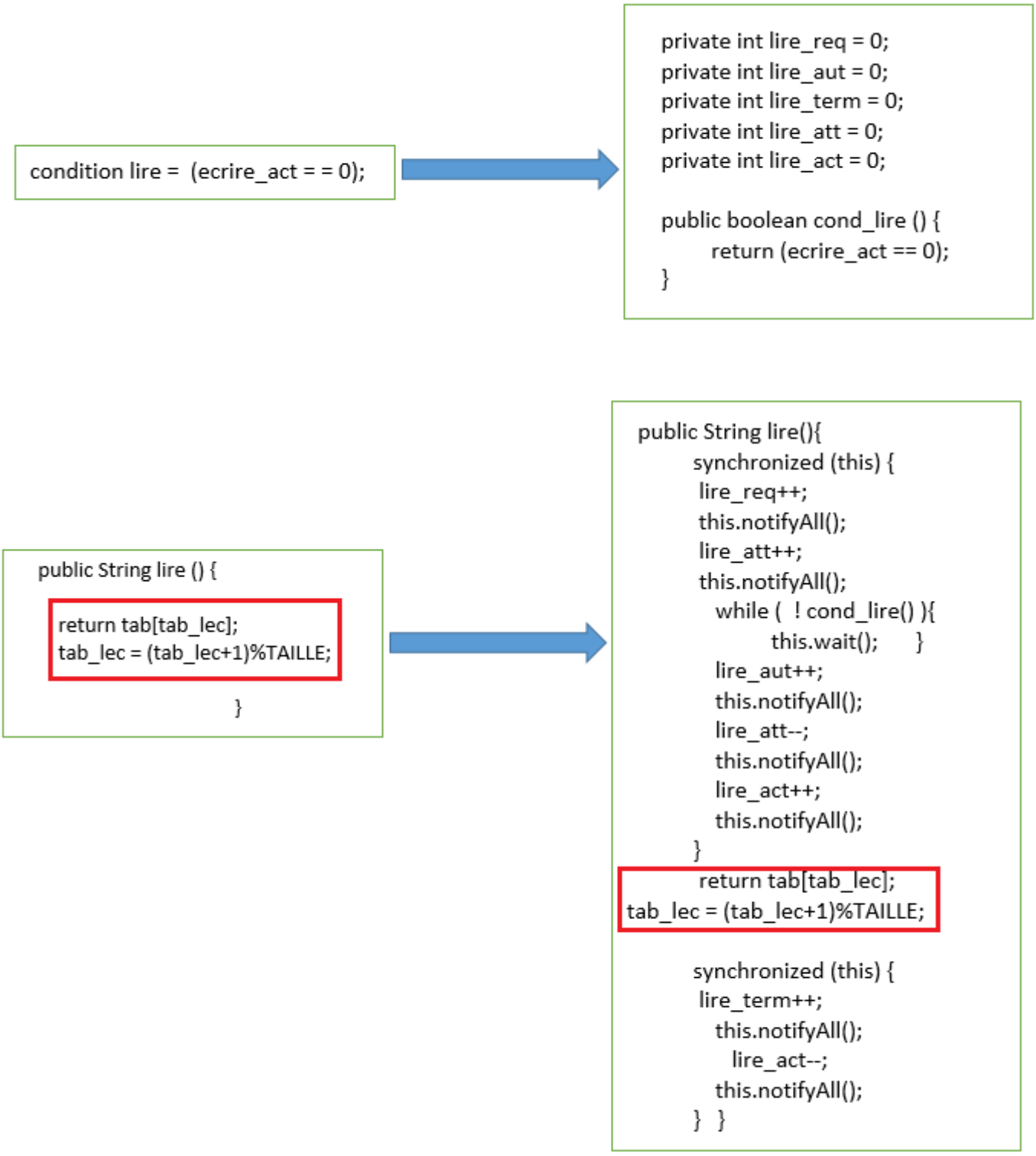
D'abord, à partir des noms des conditions déclarés par l' utilisateur, nous générons la déclaration des compteurs de synchronisation et les méthodes de test.

Ensuite, nous générons automatiquement les mises à jour des compteurs et l'appel aux primitives de synchronisation Java en fonction du nom de la méthode.

L'exemple suivant illustre ce mécanisme.

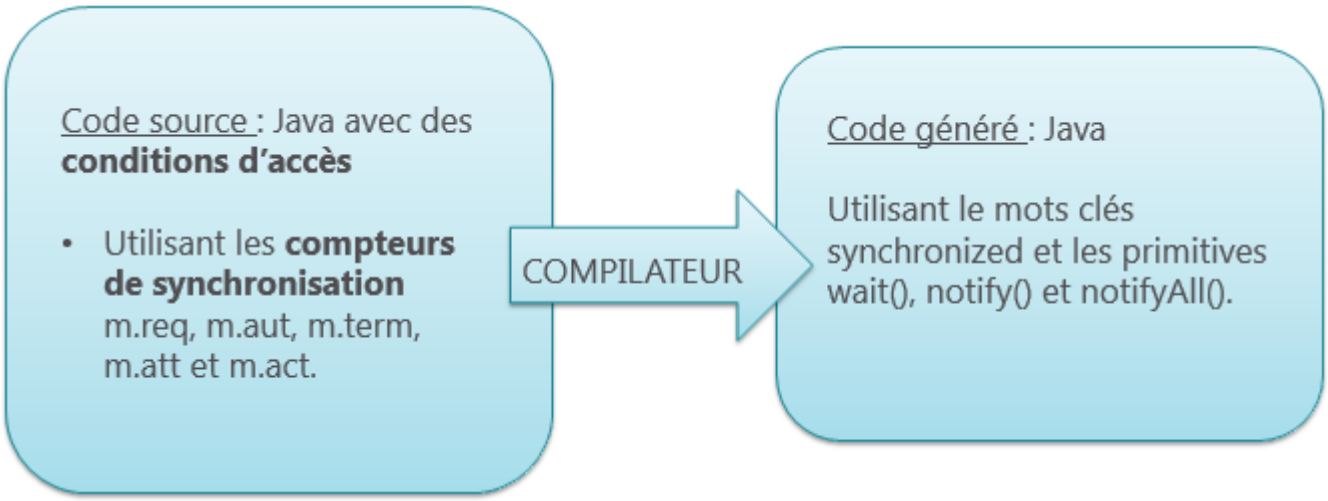
Exemple du lecteur/redacteur :

A gauche, le code tel que l'utilisateur le mettrait dans un fichier Java et à droite, le code généré en sortie après le traitement réalisé par notre compilateur.



Nous notons ici que nous avons abandonné la technique d'annotation tel que nous l'avons décrit dans le rapport de spécification pour ne garder uniquement que la technique de compilation. Nous avons jugé que l'intégration directe des conditions dans la grammaire était suffisante.

Pour résumer, le fonctionnement de notre compilateur peut s'expliquer par la figure ci-dessous :



2 Architecture logicielle interne

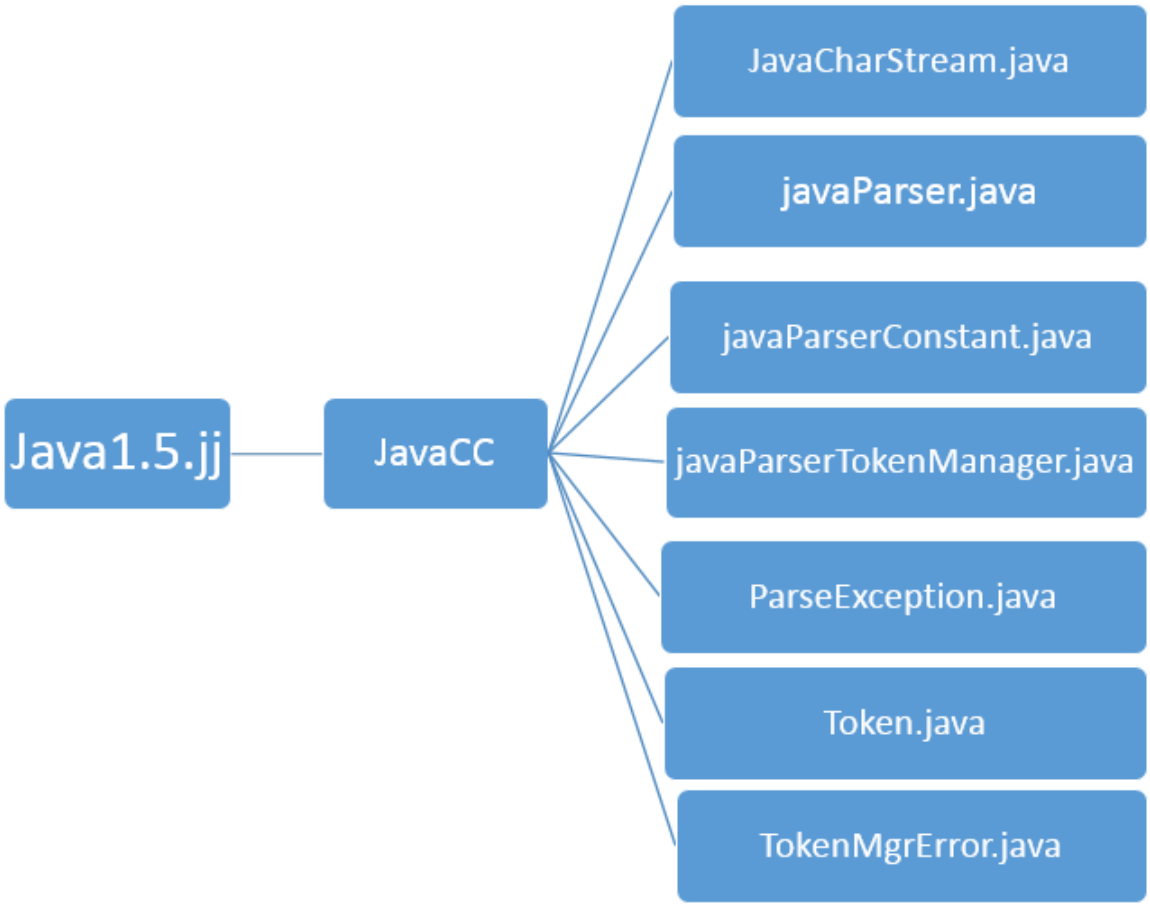
2.1 Le générateur d'analyseur syntaxique

2.1.1 JavaCC

Java Compiler Compiler (JavaCC) est le générateur d'analyseur syntaxique que nous avons choisi d'utiliser car il est adapté pour les applications Java. En plus du générateur d'analyseur lui-même, JavaCC fournit d'autres capacités standards liées à la génération de l'analyseur, comme la construction de l'arbre (via un outil appelé JJTree inclus avec JavaCC), les actions et le débogage.

JavaCC prend en entrée un fichier d'extension **«.jj»** qui contient entre autres les descriptions des règles de la grammaire et produit un parser descendant (dans ce fichier). Pour notre projet, nous travaillons avec le fichier **«java1.5.jj»** qui décrit les règles de grammaire et définit les tokens du langage Java 1.5.

JavaCC produit 7 fichiers java en sortie, la figure ci- dessous montre les différentes classes générées :



— **TokenMgrError :** classe d'erreur renvoyant les erreurs détéctées par l'analyseur lexical.

— **ParseException :** classe d'erreur renvoyant les erreurs detéctées par le parseur.
— **Token :** classe représentant les Tokens, chaque objet Token a un attribut entier"kind" qui présente le type de Token et un attribut string "image" qui présente la chaine de carctère à partir du fichier d'entrée correspondante au token.
— **SimpleCharStream :** classe qui délivre les carctères à l'analyseur lexicale.
— **JavaParserConstants :** interface qui définie un nombre de classe utilisés dans l'anlyseur lexicale et syntaxique.
— **JavaParserTokenManager :** analyseur lexicale.
— **JavaParser :** parseur.

Définition des Tokens :

La définition des lexèmes (token) se fait comme la portion de code suivante où nous spécifions les littéraux(entiers, décimaux, hexadécimaux, caractères)

```
TOKEN :
{
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
      | <HEX_LITERAL> (["l","L"])?
      | <OCTAL_LITERAL> (["l","L"])?
  >
|
  < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
|
  < #HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
|
  < #OCTAL_LITERAL: "0" (["0"-"7"])* >
}
```

Dans cette partie, pour adapter la grammaire à notre problème, nous ajoutons un nouveau Token :
<CONDITION : "condition">

Définition des règles :

Pour pouvoir reconnaitre les conditions entrées par l'utilisateur, nous avons ajouté la règle **«Condition()»** dans la grammaire qui fait appel à une règle prédéfinie **«ConditionalExpression() »**, cette dernière définit la règle pour une expression conditionnelle.

Nous pouvons voir la règle condition() dans le code suivant :

```
void Condition():{}
{
    <CONDITION> <IDENTIFIER> "=" "("  ConditionalExpression() ")" ";"
}
```

Nous pouvons également voir la règle conditionalExpression() dans le code suivant :

```
void ConditionalExpression():
{}
{
  ConditionalOrExpression() [ "?" Expression() ":" Expression() ]
}

void ConditionalOrExpression():
{}
{
  ConditionalAndExpression() ( "||" ConditionalAndExpression() )*
}

void ConditionalAndExpression():
{}
{
  InclusiveOrExpression() ( "&&" InclusiveOrExpression() )*
}
```

Nous choisissons de faire appel à la règle **Condition()** dans la règle **ClassOrInterfaceBodyDeclaration()** pour indiquer que la déclaration des conditions doit être dans le corps d'une classe et spécifiquement au même niveau que la déclaration des attributs et des méthodes.

Le code suivant illustre ce mécanisme :

```
void ClassOrInterfaceBodyDeclaration(boolean isInterface):
{
   boolean isNestedInterface = false;
   int modifiers;
}
{
  LOOKAHEAD(2)
  Initializer()
  {
     if (isInterface)
        throw new ParseException("An interface cannot have initializers");
  }
|
  modifiers = Modifiers() // Just get all the modifiers out of the way. If you want to do
                          // more checks, pass the modifiers down to the member
  (
     ClassOrInterfaceDeclaration(modifiers)
   |
     EnumDeclaration(modifiers)
   |
     LOOKAHEAD( [ TypeParameters() ] <IDENTIFIER> "(" )
     ConstructorDeclaration()
   |
     LOOKAHEAD( Type() <IDENTIFIER> ( "[" "]" )* ( "," | "=" | ";" ) )
     FieldDeclaration(modifiers)
   |
     MethodDeclaration(modifiers)
   |Condition()
  )
|
  ";"
}
```

Les règles de déclarations d'attribut et de méthodes

Vérification du nom des compteurs

Nous vérifierons les noms des compteurs en ajoutant une règle **compteur()** dans la grammaire qui n'autorise que les compteurs décrits ci-dessous :

— nomMethode_req.
— nomMethode_att.
— nomMethode_aut.
— nomMethode_act.
— nomMethode_term.

Le code suivant montre cette règle :

```
void compteur():{}
{
    <IDENTIFIER>"_"compteurExistant()
}

void compteurExistant():{}
{
    "act" | "att" | "req" | "aut" | "term"
}
```

Ajout de code Java :

JavaCC permet aussi l'ajout de code Java dans le fichier «.jj», il suffit de mettre le code entre {}.

Nous définissons donc 4 classes java en plus des fichiers générés qui servent à produire le code en sortie .

— **La classe Ecriture** a pour fonctionnalité de créer, ouvrir et écrire dans un fichier, elle sera utile pour la création du fichier généré.
— **La classe Génération** possède le role de générer la déclaration des compteurs de synchronisations, les méthodes pour chaque conditions et l'ajout des outils de synchronisations java dans les méthodes adéquates.
— **la classe Verification** controle la syntaxique et les types du code source.
— **La classe Optimisation** évite la traduction systèmatique en adaptant celle-ci à chaque problème et donc faciliter l'appentissage en simplifiant le code généré.

2.2 Implémentation

Nous avons vu précédemment comment définir les règles de grammaire dans un fichier «.jj» de JavaCC pour pouvoir reconnaître le langage définit par les compteurs de synchronisation. Dans cette partie, nous verrons la méthode de génération du code des compteurs, la méthode de vérification de la cohérence de ces derniers.

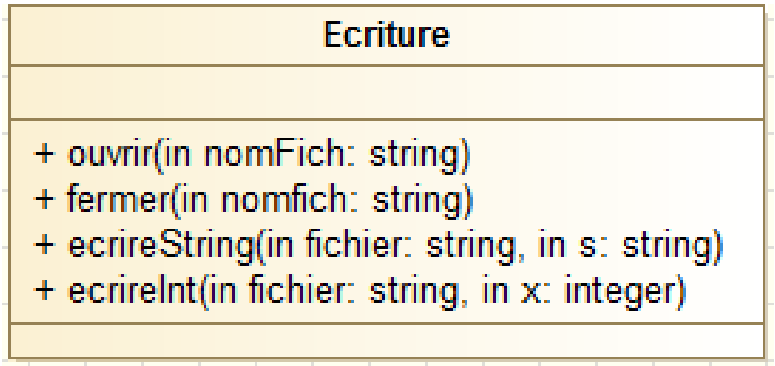
2.2.1 Génération du code

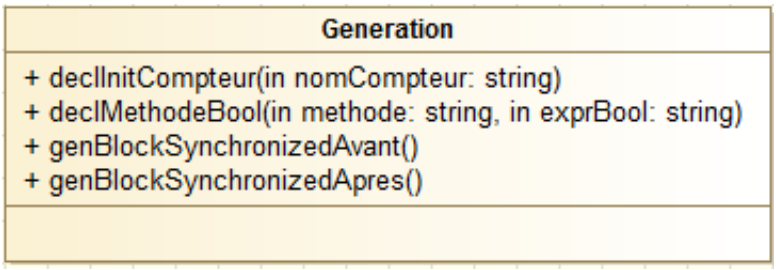
Pour la génération du code, nous implémentons une classe **Generation.java** dans laquelle nous définissons les méthodes qui permettent d'écrire dans le fichier en sortie :

— l'initialisation des compteurs de synchronisations.
— l'implémentation des méthodes qui vérifient les conditions.
— l'insertion des blocs synchronized ainsi que les primitives Java pour gérer le parallélisme.
— la récupération du nom des compteurs.
— la recopie du reste du code qui ne changera pas.

Pour améliorer la lisibilité du code, nous implémentons une classe **Ecriture** dans laquelle nous définissons quelques primitives d'écriture dans un fichier.

Ci-dessous la classe Ecriture et la classe Génération :





D'abord, lors du processus de génération du code, la déclaration et l'initialisation des compteurs se font dès la reconnaissance du mot clé **"condition"** et la récupération du nom de la méhode comme le montre le code ci-dessous :

```
void condition():{}
{
    <CONDITION> <IDENTIFIER>
    {
        //Code Java pour la déclaration et l'initisalisation des compteurs
        Generation.declInitCompteur(nomMethode);
    }
     "=" {
     "(" ConditionalExpression() ")"
     ";"
}
```

Ensuite, l'implémentation des méthodes de test des conditions se fait après la reconnaissance de l'expression conditionnelle c'est-à-dire après le non-terminale **ConditionalExpression()** comme nous le montre le code ci-dessous :

```
void condition():{}
{
    <CONDITION> <IDENTIFIER>
    {
        //Code Java pour la déclaration et l'nitisalisation des compteurs
        Generation.declInitCompteur(nomCompteur);
    }
     "=" {
     "(" ConditionalExpression() ")"
     {
        //Code Java pour générer le code des méthodes de vérificaiton des conditions
        Generation.declMethodeBool(nomMethode, exprBool);
     }
     ";"
}
```

Enfin, pour insérer le code des blocs synchronized ainsi que la mise à jour des compteurs de synchronisations dans la méthode concernée (*la méthode lire dans l'exemple décrit dans la partie rappel du contexte*). Il nous faut générer le code dans la règle **Block()** avant et après la règle **BlockStatement()** qui définit les variables locales ainsi que la suite d'instructions.

Le code suivant illustre ce mécanisme :

```
void Block():
{}
{
  "{"
    {
        //Vérification le bloc de code lu est bien le bloc d'une méthode dont la condition a été défini
        //Generation des blocs synchronized et mise à jour des compteurs avant l'exécution
        //des insructions si la condition est vérifiée
        Generation.genBlocSynchronizedAvant();
    }
    ( BlockStatement() )*

    {
        //Vérification
        //Generation des blocs synchronized et mise à jour des compteurs après l'exécution des instructions
        Generation.genBlocSynchronizedAprès();
    }

    "}"
}
```

2.2.2 Contrôle de validité des expressions

Avant de générer le code des compteurs de synchronisation, nous devons effectuer :
- un conrôle de types de l'expression booléenne de la condition définie par l'utilisateur.
- une vérification des méthodes déclarées par l'utilisateur surtout celles liées à une condition.

Contrôle de types

Pour mieux comprendre le contexte prenons un exemple : si i est un entier, b un booléen, l'instruction $i = 2 + b$ est syntaxiquement correcte mais sémantiquement incorrecte. De ce fait, la grammaire ne suffit pas.

Pour pouvoir évaluer une expression, il nous faut ajouter tout un ensemble de fonctions permettant d'effectuer le contrôle de types. Pour cela, nous implémenterons deux piles : **une pile des types d'opérandes et une pile des opérations.**

Dans le langage Java, les deux opérandes sont toujours de type compatible. L'image ci-dessous indique le type du résultat pour les opérations licites avec les opérations binaires.

Opérations (op)	int op int	Bool op bool
+,-,*,/	int	Erreur
<,>,=<,>=	bool	Erreur
==, !=	bool	bool
&&, ||	Erreur	bool

Lors de l'évaluation de $i1 + b * 2$, une première erreur est détectée lors de la manipulation. Cependant, l'analyse de l'expression doit continuer c'est-à-dire qu'aucune exception ne sera levée. Pour cela, au moment de la détection de l'erreur, nous empilerons le type **Erreur** sur la pile pour pouvoir continuer les calculs. Nous pourrons aussi grâce à ce type d'erreur, ne pas multiplier les messages d'erreur lors de l'évaluation d'une expression incorrecte.

La figure suivante illustre ce mécanisme : Prenons le cas des entiers et des booléens.

Opérations (op)	int op int	Bool op bool
+,-,*,/	int	Erreur
<,>,=<,>=	bool	Erreur
==, !=	bool	bool
&&, ||	Erreur	bool

Pour vérifier à la fin que l'expression lue est bien une expression booléenne, le dernier type présent dans la pile des opérandes doit être du type Bool. Notons que grâce au contrôle de type, nous pourrons optimiser la génération de code en optimisant le nombre de *notifyall()* à générer, nous expliquons cela dans la partie Optimisation.

Vérification des méthodes liées aux compteurs

Pour faciliter la mise à jour des compteurs, nous définissons la classe **Identificateur**.

Rappelons le code de la déclaraiton d'une condition :

```
condition ecrire = (ecrire_act == 0 && lire_act == 0)
```

Dans un objet Identificateur nous avons besoin de stocker le nom de la condition, les compteurs utilisés dans l'expression conditionnelle ainsi que l'expression booléenne associée à cette condition.

```
public class Ident {

public String nom;
public String expressionBooleen;
    public ArrayList<String> compteur;

public Ident(String n){
this.nom = n;
expressionBooleen = "";
        compteur = new ArrayList<String>();
}

}
```

Ces identificateurs seront ensuite stockés dans **la table des identificateurs**. Nous définissons alors une **HashMap<String, Ident>** et nous implémentons des méthodes qui permettront de gérer facilement les identificateurs notamment la recherche, le test d'existence et l'ajout d'un identificateur.

Le code suivant montre le squelette de la classe TabIdent :

```
public class TabIdent{
//Table des identificateurs
private HashMap<String, Ident> table;

    public TabIdent(int taille){...}

    //Permet de récupérer un identificateur à partir de son nom qui représente la clé
    public Ident chercheIdent(String clef){...}
```

```
    //Permet de vérifier si un identificateur existe dans la table
    public boolean existeIdent(String clef){...}

    //Range un identificateur dans la table
    public void rangeIdent(String clef, Ident id){...}


}
```

Ainsi nous pouvons contrôler la méthode liée à une condition ainsi que les compteurs qui y ont été utlisés.

Notamment dans l'exemple :

```
condition ecrire = (ecrire_act == 0 && lire_act == 0)
```

L'identificateur aura comme **nom : ecrire, le tableau compteur contiendra : ecrire_act et lire_act** et la méthode **ecrire(...)** devra avoir été défini.

2.3 Optimisation

Après la vérification syntaxique du code source et la génération du code synchronisé avec les mots clés java, nous nous intéressons à l'optimisation de ce code généré. Cette partie correspond à la troisième classe écrite pour notre compilateur.

Le code généré sans optimisation fonctionne correctement, cependant certaines parties sont superflues notamment la mise à jour de certains compteurs de synchronisation ou certains appel à "this.notifyAll()". Le but du projet étant de faciliter l'apprentissage des compteurs de synchronisation, il est essentiel de ne pas faire une traduction trop systématique en l'adaptant à chaque problème.

Dans un premier temps, nous allons chercher à ne mettre à jour que les compteurs qui sont utilisés dans une des conditions d'exécution des méthodes de la classe. En effet, si un des compteurs n'est présent dans aucune des conditions, sa valeur n'aura aucune influence sur l'autorisation d'exécution d'une des méthodes par un thread ; sa mise à jour est donc totalement inutile. Par exemple pour la classe LecteurRedacteur, les conditions des méthodes lire et écrire ne font appel qu'aux compteurs ecrire_act et lire_act ; il ne sera donc pas nécessaire de mettre à jour les autres compteurs (ecrire_req, acrire-aut, ecrire_term, ecrire_att, lire_req, lire_aut, lire_term, lire_att) dans les méthodes lire et écrire du code généré.

Code généré de la classe LecteurRedacteur :

```
public Class LecteurRedacteur {
private final int TAILLE = 10;
private String tab[] = new String [TAILLE];
private int tab_lec = 0;
private int tab_red = 0;

    private int lire_req = 0 ;

    private int lire_aut = 0 ;
private int lire_term = 0 ;
private int lire_att = 0 ;
```

```
private int lire_act = 0;

private int ecrire_req = 0;
private int ecrire_aut = 0;
private int ecrire_term = 0;
private int ecrire_att = 0;


private int ecrire_act = 0;

public boolean cond_lire () {
return (ecrire_act == 0 );
}

public boolean cond_ecrire () {
return (ecrire_act==0 && lire_act==0 );
}
public String lire(){
synchronized (this) {

lire_req++;
this.notifyAll();
lire_att++;
this.notifyAll();


while ( ! cond_lire() ){
this.wait();
}

lire_aut++;
this.notifyAll();
lire_att--;
this.notifyAll();


lire_act++;
this.notifyAll();
}
return tab[tab_lec];
tab_lec=(tab_lec+1)%TAILLE;
synchronized (this){

lire_term++;
this.notifyAll();


lire_act--;
this.notifyAll();

}
}


public void ecrire( String s){
synchronized ( this ) {
```

~~ecrire_req++;~~
~~this.notifyAll();~~
~~ecrire_att++;~~
~~this.notifyAll();~~

```
while ( ! cond_ecrire() ){
this.wait();
}
```

~~ecrire_aut++;~~
~~this.notifyAll();~~
~~ecrire_att--;~~
~~this.notifyAll();~~

```
ecrire_act++;
this.notifyAll();
}
tab[tab_red]=s;
tab_red=(tab_red+1)%TAILLE;
synchronized(this){
```

~~ecrire_term++;~~
~~this.notifyAll();~~

```
ecrire_act--;
this.notifyAll();
}
}
}
```

Pour cela, nous créons une liste à laquelle nous ajoutons les compteurs présents dans les conditions. Il suffit alors de rajouter une condition d'appartenance à cette liste à la méthode qui génère la déclaration des compteurs et à celle qui gère leur incrémentation et leur décrémentation ainsi que le « this.notifyAll() » qui suit cette mise à jour.

```
public void declInitCompt(String methode){
    if( Optimisation.listeCompteurs.contain( methode+"_req"){
        // génération de la déclaration du compteur methode_req
    }
    if( Optimisation.listeCompteurs.contain( methode+"_aut"){
        // génération de la déclaration du compteur methode_aut
    }
    if( Optimisation.listeCompteurs.contain( methode+"_term"){
        // génération de la déclaration du compteur methode_term
    }
    if( Optimisation.listeCompteurs.contain( methode+"_att"){
        // génération de la déclaration du compteur methode_att
    }
    if( Optimisation.listeCompteurs.contain( methode+"_act"){
        // génération de la déclaration du compteur methode_act
    }

}
```

Dans un second temps, nous allons réfléchir à la nécessité du «this.notifyAll() » suivant systématiquement la mise à jour d' un compteur. En effet, cela permet de réveiller tous les threads bloqués afin de tester de nouveau la condition d'exécution à une méthode qui peut devenir valide suite au changement de valeur d'un compteur. Cependant, le compteur peut « évoluer dans le mauvais sens ». Par exemple, dans la classe LecteurRedacteur, les conditions d'autorisation d'exécution des méthodes lire et écrire étant que les compteurs lire_act et ecrire_act soient nuls ; il est inutile de mettre « this.notifyAll() » suivant l'incrémentation d'un des compteurs (les compteurs étant toujours positifs, la condition sera forcément fausse).

```
public Class LecteurRedacteur {
private final int TAILLE = 10;
private String tab[] = new String [TAILLE];
private int tab_lec = 0;
private int tab_red = 0;

private int lire_act = 0;
private int ecrire_act = 0;

public boolean cond_lire () {
return (ecrire_act == 0 );
}

public boolean cond_ecrire () {
return (ecrire_act==0 && lire_act==0 );
}

public String lire(){
synchronized (this) {
while ( ! cond_lire() ){
this.wait();
}
lire_act++;

~~this.notifyAll() ;~~
```

```
}
return tab[tab_lec];
tab_lec=(tab_lec+1)%TAILLE;
synchronized (this){
lire_act--;
this.notifyAll();
}
}

public void ecrire( String s){
synchronized ( this ) {
while ( ! cond_ecrire() ){
this.wait();
}
ecrire_act++;

this.notifyAll() ;


Ou }
tab[tab_red]=s;
tab_red=(tab_red+1)%TAILLE;
synchronized(this){
ecrire_act--;
this.notifyAll();
}
}
}
```

Pour cela, nous analysons les expressions booléennes afin de poser des conditions sur la génération du « this.notifyAll() ».

3 Conclusion

Au cours de cette phase de conception, réfléchir à la vérification, la génération et à l'optimisation pour passer d'un code source à un code synchronisé en Java afin de faciliter l'apprentissage de celui-ci nous a permis de mieux comprendre le fonctionnement de la synchronisation grâce aux nombreuses questions qui sont soulevées au cours de ces différentes étapes.

Enfin,il nous semble opportun de dire que mener ce projet de bout en bout sera extrêmement enrichissant car toutes ces phases nous ont permis de bien comprendre l'importance de l'analyse, de la planification et de la conception qui sont primordiales et indispensables au succès de tout projet.