Design patterns pour la compilation

Référence : tigdes_20010430_v11	**Nombre de pages :** 13
Version : 1.1	**Date** : 30/04/2001
Auteur : H.Delorme , W.De Denterghem	
Liste de diffusion A.Demaille , T.Géraud , P.Laroque	

Historique du document:

N° de version	Date	Auteurs	Description des modifications
1.0	30/04/2001	H.Delorme	Création du document : syntaxe abstraite , visiteurs, modélisation des frames
1.1	09/05/2001	H.Delorme W.Denterghem	Compléments sur partage d'information entre visiteurs et suggestions Ajout « Gestion des frames » : problématique, modélisation existante, modélisation selon Appel

Glossaire

Abréviation	Description
FP	*Frame Pointer*
IR	*Intermediate Representation*

Sommaire

I. Syntaxe abstraite pour Tiger

1. Modélisation de la syntaxe abstraite

1.1. Problématique : séparation des traitements du modèle de classe

Les programmes Tiger sont représentés sous forme d'arbres syntaxiques. Sur ces arbres seront effectués des traitements d'analyse sémantique, de génération de code, d'impressions formatées, etc. La plupart de ces traitements nécessitent de traiter les nœuds différemment les uns des autres.
Dans une première approche, on pourrait implémenter directement les opérations directement dans les classes des nœuds. Mais cela pose les inconvénients suivants :

- La dissémination des opérations parmi les différents nœuds conduit à un ensemble difficile à maintenir et faire évoluer
- L'ajout d'une nouvelle opération nécessite souvent la recompilation de toutes les classes

Ces problèmes seraient résolus s'il était possible de pouvoir ajouter chaque opération séparément, et que les classes soient indépendantes des opérations qui leur sont appliquées.

1.1.1. Application du pattern Visiteur

Afin de découpler traitements et modèle, le pattern *Visiteur*[GHJV94] est appliqué dans la modélisation courante.
Un point intéressant est que ce modèle thésaurise les informations d'état : celles-ci sont accumulées au cours de la visite de chacun des éléments de l'arbre syntaxique. Sans visiteur, ces états devraient être passés comme arguments aux opérations qui effectuent le parcours, ou encore ils devraient exister en tant que variables globales.
Puisque les classes qui définissent l'arbre syntaxique ont une faible probabilité d'évoluer, en tout cas beaucoup moins que les algorithmes à appliquer à l'arbre syntaxique, son utilisation est vraiment indiquée.

La modélisation originale proposait un couplage fort entre plusieurs visiteurs pour minimiser ce problème de partage d'informations. On pouvait en effet rassembler en familles des visiteurs effectuant des tâches apparentées chronologiquement et sémantiquement. Ainsi, l'actuel *TypeVisitor* ne faisait qu'un avec le *TranslateVisitor* et *l'EscapeVisitor*, proposant ainsi une macro-étape d'analyse sémantique et de traduction en code intermédiaire sous le nom de *SemantVisitor*. Cette modélisation était intéressante car elle offrait une communication directe entre ces différentes étapes mais avant l'inconvénient de rendre indissociable leur utilisation ce qui allait à l'encontre du but fixé qui est d'obtenir un compilateur comprenant des modules ou tâches sérialisés, permettant ainsi d'arrêter le processus de compilation à une étape donnée pour en observer le résultat. La modélisation actuelle corrige ce problème en proposant des visiteurs individualisés selon leur fonction atomique.

Par contre, on peut dégager certaines critiques quant à son implémentation. Une des conditions d'utilisations du pattern indique qu'il peut être appliqué quand « *il s'agit d'effectuer plusieurs opérations distinctes et sans relation entre les objets d'une structure* »[GHJV94].
La traduction de l'arbre syntaxique en code intermédiaire est réalisée via un visiteur (*TranslateVisitor*), mais cette phase doit être précédée du calcul des *escapes*. Le calcul des *escapes* est naturellement implémenté aussi avec un visiteur(*EscapesVisitor*), et ces informations sont nécessaires pour la suite, on vient de le voir. Certes, ceci est contre-indiqué, mais la simplicité et la souplesse obtenues excusent en partie cette rupture du pattern. Pourtant, le partage d'informations entre visiteurs pose un problème de conception difficile à résoudre.

1.1.2. Partage d'informations entre visiteurs

On a évoqué précédemment que le transfert d'informations entre visiteurs pose un problème de conception. Il est intéressant d'aborder comment ces problèmes sont résolus dans l'implémentation courante de Tiger.

- Pont Analyse sémantique -> Traduction en code intermédiaire

Pour traduire Absyn en représentation intermédiaire, nous avons besoins de deux choses :

- La liste des variables s'échappant (calculée par *EscapeVisitor*)
- Le type de certaines expressions (notamment *Let* et *If*)

Ces informations ne peuvent être déduites au moment de la traduction car elles sont déterminées au moment du contrôle de type, il faut donc conserver ces informations pour la phase de traduction en code intermédiaire.

La solution retenue est de procéder à un etiquetage des nœuds d'Absyn au moment où on dispose de l'information puis d'interpréter ces etiquette dans une phase postérieure lorsque l'on souhaite exploiter l'information. Plus particulièrement, on etiquetera les nœuds *VarDec* et les nœuds *Field* dans le contexte des paramètres formels. Pour ce qui est des types d'expressions on etiquetera les nœuds *LetExp*, *IfExp*.

Dans l'état actuel des choses, cette solution pose les problèmes de conception suivants :

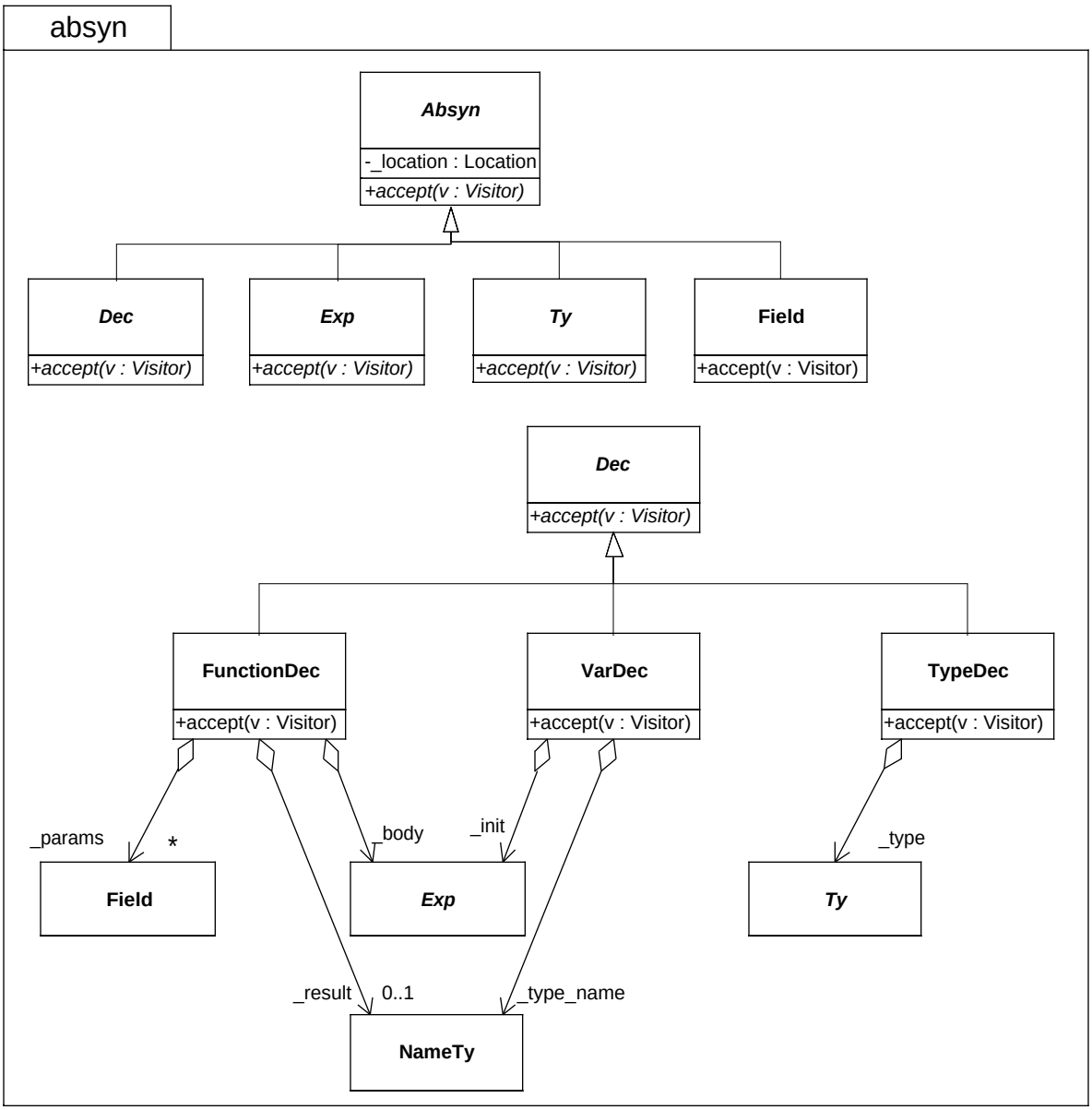
- Manipulation de nœuds d*'Absyn* const (on doit affeceter les étiquettes via des const_cast)

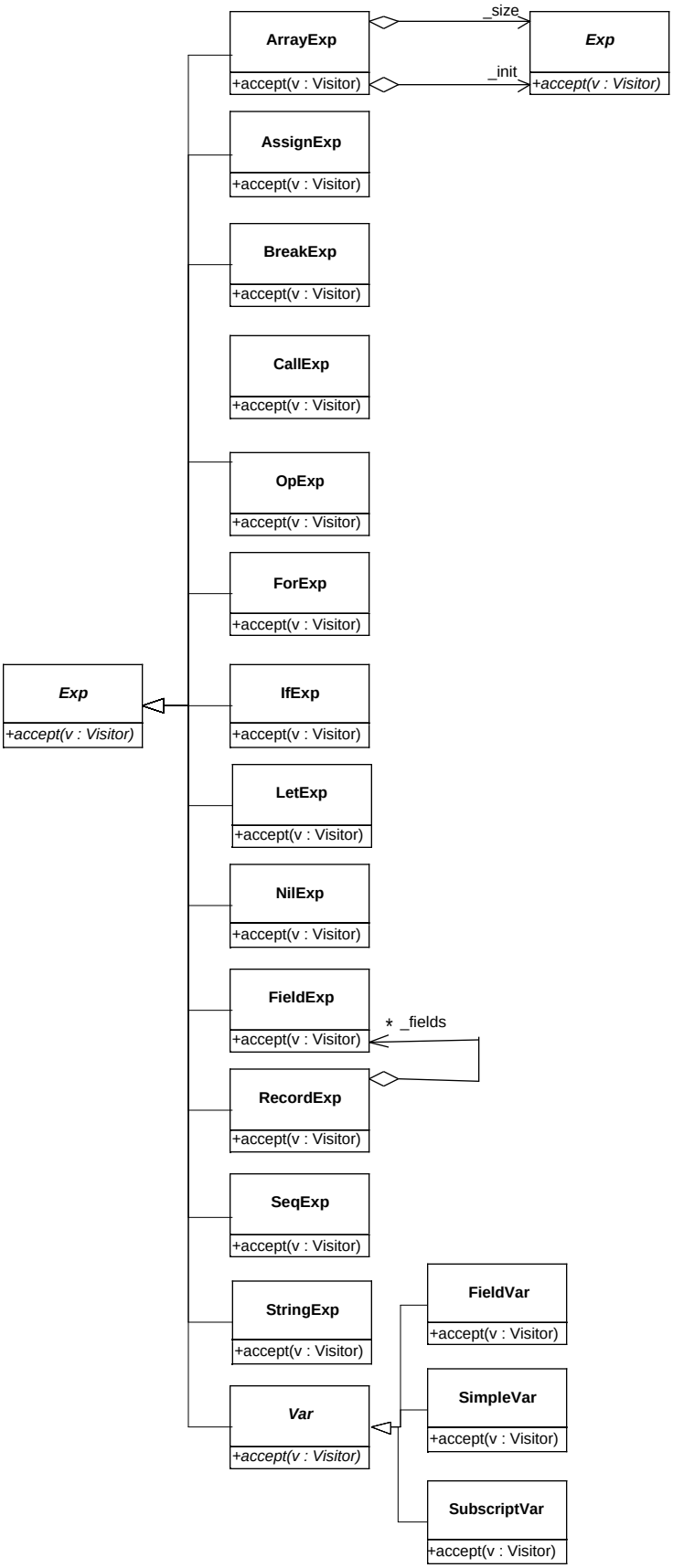
- La présence d'étiquettes altère l'intégrité de l'arbre syntaxique en introduisant des attributs nécessaires uniquement pour des opérations extrinsèques : les nœuds n'ont aucun besoin de connaître le type d'expression qu'ils représentent.

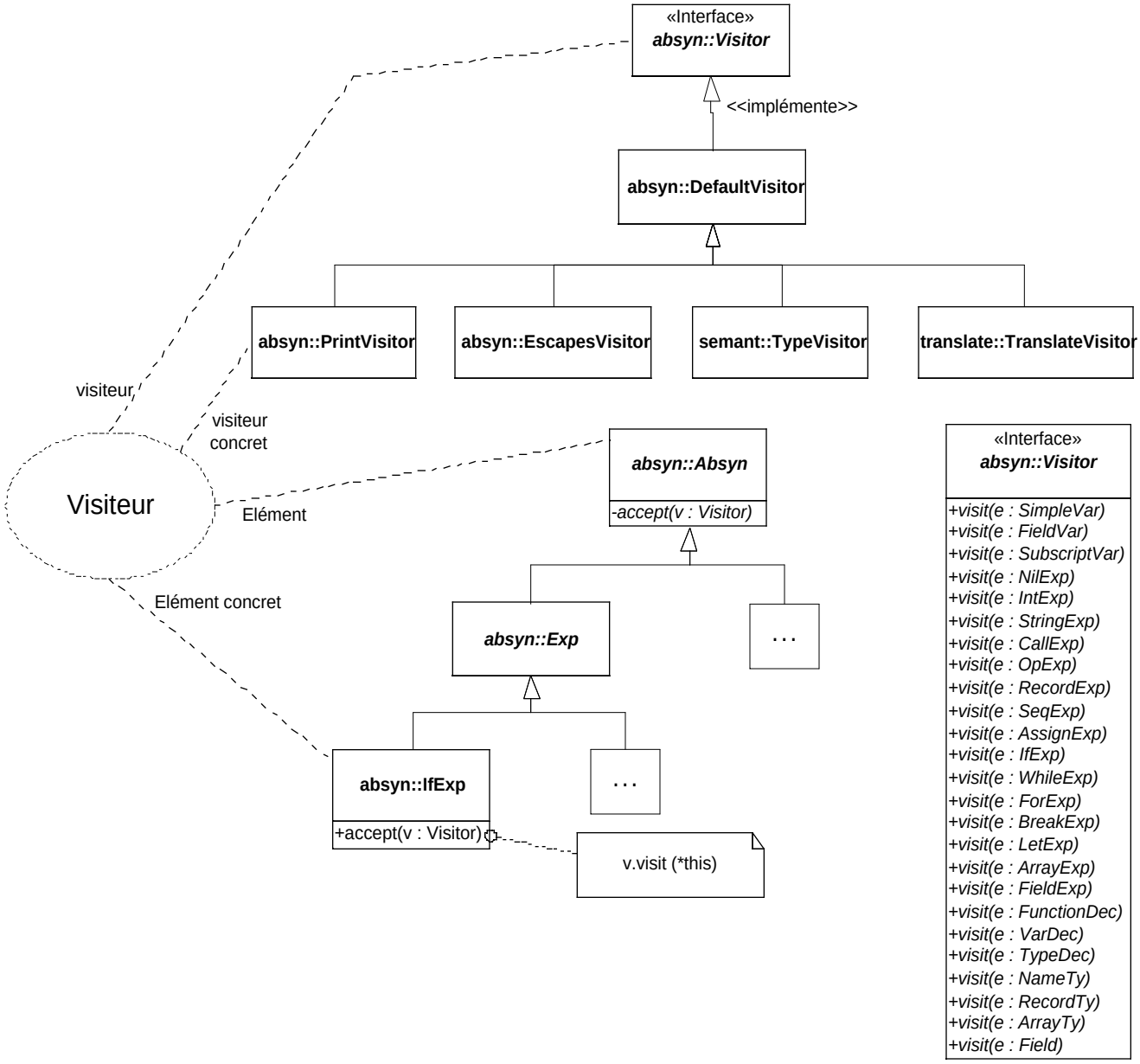
Cette solution pourrait être améliorée (dans le cas du problème 1) en rendant *Absyn* non const et donc accessible en lecture/écriture.

1.2. Modèle statique

1.2.1. Classes des noeuds




ArrayExp
+accept(v : Visitor)
_size
_init
Exp
+accept(v : Visitor)
AssignExp
+accept(v : Visitor)
BreakExp
+accept(v : Visitor)
CallExp
+accept(v : Visitor)
OpExp
+accept(v : Visitor)
ForExp
+accept(v : Visitor)
Exp
+accept(v : Visitor)
IfExp
+accept(v : Visitor)
LetExp
+accept(v : Visitor)
NilExp
+accept(v : Visitor)
FieldExp
+accept(v : Visitor)
* _fields
RecordExp
+accept(v : Visitor)
SeqExp
+accept(v : Visitor)
StringExp
+accept(v : Visitor)
Var
+accept(v : Visitor)
FieldVar
+accept(v : Visitor)
SimpleVar
+accept(v : Visitor)
SubscriptVar
+accept(v : Visitor)


«Interface»
absyn::Visitor
<<implémente>>
absyn::DefaultVisitor
absyn::PrintVisitor
absyn::EscapesVisitor
semant::TypeVisitor
translate::TranslateVisitor
visiteur
visiteur concret
Visiteur
Elément
Elément concret
absyn::Absyn
-accept(v : Visitor)
absyn::Exp
. . .
absyn::IfExp
+accept(v : Visitor)
. . .
v.visit (*this)
«Interface»
absyn::Visitor
+visit(e : SimpleVar)
+visit(e : FieldVar)
+visit(e : SubscriptVar)
+visit(e : NilExp)
+visit(e : IntExp)
+visit(e : StringExp)
+visit(e : CallExp)
+visit(e : OpExp)
+visit(e : RecordExp)
+visit(e : SeqExp)
+visit(e : AssignExp)
+visit(e : IfExp)
+visit(e : WhileExp)
+visit(e : ForExp)
+visit(e : BreakExp)
+visit(e : LetExp)
+visit(e : ArrayExp)
+visit(e : FieldExp)
+visit(e : FunctionDec)
+visit(e : VarDec)
+visit(e : TypeDec)
+visit(e : NameTy)
+visit(e : RecordTy)
+visit(e : ArrayTy)
+visit(e : Field)

II. Analyse sémantique

III. Gestion des frames

1. Rappels

1.1. Frame

1.2. Organisation

2. Problématique : indépendance des architectures cibles

On veut, à partir de code Tiger, produire de l'assembleur correspondant à différentes architectures machines cibles. Ces architectures possèdent un jeu d'instructions propre plus ou moins complexe, la gestion de la mémoire leur est spécifique, etc. Evidemment, les phases « back-end » ont besoin de connaître les détails de l'architecture cible. La gestion des frames est le premier élément dépendant de l'architecture cible à traiter. Chaque architecture gère différemment les frames(sauvegarde de registres, « view shifting », etc.), il faut donc trouver une abstraction à leur représentation et utilisation.
Récapitulons, concernant la gestion des frames, ce qui est dépendant de l'architecture :

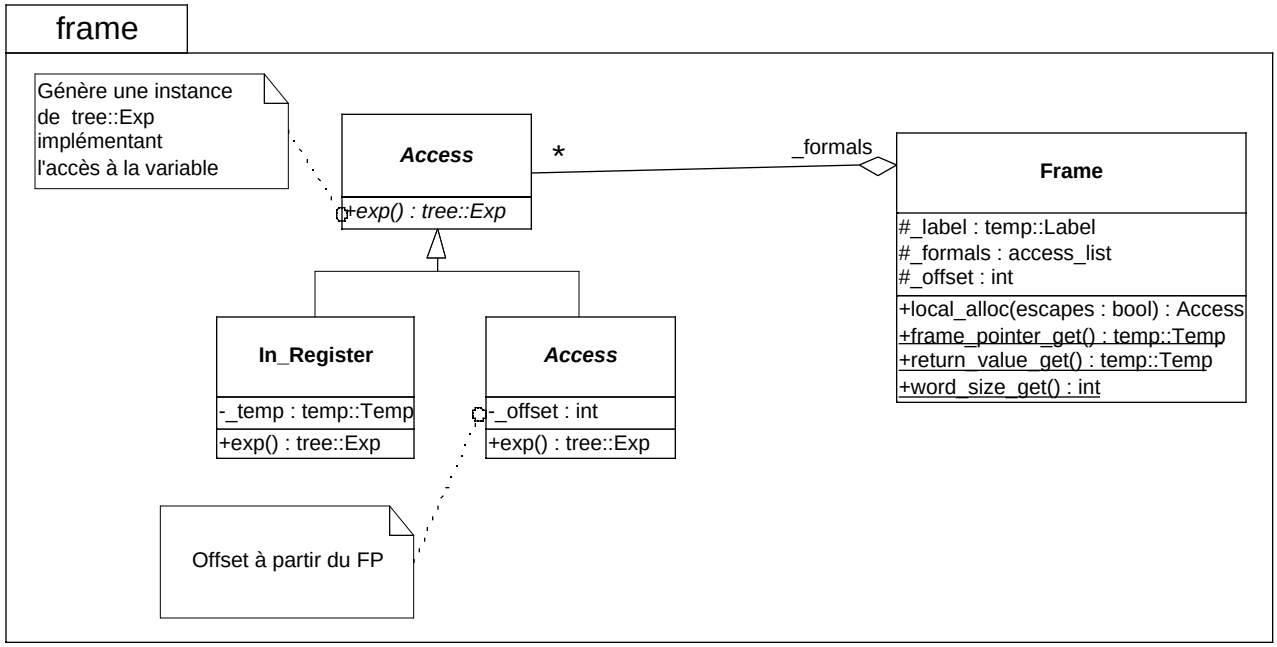
- Taille d'un mot machine (offset par rapport au FP)
- Instructions nécessaires pour implémenter le « view shift ».
- Instructions particulières pour les prologues/épilogues de fonctions(sauvegardes/restauration de registres)

Fournir une abstraction à la gestion des frames est une étape dans la résolution de la problématique : le code client de cette abstraction doit être indépendant de l'implémentation utilisée. En somme, on ne veut révéler seulement que l'interface et non l'implémentation.

De plus, la modélisation doit permettre la prise en compte de nouvelles architecture facilement.

3. Historique des modélisations

3.1. Modélisation existante



Cette modélisation reprend l'abstraction fournie par F_Access dans [App98, p.137]. La classe abstraite frame::Access décrit les paramètres ou les variables de locales qui peuvent résider dans la frame ou dans des registres. Elle fournit un accès transparent à ces variables via la méthode abstraite *exp*, générant le code IR nécessaire.

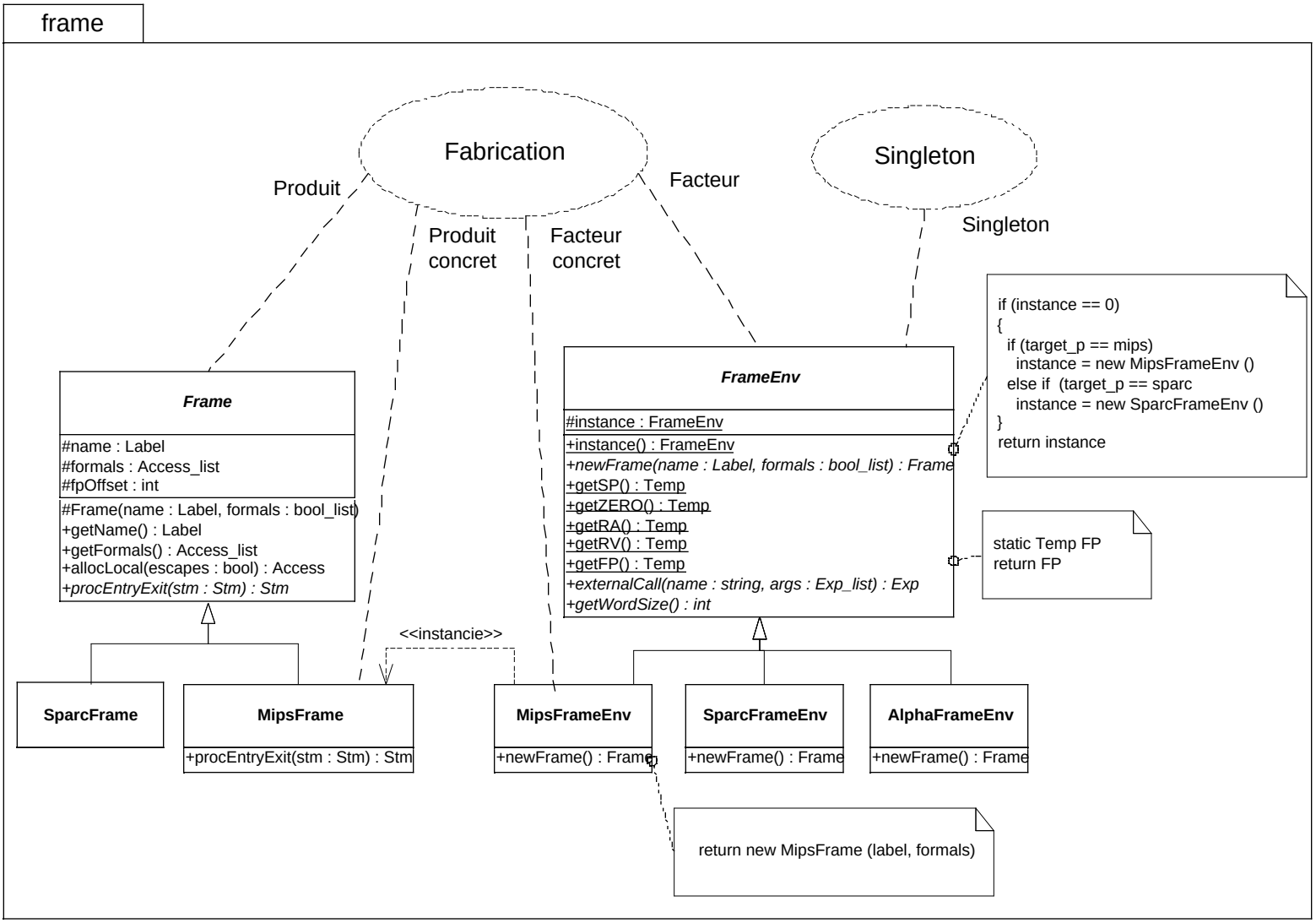
frame::Frame maintient naturellement une liste d' « accès » aux variables présentes dans le contexte de la frame, celles-ci sont ajoutées via l'opération *local_alloc*.
Mais, la modélisation des frames n'est pas complète. frame::Frame ne prend pas en compte par exemple la gestion du « view shift », ni même celle des prologues/épilogues. De plus, la méthode de classe *word_size_get* renvoie toujours la même valeur, quelle que soit l'architecture cible.
Plus généralement, cette modélisation ne prend pas du tout en compte le support de plusieurs architectures cibles, et résout peu d'éléments de la problématique : aucune capacité d'évolution, pas d'abstraction des frames.

3.2. Nouvelles modélisations

Le but de ces modélisations est surtout d'assurer l'indépendance vis-à-vis des architectures cibles pour le code client.

3.2.1. Selon Appel

Selon Appel, les notions dépendantes d'une architecture cible doivent être regroupées dans le module *frame*[App98]. Ainsi, par exemple, il préconise que les registres disponibles sur une architecture soient déclarés dans ce module.



Cette modélisation prend en compte ce concept, et résout les éléments de la problématique.
L'utilisation du pattern *Fabrication*[GHJV94] permet d'encapsuler la connaissance du type de la sous-classe de *Frame* à créer. *FrameEnv* confie à ses sous-classes la définition de la fabrication(*newFrame*) de sorte qu'elle renvoie l'instance appropriée. Le pattern nous dispense donc d'avoir à incorporer dans le code des classes

spécifiques d'une architecture. Le code ne concerne que l'interface *Frame*, il peut donc fonctionner avec toute classe implémentant *Frame*. Le pattern apporte directement la prise en charge de l'évolutivité : de nouvelles architectures peuvent être facilement supportées, il suffit pour cela d'implémenter *FrameEnv* et *Frame*.

On se retrouve tout de même avec le dilemme d'avoir à instancier des classes(celles implémentant *FrameEnv*) dont on ne peut prédire la nature. De plus, on ne veut qu'une seule instance de *FrameEnv*.
Pour résoudre le problème, le pattern *Singleton*[GHJV94] est appliqué à *FrameEnv* : il garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance. La méthode de classe *instance* représente ce point d'accès et prend en charge d'instancier la bonne classe concrète en fonction l'architecture cible visée, de ce fait le code client ne s'occupe pas de l'instanciation effective de *FrameEnv*.

Dans un premier temps, le pattern *Fabrique Abstraite*[GHJV94] avait été appliqué, mais son application dénaturait la philosophie du pattern. La fabrique abstraite fournit une interface pour la création de familles d'objets apparentés sans qu'il soit nécessaire de spécifier leurs classes concrètes. Hélas, dans notre cas, une seule famille de produit existe : les frames. Il est donc plus naturel d'appliquer la *Fabrication*.

3.2.2. Par architectures

Bibliographie

[App98] Andrew W. Appel. *Modern compiler implementation in C*. Cambridge University Press, 1998.

[GHJV94] Erich Gamma , Richard Helm , Ralph Johnson , Jhon Vlissides. *DESIGN PATTERNS, Catalogue de modèles de conception réutilisables.* Vuibert,1994.

[MuG00] Pierre Alain Muller , Nathalie Gaertner. *Modélisation objet avec UML*. Eyrolles, 2000

Sitographie