UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

Évolution et validation du modèle intentionnel

Utilisation dans un cas réel

Mémoire présenté par Frédéric PLUQUET
dirigé par Roel WUYTS en vue de l'obtention du
Diplôme d'Études Approfondies en Informatique

Année académique 2004–2005

Remerciements

La clef qui ouvre toutes les portes... La confiance.
Charlotte Savary

Je profite de ce petit espace personnel du mémoire, pour remercier toutes les personnes qui ont contribué de près ou de loin à la réussite de celui-ci.

Je voudrais remercier tout d'abord Monsieur *Roel Wuyts*, professeur qui a su voir en moi les capacités nécessaires au travail qu'il m'a proposé. J'aimerais tout particulièrement le remercier pour la confiance qu'il a placée en moi.

Je voudrais aussi remercier Messieurs *Kim Mens* et *Andy Kellens*, partenaires de cette recherche, à qui je dois l'évolution de mon anglais de base.

Je voudrais ensuite remercier Mademoiselle *Magali Molle*, qui a su, je ne sais par quel miracle, me comprendre totalement et qui, de ce fait, m'aide chaque jour à avancer dans la vie.

Je voudrais évidemment remercier *mes parents*, qui, grâce à leur confiance et au cocon familial rempli de tendresse et d'humour qu'ils ont su m'offrir, m'ont permis de choisir les bons chemins de la vie.

Enfin, je remercie toutes ces personnes, qui au détour d'un regard, d'une parole ou d'un geste, ont su m'encourager ou m'aider dans la tâche qui était la mienne.

Table des matières

Chapitre 1

Introduction

La documentation d'un système informatique joue un rôle important tout au long du développement d'un projet de taille plus ou moins grande. En effet, une documentation mise à jour permet une meilleure compréhension du système global par chaque développeur et donc, une meilleure entente sur l'implémentation du système. Un problème récurrent dans le développement d'application est la *non mise à jour* de la documentation lors de changements dans le code. La justification principale est que le temps pris à mettre à jour la documentation n'est pas utilisé pour faire avancer l'implémentation.

1.1 La documentation

La documentation d'un système peut être construite à deux moments très différents : soit avant l'implémentation d'un nouveau système, soit lors de l'étude d'un système existant, afin de l'étendre ou le remanier pour des raisons de performances ou de design.

Dans le premier cas de figure, le *foward engineering* sera utilisé. Lors de la création d'un nouveau système, cette méthode de conception définit la documentation avant de coder quoique ce soit. L'implémentation sera, normalement, basée sur cette documentation. Cette méthode part d'un niveau élevé d'abstraction composé de designs indépendants de toute implémentation pour ensuite se diriger progressivement vers l'implémentation physique d'un système. Si une approche séquentielle de cette méthode est adoptée, celle-ci passe par les étapes suivantes : analyse des besoins, conception de designs de haut niveau et enfin la phase d'implémentation [Wuy01]. La documentation est alors composée de l'architecture du projet ainsi que d'éventuelles conventions de codage. Après cette première phase de conception, l'attention se portera beaucoup plus sur l'implémentation que sur la maintenance de la documentation. Ceci a été observé pour différentes phases, et est connu sous le nom de "architectural drift and erosion" pour la phase de conception de haut-niveau [PW92].

Si on désire plutôt partir d'un projet existant pour le faire évoluer, on utilisera la méthode de conception dite de *reverse engineering* (ou *reengineering*). Partant d'un système existant, celle-ci permet d'identifier ses composants et leurs relations afin de créer plusieurs représentations du système étudié d'une

autre forme que celles existantes ou d'un niveau d'abstraction supérieur. La documentation extraite du code va donc être vitale lors de l'analyse de la solution afin de faire évoluer le système existant.

Nous pouvons dès lors remarquer que la documentation joue un rôle central lors de l'évolution d'un système.

La principale cause des écarts entre la documentation et l'implémentation peut être résumée comme ceci : dès les premières implémentations de la solution choisie, de légères modifications interviennent dans les plans originels, dues le plus souvent à quelques problèmes non perçus lors de l'analyse ou de la phase de design. La documentation n'est alors que très rarement mise à jour par les développeurs, se basant dès ce moment sur leurs connaissances acquises et leur compréhension du code.

Lors de la réutilisation de cette documentation obsolète, on peut remarquer qu'il est souvent très difficile de différencier les informations correctes de celles erronées. Il serait donc intéressant d'avoir une documentation dynamique dès le début du projet (que ce soit en forward ou reverse engineering) qui présenterait automatiquement les problèmes résidant dans celle-ci lors de l'évolution du code source. Ce même outil d'évolution simultanée du code source et de la documentation, appelée *co-évolution*, permettra, tout au long du développement, de garder la documentation à jour avec les changements de l'implémentation. C'est ce que le modèle des *vues et relations intentionnelles* (*Intensional Views and Relations*) propose.

1.2 Vers une documentation dynamique

Le modèle des vues et relations intentionnelles permet de réaliser une documentation dynamique, évoluant avec le code. L'idée est de définir des vues sur un code source, c'est-à-dire un ensemble de contraintes d'architecture (comme des patterns) et/ou de codage (conventions de nominations des classes, méthodes, ...), sur du code orienté objet. Lorsque la documentation devient inconsistante avec le code, un mécanisme permet de retrouver les entités de code (classes, méthodes, ...) ne respectant pas les contraintes définies par cette documentation.

Ce modèle a été d'abord implémenté en *SOUL* (langage de programmation logique sous *Smalltalk*[1]) pour documenter du code écrit en *Smalltalk*. Une nouvelle version a vu le jour durant cette recherche, implémentée complètement dans le paradigme orienté objet sous *Smalltalk*, pour des raisons de performances. De plus, quatre outils graphiques permettent d'utiliser ce modèle assez facilement. L'environnement offert par ces outils se nomme *IntensiVE* (**Intensi**onal **V**iews **E**nvironment).

1.3 Buts du mémoire

La recherche que retrace ce mémoire se décompose en deux phases.

[1] langage purement orienté objet

La première est de créer un outil offrant une vue globale d'une documentation écrite dans le modèle intentionnel. En effet, le modèle intentionnel est pour le moment exploité par quatre outils destinés à l'édition des définitions des vues et relations, et la vérification de la cohérence de celles-ci. Ces outils fonctionnent très bien et sont facilement manipulables. Mais très vite on peut se rendre compte qu'il est très fastidieux de rechercher des réponses à des questions globales telles que "Quelles sont toutes les vues qui ne sont pas cohérentes ?" ou encore "Quelle est la vue qui contient le plus d'entités de code de tout mon système ?". En effet, pour répondre à de telles questions pour le moment, il faut parcourir chaque vue et les comparer soi-même les unes avec les autres. L'intérêt d'un cinquième outil est apparu dans l'étude d'un cas réel où le nombre de vues devenait important et dans lequel l'utilisation des quatre outils existants ne suffisait plus.

Ni le modèle intentionnel, ni les outils l'accompagnant n'ont encore été validés. La seconde phase sera donc de valider le modèle ainsi que l'ensemble des outils présents dans *IntensiVE*, l'environnement de manipulation des vues et des relations intentionnelles. Cette validation sera effectuée par une étude d'un programme réel de taille moyenne nommé *SmallWiki*.

1.4 Remarque préliminaire

Nous attirons l'attention du lecteur sur le fait que ce mémoire a été réalisé dans le cadre d'une recherche intercommunautaire en collaboration avec la Vrije Universiteit of Brussels (VUB). Le lecteur pourra donc remarquer que plusieurs termes non traduits sont écrits en anglais, langue commune utilisée lors de la recherche.

1.5 Structure du mémoire

Le chapitre 2 explique les langages *Smalltalk* et *SOUL*, langages utilisés lors de la recherche. Le chapitre 3 définit le modèle des vues et relations intentionnelles, aussi appelé plus simplement *modèle intentionnel*. Le chapitre 4 passe en revue les quatre outils développés initialement au-dessus du modèle intentionnel. Le chapitre 5 énumère les problèmes liés à ces quatre outils, explique la nécessité d'un nouvel outil et définit une solution conceptuelle résolvant les problèmes. Le chapitre 6 explique l'outil créé pour valider la solution définie dans le chapitre 5. Le chapitre 7 effectue la validation du modèle intentionnel et des cinq outils l'accompagnant. Enfin, le chapitre 8 conclut ce mémoire en résumant les idées principales et en fournissant des critiques du travail actuel et de nouvelles pistes pour des travaux futurs.

Chapitre 2

Smalltalk et SOUL

Pour étudier le problème sur lequel nous nous pencherons dans ce mémoire, plusieurs notions, modèles et outils ont besoin qu'être définis et décrits. Dans ce chapitre, nous verrons les langages *Smalltalk*, un langage purement orienté objet, et *SOUL*, un langage du type `Prolog` permettant de faire de la méta-programmation logique sur du code *Smalltalk*. Le chapitre suivant introduit le modèle intentionnel, constituant le centre de notre recherche.

2.1 Smalltalk

2.1.1 Généralités

Le langage *Smalltalk* fut créé dans les années '80 par Alan Kay, Dan Ingalls, Ted Kaehler, Adele Goldberg au Palo Alto Research Center de Xerox. Le but premier de ce langage est d'offrir un langage de programmation si simple à utiliser que "tout le monde" pourrait l'employer sans problème. C'est la raison pour laquelle, comme nous allons le voir, sa syntaxe est très simple.

De plus, *Smalltalk* a été l'un des précurseurs dans le domaine de l'orienté objet (OO). Les concepts d'encapsulation, d'héritage et de polymorphisme y sont d'ailleurs présents. *Smalltalk* est un langage purement orienté objet, c'est-à-dire composé uniquement d'objets et de messages pouvant être envoyés sur ces derniers. Par exemple, les chaînes de caractères, les entiers, les booléens, les définitions de classes, les blocs de code, les piles et la mémoire sont représentés en tant qu'objets. L'un des autres concepts originaux de *Smalltalk* est que tout peut être modifié. Il est, par exemple, possible de changer l'IDE[1], en cours d'utilisation, sans recompiler ni redémarrer l'application. Une nouvelle instruction de contrôle dans le langage vous serait utile ? Vous pouvez l'ajouter ! De plus, le typage est dynamique : pas besoin de définir de types dans le code. Ensuite, un ramasse-miettes mémoire est intégré et transparent pour le développeur et un système de gestion d'exceptions avec reprise est fourni. Le débuggeur est très puissant. Grâce au côté OO de *Smalltalk*, le débuggeur permet de savoir où le programme s'est fourvoyé, après quelle suite d'appels, de réexécuter le code à partir du début d'une méthode avec le contexte original, de changer le code à

[1] Integrated Development Environment

la volée et de reprendre l'exécution du code en tenant compte du code modifié. Finalement, les programmes *Smalltalk* sont généralement compilés en bytecode, exécutés par une machine virtuelle, ce qui leur permettent d'être portables sur la plupart des OS actuels [2].

Plusieurs versions de *Smalltalk* sont apparues au fil des années, toutes spécialisées dans un domaine plus ou moins précis (performances, ajouts de fonctionnalités, ...). De plus, plusieurs environnements de développement existent pour *Smalltalk*. Notre choix lors de cette recherche a été d'utiliser *Smalltalk*-80 dans l'environnement *VisualWorks Smalltalk* 7.2.1. Cet outil va nous faciliter certaines tâches lors de la conception de notre application (design des fenêtres, des menus, aide au refactoring (c'est-à-dire du remaniement de code), ...).

2.1.2 Structure du code

Smalltalk est composé de classes, elles-mêmes composées de méthodes et de variables. Les méthodes sont rangées par protocoles dans chacune des classes. Chaque classe repose dans un ou plusieurs packages, qui sont des sortes de répertoires. Un mécanisme de namespaces est présent afin de limiter la portée des noms de classes au sein d'un même domaine. On retrouve le même mécanisme dans le langage C++.

Des variables partagées, qui peuvent être vues comme des variables globales, peuvent être déclarées au sein des classes.

2.1.3 Syntaxe

Smalltalk étant dynamiquement typé, sa syntaxe reste très simple. Voici les quelques éléments qui la composent[3] :

L'assignation : `:=` Ce symbole permet d'assigner une valeur (située à droite du signe) à une variable (située à sa gauche) ;

Les variables locales : `| aVar1 aVar2 |` Les variables locales, s'il en existe au moins une, doivent être déclarées en tout premier lieu dans le code *Smalltalk*. Elles sont séparées par des espaces (par exemple ici `aVar1` et `aVar2`) et l'ensemble est délimité par 2 barres verticales (`| |`) ;

Le point final : . Une expression *Smalltalk* doit toujours se terminer par un point final (sauf dans le cas où elle est la dernière, le point final n'étant pas alors obligatoire) ;

L'envoi de message : `aVar mess` En *Smalltalk*, tout n'est qu'objet et envoi de messages vers ceux-ci. Ici, le message `mess` est envoyé à l'objet représenté par la variable `aVar`.

Les types de messages Il existe 3 types de messages :

1. Les messages unaires : le nom de la méthode est simplement formé d'une suite de caractères (`factorial, open, class`).

[2] Pour le moment, sur Windows, Mac OS, Linux mais aussi AIX, HP-UX, SGI, Windows CE et bien d'autres.

[3] Nous détaillerons ici uniquement la syntaxe utile pour la compréhension du reste de ce mémoire. Pour une information plus exhaustive, veuillez consulter [Liu96]

Exemples :

```
2000 factorial.
Browser open.
```

2. Les messages binaires : le nom de la méthode est formé d'un ou deux caractères choisis parmi les symboles `+ - / \ * ~ < > = @ & ?` comme par exemple : `>=` .
   Exemple :

   ```
   100@100
   ```

   Ce code permet de créer un point ayant comme coordonnées (100,100).
3. Les messages à base de mots-clés : le nom de la méthode est formé d'un ou plusieurs mots terminés par un `:` qui spécifie qu'un argument est attendu.
   Exemples
   `r: g: b:` (trois arguments), `playFileNamed:` (un argument), `at: put:` (deux arguments),

   ```
   Color r: 1 g: 0 b: 0
   MIDIFileReader playFileNamed: 'LetItBe.MID'
   ```

L'envoi multiple de messages sur un même objet : `;` Si plusieurs messages consécutifs doivent être envoyés vers un même objet, un raccourci peut être employé via le symbole `;`. Si nous avons le code suivant :

```
aVar mess1. aVar mess2. aVar mess3.
```

nous pouvons le réduire dans la forme plus simple suivante :

```
aVar mess1 ; mess2 ; mess3.
```

Passage d'argument à un message : `aVar mess4: anObject` Des arguments (ici `anObject`) peuvent être passés en paramètre à un message (`mess4:`) lors de son envoi vers un objet (`aVar`). C'est lors de la création du message que le nombre de paramètres est fixé.

Référence à l'objet courant : `self` On peut toujours accéder à l'objet courant via le receveur spécial `self`.

Référence à l'objet parent : `super` On peut aussi accéder à l'objet parent par le receveur spécial `super`.

2.1.4 Les blocs

Les blocs *Smalltalk* sont des objets qui contiennent des expressions *Smalltalk*, qui peuvent accepter des paramètres et dont on peut calculer la valeur, c'est-à-dire le résultat renvoyé par l'exécution du code contenu dans le bloc. Nous verrons que ces objets seront d'une grande utilité lors de la recherche.

```
| aBlock |
aBlock := [:a :b |
              Transcript
                       show: a asString;
```

```
                        cr;
                        show: b asString.
                a + b.].
aBlock value: 5 value: 6.
```

Sur cet exemple de code, on peut voir que l'on déclare une variable locale nommée `aBlock`, qui est instanciée à la ligne suivante par un nouveau bloc (délimité par les crochets), qui prend 2 paramètres (`a` et `b`), qui affiche ces deux paramètres sur le `Transcript` (qui ressemble à une fenêtre de log et qui est défini comme une variable partagée) en lui envoyant le message `show:` qui prend comme paramètre une chaîne de caractères (pour transformer n'importe quel objet en chaîne de caractères, nous pouvons utiliser la méthode `asString`). Les ";" permettent, comme nous venons de le voir, d'envoyer plusieurs messages vers le même objet (ici `Transcript`). Le deuxième message envoyé à cet objet est `cr`, qui n'accepte aucun paramètre et qui permet, simplement, de revenir à la ligne sur la fenêtre de log. La dernière instruction du bloc est `a + b`. Cette instruction renvoie, comme on peut s'en douter, la somme des 2 objets. Il faut savoir qu'un bloc renvoie toujours comme valeur le résultat de la dernière instruction exécutée. La valeur de ce bloc sera donc le résultat de l'addition.

A ce moment de l'exécution, le bloc déclaré dans la variable `aBlock` n'a pas encore été évalué. La dernière instruction de cet exemple va évaluer ce bloc avec les arguments `5` et `6`.

L'exécution complète de ce bout de code donnera donc comme résultat l'affichage suivant sur le `Transcript` :

```
5
6
```

et renverra la valeur 11.

2.2 SOUL

Acronyme de *Smalltalk Open Unification Language*, *SOUL* peut être vu comme un langage du type de Prolog où les symboles ont disparu au profit d'objets, c'est-à-dire qu'il est possible d'écrire des faits et des règles tels qu'en Prolog mais avec de nouveaux termes et prédicats qui, une fois évalués, sont reliés à un objet de *Smalltalk*[WBM03]. La librairie *LiCoR* utilise cette symbiose entre *SOUL* et *Smalltalk* pour offrir une suite de règles décrites en *SOUL* permettant des raisonnements puissants sur du code *Smalltalk*. Cette partie du chapitre se concentre sur l'explication de ces nouvelles notions.

Ce langage a été construit dans l'approche du Declarative Meta Programming, expliquée ci-après.

2.2.1 Declarative Meta Programming

Derrière le terme *Declarative Meta Programming* (ou DMP pour les intimes) se cache une manière déclarative de construire des programmes raisonnant sur d'autres programmes. Décomposons ce terme pour mieux le comprendre.

La technique de programmation dite *déclarative* ne se focalise uniquement que sur la question "Que recherche-t-on ?" et non sur : "Comment va-t-on trouver la solution ?". C'est-à-dire que l'on va *déclarer* ce que le programme est supposé faire plutôt que de *définir* les étapes nécessaires à la résolution du problème posé. De tels programmes sont généralement utilisés dans la méta-programmation, le traitement d'un langage, le raisonnement sur une base de connaissances (tel que SQL), l'unification et le backtracking.

"Meta programming" ou la *méta-programmation* en français, permet de raisonner sur d'autres programmes mais aussi de les manipuler. Les exemples les plus connus sont les interpréteurs et les compilateurs. En effet, de tels programmes lisent en entrée d'autres programmes pour définir leur propre comportement (exécuter des actions ou créer un exécutable par exemple). Un *langage* de méta-programmation est un langage dans lequel on peut écrire des méta-programmes et qui permet de manipuler et de raisonner sur les éléments du langage dit *de base*.

Declarative Meta Programming est donc une combinaison d'un méta-langage avec la puissance des langages déclaratifs permettant de raisonner sur un langage de base. Ainsi, les programmes de bas niveau (c'est-à-dire ceux étudier par le DMP) sont exprimés par des termes, des faits et des règles dans le niveau haut. A l'inverse, les programmes de haut niveau (c'est-à-dire ceux du DMP) peuvent manipuler et raisonner sur les programmes de niveau bas.

Dans notre cas, nous utilisons le DMP avec un langage de base orienté objet : *Smalltalk* est utilisé comme langage de base et *SOUL* comme méta-langage déclaratif (car Prolog est un langage logique, qui est déclaratif).

2.2.2 Environnement

SOUL a été implémenté sous *VisualWorks Smalltalk* 3.0 et a ensuite évolué pour s'intégrer dans chaque version de *VisualWorks Smalltalk*. Il fait partie intégrante de l'environnement de *Smalltalk*. *SOUL* n'est pas seulement un langage ; c'est aussi un environnement avec 2 interfaces principales permettant d'entrer des clauses, des règles, de les gérer et d'exécuter des requêtes.

2.2.3 Syntaxe

Le langage *SOUL* est essentiellement le même que Prolog. Il met à disposition des constantes, des variables, des listes et un "cut". De plus, pour garantir la symbiose de *SOUL* avec *Smalltalk*, le langage permet de manipuler des objets, des termes et des clauses *Smalltalk*.

La syntaxe diffère de celle de Prolog sur plusieurs points :

- Les variables sont préfixées du signe `?` ;
- Les listes sont maniables par l'utilisation de `< >` ;
- `append(< >, ?Lst, ?Lst).` représente un fait ;
- `append(<?First | ?Rest>, ?Lst2, <?First | ?Lst>)`
  `if append(?Rest, ?Lst2, ?Lst).` déclare une nouvelle règle. On peut remarquer que le `if` remplace le `:-` de Prolog ;

– les arguments des têtes de règles peuvent être préfixées par les signes `+` et `-`. Le signe `+` signifie que la variable doit être une valeur bien définie dès l'entrée dans la règle. À l'inverse, le `-` indique que la valeur de la variable sera calculée au sein de la requête. Grâce à cette particularité de *SOUL*, on peut définir des règles plus rapides si on connaît certains paramètres par exemple ;
– `if append(?L1,?L2, <1,2,3,4,5>)` formule une requête.

Lorsqu'on parlera des règles dans le reste de ce mémoire, nous utiliserons parfois la notation plus courte `append/3` pour parler d'un prédicat de nom `append` acceptant 3 arguments.

2.2.4 Symbiose avec Smalltalk

Comme annoncé plus haut, *Smalltalk* est complètement intégré dans *SOUL*, particularité qui fait de *SOUL* un langage puissant pour manipuler des objets *Smalltalk*. Mais tout d'abord, pourquoi est-ce intéressant ?

Pour qu'un langage de méta-programmation logique puisse raisonner sur un certain programme de base, il est possible de construire une base de connaissances qui contiendrait l'ensemble du code source à étudier dans un format logique. Il est alors aisé de raisonner avec le langage logique sur les faits logiques de la librairie. Cependant, cette technique impose que tout programme étudié soit représenté deux fois, présentant, comme on peut s'en douter, des problèmes de synchronisation. Un autre problème se pose aussi quant à savoir jusque où faut-il transformer le programme de base en logique : s'il utilise d'autres librairies pour fonctionner, faut-il les transformer aussi en faits logiques ? Et qu'en est -il des libraires utilisées par ces librairies ? Pour contourner ces problèmes, la symbiose entre *Smalltalk* et *SOUL* a été présentée comme solution.[Wuy01]

Cette symbiose permet à la fois d'utiliser n'importe quel objet de *Smalltalk* comme un terme logique dans *Soul*, et d'écrire des expressions *Smalltalk* qui peuvent être paramétrées par des variables logiques.

Pour intégrer une valeur de *Smalltalk* dans du code *SOUL*, il suffit d'entourer le code *Smalltalk* par des crochets (`[ ]`). De plus, des raccourcis sont prévus pour alléger l'utilisation de *Smalltalk* dans *SOUL*. Pour les entiers, les crochets ne sont pas nécessaires. Nous pouvons donc écrire

```
if factorial(4,?X)
```

au lieu de

```
if factorial([4],?X)
```

Pour les symboles, un raccourci existe aussi :

```
if write(Symbol)
```

à la place de

```
if write([#Symbol])
```

Remarquez bien que `Symbol` n'est pas une variable mais une constante.

Nous pouvons donc mettre entre crochets n'importe quelle expression *Smalltalk* : non seulement des constantes (comme le `4` utilisé un peu plus haut) mais aussi toute expression qui peut être évaluée vers un objet. Ces expressions ont

la même syntaxe et la même sémantique que les termes *Smalltalk*. De plus, ces expressions peuvent être paramétrisées par des variables logiques (qui sont supposées être comprises par l'évaluation de l'expression logique incluant le code *Smalltalk*) qui sont substituées par leur valeur avant l'évaluation de l'expression.

Quelques exemples de cette symbiose :

- `if class([Array])`
- `allClasses([Smalltalk allClasses])`
- `plus(?x,?y, [?x +?y])`

Si les expressions *Smalltalk* sont utilisées à la place de clauses logiques (plutôt que des termes logiques comme on vient de le voir), elles doivent impérativement renvoyer `true` ou `false` après leur évaluation. Deux exemples illustrent cela :

- ```
  write(?test) if
        [Transcript show: (?text asString).
         true].
  ```
- ```
  smallerThan(?x, ?y) if
        atom(?x),
        atom(?y),
        [?x < ?y].
  ```
  qui peut s'écrire aussi comme ceci :
  ```
  smallerThan(+?x, +?y) if [?x < ?y].
  ```

2.2.5 LiCoR

SOUL offre, en plus de son langage, un ensemble de prédicats de base pour faire de la programmation logique standard. Pour faire du raisonnement avancé sur du code source *Smalltalk* et particulièrement sur les concepts de l'OO, il existe une librairie pour *SOUL* appelée *Library for Code Reasonning*, ou LiCoR.

Cette librairie offre des prédicats pour le typing (*Smalltalk* est dynamiquement typé, LiCoR permet de les reconstruire plus facilement), pour les conventions de code, pour manipuler les design patterns et pour l'UML. N'ayant pas la place dans ce mémoire de décrire l'ensemble des prédicats formant la librairie LiCoR, en voici seulement quelques exemples typiques :

- `class(?aClass)` renvoie vrai si `?aClass` est une classe définie dans *Smalltalk* ;
- `methodWithNameInClass(?method,?selector,?class)` est vrai s'il existe un objet `?method` qui possède comme nom `?selector` dans la classe `?class` ;
- Pour connaître le nombre d'appel au sein d'une méthode, il suffit d'utiliser la requête `sendCount/2` ;
- Pour compter le nombre de lignes de code comprises dans une méthode, nous pouvons utiliser `statementCount/2` ;
- Des prédicats un peu plus complexes existent aussi. `sameStatements/4` permet par exemple de retrouver toutes les mêmes lignes de code qui se retrouvent recopiées à deux endroits différents (par simple copier-coller).

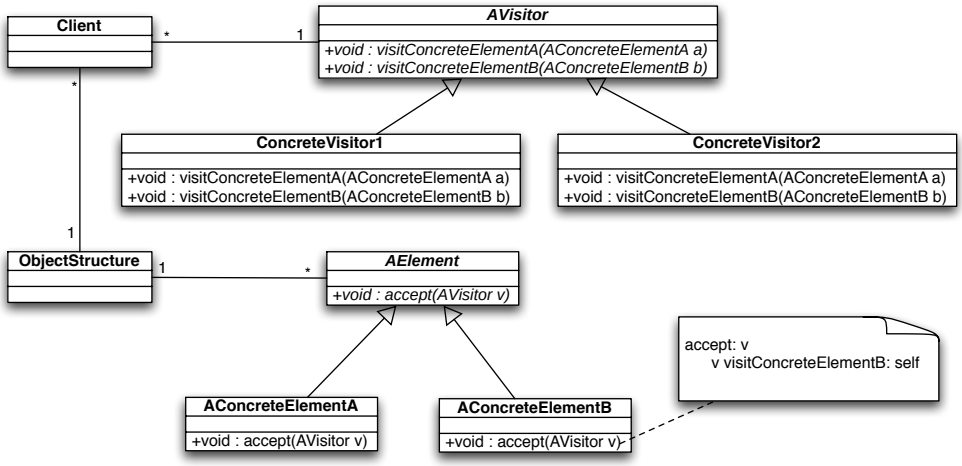

FIG. 2.1 – Structure proposée par le pattern du *Visiteur*

SOUL est donc bien le langage et LiCoR, la librairie que nous allons utiliser pour raisonner sur du code *Smalltalk*.

Pour expliquer LiCoR de manière plus approfondie, nous avons choisi de définir avec des règles logiques un pattern de design appelé "Visiteur". Nous allons d'abord expliquer ce pattern et ensuite, montrer ce que LiCoR peut nous apporter.

2.2.6 Le pattern du visiteur

Un *pattern de design* (ou *design pattern* en anglais) est une réponse générique et élégante à un problème récurrent de design[GHJV94]. Celui du *visiteur* permet de découpler le comportement et les données d'objets. Grâce à cela, on peut changer le comportement d'une structure sans changer les classes des éléments de cette structure.

L'utilisation de ce pattern est conseillée lorsque les opérations sur un ensemble d'éléments changent beaucoup et surtout pas si les éléments à visiter changent énormément.

2.2.6.1 Le fonctionnement

Le pattern du visiteur propose comme solution la structure expliquée par le graphe UML de la figure 2.1. Chaque élément de la structure à visiter doit comprendre une méthode `accept:` qui prend comme paramètre un objet du type "Visitor" (la traduction de "visiteur" en anglais). Cette méthode va alors envoyer un message à cet objet Visitor en lui passant en paramètre une référence vers lui-même. Le visiteur exécute alors son algorithme pour cet élément. Ce processus est appelé communément *Double Dispaching* : l'objet de la structure fait appel à un visiteur, en se passant lui-même comme paramètre, et le visiteur exécute cet algorithme sur cet objet. Nous pouvons donc remarquer que l'appel

dépend du type de visiteur et de l'objet à visiter, et non pas juste d'un seul élément (d'où le nom de *Double Dispatching*) [GHJV94].

Tout le mécanisme de découplage des données et des opérations de ce pattern repose donc sur la méthode `accept`: des éléments définis par les sous-classes de *AElement*. Les opérations sont définies dans les différents visiteurs, séparées de la structure de données à visiter.

Par exemple, on peut rencontrer des visiteurs dans les compilateurs afin qu'ils parcourent l'arbre de "parsing". Chaque visiteur effectuera une action particulière sur chaque nœud de l'arbre. Si une nouvelle opération a besoin d'être définie, il suffit de créer un nouveau visiteur qui parcourra l'arbre, les classes définissant ce dernier restant inchangées.

2.2.7 Utilisation concrète de LiCoR

Nous pouvons utiliser LiCoR pour raisonner sur le pattern de visiteur. Examinons la règle suivante :

```
(1) visitor(?visitor,?element,?accept,?visitSelector)
(2)  if class(?visitor),
(3)      classImplements(?visitor,?visitSelector),
(4)     class(?element),
(5)     methodWithNameInClass(?acceptBody,?accept,?element),
(6)     argumentsOfMethod(?acceptArgs,?acceptBody),
(7)     statementsOfMethod(<return(send(?v,?visitSelector,?visitArgs))>,
           ?acceptBody),
(8)     member(variable([#self]),?visitArgs),
(9)    member(?v,?acceptArgs)
```

Comme on peut le voir à la ligne (1), cette règle prend 4 paramètres :

1. `?visitor` : la classe du visiteur ;
2. `?element` : la classe de l'élément visité par `?visitor` ;
3. `?accept` : la méthode utilisée par la classe `?element` pour accepter le parcours du visiteur ;
4. `?visitSelector` : le message envoyé à la classe `?visitor` pour exécuter l'algorithme du visiteur sur l'élément `?element`.

Le moteur de résolution de règles va rechercher toutes les classes de visiteurs (ligne (2)) et d'éléments à visiter (4) (à moins qu'elles soient définies par les paramètres) qui répondent à la condition suivante : une classe d'éléments doit posséder une méthode (5) qui prend comme paramètre un objet (6 et 9) sur lequel sera envoyé un message (7) défini dans une classe du visiteur `?visitor` (3) prenant comme paramètre l'élément lui-même (8).

Grâce à cette règle, nous pouvons maintenant poser des questions au moteur de recherche de *SOUL*. On peut par exemple demander si deux classes sont reliés par une relation de visiteur.

```
if visitor([BlockAnalyzer], [ReturnNode], [#nodeDo:],[#doReturn:value:])

--> succeeds in 26 milliseconds
```

Cette autre requête nous donne les sélecteurs utilisés par ces mêmes classes `BlockAnalayzer` et `ReturnNode` pour cette relation de visiteur.

```
if visitor([BlockAnalyzer], [ReturnNode], ?acceptSel, ?visitSel)

--> succeeds with solutions in 773 ms
```

Cette dernière requête permet de savoir si la classe `BlockAnalyzer` est un visiteur et si oui, pour quelles classes d'éléments et via quelles méthodes.

```
if visitor([BlockAnalyzer], ?element, ?acceptSel, ?visitSel)
```

2.3 Conclusion

Nous avons vu dans ce chapitre les deux langages que nous utiliserons pour effectuer la recherche demandée.

Le prochain chapitre continue à expliquer le contexte dans lequel nous nous plaçons en introduisant le modèle intentionnel, base des outils de documentation co-évolutive que nous devons créer et valider.

Chapitre 3

Le modèle intentionnel

Maintenant que nous avons pris connaissance des langages utilisés dans cette recherche grâce au précédent chapitre, nous allons définir dans celui-ci le modèle intentionnel, modèle qui, à partir de vues et relations définies au-dessus d'un code source orienté objet, permet de faire co-évoluer un code et la documentation l'accompagnant.

Cette introduction des vues intentionnelles est inspirée des articles suivants [MMW02, TBKG03].

3.1 Introduction

La maintenance et l'évolution du code source d'un grand système informatique est assez complexe à gérer. En effet, les développeurs sont dans l'obligation de comprendre complètement la structure interne du programme considéré avant d'effectuer quelque modification que ce soit, tout en étant sûr de ne pas en modifier son comportement. Cette action est facilitée par la documentation jointe au code source. Généralement, celle-ci est écrite soit avant l'implémentation (lors de l'analyse du problème), soit à un moment où le code source est plus ou moins stable. Lors de changements dans le code, la documentation n'est pas mise à niveau *automatiquement*, les développeurs ne prenant pas de temps superflu sur leurs calendriers déjà assez serrés pour le faire. De modifications en légères mises-à-jour du code, la documentation devient très rapidement obsolète et s'en servir pour retrouver des informations qui resteraient correctes (même suite à toutes les modifications apportées au code) prendrait un temps considérable et serait, très probablement, source d'erreurs.

Ce que propose le modèle de vues intentionnelles est de palier à ce problème en documentant des structures ou des patterns et d'*automatiquement* mettre à jour cette documentation lorsque qu'un changement est effectué. En effet, comme nous le verrons, un mécanisme simple et puissant permet de connaître quelles parties de la documentation ne sont pas cohérentes avec le code.

Les résultats produits par ce modèle permettent d'améliorer la compréhension, la modularité et l'exploration du code en présentant comme documentation un ensemble de vues regroupant des *entités de code* (classes, méthodes, variables, namespaces, ...) qui s'accordent sur un même critère. Chaque vue in-

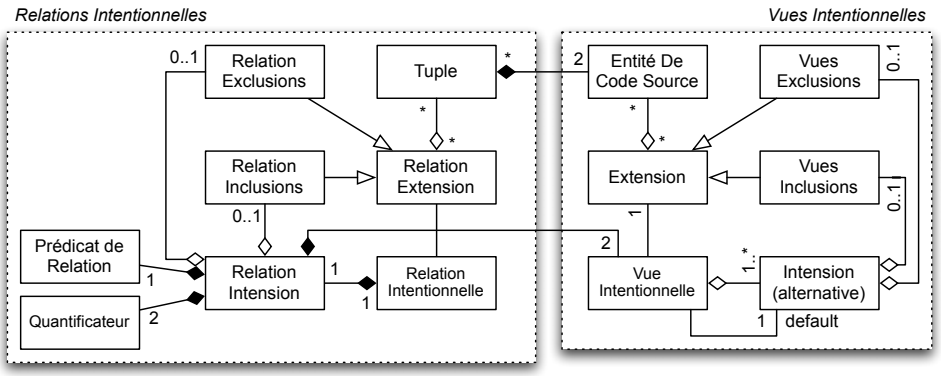

FIG. 3.1 – Diagramme UML du modèle intentionnel

tentionnelle correspondra à un critère important, tel qu'un pattern (de visiteur par exemple, voir page 11) ou l'ensemble des méthodes accédant à une variable de classes, portant sur tout le code source du programme.

Ce modèle est composé de plusieurs concepts différents définis ci-après et repris dans la figure 3.1.

3.2 Vue intentionnelle

Une vue intentionnelle est une *vue* dans le sens où elle fournit un sous-ensemble d'informations sur le code source et qu'elle est définie au-dessus du code source. La déclaration d'une vue laisse donc le code complètement intact. Elle est *intentionnelle* car elle décrit les caractéristiques des éléments qui doivent composer la vue, non pas de manière énumérée mais sous forme d'une demande abstraite et intuitive qui exprime clairement l'intention de la vue. Cette requête est nommée *intention*. Comme nous le verrons, dans les implémentations courantes, elle peut être donnée soit par une requête logique définie en *SOUL*, soit par un bloc de code *Smalltalk*. Une vue intentionnelle possède toujours au moins une vue alternative, qui est celle appelée *vue alternative principale*. C'est cette vue qui contiendra l'intention de la vue intentionnelle, cette dernière n'étant que le concept d'une vue sur le code.

L'intention d'une vue intentionnelle n'est calculée que lorsqu'on la demande. Ainsi, l'*extension*, c'est-à-dire le calcul de l'intention sur le code source, reflète toujours l'ensemble *courant* d'entités de code représenté par la vue sur le code.

3.3 Vues alternatives

Chaque vue intentionnelle est composée de vues dites *alternatives* qui décrivent, selon des critères différents, le même ensemble d'entités de code. Il existe donc une relation intrinsèque d'égalité entre les ensembles d'entités obtenus par chaque vue alternative d'une même vue intentionnelle. Cette relation sera

nommée dans le reste de ce document *relation interne*. Sa cohérence est vérifiée si toutes les vues alternatives sont effectivement égales "extentionnellement" entre elles. Cette cohérence sera nommée *cohérence interne*.

Pour illustrer cela, prenons un exemple. Lors de la production de code, une convention simple de nomination a été établie : toute méthode classée dans le protocole "action" doit avoir un sélecteur(un nom) qui commence par le mot 'execute'. Nous pouvons alors décrire une vue intentionnelle comprenant deux vues alternatives décrites en *SOUL*.

```
selectorOfClassInProtocol(?selector,?class, action),
methodWithNameInClass(?entity,?selector,?class)
```

FIG. 3.2 – Vue alternative basée sur le protocole

```
selectorOfClassInProtocol(?selector,?class,action),
methodWithNameInClass(?entity,?selector,?class),
startsWith(?s,['execute'])
```

FIG. 3.3 – Vue alternative basée sur le protocole et le nom

Remarque importante Lorsque les requêtes sont exprimées dans le langage *SOUL*, la variable logique `?entity` permet de récupérer le résultat de la requête.

La vue alternative exprimée par la figure 3.2 permet de prélever toutes les méthodes catégorisées dans le protocole "action", et ce, par défaut, dans tout le namespace *Smalltalk*(c'est-à-dire l'ensemble du système). La seconde vue alternative (Fig. 3.3) reprend également les dites méthodes mais en ne gardant que celles qui commencent par 'execute'. Pour que la règle de conception soit respectée, les deux vues alternatives doivent donner exactement le même sous-ensemble de méthodes.

Le modèle supporte également l'exclusion explicite (respectivement inclusion) d'une entité d'une vue alternative. Cela permet de définir de manière énumérée des éléments qu'il faut exclure (respectivement inclure) de la vue. Ces *cas de déviations* pourrait être utilisés dans l'exemple précédent, si une méthode appartient au protocole "action" mais ne commence pas par le mot 'execute'. Si une telle méthode existe, alors la cohérence interne de la vue intentionnelle est invalide. Pour qu'elle devienne cohérente, nous pouvons exclure explicitement cette méthode de la première vue intentionnelle, indiquant que cette méthode est un cas exceptionnel de déviation. Lors du calcul de l'extension de la vue, l'intention sera alors évaluée sur le code, donnant un ensemble de méthodes dans lequel la méthode en question sera retirée. L'extension de la première vue alternative comprendra donc bien les mêmes entités de code que l'extension de la seconde, et la cohérence interne sera respectée.

3.4 Les relations

Des relations peuvent être ajoutées afin d'établir des contraintes entre les différentes vues intentionnelles. Ces relations sont définies entre deux vues intentionnelles d'un même layer (notion définie à la section 3.6) et sont caractérisées par deux critères :

1. un prédicat qui définit la relation entre les deux vues, telle que l'égalité ;
2. deux quantificateurs logiques (un par vue) qui définissent le nombre d'éléments à prendre dans chaque vue pour valider le prédicat, tels que "pour tout", "il existe".

De manière plus rigoureuse, seules les relations de la forme suivante sont acceptées :

$$\mathcal{Q}_1\ x \in Source : \mathcal{Q}_2\ y \in Target : x\ R\ y$$

où $\mathcal{Q}_1$ et $\mathcal{Q}_2$ sont soit des quantificateurs logiques $\forall$, $\exists$, $\exists!$, $\nexists$, soit des quantificateurs numériques comme *some*, few, *many* or *most*[1]. $Source$ et $Target$ représentent des vues intentionnelles et R est un prédicat binaire raisonnant sur les entités de code source (dénotés par x et y) contenues dans ces vues.

Une relation est dite cohérente si la relation est vérifiée. Cette cohérence sera appelée *cohérence de relation*.

Prenons un dernier exemple pour mieux comprendre. Si nous avons défini deux vues intentionnelles, l'une (appelée *Vue1*) reprenant un ensemble de méthodes et l'autre(*Vue2*) d'autres méthodes, nous pourrions déclarer une relation intentionnelle entre ces deux vues qui contraint toute méthode de la première vue à appeler au moins une méthode de la seconde. Cette relation peut être représentée de la manière suivante :

$$\forall\ x \in Vue1 : \exists\ y \in Vue2 : x\ appelle\ y$$

Le prédicat *appelle* devra alors être défini en *SOUL* ou en *Smalltalk*.

De la même manière que dans les vues, le modèle intentionnel accepte les cas de déviations d'une relation en donnant la possibilité d'inclure ou d'exclure explicitement des tuples de celle-ci. En reprenant l'exemple précédemment cité, si il existe une méthode m de *Vue1* qui n'appelle pas une méthode de *Vue2*, on peut inclure explicitement le tuple (m, *) dans la relation, apportant alors la relation manquante.

3.5 La hiérarchie des vues

Il est possible de créer une hiérarchie entre les vues. La sémantique d'une sous-vue est la suivante : les éléments qui composeront l'extension de la sous-vue seront toujours un sous-ensemble de l'extension de la vue située au-dessus.

Supposons que nous déclarons en *Smalltalk* une première vue comme ceci :

```
1 to : 10
```

[1] *some* : un peu, few : quelques, *many* : beaucoup, *most* : la plupart. Ces quantificateurs sont définis en termes d'un minimum et d'un maximum d'éléments pour lesquels la condition doit être respectée.

Ce code représente une collection comprenant les 10 nombres compris entre 1 et 10. Définissons maintenant une sous-vue de celle-ci :

```
5 to : 15
```

Celle-ci exprime donc une vue comprenant les nombres de 5 à 15.

Si nous calculons l'extension de la sous-vue, celle-ci sera uniquement composée des nombres de 5 à 10, intersection de la sous-vue avec la première vue.

3.6 Les layers

Les vues intentionnelles peuvent être regroupées dans des *layers*, c'est-à-dire des sortes de répertoires où les vues ne sont connues que dans celui-ci. Deux vues portant le même nom dans deux layers différents ne posent donc aucun problème. Cette séparation des vues peut s'avérer très pratique lors d'études de différents systèmes en même temps, ou pour caractériser plusieurs ensembles de vues différents sur le même système (comme par exemple, l'évolution de la documentation).

3.7 Système intentionnel

Nous définirons par *système intentionnel* l'ensemble des vues et des relations intentionnels définies pour un code particulier.

3.8 Conclusion

Nous avons pu, dans ce chapitre, parcourir ce modèle des vues et relations intentionnelles qui ouvrent des voies intéressantes pour maintenir une documentation à jour avec un code source donné. Le prochain chapitre décrit les outils graphiques créés au-dessus de ce modèle afin de pouvoir le manipuler.

Chapitre 4

IntensiVE

Les outils de base offerts avec les vues intentionnelles permettent de définir des vues, leurs vues alternatives, des relations et de vérifier les cohérences internes et de relations. Quatre outils ont été conçu pour cela : the Intensional View Editor, the View Consistency Checker, the Relation Editor et the Relation Checker. Ceux-ci constituent l'environnement *IntensiVE*, **Intensi**onal **V**iew **E**nvironment. Ce chapitre décrit rapidement l'utilisation de ces outils.

4.1 Star Browser 2

Ces outils sont intégrés dans un environnement de développement plus large nommé *StarBrowser* 2 [WD04]. Celui-ci a été développé afin d'offrir un environnement facilement extensible utilisant le modèle des classifications. *StarBrowser* 2 permet de parcourir un environnement *Smalltalk* et de classifier n'importe quel élément de code rencontré. Ainsi, des méthodes, des classes intéressantes peuvent être regroupées dans une classification par simple *drag and drop* afin de les garder rapidement sous la main. *StarBrowser* 2 gère aussi les classifications plus automatiques regroupant un ensemble d'objets possédant une même caractéristique. Nous pouvons, par exemple, créer une classification qui comprendrait l'ensemble des classes dont le nom commence par tel ou tel préfixe. Ces éléments seront alors calculés à chaque fois que l'utilisateur le demande.

Les classifications comprises dans le système courant sont affichées dans l'arbre de gauche de l'interface de *StarBrowser* 2. Chaque sous-arbre est une classification. Lorsque l'on demande d'ouvrir un arbre, les éléments composant son sous-arbre sont calculés et affichés en dessous de celui-ci.

Il existe un lien évident entre ces classifications et le modèle intentionnel. En effet, les classifications peuvent être vues comme des vues intentionnelles sans les cas de déviation. Que ce soit les classifications ou les vues, elles possèdent une intention qui sera calculées uniquement lorsque l'utilisateur le demande, leur permettant de rester toujours à jour avec le code.

Dans le cas qui nous occupe, les layers sont définis comme des classifications automatiques. A chaque demande, l'ensemble des vues et relations comprises dans le layer est calculé. On peut voir ces différentes classifications dans la fig. 4.1, panneau de gauche. La classification *SmallWikiViews* comprend 2

éléments : *Views* et *Relations*. *Views* est composée de l'ensemble des vues du layer SmallWikiViews et *Relations* de toutes les relations correspondantes.

Le panneau de droite est la pièce maîtresse de la puissance de *StarBrowser* 2. Elle va permettre d'accueillir les outils nécessaires à la visualisation ou à l'édition de ce que représente l'objet sélectionné dans l'arbre. Par exemple, lorsque l'on sélectionne une classe, l'éditeur de classe s'affiche. Cette fonctionnalité va nous permettre d'afficher un éditeur de vue intentionnelle (respectivement d'une relation) lors de la sélection d'une vue (d'une relation), comme nous allons le voir dans ce chapitre.

4.2 Implémentation revue et corrigée

L'implémentation du modèle a subi, au cours de ce mémoire, un changement fondamental. Dans la première version, les vues et relations intentionnelles étaient complètement enregistrées et calculées en *SOUL*. Chaque vue et relation étaient donc des prédicats logiques. La seconde version implémente ce modèle via des objects *Smalltalk*. Cette modification du code permet d'utiliser les apports du paradigme orienté-objet, dont l'héritage et le polymorphisme.

Ces transformations étant purement techniques, nous n'en parlerons pas plus au sein de ce mémoire. Il faut néanmoins remarquer que ce changement de code a eu des répercussions sur celui de l'outil graphique développé ici. Ces changements ont été plutôt bénéfiques grâce à l'utilisation de l'OO mais une perte de temps s'est faite ressentir. Désormais, nous ne parlerons dans ce mémoire que de la dernière version de ces outils.

Même si les mécanismes de sauvegarde et de calcul des vues et relations intentionnelles sont différents d'une version à l'autre, les interfaces n'ont pas été modifiées. Les différentes interfaces reprises dans ce chapitre sont donc valables dans les deux versions.

4.3 Intensional View Editor

Intensional View Editor (Fig. 4.1) est l'outil principal pour créer et manipuler les vues. Sur la copie d'écran, le panneau de gauche montre toutes les vues précédemment définies sous forme d'un arbre. Le reste de l'écran, le panneau droit, montre l'Intensional View Editor ouvert sur la vue nommée "Executed Methods". Cette vue regroupe toutes les méthodes responsables de l'exécution des actions sur les pages Wiki (l'édition, la suppression, l'enregistrement, ...). Toutes ces méthodes sont rangées dans le protocole "action". Nous pouvons fournir l'intention de cette vue de la manière suivante :

```
methodInProtocol(?entity, action).
```

Cette requête, écrite dans le langage *Soul*, fait correspondre toutes les méthodes dans le protocole "action" avec la variable logique libre `?entity`. Dans la figure 4.1(dans le panneau de gauche), nous pouvons noter que la vue "Executed Methods" est définie comme une sous-vue de "allSmallWikiMethods" (car

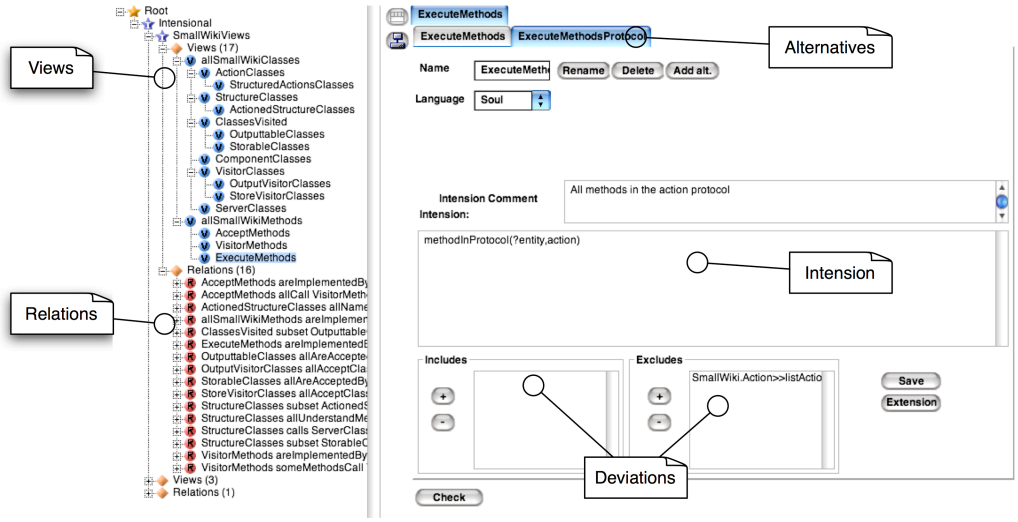

FIG. 4.1 – Intensional View Editor

elle se trouve dans son sous-arbre), qui contient, comme son nom l'indique, l'ensemble des méthodes définies dans le namespace "*SmallWiki*".

Cet outil supporte aussi l'exclusion (respectivement l'inclusion) explicite d'une entité de la vue. Dans l'exemple, la méthode `listActions`, implémentée dans la classe `Action`, fait partie de l'extension calculée de la vue, mais n'est pas réellement une méthode dite "d'exécution". Si bien que nous pouvons l'exclure explicitement de la vue, en la mettant dans sa liste nommée "Excludes". De manière analogue, nous avons une liste dite "d'inclusion" d'entités qui doivent être incluses dans la vue et qui ne satisfont pas l'intention.

Comme nous l'avons vu dans le chapitre précédent, le modèle intentionnel accepte la définition de plusieurs vues alternatives pour une vue intentionnelle particulière. La vue intentionnelle est dénotée dans cet outil par l'onglet situé tout en haut à gauche de l'interface. Les deux autres onglets représentent les vues alternatives. La vue alternative principale est représentée par le premier onglet. Elle est la définition principale de la vue intentionnelle et porte le même nom. La seconde vue alternative est celle éditée lors de cette copie d'écran. Des boutons en haut du panneau central permettent de renommer ou de supprimer la vue alternative courante. Le troisième bouton permet d'ajouter une nouvelle vue alternative à celles présentes.

Le bouton "Check" situé en bas du panneau central permet de tester la cohérence interne de la vue intentionnelle en cours d'étude en ouvrant une fenêtre de l'outil défini ci-après : View Consistency Checker.

4.4 View Consistency Checker

La figure 4.2 montre le *View Consistency Checker*. Cet outil est utilisé pour vérifier la cohérence interne, c'est-à-dire que les différentes descriptions alternatives d'une même vue sont "extensionnellement consistante", qu'elles produisent

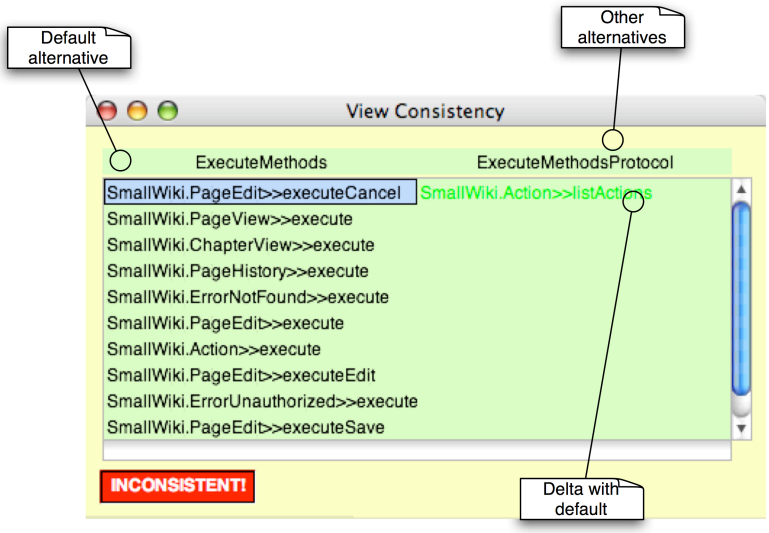

FIG. 4.2 – View Consistency Checker

la même extension. Lorsque cette contrainte est violée, l'outil fournit un retour approprié sur les entités en cause.

Pour illustrer ceci, nous utilisons la même vue que précédemment : "Execute Methods". En plus de l'intention déjà décrite précédemment, nous définissons une seconde description alternative basée sur le fait que le nom de chaque méthode dite "d'exécution" commence par la chaîne de caractères 'execute'. La figure 4.2 montre le résultat du test de la consistance interne de ces deux vues alterntives. Notons que ce test a été réalisé avant d'avoir explicitement exclu `listActions` de la seconde alternative. En fait, c'était précisément le résultat donné par le View Consistency Checker qui nous a motivé à regarder l'implémentation de cette méthode et à décider que c'était un cas de déviation.

Cet outil montre à l'utilisateur une colonne par description alternative de la vue considérée. La première colonne contient l'extension de l'alternative principale (par défaut celle-ci est la première alternative de la vue, mais une double-clic sur une colonne active la description alternative comme étant l'alternative principale) ; les autres colonnes contiennent la différence entre l'extension de la vue principale et celle de l'alternative représentée par la colonne. Si un élément n'existe pas dans l'alternative principale, il apparaît en vert. À l'inverse, tout élément existant dans l'alternative principale et pas dans la vue alternative définie par la colonne, apparaît en rouge.

4.5 Relation Editor

La figure 4.3 montre le *Relation Editor* en action. L'exemple repris par cette figure est la relation intentionnelle qui est représentée dans la forme canonique suivante (dont la syntaxe est définie en page 17) :

$$\forall\, x \in ExecuteMethods : \exists!\, y \in ActionClasses : x\ methodInClass\ y$$

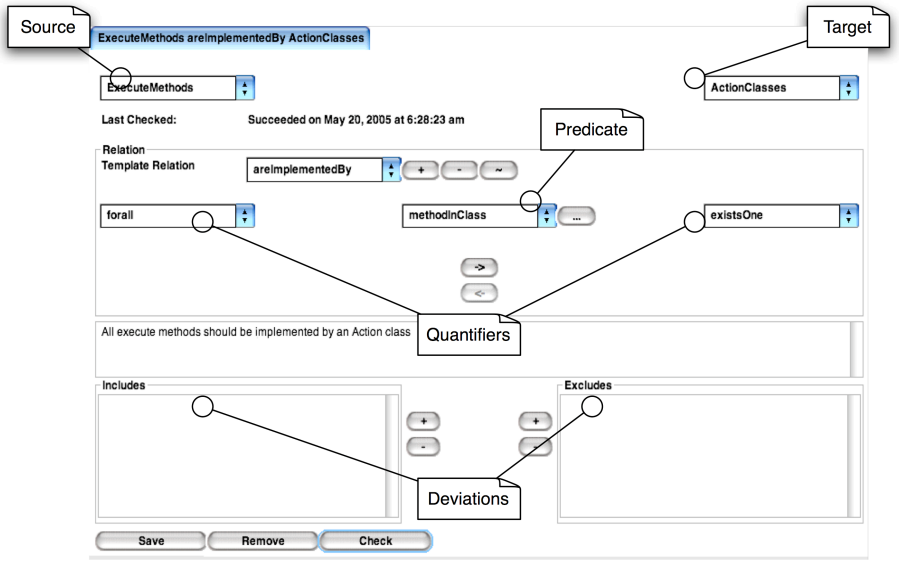

FIG. 4.3 – Relation Editor

c'est-à-dire toute méthode "d'exécution" est implémentée par une et une seule classe "d'action". Cette deuxième vue reprend simplement toutes les classes qui représentent des actions possibles sur les pages web de *SmallWiki*. Nous étudierons ces vues en détail lors de la validation de ces outils.

Cette interface offre deux moyens pour définir un prédicat binaire R sur des entités du code source. En plus de pouvoir définir un prédicat directement en *Smalltalk* (en utilisant un bloc *Smalltalk* qui prend 2 arguments et renvoie un booléen), l'utilisateur peut utiliser un prédicat *Soul* (typiquement en utilisant LiCoR, dont nous avons parlé à la page 10).

Comme l'*Intensional View Editor*, le Relation Editor supporte de manière explicite la déclaration des cas de déviation (inclusions et exclusions). Ceci permet à l'utilisateur de spécifier explicitement les tuples d'entités qui doivent être inclus ou exclus de la relation.

4.6 Relation Checker

Lorsque vous cliquez sur le bouton "Check" dans le *Relation Editor* (Fig. 4.3), la validité de la relation courante est vérifiée et l'utilisateur obtient à l'écran une instance du *Relation Checker* (Fig. 4.4). Non seulement cette fenêtre permet de voir aisément si une relation donnée est vérifiée ou pas, mais en plus elle donne un retour précis des tuples intervenant ou n'intervenant pas (et qui le devraient) dans la relation. La liste principale située au centre de la fenêtre reprend les tuples respectant la relation. La liste en bas à gauche reprend les entités qui ne sont pas dans le *domaine* (c'est-à-dire l'ensemble des entités de la vue source qui ne valident pas dans la relation), tandis que celle de droite affiche celles qui ne sont pas dans le *range* (idem pour la vue cible).

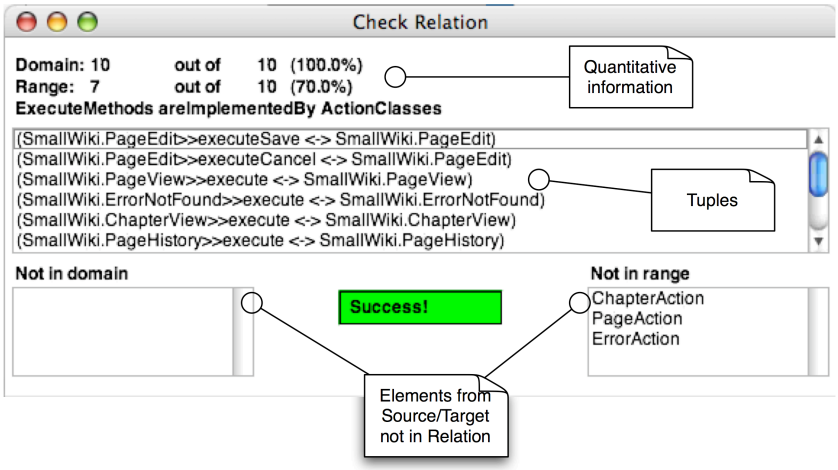

FIG. 4.4 – Relation Checker

4.7 Conclusion

Les quatre outils présentés ici ont été conçus pour utiliser pratiquement le modèle intentionnel. Ils permettent de manipuler un ensemble de vues et de relations intentionnelles définies au-dessus d'un code *Smalltalk*. Ceux-ci ne permettent pas d'utiliser le modèle défini dans le chapitre précédent au maximum. Car, comme nous allons le voir dans le chapitre suivant, des manques se font ressentir lors de l'utilisation pratique de ceux-ci.

Chapitre 5

Évaluation pratique du modèle intentionnel

Le but du modèle des vues et relations intentionnelles décrit dans les chapitres précédents est de nous offrir un outil puissant pour vérifier des contraintes structurelles sur du code *Smalltalk*. À présent, la question que nous nous posons est la suivante : comment utiliser cette approche dans des cas réels ? Nous avons alors relevé deux choses importantes. Premièrement, des manques se font sentir lorsque l'on se pose des questions générales sur un système intentionnel. Par exemple, il est très fastidieux pour le moment de répondre à la question : "quelles sont toutes les vues intentionnelles inconsistantes actuellement dans l'ensemble du système intentionnel ?". Deuxièmement, aucune validation de ce modèle et de ses outils n'a encore été effectuée. Elle est pourtant nécessaire pour connaître l'utilité du modèle et l'"utilisabilité" (c'est-à-dire le caractère utilisable) des outils.

Dans le temps imparti à ce mémoire, nous avons pu réaliser un outil graphique palliant le premier problème. Cet outil a été rapidement créé si bien que, lors de la phase de validation, nous avons pu l'inclure également.

Ce chapitre définit les problèmes rencontrés et étudie la solution apportée. Le chapitre suivant valide cette solution. Le chapitre 7 effectue la validation du modèle intentionnel ainsi que des cinq outils par une étude sur un cas concret.

5.1 Imperfections des outils

Le but du modèle des vues et relations intentionnelles, défini dans le chapitre 3, est d'offrir un outil puissant aux développeurs pour garder à jour la documentation d'un code source orienté objet. Cette documentation se compose de vues et relations décrivant des contraintes architecturales, qui peuvent se révéler assez complexes. Il est important de souligner que cette documentation est dynamique sur le code, c'est-à-dire que les extensions des vues sont toujours recalculées pour correspondre à la version courante du code. De ce fait, la cohérence interne de chaque vue intentionnelle ainsi que la cohérence de chaque relation sont remises en cause à chaque changement de code. C'est ce mécanisme qui permet une automatisation du processus de co-évolution du

code et de la documentation.

Comme nous l'avons vu dans le chapitre 4, les quatre outils graphiques permettant l'utilisation pratique de ce modèle nous permettent d'ajouter, de modifier, de voir et de supprimer une vue intentionnelle, une vue alternative ou une relation intentionnelle. De plus, ils permettent de vérifier la cohérence interne d'une vue intentionnelle (c'est-à-dire la cohérence entre les vues alternatives) ainsi que celle des relations intentionnelles. Remarquez bien que les outils actuels permettent de faire toutes ces opérations sur une seule vue (ou une seule relation) à la fois.

L'idée primaire de ce modèle, c'est-à-dire de documenter une architecture et vérifier que les contraintes définies par cette documentation soient respectées en tout temps, n'est donc pas exploitée à cent pour cent. En effet, le seul moyen qu'offrent ces outils pour connaître les vues ou les relations non cohérentes d'un système est de vérifier la cohérence de chacune d'entre elles, les unes après les autres. Ce procédé est inconcevable sur une documentation à grande échelle.

Nous pouvons aussi remarquer que dans les outils actuels, il n'est pas simple de savoir quelles sont les relations qui impliquent telle ou telle vue. Une nouvelle fois, cette question peut être résolue en examinant chaque relation définie dans le système, processus bien trop lourd sur de grands codes.

Ce sont pourtant ces deux types de questions qui permettent de trouver les failles d'un code ne répondant pas aux règles établies. Elles doivent donc être traitées de la manière la plus efficace possible.

5.2 Utilisation graphique et globale du modèle intentionnel

De prime abord, une question se pose : faut-il un nouvel outil ou améliorer les outils existants ? Si nous regardons de plus près les buts des 4 outils (la création, l'édition, la suppression d'une vue ou d'une relation), nous pouvons remarquer que ces outils sont intrinsèquement unitaires sur l'objet manipulé. En effet, une et une seule vue ou relation peut être éditée à la fois. Faut-il les rendre moins unitaires ? Leur fonction étant unitaire à la base, les changer serait complexifier leur fonction et voire même, les rendre inutilisables. C'est pour ces raisons que nous pensons qu'il est préférable de créer un nouvel outil offrant une vue plus globale du système, tout en offrant des facilités pour accéder aux quatre anciens outils.

L'outil à développer a donc plusieurs buts. D'une part, il serait intéressant que celui-ci nous montre quelles sont les vues et relations non cohérentes du système. De cette manière, la prise rapide de connaissance des problèmes issus de la documentation permettra une résolution de ceux-ci avec une grande efficacité. D'autre part, une représentation du système intentionnel global ou partiel faciliterait la compréhension des relations internes ou externes entre les vues.

D'autres informations utiles à la compréhension à la fois des vues intentionnelles et des problèmes couverts par les incohérences, pourraient aussi nous être délivrées. Il serait par exemple intéressant de savoir quelles sont les vues

alternatives qui sont les plus différentes de leur vue intentionnelle. Nous pourrions alors les examiner en premier lieu et résoudre ainsi les plus gros problèmes d'abord.

La solution présentée dans ce chapitre est une représentation graphique munie de métriques permettant de connaître les vues et les relations importantes, celles qui posent des problèmes, celles qui sont incohérentes et permettant également une meilleure compréhension de la documentation du système intentionnel grâce à une vue globale du système.

5.2.1 La représentation graphique

Pour résoudre les problèmes précédemment cités, la solution d'une représentation visuelle sous forme de graphes des vues et relations intentionnelles a été choisie pour plusieurs raisons :

1. Une représentation sous forme de graphe permet de condenser beaucoup d'informations sur peu de place ;
2. Cette représentation peut être exportée sur des supports papiers, afin d'en discuter entre développeurs ;
3. Des métriques peuvent être ajoutées afin d'apporter de nouvelles informations sur les composants du graphe ;
4. Le système hiérarchique des vues intentionnelles convient parfaitement pour en faire une représentation sous forme d'un arbre, les relations étant alors représentées par des liens entre ces arbres, reliant les différentes vues faisant partie de la relation intentionnelle.

Dans cette solution, une vue (intentionnelle ou alternative) sera représentée par un rectangle où le nom de cette vue est affiché en son centre. Comme le nom des vues est laissé au libre arbitre de l'utilisateur du modèle intentionnel, la longueur des différents noms ne sera que très rarement constante. Nous avons donc décidé que la largueur du rectangle entourant ce nom sera proportionnelle à la longueur de ce nom. En d'autres termes, le rectangle sera ajusté au nom de la vue qu'il représente. Par contre, la hauteur sera toujours identique.

Le modèle intentionnel est composé de vues et de relations. Les vues sont déclarées soit à la base du système intentionnel, soit comme des sous-vues d'autres vues. Nous avons donc une hiérarchie de vues intentionnelles. Une représentation typique d'une telle hiérarchie est celle sous forme d'arbre. En effet, nous n'aurons que des vues ayant au plus un parent dont au moins une n'en a pas. Aucun cycle ne peut apparaître. Les relations sont alors vues comme des liens entre les différents éléments composant les arbres.

La figure 5.1 nous montre le graphe d'un système composé de 6 vues, avec 2 vues de base (sans parent). Remarquez bien que les relations intentionnelles sont définies entre les différents éléments du système, que ce soit dans une autre hiérarchie de vues ou pas (relation `relation3` entre les vues `Vue1.1` et `Vue1.1.1`).

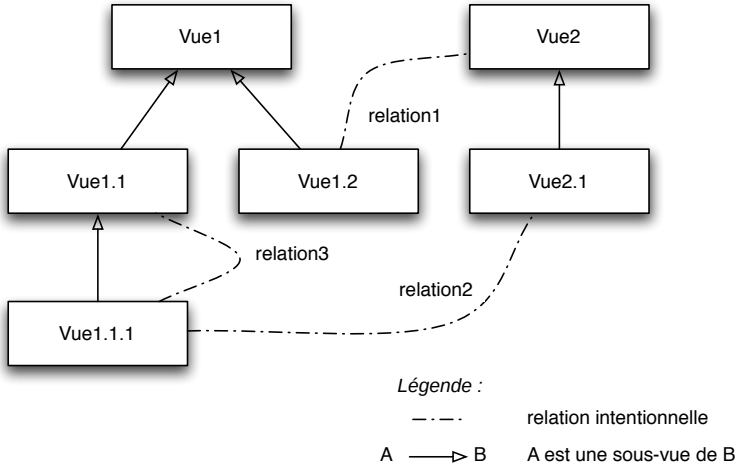

FIG. 5.1 – Idée de la représentation du système intentionnel

5.2.2 Métriques

Le graphe tel que nous l'avons entrevu dans la figure 5.1 permet de répondre à un des manques signalés plus haut. En effet, nous pouvons répondre à la question : *"Quelles sont les relations dans lesquelles* `Vue2` *intervient ?"* et ce, sans parcourir toutes les vues les unes après les autres.

Désormais, nous voudrions utiliser des métriques pour venir enrichir ce graphe. Les métriques que nous allons utilisées dans ce mémoire sont simplement une mesure quantitative sur les vues et relations intentionnelles afin d'en extraire certaines propriétés intéressantes.

5.2.2.1 Définition des métriques

Lorsque l'on se penche sur les vues intentionnelles, plusieurs métriques apparaissent naturellement. La première est le nombre d'objets composant l'extension de la vue alternative principale d'une vue intentionnelle. Cette métrique nous montre bien évidemment la généralité d'une vue : au plus, une vue contiendra d'éléments, au plus sa définition se rapproche d'un trait commun à un nombre important d'objets de code.

Une seconde métrique est la différence entre une vue alternative et sa vue alternative principale, dans le cas où la vue intentionnelle correspondante est composée au moins de deux vues alternatives. Si nous examinons cette métrique d'un peu plus près, nous pouvons remarquer que celle-ci doit être divisée en deux métriques pour qu'elle soit vraiment informative : en prenant comme modèle une des vues alternatives, nous pouvons calculer le nombre d'éléments en plus et le nombre d'éléments en moins pour chaque autre alternative par rapport à ce modèle. De cette manière, ces deux métriques nous informent sur le type d'erreurs qui se trouvent dans la documentation : si une vue alternative a un objet en plus dans son extension qu'une de ces congénères soit la première est

trop large et ne devrait pas le contenir, soit c'est la définition de la seconde qui trop stricte, soit l'erreur est logée dans le code (auquel cas, il est simple, via la définition des vues alternatives non cohérentes, de retrouver les objets du code qui ne sont pas correctement écrits).

Pour les relations, nous pouvons aussi prendre comme métrique le nombre d'éléments qui rendent la relation non cohérente. Cela permet de connaître les relations les moins respectées.

5.2.2.2 Utilisation graphique des métriques

Pour utiliser graphiquement ces métriques, nous devons les appliquer aux rectangles représentant les vues et aux lignes pour les relations. Nous utiliserons tout d'abord **la couleur** pour montrer les incohérences (et donc aussi de ce fait, les cohérences respectées) de l'ensemble du système. Les vues et relations incohérentes seront teintées de rouge pour qu'on puisse les remarquer le plus rapidement possible.

Ensuite, nous allons utiliser **la hauteur** ou/et **la largeur** des rectangles représentant les vues. La hauteur ou la largeur (ce choix peut être fait pour des raisons esthétiques par l'utilisateur) des rectangles des vues *intentionnelles* représentera le nombre d'éléments compris dans l'extension de la vue. Pour les vues *alternatives*, l'une des deux grandeurs représentera les éléments en plus et l'autre les éléments en moins dans l'extension de la vue alternative par rapport à celle de la vue alternative principale.

En ce qui concerne les relations, nous avons essayer d'appliquer la métrique définie plus haut sur la grosseur du trait représentant la relation. Le graphique en résultant ne fut vraiment pas très lisible et l'idée fut abandonnée.

Il nous reste encore une difficulté pour représenter les métriques sur les rectangles des vues intentionnelles. La longueur des noms des vues étant variable, les rectangles seront donc aussi d'une largeur variable. Nous devons donc trouver un moyen d'estomper ce manque d'homogénéité pour représenter la métrique sur la largeur du rectangle. L'idée est de ne pas changer le rectangle initial mais de lui accoler deux rectangles (un pour la métrique sur la largeur et l'autre pour la hauteur), comme le montre la figure 5.2. Chacun de ces deux rectangles aura une couleur différente. De cette façon, si nous définissons la métrique de la largeur d'une vue alternative comme étant le nombre d'entités en plus par rapport à l'extension de la vue intentionnelle s'y reportant, nous pouvons alors nous concentrer sur les rectangles bleus du graphique, le plus grand indiquant le nombre maximum d'éléments supplémentaires contenus dans une vue alternative par rapport à la vue principale sur l'ensemble système.

5.3 Comparaison avec d'autres solutions

Bien sûr, il existe sûrement beaucoup de solutions répondant à ces problèmes. Nous allons en discuter deux dans cette section.

Une solution simple est d'utiliser des boîtes sans nom et d'y appliquer des métriques de manière plus commune, c'est-à-dire que chaque vue serait

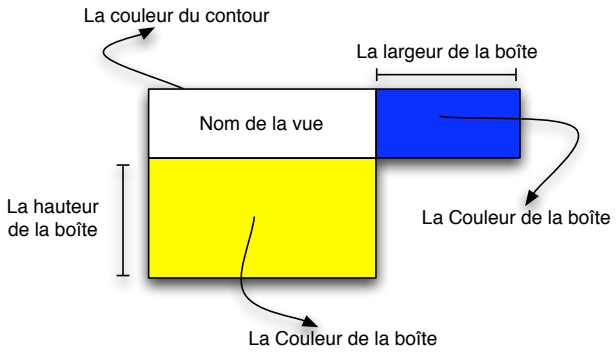

FIG. 5.2 – Les métriques des vues

représentée par un rectangle d'une certaine couleur sur lequel on pourrait appliquer des métriques sur la largeur et la hauteur. Nous perdons alors les noms, récupérés par une infos-bulle apparaissant lorsque l'on reste plus de une seconde sur un rectangle. Nous n'avons pas adopté cette solution pour la simple raison que le but du nouvel outil est d'afficher une vue globale du système intentionnel permettant une meilleure compréhension de la documentation. Si nous y retirons les noms, elle ne devient vraiment compréhensible que lorsque l'on a la souris en main et que l'on a une excellente mémoire pour relier chaque rectangle à un nom de vue mentalement. De plus, le nom de la vue n'est qu'une description partielle de ce qu'elle représente. Or, lors de la lecture de la documentation, il est impératif de comprendre à la fois la structure de la documentation mais également le rôle (et donc la définition) de chaque vue. S'il faut non seulement retenir les noms, mais de plus relier chaque nom à une définition, l'utilisateur va très vite s'y perdre.

5.4 Conclusion

Lors de l'analyse des outils graphiques d'*IntensiVE*, nous avons pu remarquer un manque de globalité ne permettant pas une utilisation optimale du modèle intentionnel. Nous avons donc élaboré une solution qui consiste en un nouvel outil graphique, reprenant les vues et relations sous forme d'un graphe composé de rectangles représentant les vues intentionnelles et des liens entre ces rectangles pour les relations. Les liens du type "est une sous-vue de" et "est une vue alternative de" seront aussi représentés dans le graphe pour avoir une représentation globale du système étudié.

Des métriques seront utilisées afin de donner au graphe des dimensions supplémentaires permettant à l'utilisateur de se renseigner sur telle ou telle caractéristique globale sur le système étudié. Outre la mise en évidence des vues et relations non cohérentes du système, celles-ci permettront de connaître par exemple les vues les plus peuplées en terme d'objets contenus dans leur extension ou encore les vues alternatives les plus différentes de leur vue alternative principale.

Chapitre 6

Intentional View Displayer

Pour valider la solution décrite dans le chapitre précédent, nous avons implémenté un nouvel outil appelé *Intensional View Displayer* permettant de mettre en pratique les différents concepts définis dans le chapitre 5.

6.1 CodeCrawler

Notre nouvel outil, *Intensional View Displayer*, sera une extension de *CodeCrawler*, programme libre écrit en *Smalltalk*.

6.1.1 Introduction

L'idée de l'environnement qu'offre *CodeCrawler* est de permettre une lecture plus aisée d'un code source programmé dans le paradigme orienté objet [DL05]. En effet, il n'est pas évident de lire un code orienté objet et ce, pour plusieurs raisons telles que les difficultés engendrées par les concepts d'héritage et de polymorphisme, et l'ordre de la définition des classes qui n'est pas imposée (contrairement au code procédural où l'ordre de la déclaration des procédures est important et l'utilisation des seules procédures précédemment déclarées est requise) [Dek02].

Ce programme repose sur un autre programme appelé *MOOSE* [SDD05]. Ce dernier scanne un code source orienté objet (*Smalltalk*, `C++`, `Java`, ...) pour créer un modèle abstrait du programme dans un langage indépendant. De nouveaux outils de rétro-conception complètement indépendants du langage utilisé par le programme de base, peuvent alors être développés au-dessus de *MOOSE*. *CodeCrawler* transforme les informations fournies par ce dernier pour créer un graphique représentant les méthodes et les attributs de chaque classe du code étudié ainsi que les appels et accès des méthodes aux autres méthodes et attributs. Les attributs et méthodes sont représentés par des noeuds (sous forme de rectangles) et les accès et appels par des arcs rejoignant deux noeuds (représentés par des segments de droite). Des métriques, telles que la couleur ou la hauteur des noeuds, sont utilisées pour permettre au développeur de se faire un modèle mental des classes qu'il étudie et d'offrir un support pour la reconstruction du flot logique des appels des méthodes [DL05].

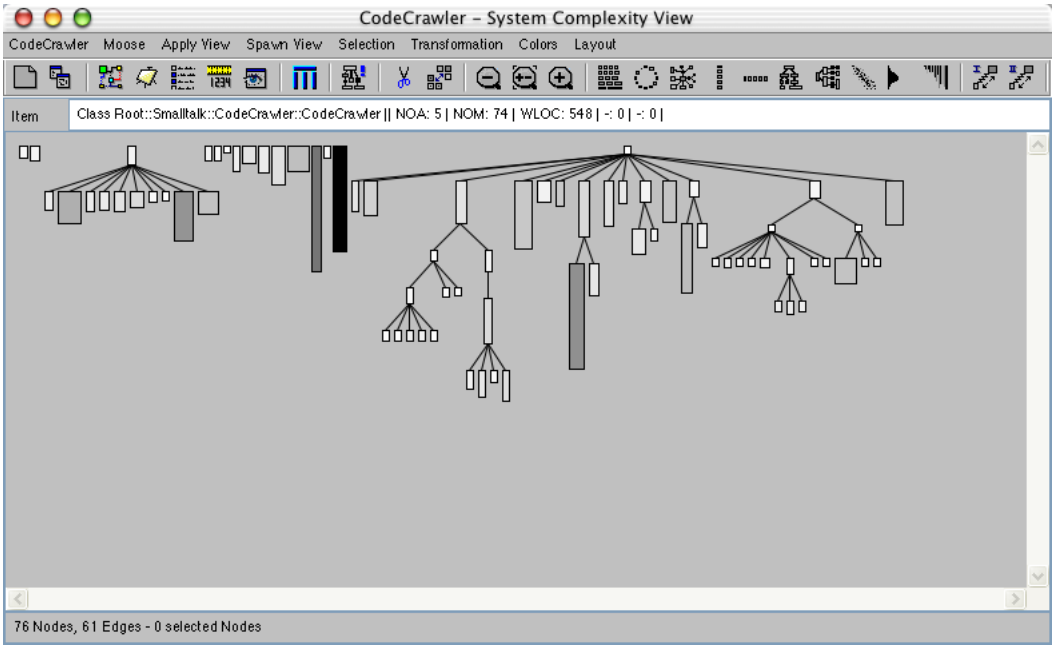

FIG. 6.1 – CodeCrawler en action

6.1.2 Métriques

La figure 6.1 montre *CodeCrawler* en action, étudiant la complexité d'un système. Chaque rectangle représente une classe du programme étudié. Les liens représentent simplement l'héritage d'une classe par une autre.

Comme on peut le remarquer, les rectangles ne sont pas de la même taille, ni de la même couleur. Ce sont les métriques utilisées dans cet exemple. Elles représentent le nombre d'attributs d'une classe pour la largeur, le nombre de méthodes pour la hauteur et la noirceur de la boîte représente le nombre de lignes de code dans la classe. Grâce à ce graphique généré automatiquement, nous pouvons discerner les classes qui comprennent par exemple énormément de méthodes mais pratiquement pas de lignes de code. Le rectangle noir à gauche du centre, nous indique une classe avec de nombreuses méthodes avec énormément de lignes de code par rapport aux autres classes.

Ces métriques peuvent bien sûr être changées via une simple interface fournie par *CodeCrawler*. Ce changement permet d'obtenir un nouveau graphique étudiant les mêmes classes mais d'un point de vue différent. Les métriques définissables sous *CodeCrawler* sont les suivantes.

Pour les rectangles Il existe 3 métriques :

- la couleur ;
- la hauteur ;
- la largeur.

Pour les arcs Il en existe 2 :

- la couleur ;
- la grosseur.

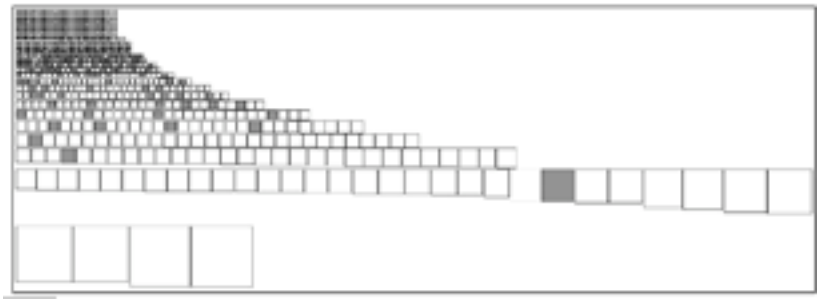

FIG. 6.2 – D'autres métriques et layout sous CodeCrawler

De plus, les classes sont placées pour offrir la meilleure clarté possible. Le *layout*, c'est-à-dire la façon dont les rectangles sont placés automatiquement, est "Arbre vertical" dans ce cas-ci. Une fois de plus, une simple configuration du graphique que l'on veut obtenir permet de changer cela. *CodeCrawler* offre une foule de layout tels que "Arbre horizontal", "En rond", "Ligne verticale". Les calculs savants requis pour le placement de ces rectangles sont donc écrits dans *CodeCrawler*.

La figure 6.2 nous montre l'étude d'un autre programme. Un layout différent est appliqué à d'autres métriques. *CodeCrawler* permet, comme on peut aisément s'en rendre compte, de créer des vues très différentes d'un système donné.

Les métriques et les layouts seront d'une importance considérable dans le développement de notre outil. En effet, comme nous l'avons vu dans le chapitre précédent, nous allons utiliser des métriques pour connaître par exemple la taille de l'extension d'une vue ou encore le nombre d'éléments différents entre les extensions des différentes vues alternatives d'une même vue intentionnelle.

6.1.3 Framework

Comme la solution définie dans le chapitre précédent le montre, nous avons besoin de ce que *CodeCrawler* propose. En effet, nous avons les rectangles qui peuvent être repris pour les vues intentionnelles et les arcs pour les relations. Nous allons donc créer notre programme comme une extension de *CodeCrawler*, récupérant de cette manière la puissance de ces layouts et le mécanisme mis en place pour appliquer des métriques sur des rectangles et des arcs.

CodeCrawler a été construit sous *Smalltalk*, dans le paradigme de l'orienté-objet et a été conçu pour qu'il puisse évoluer dans le temps. Chaque élément graphique est un objet en mémoire provenant d'une classe bien définie. L'héritage de ces classes et la surcharge de quelques unes de leurs méthodes permet de récupérer la puissance de *CodeCrawler* (métriques, layout, ...) pour en faire un nouveau programme de visualisation.

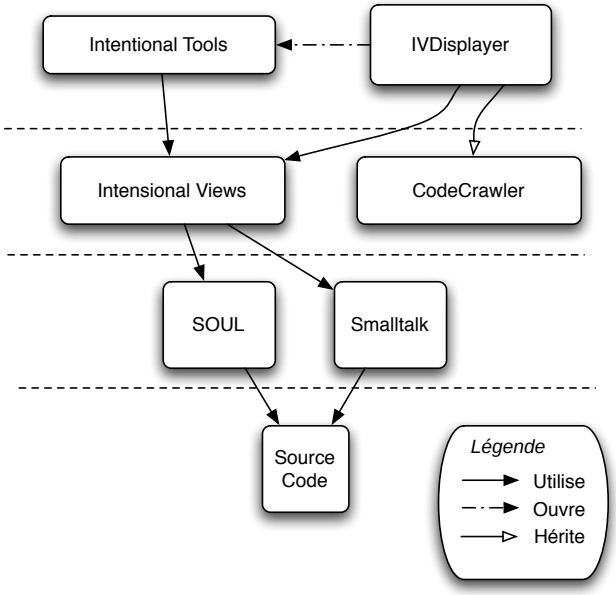

FIG. 6.3 – L'architecture en couches

6.2 Architecture en couche

Lors du développement de l'application, nous avons pu remarquer qu'il serait important de garder une architecture en couches, c'est-à-dire dans laquelle chaque composant d'une couche ne peut utiliser que les composants de cette même couche ou ceux situés dans une couche directement inférieure. De cette manière, les composants de chaque couche sont indépendants des couches supérieures.

Comme on peut le voir sur la figure 6.3, notre architecture est divisée en quatre couches. La couche la plus basse correspond au système à analyser. La couche comprenant "SOUL" et "Smalltalk" correspond aux requêtes faites sur le code source. Ce sont les intentions des vues intentionnelles reprises dans la troisième couche. Celle-ci comprend aussi *CodeCrawler* qui est hérité par le *Intensional View Displayer*. Nous avons enfin la couche la plus élevée qui reprend les outils graphiques qui permettent la gestion des vues intentionnelles, dont le *Intensional View Displayer*.

Nous pouvons donc remarquer plusieurs propriétés importantes :

- *CodeCrawler* reste indépendant du nouvel outil développé dans le cadre ce mémoire ;
- le code source est indépendant de la documentation. Cette technique de documentation, appelée *non intrusive* (car le code du programme étudié reste inchangé), apporte la certitude que la documentation ne changera pas le comportement du programme et n'introduira pas de nouveaux bugs.

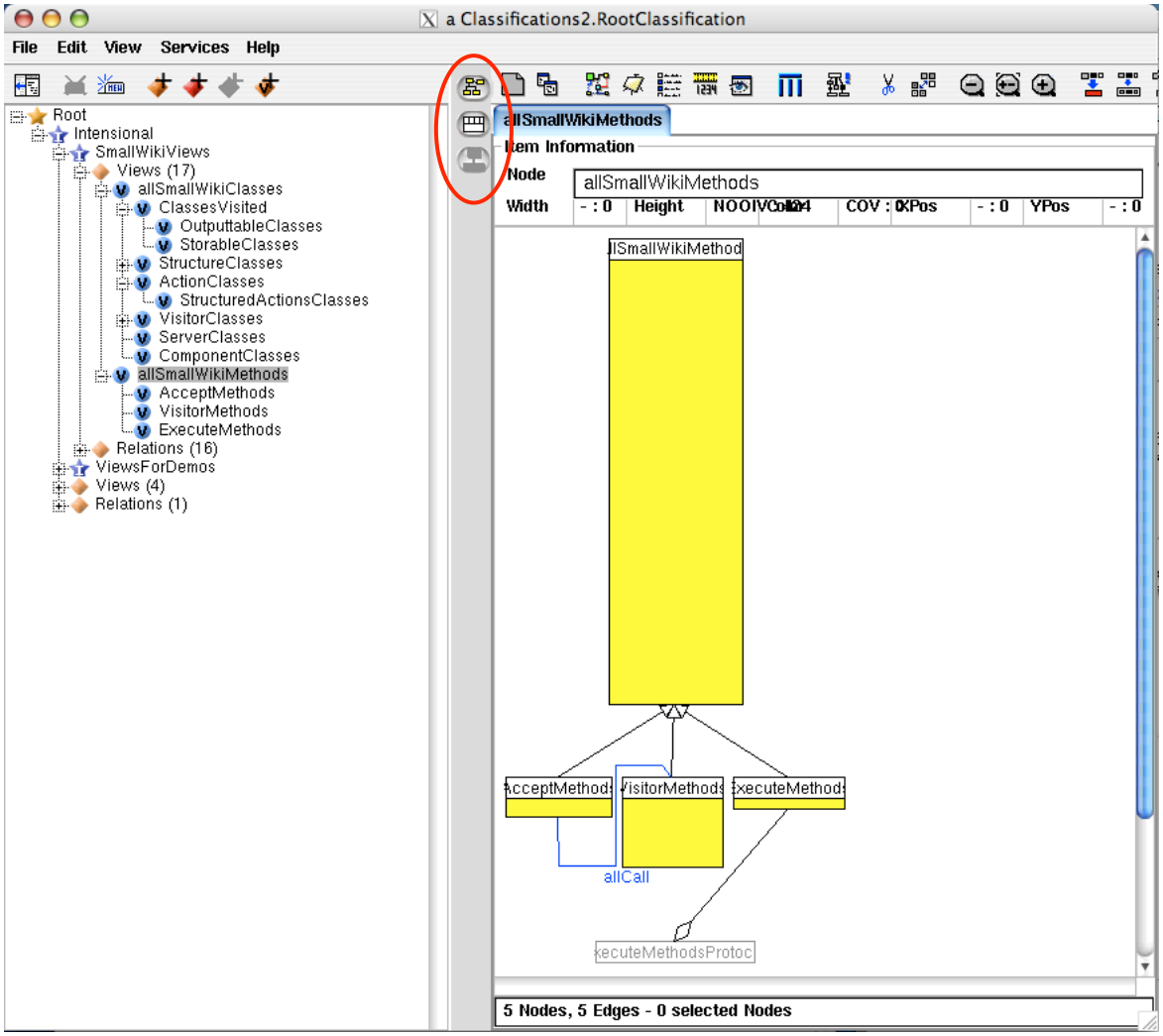

FIG. 6.4 – Intensional View Displayer en action

6.3 *Intensional View Displayer*

6.3.1 Introduction

Intensional View Displayer est le nom de l'outil développé dans le cadre de ce mémoire. Il est une extension de *CodeCrawler*, en exploitant tous les outils offerts par ce dernier. En effet, grâce aux métriques, aux layouts, au reliage des différents noeuds par des arcs, l'environnement qu'offre *Intensional View Displayer* est configurable à souhait et offre différentes visions sur un même ensemble de vues intentionnelles.

Comme les quatre autres outils, il est complètement intégré dans *StarBrowser* 2. Un simple bouton accessible en tout temps, permet à l'utilisateur de passer du mode *édition* du système intentionnel (composé des quatre outils) au mode *visualisation* et inversement (sur la copie d'écran fig. 6.4, au milieu

entouré en rouge, il y a 3 boutons : le premier nous importe peu ici, le second passe en mode édition et le dernier en mode visualisation). Comme on peut le remarquer sur le copie d'écran, l'outil de visualisation vient se positionner à la même place que les autres outils.

La sélection d'une vue autre que la vue courante (dans le panneau de gauche) commande un nouveau graphique. L'outil calcule alors l'extension de la vue sélectionnée. Deux cas de figure peuvent se présenter : la vue choisie possède une extension qui comprend soit uniquement d'autres vues, soit d'autres objets (classes, méthodes, ...).

Si nous nous trouvons dans le deuxième cas, un graphique sera calculé et affiché dans l'outil représentant la vue sélectionnée ainsi que toute sa hiérarchie et les relations reliant les différentes vues présentes dans le graphe. La hiérarchie est affichée sous la forme d'un arbre où les vues sont représentées par des rectangles comme définies par la figure 5.2.

Une flèche d'une vue vers une autre représente une relation de sous-vue/super-vue dans laquelle la super-vue est pointée par la flèche. Une ligne finissant par un losange définit quant à elle une relation de vue intentionnelle/vue alternative où la vue alternative est accolée au losange. Ce changement de sens pour les annotations permet de les différencier très facilement lors de la lecture du graphique.

Les relations sont représentées par des lignes étiquetées par le modèle de la relation. Des métriques par défaut ont été définies pour les vues et les relations. Elles seront décrites dans une prochaine section.

Le premier cas de figure peut se produire si l'on construit des vues intentionnelles sur le système intentionnel, une sorte de méta-modèle sur le méta-modèle d'un code source. Cela peut paraître incongru de prime abord mais le simple fait de ne vouloir reprendre qu'une partie des vues du système intentionnel pour les étudier plus simplement constitue déjà un méta-modèle. Le graphique représentera alors les vues contenues dans l'extension et pas la hiérarchie de la vue sélectionnée comme dans le second cas.

6.3.2 Le layout

Pour que l'utilisateur ait une vue globale et simple du système des vues et relations intentionnelles, le layout doit être suffisamment bien pensé afin qu'en un coup d'œil, le développeur remarque d'une part l'architecture du système et d'autre part, toutes les erreurs au sein des vues et relations intentionnelles définies et l'importance de celles-ci.

Comme nous pouvons le voir dans la figure 6.4, le layout d'arbre vertical se prête très bien au système des vues intentionnelles. En effet, les vues peuvent être des vues ou des sous-vues d'un seul parent à la fois. Il ne peut avoir de cycle et la structure d'arbre se prête bien à la visualisation d'un tel système. C'est ainsi que, par défaut, l'outil adopte un layout d'arbre vertical, disposant la hiérarchie de la vue sélectionnée sous forme d'arbres verticaux et affichant les différentes relations entre les vues disponibles dans le graphe. Toutes les vues n'ayant aucun parent sont alignées horizontalement sur le haut du graphique.

Le layout peut être changé par une simple interface offerte par *CodeCrawler*.

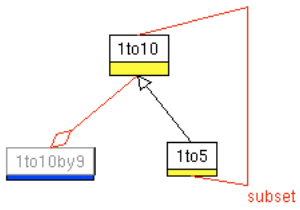

FIG. 6.5 – Intensional View Displayer et les métriques

6.3.3 Les métriques

Les métriques décrites dans le chapitre précédent ont été implémentées dans *Intensional View Displayer*. Les métriques sur les vues et les relations sont paramètrables via une interface héritée de *CodeCrawler*.

Comme nous pouvons le voir dans la figure 6.4, les vues sont représentées par des rectangles plus au moins longs, s'adaptant à la longueur du nom de la vue. Les rectangles jaunes, accolés à ceux des vues, représentent la métrique "Nombre d'objets contenus dans l'extension de la vue". Par l'application de cette métrique, le graphique nous révèle l'importance des vues intentionnelles. Nous pouvons constater que l'extension de la vue *AllSmallWikiMethods* comprend un nombre d'objets beaucoup plus grand que l'extension de ses sous-vues.

Afin de montrer l'utilité des métriques, prenons un exemple extrêmement simpliste. Nous définissons premièrement une vue nommée *1to10* qui donne comme extension les nombres de 1 à 10. Nous lui ajoutons une vue alternative (non cohérente pour la démonstration) donnant les nombres de -3 à 9 nommé *1to10by9*. Nous créons ensuite une sous-vue de *1to10* nommée *1to5* et une relation définie comme suit : tout élément de *1to10* existe dans *1to5*. Cette relation est évidement non cohérente par sa définition.

Ces cas d'école vont nous permettre de connaître le comportement de notre nouvel outil. Le graphique obtenu est celui de la figure 6.5. Nous sommes attirés immédiatement par les traits rouges qui ressortent du graphe. Ceux-ci mettent en évidence les incohérences relevées dans le système étudié. La ligne rouge munie d'un losange indique que la vue alternative *1to10by9* est incohérente avec *1to10*. Les métriques demandées pour les vues alternatives sont le nombre d'éléments en moins par rapport à la vue principale pour la largeur et le nombre d'éléments en plus pour la hauteur. Grâce à celles-ci, nous apprenons que la vue alternative ne possède plus un élément que la vue principale possédait (c'est la ligne noir sur le côté droit du rectangle intitulé *1to10by9*) et qu'elle possède 4 éléments en plus (le rectangle bleu situé en dessous). La seconde ligne rouge, reliant les deux vues concernées, représente la relation incohérente.

Les métriques des vues intentionnelles sont le nombre d'éléments compris dans l'extension de la vue comme hauteur. Elles sont représentées par les rectangles jaunes. *1to5* a effectivement 2 fois moins d'éléments que *1to10*.

L'utilisation des métriques permet donc de comprendre beaucoup plus facilement le système dans sa globalité, de prendre conscience des incohérences

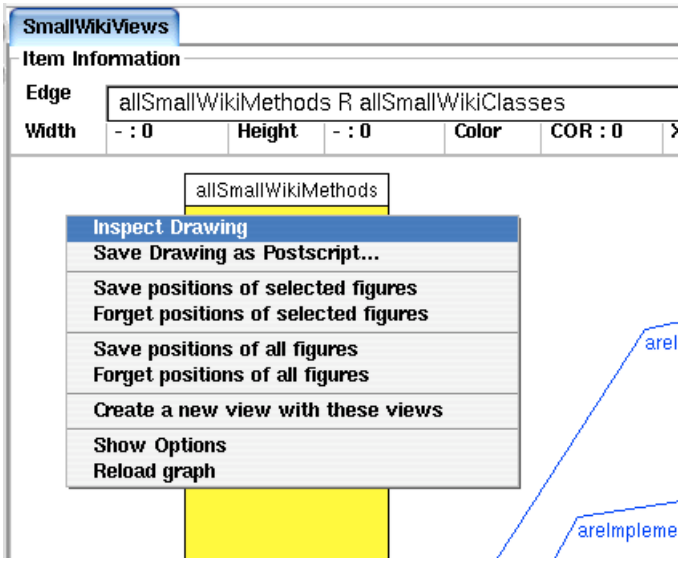

FIG. 6.6 – Options d'*Intensional View Displayer*

et d'évaluer leur importance (via les métriques sur les vues alternatives par exemple).

6.3.4 La configuration

Lors la validation décrite dans le chapitre suivant, de nombreux utilisateurs ont utilisé cet outil dans des buts parfois différents (vérifier certaines contraintes, avoir une vue globale de la documentation ou seulement une vue partielle, ...). Une configuration plus fine a donc été mise en place afin de répondre aux exigences des utilisateurs.

Quelques options sont donc disponibles par un clic droit dans un espace exempt de tout rectangle ou arc (Fig. 6.6). Les deux premiers items de ce menu contextuel sont ceux hérités de *CodeCrawler*. Le premier permet d'inspecter l'objet *Smalltalk* `Drawing`, objet qui contient toutes les informations concernant le graphe. Cette option est très utile lorsque l'on veut débugger le mécanisme du graphe. La seconde option permet d'exporter le schéma dans un fichier au format Postscript. Celle-ci est discutée plus loin dans ce chapitre.

Détaillons le reste des options :

Save positions of selected figures et **Forget positions of selected figures** Lorsque vous manipulez le graphe du système intentionnel, vous pouvez bouger les rectangles, ajouter des points intermédiaires dans les arcs (afin de leur donner une trajectoire rendant le graphe plus lisible). Ces deux items permettent respectivement de sauver et d'oublier les positions des figures sélectionnées dans le graphe pour la vue courante (et seulement pour celle-ci). Après une opération de sauvegarde des positions, lorsque l'utilisateur rechargera la vue, les figures possédant un enregistrement pour cette vue seront repositionnées automatiquement.

Save positions of all figures et **Forget positions of all figures** réalisent respectivement les mêmes opérations que celles citées ci-dessus mais sur l'ensemble des figures présentes dans le graphe (et non pas seulement pour les figures sélectionnées).

Create a new view with these views Cette fonctionnalité permet de créer une nouvelle vue intentionnelle reprenant les vues intentionnelles et alternatives présentes dans le graphe actuel. Celle-ci est très utile pour regrouper un sous-ensemble de vues dans une autre visualisation de cette partie des vues.

Show Options Cet item amène sur une nouvelle fenêtre d'options dites *générales* détaillées ci-après.

Reload Graph permet de rafraîchir le graphe, mais sans qu'aucun nouveau calcul d'extension ne soit fait. Ceci est très pratique lorsque l'on change le layout ou les métriques utilisées : le code restant inchangé, l'extension de chaque vue le sera aussi, il n'est donc pas nécessaire de devoir les recalculer.

Les options *générales*, accessibles par le sous-menu "Show Options" (Fig. 6.7), sont les suivantes :

Show alternate views Cette option permet d'annihiler l'affichage des vues alternatives dans le graphe. Cela peut s'avérer utile pour alléger le graphe et ne se concentrer que sur ce que représentent abstraitement les vues intentionnelles (sans se soucier de comment on peut demander les différents éléments de la vue).

Bring to front all edges Cette option cochée, tous les arcs (et symboles s'y attachant) seront mis au premier plan. Si elle ne l'est pas, ce seront les rectangles des vues qui seront au premier plan.

Relation with predicate and quantifiers Les relations intentionnelles sont définies comme nous l'avons vu par deux quantificateurs et un prédicat. Dans l'Intentional Relation Editor, il est possible d'enregistrer ces trois paramètres, formant un type de relation, sous un nom décrivant ce type. En décochant cette case, les relations ne présenteront plus ni les prédicats ni les quantificateurs mais juste un label avec le type de relation. Pour les relations n'ayant pas de type, leur prédicat et leurs quantificateurs apparaîtront bien évidemment.

Show integral popup-menu Héritant de *CodeCrawler*, *Intensional View Displayer* ajoute de nouveaux items au menu contextuel des entités graphiques (rectangles, arcs, ...), laissant ceux définis par *CodeCrawler* totalement intacts. En décochant cette option, les anciens items des menus contextuels seront cachés pour ne laisser voir que ceux vraiment utiles à la manipulation du système intentionnel.

6.3.5 L'export

Le graphique généré par *Intensional View Displayer* est exportable directement vers des fichiers du type PostScript. Le fait même que ce soit un graphe qui

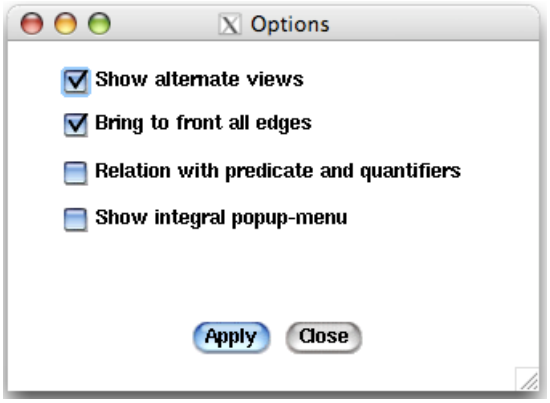

FIG. 6.7 – Options générales d'*Intensional View Displayer*

soit généré permet aux développeurs d'avoir une base de travail pour discuter et raisonner sur des problèmes avec d'autres personnes.

Il est à noter que le PostScript généré est un fichier stockant les graphiques de manière vectorielle. Cette particularité permet d'agrandir le graphique autant que l'on veut sans perte de qualité (l'image est toujours recalculée pour donner le meilleur résultat). Les développeurs peuvent exporter leur graphique vers des formats de papier très grand (tels que l'A1 ou l'A2) afin que, lors de réunion, un grand nombre de personnes peuvent discuter en même temps autour d'un même plan.

6.3.6 Le lien avec les autres outils graphiques

Lorsque l'on clique droit sur un rectangle, le menu contextuel permet de se rendre sur l'Intensional View Editor, pour l'édition ou sur le View Consistency Checker pour vérifier la cohérence interne d'une vue.

La même technique sur les lignes permet de se rendre à l'Intensional Relation Editor (pour l'édition de la relation pointée) ou au Relation Checker (pour vérifier la cohérence de relation). Ce nouvel outil permet donc une édition rapide des vues et des relations intentionnelles.

Cette amélioration par rapport aux autres outils est très importante. L'efficacité de la détection des incohérences couplée à la rapidité d'édition des vues et relations permet une mise à jour plus rapide de la documentation.

6.4 Conclusion

L'outil présenté au sein de ce chapitre offre une vue globale d'un système intentionnel, en y incluant des métriques afin d'obtenir encore plus d'informations en un seul coup d'œil.

De plus, des menus contextuels sur les vues et relations permettent un lien avec les outils d'édition et de vérification de cohérence.

Le prochain chapitre propose une validation du modèle intentionnel et de

ces cinq outils par l'étude d'un programme existant *SmallWiki*, un framework pour des serveurs Web coopératifs.

Chapitre 7

Validation d'IntensiVE

La deuxième partie de la recherche se concentre sur la validation du modèle des vues intentionnelles ainsi que de l'ensemble des outils dont se compose *IntensiVE*, l'environnement de manipulation des vues intentionnelles.

7.1 SmallWiki

Un *wiki* est une application web de collaboration qui permet aux utilisateurs enregistrés dans celui-ci d'ajouter du contenu mais permet surtout à n'importe qui d'éditer ce contenu. Il est important de noter que chaque version d'une page d'un wiki est sauvegardée et les administrateurs peuvent à tout moment revenir à des versions plus anciennes de leurs pages. Ce système permet vraiment qu'un site web soit créé en groupe, où n'importe qui peut venir poser sa pierre à l'édifice. On peut rencontrer ce type de système de plus en plus sur Internet. Pour n'en citer qu'un seul : WikiPedia[1] est une encyclopédie construite par des internautes du monde entier. Ainsi, la base de connaissances offerte par ce site grandit de jour en jour, et ce en plus de 10 langues !

SmallWiki[Ren05] est un framework pour wiki complètement orienté-objet et extensible. Il est développé entièrement sous *VisualWorks Smalltalk*. Contrairement aux autres implémentations de wiki, qui sont souvent difficiles à adapter, *SmallWiki* a été construit depuis le début avec un design permettant l'extension. Il est basé sur un design clair orienté-objet dans lequel toutes les entités qui peuvent se retrouver dans une page web(textes, liens, tables, listes) sont explicitement modélisés comme des objets. Tout dans *SmallWiki* est pensé pour être étendu : les types de page, le mécanisme de sauvegarde, les actions, le mécanisme de sécurité, le serveur web, etc. Les plug-ins peuvent être partagés dans la communauté et chargés indépendamment les uns des autres dans le système.

Nous avons décidé d'utiliser *SmallWiki* pour notre étude de cas pour plusieurs raisons. Premièrement, il est open source et ses sources sont libres d'utilisation. Deuxièmement, beaucoup de versions sont accessibles, depuis les toutes premières versions jusqu'aux versions stables utilisées dans plusieurs entreprises.

[1] http ://www.wikipedia.fr

Troisièmement, ce programme n'est pas trivial et reste encore manipulable en taille et en complexité.

Pour effectuer la validation, nous avons étudié les trois versions suivantes de *SmallWiki* :

Version 1.54 (14-12-2002) Cette version est la première 'internal release' (c'est-à-dire à peu près stable) de *SmallWiki*. Celle-ci offre déjà un serveur Wiki opérationnel avec peu de fonctionnalités : seuls l'édition et le rendu des pages Wiki assez simples étaient supportés. Cette version est composée de 63 classes et 424 méthodes.

Version 1.90 (15-01-2003) Cette version n'est séparée que d'un mois de développement seulement de la précédente (ce qui limite le risque d'avoir une version trop différente de la première version étudiée). Néanmoins, ce mois représente une activité intense de développement, avec parfois plus d'une nouvelle version par jour ! Ce n'est donc pas une version triviale à étudier. Cette version compte 8 classes de plus (donc 71 au total) et beaucoup plus de méthodes(633) que la version précédente. Une différence importante entre ces 2 versions est que dans la version la plus récente, les méthodes responsables du rendu du code HTML ont subi une restructuration.

Version 1.304 (16-11-2003) Cette version a été choisi car elle couvre une large période de développement (presque un an) avec beaucoup de versions intermédiaires. Cette version va nous permettre d'étudier le problème de synchronisation de la documentation du design et du code source après un laps de temps assez conséquent. Avec 108 classes et 1219 méthodes, cette version est clairement plus large que les deux autres.

Pour étudier l'utilité et l'utilisabilité du modèle intentionnel et de ses outils pour documenter l'architecture d'un système informatique en évolution, nous conduiront les différentes expériences suivantes :

1. Nous allons commencer par étudier le design de la version 1.54 et analyser comment cette documentation peut nous aider à mieux comprendre la structure du code, en relevant, par exemple, les conventions de nomination et de codage.
2. Nous allons ensuite comparer cette documentation à la version 1.90 et en tirer les conclusions concernant l'évolution des méthodes de *SmallWiki* et les conséquences de cette évolution sur l'architecture.
3. Finalement, nous comparerons la documentation obtenue à la version la plus récente des trois, et nous observerons que le design reste relativement stable, même après cette longue période de développement.

7.2 Expériences

Après avoir présenté les outils d'*IntensiVE* en détail dans le chapitre 4, nous allons effectuer quelques expériences sur *SmallWiki*. Dans la première expérience, nous essayons de documenter le design de *SmallWiki* version 1.54

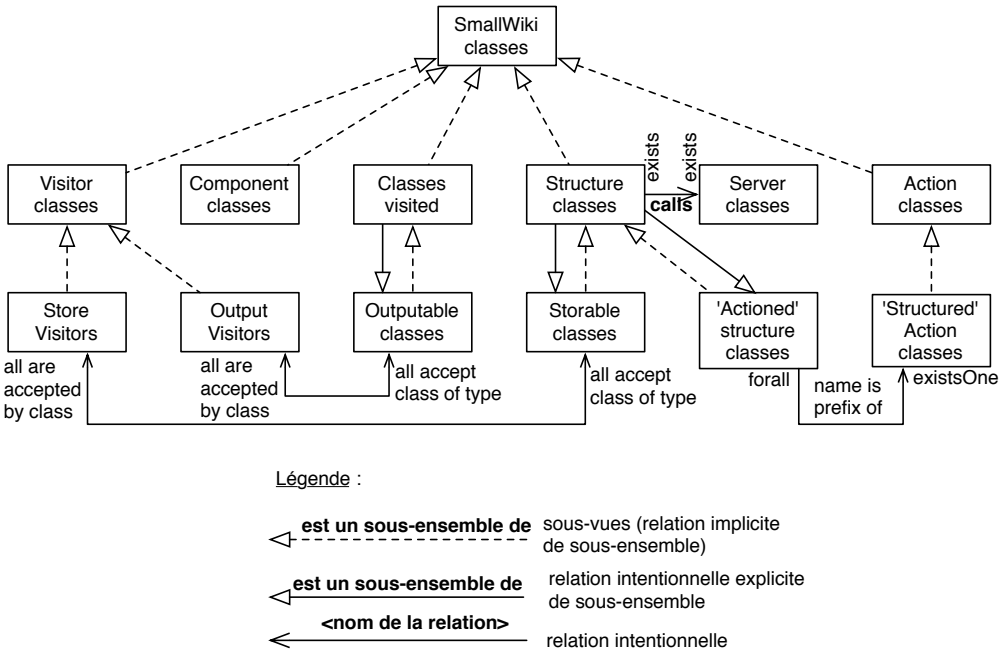

FIG. 7.1 – Vues et relations sur SmallWiki (concernant les classes)

et de découvrir comment cette documentation peut nous aider à mieux comprendre la structure de l'implémentation ainsi que les conventions de codage et de nomination qui ont été utilisées.

7.2.1 Expérience 1 (Documenter la structure de *SmallWiki* 1.54)

À cause du manque de documentation adéquate sur cette version précise, l'approche que nous adoptons est pratiquement entièrement manuelle. Nous inspectons manuellement le code, regardons les groupes intéressants de classes ou de méthodes, codifions ces groupes en des vues intentionnelles, vérifions la validité de ces vues par rapport au code source, nous les affinons quand cela est nécessaire, nous inspectons les éléments des vues définies pour découvrir des relations avec d'autres vues (potentiellement à définir), etc.

Finalement, nous avons un total de 17 vues intentionnelles, reliées par 14 relations de sous-vues et 16 relations intentionnelles. Les figures 7.1 et 7.2 résument toutes les vues et les relations définies. La figure 7.1 montre les vues contenant des classes ainsi que leur interrelations tandis que la figure 7.2 se focalise sur les vues contenant des méthodes.

Les vues

All SmallWiki Classes Premièrement, nous définissons une vue contenant toutes les classes de l'application étudiée. Cette vue est codifiée par la requête

Smalltalk suivante : `SmallWiki allClasses.`[2]

Pour restreindre leur domaine aux classes de *SmallWiki* uniquement, toutes les autres vues sont définies comme des sous-vues de celle-ci. Par exemple, nous définissons une série de vues correspondant à d'importantes hiérarchies de classes dans le code. Celles-ci sont toutes définies par une requête *Soul* ayant la forme suivante : `classInHierarchyOf(?entity,[`*classe racine de la hiérarchie*`])`.

Structure Classes (les classes dans la hiérarchie de `Structure`) représente les entités de *SmallWiki* qui peuvent être référencées par une adresse URL unique, comme une page web par exemple.

Component Classes (hiérarchie de `PageComponent`) représente les composants à partir desquels une page web peut être construite : un texte, des lignes, des tables, des listes, ...

Visitor Classes (hiérarchie de `Visitor`) visite la structure et les classes de composants (définies plus haut) et joue un rôle crucial dans *SmallWiki*, pour le rendu et la sauvegarde des pages web. entre autres.

Action Classes (hiérarchie de `Action`) modélise les actions que l'on peut effectuer sur les pages Wiki.

Server Classes (hiérarchie de `WikiServer`) représente les différentes sortes de serveurs Wiki supportées par *SmallWiki* (la version 1.54 ne supporte uniquement que Swazoo[3]).

Les autres vues que nous avons définies, principalement en recherchant manuellement dans le code, sont :

Visitor methods Cette vue reprend toutes les méthodes implémentées dans l'hiérarchie de la classe `Visitor` qui font partie d'un protocole nommé "visiting". Avec *Soul*, on peut l'exprimer comme suit :

```
classInHierarchyOf(?class,[SmallWiki.Visitor]),
methodOfClassInProtocol(?entity,?class,?protocol),
['visiting*' match:?protocol]
```

En fait, c'est un exemple d'une requête hybride où nous utilisons la logique pour raisonner sur la structure du code et nous évaluons une expression *Smalltalk*, paramétrisée par une variable logique (c'est une particularité de *Soul*), pour raisonner sur les chaînes de caractères.

Accept methods sont les méthodes nommées `accept` et qui jouent un rôle important dans le pattern du visiteur (voir page 11). Nous définissons cette vue par la requête *Soul* `methodWithName(?entity,[#accept:])`.

Actioned structure classes Nous définissons le groupe de toutes les classes de structures sur lesquelles des actions peuvent être effectuées, comme une sous-vue de la vue nommée "Structure Classes". Elles peuvent être reconnues aisément car elles possèdent une classe `Action` correspondante, c'est-à-dire que chaque classe de structure possède une autre classe cousine qui porte le même nom qu'elle, suffixée du mot 'Action' :

```
classInViewNamed(?c, ActionClasses),
['*Action' match: ?c name],
[(?entity name, 'Action') = ?c name asString]
```

[2]Nous pourrions aussi déclarer cette même demande dans le language *Soul* comme ceci : `classInNamespace(?entity, [SmallWiki])`

[3]Serveur Web développé en *Smalltalk*

Structured action classes De manière inverse, nous définissons la vue des classes d'actions pour une classe de structure particulière comme une sous-vue de la vue "Actions Classes".

Execute methods sont les méthodes responsables de l'exécution des différentes actions sur les pages Wiki, comme le rendu, l'enregistrement, l'annulation et l'édition. Leur nom comportent un suffixe commun : 'execute'. Nous définissons donc cette vue par l'intention : `['execute*' match: ?entity selector asString]`. Comme nous avons pu observer que les développeurs de *SmallWiki* ont, de manière consistante, adopté la convention de mettre ces méthodes dans un protocole nommé "action", nous pouvons aussi déclarer une intention alternative : `methodInProtocol(?entity,action)`.

En définissant les vues que l'on vient de décrire, nous en avons encore découvert de nouvelles :

Store Visitors et Output Visitors Après avoir défini la vue Classes Visited, nous nous sommes demandés quelles classes étaient visitées et pour quelles raisons. En inspectant les classes de visiteurs plus en détail, nous avons appris que dans *SmallWiki* 1.54, il y avait deux principaux visiteurs : un visiteur d'enregistrement (c'est-à-dire qui parcourt les objets pour les sauver physiquement) et un visiteur de rendu (qui permet de transformer un objet représentant une entité de page Web en code HTML). Nous codifions simplement ces derniers comme des sous-vues de la vue Visitor Classes : ces deux vues reprennent toutes les classes de visiteurs nommées respectivement `VisitorStore*` et `VisitorOutput*`.

Storable Classes et Outputtable Classes Nous définissons aussi une vue représentant les classes dites 'enregistrables' ('storable' classes, en anglais), c'est-à-dire les classes visitées par un visiteur d'enregistrement, et une autre vue représentant les classes rendues, comme des sous-vues de Classes Visited. Nous ne montrons uniquement ici que la définition de la vue Storable Classes, étant donné que la vue Outputtable Classes est analogue à celle-ci. Nous définissons la vue en terme de la nouvelle vue Store Visitors : les classes de visiteurs d'enregistrement ont besoin d'implémenter une méthode spécifique `accept<nom de la classe>:` pour chaque classe qu'elles ont besoin de visiter. Sans aller dans tous les détails, la requête hybride (code *SOUL* et *Smalltalk*) suivante extrait les classes visitées à partir des noms des méthodes des classes de visiteurs d'enregistrement :

```
classInViewNamed(?class,StoreVisitors),
methodWithNameInClass(?method,?selector,?class),
[?selector = (\#accept, ?entity name, ':') asSymbol]
```

Comme second exemple de la réutilisation graduelle des vues existantes afin de raffiner et comprendre la structure de code, la définition de Visitor Classes a déclenché la définition d'une autre vue contenant toutes les classes *étant* visitées :

Classes Visited sont ces classes qui peuvent être visitées par les classes de visiteurs. Vu que le design pattern de visiteur (décrit à la section 2.2.6) utilise un protocole de double dispatching où les classes visitées implémentent une méthode `accept:` prenant un visiteur en argument, nous définissons cette vue en utilisant la requête *SOUL* `methodWithNameInClass(?M, [#accept:],?entity)`. De plus, toutes ces méthodes étant normalement dans le protocole 'visiting', nous ajoutons une vue alternative dont l'intention est `protocolInClass(visiting,?entity)`.

En vérifiant la consistance de cette vue, nous apprenons que l'utilisation du protocole 'visiting' a été extrêmement bien respectée : toutes les classes qui implémentent une méthode `accept:` possèdent aussi un protocole 'visiting' et vice versa. Pour documenter cette propriété, nous définissons la vue alternative suivante pour la vue Accept Methods précédemment définie : `methodInProtocol(?entity,visiting)`.

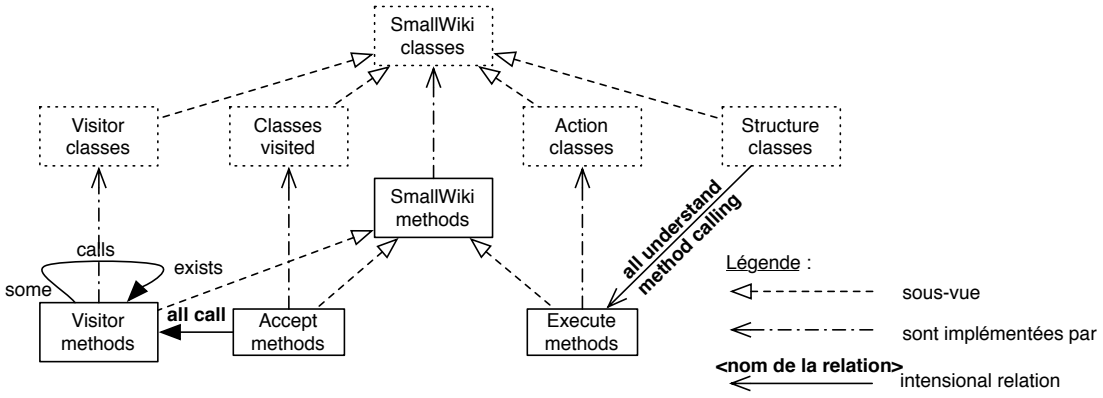

FIG. 7.2 – Vues et relations sur SmallWiki (concernant les méthodes)

Cependant, vu que toutes les descriptions alternatives devraient produire la même extension, cela n'implique pas seulement que toute méthode `accept:` est comprise dans un protocole nommé 'visiting' mais que, de plus, toute méthode dans le protocole 'visiting' est une méthode `accept:`. Cette contrainte est clairement trop forte, comme nous l'apprenons lorsque nous utilisons le View Consistency Checker : la classe `Visitor` n'implémente pas une méthode `accept:`, mais contient plusieurs méthodes de 'visite' dans le protocole 'visiting'. En excluant cette classe de la nouvelle alternative, la cohérence interne redevient valide.

Bien que le pattern de visiteur soit bien connu et compris, notons qu'il y a quelques vues (et relations, comme nous le verrons dans la section 7.2.1) dans notre documentation qui couvrent cette configuration spécifique.

Nous avons décidé de le documenter explicitement non seulement en raison du rôle crucial que ce pattern joue dans l'implémentation de *SmallWiki*, mais surtout parce que nous voulions pouvoir vérifier si les contraintes d'implémentation implicites de ce pattern demeurent uniformément respectées dans les versions futures de *SmallWiki*.

Relations entre les vues intentionnelles

Toutes les relations que nous allons définir entre des vues intentionnelles précédemment définies contenant des classes sont résumées dans la figure 7.2. Les lignes hachurées finissant avec un triangle représentent les sous-vues d'une vue intentionnelle. En plus de ces relations implicites de sous-ensembles, nous allons coder quelques relations explicites de sous-ensembles entre des vues non reliées directement par une relation implicite de sous-ensembles.

Classes Visited *est un sous-ensemble de* Outputtable Classes Non seulement toutes les classes que l'on peut rendre en sortie sont une sorte particulière de classes visitées (qui est définie par la sémantique de la sous-vue), mais de même, *toutes* les classes visitées peuvent être rendues.

Structure Classes *est un sous-ensemble de* Storable Classes Même si toutes les classes visitées peuvent être rendues, seules quelques unes sont enregistrables. D'un autre côté, toutes les classes de structures, à l'exception de la super classe `Structure`, sont enregistrables. Vu que cela semble potentiellement être une contrainte importante de dsign, nous la documentons comme une relation intentionnelle avec un cas de déviation explicite pour la classe `Structure`.

Structure Classes *est un sous-ensemble de* Actioned Structure Classes Bien que les classes de structure 'actionnée' sont définies comme une sous-vue de Structure Classes, nous observons que toutes les classes de structures (une nouvelle fois à l'exception de la classe `Structure`) sont 'actionnées', c'est-à-dire possèdent une classe `*Action` correspondante.

'Actioned' Structure Classes versus 'Structured' Action Classes Cette même observation nous a amenés à définir la relation intentionnelle suivante entre les classes de structures 'actionnées' et les classes d'actions 'structurées' :

$\forall\ x \in ActionedStructureClasses : \exists!\ y \in StructuredActionClasses :$
$x\ has\ name\ which\ is\ prefix\ of\ name\ of\ y$

dans laquelle le prédicat est défini par le code *Smalltalk* suivant :

```
[:class1 :class2 | (class1 name asString),'*'
                   match: (class2 name asString)]
```

Nous allons maintenant définir les relations entre les visiteurs et les classes visitées.

Output Visitors *acceptent tous une classe du type* Outputtable Classes et **Store Visitors *acceptent tous une classe du type* Storable Classes** Puisque le mécanisme de double dispatching est utilisé par le pattern de Visiteur (voir la section 2.2.6), nous savons que *toutes* les classes de visiteurs qui peuvent manipuler un certain type de classe ont besoin d'implémenter *une* méthode spécifique `accept:`, prenant des objets de ce type comme argument. Ceci est particulièrement vrai pour les visiteurs de rendu et les classes qui peuvent être rendues. Il en est de même pour les visiteurs d'enregistrement et les classes enregistrables. Le prédicat (hybride) *SOUL* qui est utilisé pour ces relations intentionnelles est donné plus bas. À cause du manque de type statique dans *Smalltalk*, le prédicat se base sur le fait que les paramètres formels de la méthode sont nommés du type prévu.

```
acceptsClassOfType(?VisitorClass,?VisitedClass) if
   methodWithNameInClass(?Method,?Selector,?VisitorClass),
   ['accept*' match:?Selector asString],
   argumentOfMethod(?Argument,?Method),
   ['*',(?VisitedClass name asString),'*' match:?Argument asString]
```

Outputtable Classes *sont toutes acceptées par* Output Visitors et **Storable Classes *sont toutes acceptées par* Store Visitors** Réciproquement, toutes les classes visitées sont supposées être acceptées par au moins une classe de visiteur. En particulier, les classes qui peuvent être rendues et les visiteurs de rendu, ainsi que les classes enregistrables et les visiteurs d'enregistrement, respectent cette contrainte. Ce prédicat logique intervenant dans cette relation est l'inverse du précédent :

```
isAcceptedByClass(?VisitedClass,?VisitorClass) if
   acceptsClassOfType(?VisitorClass,?VisitedClass)
```

Désormais, nous allons documenter les classes de serveur, qui sont invoquées par les classes de structures.

Structure Classes *appellent* Server Classes En considérant qu'il ne faut pas que toutes les classes de serveurs soient appelées (il suffit d'avoir un seul serveur qui tourne) ni que toutes les classes de structures appellent les classes de serveurs, cette relation intentionnelle est définie comme :

$\exists\ x \in StructureClasses : \exists\ y \in ServerClasses : x\ classCallsClass\ y$

où x *classCallsClass* y vérifie si une classe x possède une méthode qui appelle une méthode de la classe y. Une librairie logique rassemblant l'ensemble des prédicats logiques utiles (tels que *classCallsClass*) est fournie avec l'environnement *IntensiVE*.

Alors que la figure 7.1 se focalise sur les vues contenant des classes, la figure 7.2 résume les différentes relations entre les vues intentionnelles contenant des méthodes. En tout premier lieu, nous avons les relations évidentes d'implémentation :

Accept Methods *sont implémentées par* Classes Visited
Execute Methods *sont implémentées par* Action Classes
Visitor Methods *sont implémentées par* Visitor Classes

Les autres relations intentionnelles que nous documentons sont :

Accept Methods *appellent toutes* Visitor Methods En effet, les méthodes `accept:` ont toutes le pattern suivant pour appeler une méthode appropriée dans le visiteur :

```
accept: aVisitor
   aVisitor accept<nom de la classe>: self
```

Pour exprimer cette relation, nous utilisons deux quantificateurs universels et un prédicat `methodCallMethod` que nous déclarons en *Smalltalk* en utilisant le bloc suivant :

```
[:method1 :method2 | method1 sendsSelector:
                     (method2 compiledMethod selector)]
```

Visitor Methods *peu(some) appellent* Visitor Methods Cette relation représente le fait qu'une méthode de visiteur est parfois implémentée en termes d'autres méthodes de visiteurs. Par exemple, toute une partie des méthodes `accept:` des visiteurs font appel à leur propre méthode `visit:`. Pour exprimer cette relation intentionnelle, nous utilisons le même prédicat que plus haut, et un quantificateur numérique *some* qui requiert que la relation est valide pour au moins 25% de son domaine.

Structure Classes comprennent toutes une méthode appelant Execute Methods Ce sont les actions peuvent être exécutées sur des éléments comme des pages web (qui sont représentés par les classes de structures). Il faut que chaque classe de structure comprenne au moins une méthode qui appelle une méthode d'exécution appropriée pour manipuler réellement les actions. Par exemple, la classe abstraite `Structure` implémente une méthode nommée `evaluateActionWithRequest: response:` qui appelle la méthode `execute` sur la classe d'action appropriée. Nous définissons cela via un prédicat logique que nous ajoutons à la librairie logique.

7.2.2 Expérience 2 (Comparer la documentation avec *SmallWiki* 1.90)

Dans cette deuxième expérience, nous allons comparer le design documenté de la version 1.54 avec la version 1.90, version plus récente. Nous allons essayer de comprendre comment *SmallWiki* a évolué et quelles sont les conséquences de cette évolution sur la documentation du design. Pour faire cela, nous chargeons la nouvelle version et nous recalculons et visualisons toutes les vues et relations intentionnelles grâce au *Intensional View Displayer*. Comme expliqué à la section 6.3 et illustré dans la figure 7.3, toutes les vues et relations conflictuelles sont surlignées en rouge. Nous inspectons alors ces conflits et nous essayons de comprendre les problèmes qui sont apparus.

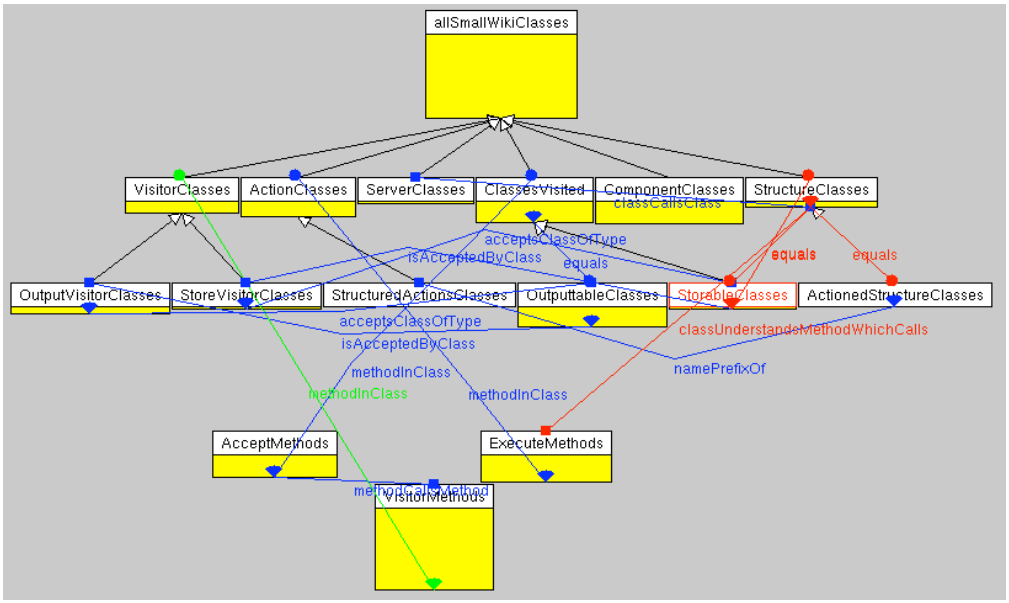

FIG. 7.3 – Vues et relations intentionnelles sur la version 1.90

Vues inconsistantes

Storable Classes est devenue inconsistante parce qu'un cas de déviation explicite dans la troisième description alternative n'était plus nécessaire. Plus précisément, nous avions défini que toutes classes enregistrables, à l'exception de la classe `Document` étaient des classes de structures. Dans la version 1.90, le système a été refactorisé de telle manière que la classe `Document` a bougé dans une autre hiérarchie. Nous mettons à jour la documentation en supprimant cette déviation obsolète.

Relations inconsistantes

Structure Classes *comprennent toutes une méthode appelant une* Execute Methods
Cette relation est en rouge car, dans *SmallWiki* 1.54, les méthodes d'exécution devaient en plus s'occuper du rendu des pages Web dans le format HTML. Dans la nouvelle (et plus propre) version, le rendu des pages est effectué par des méthodes de rendu séparées. Le seul but des méthodes `execute` est alors de dispatcher les requêtes d'actions vers des méthodes plus appropriées en tenant compte de l'action à réaliser. Par conséquent, la relation invalide a fait ressortir une restructuration intéressante de l'application. Pour mettre à jour cette documentation, nous faisons deux choses :

1. nous documentons le mécanisme de dispatching en définissant une relation intentionnelle qui exige que les méthodes `execute` fassent un appel à une de leurs méthodes `execute*` (où `*` est une chaîne non vide de caractères). Par exemple, la méthode `execute` de la classe `ChapterEdit` appelle soit `executeCancel` soit `executeSave` si le bouton correspondant est poussé et exécute la méthode `executeEdit` sinon ;
2. nous documentons le fait que les méthodes `execute` ne peuvent plus envoyer de messages vers une variable d'instance nommée `html` (qui était typiquement le moyen de faire le rendu).

Après avoir réalisé ces changements, nous trouvons quelques violations sur ces nouvelles contraintes mais nous ne les codons pas d'explicites déviations, pour souligner qu'ils sont de vrais conflits de conception qui devraient être fixés dans une future version du code.

Structure Classes *est un sous-ensemble de* Storable Classes Cette relation est fausse à cause de l'ajout de deux nouvelles super classes intermédaires. Nous définissons ces deux classes comme des cas de déviations de la relation.

Comparaison de la taille des vues

En utilisant le *Intensional View Displayer*, nous pouvons comparer la taille de toutes les vues de la version 1.54 avec celles de la version 1.90. Nous voulons trouver s'il existe d'importantes différences de tailles, ce qui pourrait indiquer des problèmes potentiels. Nous réalisons cela en utilisant le *Intensional View Displayer* et nous choisissons le nombre d'entités dans l'extension comme la hauteur de la vue. Cette opération ne fait pas apparaître de réels problèmes, excepté pour la vue Actioned Structure Classes et sa vue cousine Structured Action Classes qui deviennent toutes les deux vides. En essayant de comprendre la raison, nous trouvons que la définition de la vue a besoin d'un raffinement. L'introduction de quelques classes intermédiaires dans la version 1.90 nous force à utiliser le prédicat `classInHierarchyOf` au lieu de `subclassOf` [4].

Nouvelles vues et relations

À cause de la restructuration du code dans la version 1.90, nous avons besoin d'ajouter une nouvelle vue et une nouvelle relation :

Rendering Methods La restructuration des méthodes d'exécution nous décide à définir une nouvelle vue intentionnelle regroupant toutes les méthodes de rendu.

Execute Methods *appellent* Rendering Methods La restructuration a causé le déplacement de la responsabilité du rendu des pages Web des méthodes d'exécution vers les méthodes de rendu, mais le rendu est toujours déclenché par les méthodes d'exécution.

7.2.3 Expérience 3 (Vérifier le design avec *SmallWiki* 1.304)

Dans la troisième et dernière expérience, nous vérifions une nouvelle fois notre documentation sur le design sur une version encore plus récente de *SmallWiki*(celle-ci est représentée visuellement sur la figure 7.4) et tirons les conclusions sur l'utilité des vues et relations intentionnelles pour documenter l'architecture d'un programme en évolution sur une longue période de développement. Encore une fois, la documentation semble tout à fait stable, mais néanmoins nous découvrons quelques vues et relations inconsistantes intéressantes, qui sont discutées ci-après. Nous avons aussi comparé la taille des vues avec celles de la version précédente.

Vues inconsistantes

Outputtable Classes devient inconsistante à cause de l'ajout de quatre nouvelles classes. Au lieu d'avoir une méthode `accept*` comme les autres classes qui peuvent être rendues (et comme suggéré par la définition de la vue), ces classes délèguent leur méthode `accept*` à une autre plus générale. Nous résolvons ce

[4] `subclassOf` donne toutes les sous-classes d'une classe donnée tandis que `classInHierarchyOf` donne toutes les classes de la hiérarchie d'une classe donnée (comprenant donc la classe elle-même).

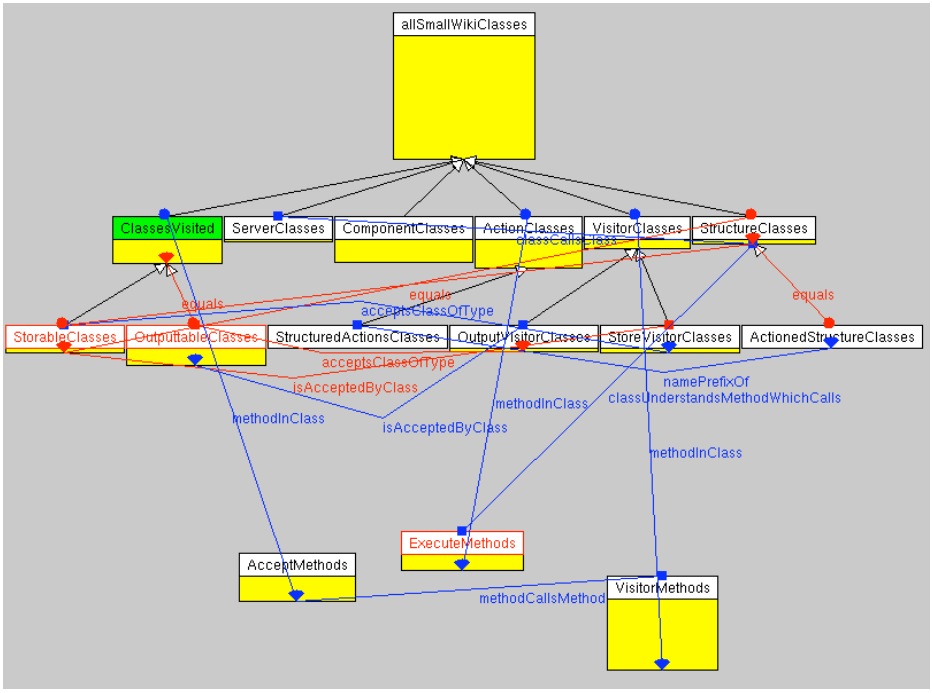

FIG. 7.4 – Vues et relations intentionnelles sur la version 1.304

problème en raffinant la définition de la vue en la déclarant comme une conjonction des classes possédant une méthode `accept*` avec les classes qui délèguent leur méthode `accept*`.

Storable Classes devient également inconsistante, comme on peut le voir sur la figure 7.4, pour les mêmes raisons que la vue Outputtable Classes. En redéfinissant cette vue de manière analogue, la consistance de cette vue est retrouvée.

Execute Methods n'est plus conforme parce que certaines conversions de codage n'ont pas été respectées dans cette nouvelle version : il existe deux méthodes `execute` qui ne sont pas implémentées dans le protocole correct, et il y a deux autres méthodes qui sont dans le bon protocole mais ne sont pas préfixées par 'execute'. Pour fixer ce problème, il suffit juste bouger les méthodes dans le protocole correct tandis que les dernières ont besoin d'être soit renommées, soit placées dans un protocole plus approprié.

Relations invalides

Structure Classes *est un sous-ensemble de* Storable Classes et **Classes Visited *est un sous-ensemble de* Outputtable Classes** est invalide à cause des problèmes que nous venons de citer concernant les vues Storable Classes et Outputtable Classes. Après avoir résolu les problèmes de ces vues, ces relations redeviennent valides.

Storable Classes *sont toutes acceptées par* Store Visitors est elle aussi invalide car l'argument passé à la méthode `accept`: de la classe `LinkInternalVisitor` s'appelle '`anInternalLink`' et non 'aLinkInternal' comme prévu (il faut se souvenir que le prédicat défini pour cette relation relie le fait que les noms des

arguments respectent une convention de nomination particulière). Nous fixons ce problème en renommant cet argument.

Store Visitors *acceptent toutes les classes du type* Storable Classes n'est pas consistante à cause de l'ajout de nouvelles classes d'enregistrement qui ne sont pas prises en considération par la vue Storable Classes. Nous résolvons ce conflit en étendant explicitement la vue Storable Classes.

Output Visitors *acceptent toutes les classes du type* Outtupable Classes est invalide car, dans la version originale de cette relation intentionnelle, nous documentions les classes `AnObsoleteVisitorOutput` et `AnObsoleteVisitorHtml` comme des cas de déviation explicites de la relation. Ces classes ont été supprimées du code entre l'expérience 2 et 3, ce qui cause la défaillance de la cohérence de cette relation. On fixe ce problème en supprimant les cas de déviation de la documentation.

Comparaison de la taille des vues

Nous comparons la taille des (extensions des) vues intentionnelles de la version 1.90 avec celles de la version 1.304. Nous observons alors deux différences importantes : le nombre de classes reprises par la vue Action Classes a pratiquement doublé (de 13 à 25), dû à l'ajout de beaucoup de fonctionnalités dans *SmallWiki*, tandis que le nombre de méthodes définies par Execute Methods a diminué de 23 à 14, illustrant la migration continue de l'ancien style des méthodes d'exécution vers celles qui utilisent le pattern de visiteur.

7.3 Analyse critique et leçons tirées

Dans cette section, basée sur les expériences menées sur le cas d'étude *SmallWiki*, nous analysons de manière critique la génération actuelle des outils — incluant les nouvelles opportunités offertes par l'outil de visualisation développé dans le cadre de ce mémoire — et du modèle fondamental des vues et relations intentionnelles, afin de supporter la co-évolution d'un design de haut niveau et le code source d'une application *Smalltalk* de taille moyenne.

Cas de déviations Les expériences montrent l'importance de la possibilité de définir des cas de déviations explicites (inclusions et exclusions) pour les vues et les relations intentionnelles. Ces cas apparaissent lorsque l'implémentation devrait convenir à une intention ou à une relation, mais que pour des raisons variées, l'implémentation ne convient pas. Typiquement, ceux-ci indiquent une opportunité de refactoriser le code ou de raffiner l'intention exprimée trop largement. Pour ces deux cas, il était utile de documenter les cas de déviation explicitement. Quand on modifie le code ou l'intention et qu'on vérifie de nouveau la consistance, l'outil avertit l'utilisateur, d'une manière ou d'une autre, des déviations qui sont devenues désuètes, nous confirmant alors que ces cas exceptionnels sont résolus, si bien que nous pourrions sans risque enlever la déviation correspondante.

Complétude Bien que les vues et les relations intentionnelles nous aient permis d'exprimer et vérifier des contraintes structurales intéressantes au sujet du code source, la documentation obtenue du design était nullement complète. Par exemple, il pourrait s'avérer utile de compléter cette documentation avec des informations plus dynamiques produites par d'autres outils.

Informations statiques versus informations dynamiques En effet, les deux langages d'interrogation supportés par notre outil (*SOUL* et *Smalltalk*) nous ont permis de définir les vues et les relations qui raisonnent au sujet de *la structure*

statique d'un système uniquement. Bien que nous n'ayons pas éprouvé le manque d'information dynamique comme obstacle grave en documentant l'architecture de *SmallWiki*, nous convenons que cette restriction peut nous interdire de documenter quelques contraintes de design intéressantes. Par exemple, il est très difficile d'exprimer le concept d'une architecture en couche sans l'utilisation d'informations dynamiques.

Approche incrémentale Lors de nos expériences, nous avons adopté une approche incrémentale pour documenter *SmallWiki*. À partir d'une connaissance minimale du fonctionnement du programme, nous avons graduellement raffiné et documenté notre connaissance au sujet du système en alternant l'inspection manuelle du code et la définition de vues et de relations, en validant toujours celles-ci sur le code source. Les outils nous ont aidés dans la codification et les tests de nos présomptions sur la structure du code et en découvrant où nos idées étaient (ou sont devenues) invalides et pourquoi. Cette approche nous a non seulement permis d'obtenir une documentation assez précise de la structure de *SmallWiki*, mais nous a en même temps aidés à obtenir une meilleure compréhension de l'implémentation du système. De plus, nous avons observé que vérifier la documentation sur de plus récentes versions du code nous a souvent fournis des informations dont le design de l'application a évolué.

Co-évolution Le but de la suite d'outils *IntensiVE* est de supporter la co-évolution du code et de la documentation du design l'accompagnant. À cette fin, nos outils supportent la détection de conflits structurels entre la documentation et le code, lorsque l'un des deux évoluait. Nous pouvons distinguer deux genres de conflits. Un premier type de conflit arrive lorsque la documentation est conceptuellement correcte, mais que quelques éléments dans le code la violent. Cela apparaît quand un bug est introduit dans le code (par exemple, supprimer une méthode qu'il ne fallait pas) ou lorsqu'une certaine convention de codage ou de nomination (par exemple, en mettant une méthode dans un mauvais protocole) ou d'architecture est violée (par exemple, en ajoutant une classe qui peut être visitée mais dont les méthodes appropriées n'ont pas été implémentées). Pour fixer ce genre de conflit, le code a besoin d'être adapté. L'autre type de conflit qui peut arriver est causé par les restructurations de code qui affectent la documentation originale du design. De tels conflits doivent typiquement être résolus en modifiant la documentation, c'est-à-dire en adaptant les vues et les relations.

Étonnamment, la majorité des conflits que nous avons détectés étaient de la deuxième sorte, c'est-à-dire qu'ils ont été provoqués par des restructurations de code de *SmallWiki*. Une explication possible de la pauvreté des conflits de la première sorte est que nous n'avons pas appliqué la documentation à un système en cours de construction, mais plutôt 'à posteriori' sur des versions de *SmallWiki* qui étaient déjà stables et testées.

Visualisation *Intensional View Displayer* a été utilisé lors de ces expériences. En utilisant d'une bonne manière cet outil basé sur *CodeCrawler*, nous pourrions l'employer non seulement pour afficher les vues et les relations avouées, mais également mettre en évidence les vues et les relations inconsistantes et nous aider à évaluer l'impact d'une évolution du système. En utilisant cette intégration dans *CodeCrawler*, avec des métriques appropriées, nous pourrions par exemple visualiser la taille des vues et la cardinalité des différences entre les diverses alternatives d'une vue. C'est un apport non négligeable aux précédents outils où nous devions manuellement examiner toutes les vues et relations afin d'avoir une idée de l'impact de l'évolution de la documentation. L'un des inconvénients de l'utilisation de la visualisation est que, lorsque le nombre de vues et de relations augmente, la représentation visuelle devient encombrée. Une solution partielle de

ce problème est de visualiser uniquement une sélection des vues et des relations.

Choix du langage d'interrogation Une question intéressante lors de l'utilisation d'*IntensiVE* est de savoir quel langage d'interrogation choisir. Lors de la définition d'une vue ou d'une relation intentionnelle, devons-nous choisir de préférence la requête logique plutôt que les requêtes *Smalltalk*, ou peut-être utiliser des requêtes hybrides ? Le principe de base que nous avons adopté était de choisir toujours le langage le plus adapté à nos besoins, c'est-à-dire le langage dans lequel nous pourrions exprimer la requête ou le prédicat de manière la plus compacte, tout en restant dans l'approche déclarative. En pratique, il s'est souvent avéré qu'une requête hybride était la plus appropriée. Par exemple, nous pourrions définir la vue Structured Action Classes par la requête logique suivante :

```
classWithName(?entity,?ename),
endsWith(?ename,['Action']),
classInViewNamed(?c,StructureClasses),
classWithName(?c,?cname),
equals([?cname, 'Action'], [?ename asString])
```

En utilisant un mélange de code *Smalltalk* et logique, cependant, nous pouvons écrire la requête de manière plus compacte, en faisant une recherche de modèle sur une chaîne de caractères en *Smalltalk* et le raisonnement sur la structure du code en langage logique :

```
['*Action' match: ?entity name],
subclassOf(?c, [SmallWiki.Structure]),
[(?c name, 'Action') = ?entity name asString]
```

Dans un cas extrême, nous avons même trouvé une requête hybride qui prenait 4 lignes de code, alors que la même requête, écrite en *Smalltalk* en prenait 17.

Néanmoins, sans aller dans des détails techniques, lors de l'utilisation d'*IntensiVE*, nous avons de temps en temps noté quelques limitations quand nous essayions de mélanger des requêtes et des prédicats définis dans des différents langages. Pour résoudre ces limitations, une meilleure intégration et symbiose des langages de raisonnements (*Smalltalk* et *SOUL*) et des librairies est requise (comme celle proposée dans [Gyb03]).

A-t-on encore besoin de la logique ? D'un autre côté, aucune des vues et relations déclarées dans ce cas d'étude n'a fait appel à toute la puissance offerte pour notre langage de programmation logique *SOUL*. Par conséquent nous pourrions utiliser probablement un mécanisme de requête moins expressif mais plus rapide comme SmallLint [RBJO96], tout en gardant la possibilité de coder exactement les mêmes vues. Mais alors nous perdrions également les facilités d'abstraction offerts par notre langage de programmation logique, ainsi que sa bibliothèque de logique contenant un set extensif de prédicats pouvant raisonner sur du code *Smalltalk*.

Chapitre 8

Conclusion

Nous sommes arrivé à la fin de ce mémoire. Ce dernier chapitre va nous permettre de résumer le contenu de celui-ci et de développer quelques idées pour un travail futur concernant le modèle des vues et relations intentionnelles et de l'environnement graphique l'accompagnant.

8.1 Résumons un peu...

Ce mémoire tourne autour d'un modèle conceptuel qui permet de documenter, via des vues et relations intentionnelles écrites dans un langage déclaratif, un code source orienté objet. Quatre outils graphiques permettant de le manipuler ont été développés au-dessus de ce modèle.

La recherche retracée par ce mémoire se compose essentiellement de deux parties.

Lors de la présentation des outils de manipulation des vues et des relations intentionnelles, nous avons pu relever quelques manques pour utiliser le plus efficacement possible le modèle initial. En effet, aucun outil ne permet de connaître, par exemple, quelles sont les relations ou les vues inconsistantes présentes dans le système de manière simple et rapide. Un cinquième outil a donc été développé et une validation de celui-ci a été effectuée. En donnant une vue à cette outil, celui-ci va créer un graphique représentant la hiérarchie dont cette vue est le sommet sous la forme d'un arbre de boîtes annotées des noms des vues qu'elles représentent. Le design de ces boîtes a été spécialement étudié afin d'offrir un maximum de clarté à l'utlisateur. Des métriques appliquées à ces boîtes viennent ajouter des informations primordiales à la compréhension et la correction de la documentation. Des liens rapides permettent de revenir aux outils d'édition des vues ou relations.

La deuxième partie valide à la fois le modèle intentionnel et les outils l'accompagnant. Cette validation a été réalisée sous forme d'expériences menées sur un cas réel : *SmallWiki*, un framework pour créer des serveurs web permettant d'élaborer en groupe des pages web. Nous avons commencé par essayer de documenter une première version de *SmallWiki* se composant du stricte nécessaire, à savoir la possibilité de lancer un serveur Web simpliste, de créer et d'éditer une page web. Cette première version de la documentation recueille un nombre de vues non négligeables et des relations très intéressantes telles que la documentation sur le pattern de Visiteur. Nous avons ensuite pris une seconde version plus élaborée à laquelle nous avons appliqué la documentation préalablement construite. Nous avons alors pu remarquer la puissance que nous offre le modèle et les outils l'accompagnant : les vues et les relations non valides sur le code source sont indiquées en rouge sur le graphe que nous donne l'Intentional View Displayer, outil développé dans la première phase de la recherche. Les outils de confor-

mance (Relation Checker et View Consistency Checker) permettent alors de retrouver les éléments du code source qui ne respectent pas la documentation définie. Il faut alors soit changer la documentation soit changer le code. Ces deux cas sont discutés dans la validation. Une troisième expérience menée sur l'une des dernières versions de *SmallWiki* montre que le design défini par la documentation évoluée (lors de la deuxième expérience) reste à peu près constant.

8.2 Idées pour de futurs travaux

La recherche peut toujours être poursuivie pour améliorer l'existant. Les quelques idées qui suivent sont des pistes pour la poursuite de cette recherche.

8.2.1 Une validation sur un système plus grand

Il serait intéressant de tester ce type de documentation sur des systèmes de plus grande envergure. *SmallWiki* n'est pas trivial mais n'est pas extrêmement grand si l'on considère les programmes utilisés dans la plupart des entreprises. Combien de temps faudrait-il pour effectuer le calcul de l'extension des vues lors de la création d'un graphique représentant un grand nombre de vues intentionnelles ? La visibilité serait-elle encore au rendez-vous ? On peut se demander de plus si une documentation écrite via les vues et relations intentionnelles de grandes envergures pourrait encore être gérée efficacement et rester compréhensible.

Dans ce but, nous avons pris contact avec une entreprise intéressée par ce système afin de documenter un de leur programme utilisant plus de trois milles classes, développé depuis dix ans et utilisé dans un monde professionnel réel. Cette expérience nous donnera les réponses manquantes pour le moment.

8.2.2 La visualisation

Une meilleure visualisation d'un système intentionnel peut toujours être trouvée. Nous pourrions par exemple mettre un sens aux lignes représentant les relations intentionnelles. Cela permettrait de connaître facilement le sens de lecture de la relation. Mais *CodeCrawler* ne permet pas de le faire de manière propre et simple.

Nous avons pu aussi relever un autre problème. Les vues alternatives sont aux mêmes niveaux que les sous-vues. Même si les annotations sur les lignes de relations "Vue/Sous-vue" et "Vue principale/Vue alternative" permettent de différentier les deux cas, cela reste potentiellement perturbant pour l'utilisateur.

Nous pouvons aussi nous demander si l'utilisation de *CodeCrawler* est suffisante. Bien que *CodeCrawler* nous permet de créer facilement des graphiques avec des métriques et des couleurs, dès que nous voulons ajouter de nouvelles fonctionnalités graphiques cela se corse. Cela est dû au fait que *CodeCrawler* n'est pas un outil pour faire du graphisme mais pour étudier du code graphiquement.

8.2.3 Représentations alternatives de programme

Pour le moment, la documentation se base directement sur du code *Smalltalk*. Il serait peut-être intéressant d'utiliser un programme tel que *MOOSE*(cfr. 31) afin de travailler sur une représentation d'un programme informatique indépendante de tout langage de programmation.

Cela permettrait d'utiliser le modèle des vues et relations intentionnelles sur des langages différents sans changer les outils graphiques pour autant. *CodeCrawler* utilise d'ailleurs ce même procédé pour analyser n'importe quel programme orienté objet.

8.3 Le point final

Cette année de DEA a été riche en expériences. Ne connaissant ni la langue de Shakespeare ni la plupart des outils que j'ai dû utiliser pour avancer dans cette recherche, j'ai dû faire face à un monde que je ne connaissais que trop mal. Mais petit à petit, conférence après conférence, lectures après lectures, ainsi que toutes les réunions de recherche, les nombreux séminaires, les colloques, les symposiums, j'apprends un métier fascinant rempli de défis et de connaissances.

Bibliographie

[Dek02] U. Dekel. Applications of concept lattices to code inspection and review. *technical report, Dept. of Computer Science*, 2002.

[DL05] Stéfane Ducasse and Michele Lanza. The class blueprint : Visually supporting the understanding of classes. *IEEE Transaction on software engineering*, 2005.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Languages and Systems*. Addisson-Wesley, 1994.

[Gyb03] Kris Gybels. Soul and smalltalk - just married : Evolution of the interaction between a logic and an object-oriented language towards symbiosis. In *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*, 2003.

[Liu96] Chamond Liu. *Smalltalk, Objects, and Design*. 1996.

[MMW02] Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. *Proceedings of Conference on Software Engineering and Knowledge Engineering (SEKE2002)*, pages 289–296, 2002.

[PW92] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4) :40–52, 1992.

[RBJO96] D. Roberts, J. Brant, R. Johnson, and B. Opdyke. An Automated Refactoring Tool. In *Proceedings of ICAST 1996, Chicago, IL*, April 1996.

[Ren05] Lukas Renggli. Collaborative web : Under the cover. Master's thesis, University of Berne, 2005.

[SDD05] Michele Lanza Stéphane Ducasse, Tudor Gîrba and Serge Demeyer. Moose : a collaborative and extensible reengineering environment. *Tools for Software Maintenance and Reengineering*, 2005.

[TBKG03] Tom Tourwé, Johan Brichau, Andy Kellens, and Kris Gybels. Induced intentional software views. *Elsevier Science. In "Special Edition of Elsevier's Computer Languages Journal"*, 2003.

[WBM03] De Meuter Wolfgang, Johan Brichau, and Kim Mens. Soul manual, 2003.

[WD04] Roel Wuyts and Stéphane Ducasse. Unanticipated integration of development tools using the classification model. *Computer Languages, Systems and Structures*, 30(1-2), 2004.

[Wuy01] Roel Wuyts. A logic meta-programming approach to support the co-evolution of object-oriented design and implementation. Master's thesis, Vrije Universiteit Brussel, January 2001.