COMMUNAUTE FRANCAISE
HAUTE ECOLE ROBERT SCHUMAN
CATEGORIE ECONOMIQUE
6800 LIBRAMONT

Création d'un éditeur de pseudo-code multiplate-forme pour fournir un support pratique au cours de « *principes de programmation* ».

Travail de fin d'études présenté
pour l'obtention du grade académique de
Bachelier en Informatique de gestion
par
DEBOT Jérôme
Septembre 2017

Remerciements

En premier lieu, je voudrais remercier l’ensemble du personnel de la Haute École Robert Schuman pour leur travail, leur engagement et leur patience qui font de la Haute École Robert Schuman, un établissement renommé.

Aussi, je voudrais remercier plus particulièrement :

- monsieur C. Peters, mon promoteur de TFE, pour sa patience, ses encouragements et ses conseils ;

- madame B. Naisse, ma cliente et ex-promotrice, pour avoir accepté mon projet, ainsi que pour sa patience et sa précision.

- monsieur V. Spies, enseignant en programmation web, pour son aide et ses bonnes idées ;

- madame I. Dony, enseignante en programmation orientée objet, pour ses impressions et ses commentaires positifs ;

Table des matières

1. Résumé

Création d'un éditeur de pseudo-code multiplate-forme pour fournir un support pratique au cours de « *principes de programmation* » de la section informatique de gestion à la Haute École Robert Schuman.

Le pseudo-code est un langage non standardisé et informel, utilisé pour publier et enseigner des algorithmes. Il se veut abstrait et n'emploie pas la syntaxe d'un langage particulier.

Une variante très illustrée de pseudo-code est utilisée à la Haute École Robert Schuman de Libramont dans le cadre du cours de « *principes de programmation* ». Les étudiants et les enseignants sont régulièrement amenés à dessiner ce langage sur papier, à défaut de disposer d'un logiciel adapté et intuitif.

Le but de l'éditeur de pseudo-code que je développe, est de palier à ce problème en fournissant une solution informatisée et simple d'utilisation, qui permettra la rédaction, l'enregistrement et le chargement de documents pseudo-code. L'implémentation de fonctionnalités supplémentaires n'est pas à exclure : impression, exécution, commentaires, etc.

2. Abstract

Creating a cross-plateform pseudocode editor to provide a convenient medium for the « *programming principles* » course of the business informatics division at the Robert Schuman Business College.

The pseudocode language is a non-standard and informal language used to ease the publishing and the learning of algorithms. It tends to be abstract and keeps its distance from any particular language's syntax.

At Robert Schuman Business College, the « *programming principles* » course utilize a very illustrated variant of pseudocode. In the absence of an intuitive and suitable software to sketch pseudocode, students and teachers commonly draw pseudocode on paper.

The goal of the editor, I'm currently developing, is to provide an easy-to-use digital solution for sketching, saving and loading pseudocode documents. Also, some optional features could be added to the project : printing, running, comment, etc.

3. Introduction

Afin de terminer leur cursus de baccalauréat, les étudiants en informatique de gestion de la Haute École Robert Schuman doivent réaliser un travail de fin d'études personnel. Le contenu de ce travail est à l'initiative de l'étudiant. Cependant, ce contenu doit s'apparenter soit à la création d'une solution logicielle, soit à l'étude exhaustive d'un sujet lié à la formation. La décision de l'étudiant doit être approuvée par le conseil de classe qui lui attribuera un promoteur afin de le suivre et de le superviser dans son travail.

Mon travail de fin d'études consiste en la création d'une application à but éducatif pour le compte de madame Naisse, dans le cadre de son travail à la Haute École Robert Schuman. Cette application est conçue pour assister les étudiants et les enseignants dans la rédaction et l'apprentissage du pseudo-code. Le développement de ce logiciel a été motivé par l'observation de difficultés d'apprentissage chez certains étudiants et par la pénibilité éprouvée lors de la rédaction de documents pseudo-code.

Afin d'évaluer les travaux de fin d'études des étudiants, le corps enseignant enjoint ces derniers à rédiger un rapport descriptif de leur travail. Ce rapport doit être conforme aux exigences présentées dans le règlement des études de la section informatique. Ainsi, conformément à ce règlement et tout en prenant certaines libertés, le rapport ci-présent développera les thèmes suivants :

- une courte présentation du contexte dans lequel s'écrit le travail : entreprise, client demandeur, utilisateurs, environnement informatique préexistant ;
- un exposé du problème que le travail vise à résoudre : demandes du client, définition claire et complète du problème, liste des atouts et contraintes matériels et logiciels, etc ;
- une analyse fonctionnelle et décisionnelle : schémas des données, choix techniques, croquis ou captures d'écran, schémas métier, etc ;
- un passage en revue des différentes technologies abordées dans ce document ou utilisées dans le cadre du projet ;
- un résumé de la phase de développement de la solution : les différentes approches essayées, les principales difficultés rencontrées et les réponses apportées à ces dernières ;
- une brève conclusion.

Une bibliographie des sources utilisées pour réaliser ce travail en annexe à la fin du document.

4. L’établissement

La Haute École Robert Schuman (HERS) de Libramont est un établissement d’enseignement supérieur, situé au 64 rue de la Cité à Libramont, dans la province de Luxembourg, en Belgique. Elle fait partie intégrante du réseau des hautes écoles Robert Schuman et propose des formations dans les domaines de l’économie et du paramédical.

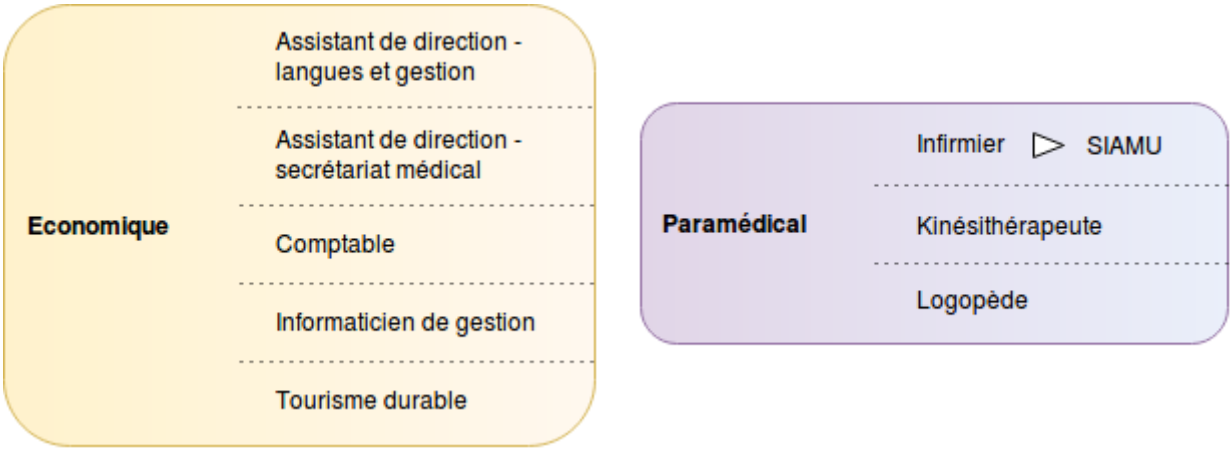


Illustration 1: Un représentation des sections d'enseignement de la HERS regroupées dans leur catégorie respective.

4.1. La section informatique de gestion

Le baccalauréat (+3) en informatique de gestion est l’une des formations proposées par la Haute École Robert Schuman de Libramont. Ce baccalauréat est dédié aux domaines de l’informatique commerciale et de l’informatique des services.

Au cours de cette formation, les étudiants se familiarisent avec les mondes de la programmation, de l’entreprise et de l’économie. Aussi, à la fin de leur cycle d’études, les étudiants d’informatique de gestion doivent :

- posséder des connaissances approfondies en méthodes de programmation, théorie des langages, gestion et structuration des données, études des systèmes d’exploitation, études des réseaux, analyse fonctionnelle, modélisation et gestion de projets ;
- savoir évoluer continuellement et de manière autonome dans l’apprentissage de nouvelles technologies et de nouvelles techniques ;
- être aptes à travailler en équipe, respecter des échéances et assumer des responsabilités dans une entreprise.

4.2. Infrastructure logicielle et matérielle

La Haute École Robert Schuman possède plusieurs salles informatiques utilisées dans le cadre des cours, ainsi qu’un réseau filaire et wifi connecté à internet et mis à libre disposition des

étudiants. Aussi, les étudiants en informatique préfèrent, quant à eux, généralement apporter et utiliser leur propre matériel informatique.

5. Sujet du travail de fin d'études

Le but de mon travail de fin d'études est concevoir une application permettant aux étudiants et aux enseignants de la section informatique de gestion de rédiger des documents pseudo-code. Pour ce faire, je dois prendre du recul et collecter les données nécessaires avant de démarrer mon projet. En rédigeant un cahier des charges, je m'assure de définir de façon claire, les contours de la future application, j'améliore la communication qui me lie à mon client et je dresse un contrat entre lui et moi.

5.1. Cahier des charges

5.1.1. Introduction

Dans le cadre du cours de « *principes de programmation* » (cours qui s'étend sur le premier et deuxième quadrimestres), les étudiants du baccalauréat en informatique de gestion de la Haute École Robert Schuman sont amenés à étudier et à utiliser un sous-ensemble *impératif* d'un langage non normalisé : le pseudo-code. Le pseudo-code est un pseudo langage expressif, non normalisé et indépendant de toute technologie. Il est principalement utilisé par les algorithmiciens pour exprimer des algorithmes en des termes compris par tous, mais aussi par les enseignants et les étudiants en programmation informatique dans un but didactique, afin d'approcher la logique de programmation sans pour autant entrer dans les carcans d'un langage spécifique.

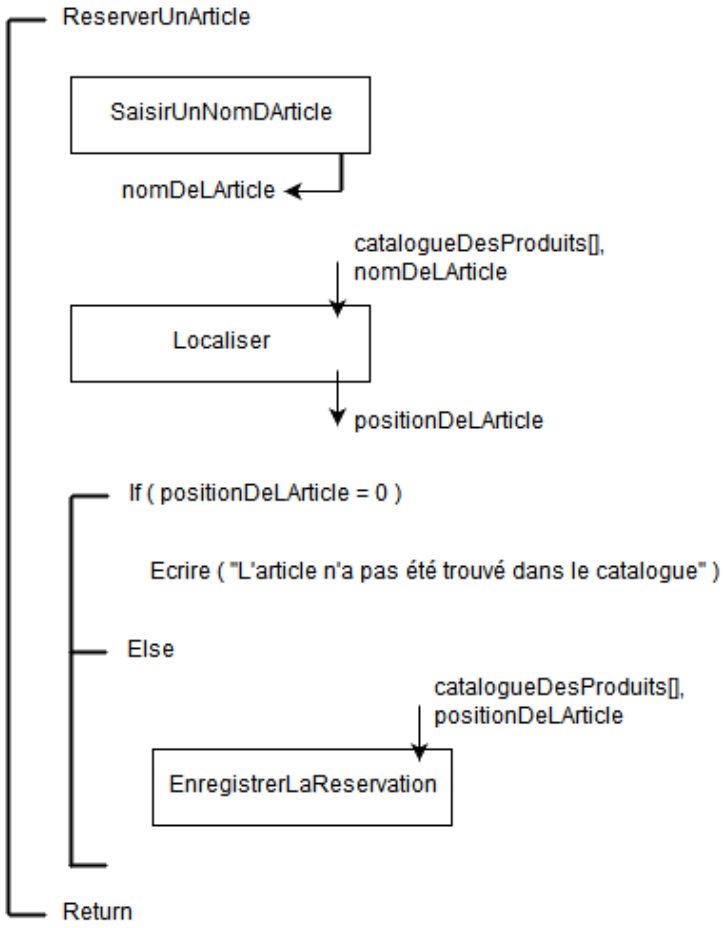


Illustration 2: Un extrait de pseudo-code tel qu'il est enseigné à la HERS.

5.1.2. Contexte du projet

Actuellement la majorité des étudiants participant au cours de « *principes de programmation* » utilisent un support papier pour rédiger leurs documents pseudo-code. D'autres, qui possèdent des connaissances approfondies dans un logiciel de traitement de texte, savent en utiliser les fonctionnalités avancées pour appliquer à leurs documents pseudo-code, la mise en forme convenue et adoptée à la Haute École Robert Schuman. Cependant, aucune de ces méthodes n'est irréprochable.

La première est facilement accessible à tous les étudiants, mais son utilisation devient cauchemardesque si des modifications ou des corrections doivent être apportées au beau milieu de l'algorithme.

La deuxième, quant à elle, est moins accessible aux étudiants et les documents qu'elle produit peuvent s'avérer difficiles à partager dans un format autre que PDF, car cela abîmerait la mise en page (à cause des problèmes de compatibilité entre les différents logiciels de traitement de texte).

Aussi, aucune de ces solutions ne permet de faire plus que de la rédaction de documents pseudo-code. Elles ne permettent pas, par exemple, de vérifier la validité du code écrit ou de

l'exécuter en temps réel. Cette dernière possibilité pourrait s'avérer extrêmement intéressante dans un contexte éducatif.

Le cours de « *principes de programmation* » n'est pas tenu dans l'une des salles informatiques de l'école. Le matériel informatique utilisé dans le cadre de ce cours est donc d'origine personnel. Ainsi une grande variété d'architectures matérielles et de systèmes d'exploitation peut être rencontrée : Windows, Mac, Linux, ChromeOS, X86-64, ARM, etc. Aussi, il paraît intéressant de noter que les étudiants peuvent aussi être amenés à réaliser des travaux de pseudo-code à domicile.

5.1.3. Parties prenantes

Demandeurs

J'ai travaillé pour le compte de madame B. Naisse, enseignante de la section informatique de gestion, de la catégorie économique à la Haute École Robert Schuman de Libramont. Le logiciel doit être utilisé dans le cadre de ses cours.

Futurs utilisateurs de l'application

- Les étudiants :

  Les profils des étudiants qui utilisent le pseudo-code, qu'ils participent ou non au cours de « *principes de programmation* », ne varient que très peu. Ils sont à la recherche d'une solution qui leur permettrait de composer, d'enregistrer et de charger des documents pseudo-code respectant les conventions et la mise en page adoptés à la HERS.

  Aussi, les étudiants sont curieux et intéressés par ce qui pourrait les aider à apprendre.

- Les enseignants :

  Comme pour les étudiants, les profils des enseignants divergent peu. Ils sont à la recherche d'une solution qui leur permettrait de composer, d'enregistrer et de charger des documents pseudo-code respectant les conventions et la mise en page adoptés à la HERS.

  Ils voudraient aussi pouvoir commenter et corriger les travaux de leurs étudiants. Ils pourraient être également intéressés à l'idée de fournir une expérience plus dynamique à leurs étudiants : exhiber l'exécution d'un extrait de pseudo-code en temps réel, etc. D'autres fonctionnalités qui pourraient intéresser les enseignants existes, mais elles ne peuvent être implémentées dans l'immédiat.

Autres parties prenantes

- La Haute École Robert Schuman :

  L'établissement sera le propriétaire du projet (voir règlement des études 2017).

- Monsieur C. Peeters :

  Enseignant de la section informatique de gestion de la HERS et promoteur de ce travail de fin d'études.

5.1.4. Buts du logiciel

L'objectif primaire de ce projet est de fournir un logiciel qui permet de rédiger des documents pseudo-code, le plus simplement possible. Le pseudo-code présenté dans ce projet s'inscrit dans le sous-ensemble de pseudo-code utilisé dans le contexte du cours de « *principes de programmation* » et respecte également la charte graphique adoptée à la HERS.

Voici une liste des instructions pseudo-code présentes dans le sous-ensemble exploité à la HERS.

Affectation :

Illustration 3: Représentation d'une affectation en pseudo-code selon la charte graphique de la HERS.

Une affectation peut s'exprimer en une seule ligne de pseudo-code. Elle indique l'attribution d'une valeur à une variable. Si une expression est utilisée, alors elle doit se résoudre en une valeur.

Lecture :

Illustration 4: Représentation d'une lecture en pseudo-code selon la charte graphique de la HERS.

Une lecture peut s'exprimer en une seule ligne de pseudo-code. Elle signale la capture d'une valeur auprès de l'utilisateur et l'attribution de cette valeur à une variable.

Écriture :

Illustration 5: Représentation d'une écriture en pseudo-code selon la charte graphique de la HERS.

Une écriture peut s'exprimer en une seule ligne de pseudo-code. Elle consiste en l'affichage à l'utilisateur d'une valeur ou d'une expression.

Condition <If> :

Illustration 6: Représentation d'une condition <If> en pseudo-code selon la charte graphique de la HERS.

Une condition <If> s'exprime sur plusieurs lignes de pseudo-code. Elle enclave d'autres lignes et en bloque l'accès si l'on ne respecte pas sa condition de garde.

Condition <If-Else> :

Illustration 7: Représentation d'une condition <If-Else> en pseudo-code selon la charte graphique de la HERS.

Une condition <If-Else> s'exprime sur plusieurs lignes de pseudo-code. Elle enclave d'autres lignes en deux sous-domaines. Le premier n'est accessible que si la condition de garde est respectée et le deuxième ne l'est que si la condition de garde n'est pas respectée.

Boucle <While> :

Illustration 8: Représentation d'une boucle <While> en pseudo-code selon la charte graphique de la HERS.

Une boucle <While> s'exprime sur plusieurs lignes de pseudo-code. Elle enclave d'autres lignes et en force l'exécution en boucle tant que son expression conditionnelle est satisfaite.

Boucle <Until> :

Illustration 9: Représentation d'une boucle <Until> en pseudo-code selon la charte graphique de la HERS.

Une boucle <Until> s'exprime sur plusieurs lignes de pseudo-code. Elle enclave d'autres lignes et en force l'exécution en boucle jusqu'à ce que son expression conditionnelle soit satisfaite.

Boucle <Do While> :

Illustration 10: Représentation d'une boucle <Do While> en pseudo-code selon la charte graphique de la HERS.

Une boucle <Do While> s'exprime sur plusieurs lignes de pseudo-code. Elle enclave d'autre lignes et en force l'exécution en boucle tant que son expression conditionnelle est satisfaite. Toutefois, les lignes comprises dans un <Do While> s'exécuteront une première fois sans tenir compte de son expression conditionnelle.

Boucle <Do Until> :

Illustration 11: Représentation d'une boucle <Do Until> en pseudo-code selon la charte graphique de la HERS.

Une boucle <Do Until> s'exprime sur plusieurs lignes de pseudo-code. Elle enclave d'autres lignes et en force l'exécution en boucle jusqu'à ce que son expression conditionnelle soit satisfaite. Toutefois, les lignes comprises dans un <Do Until> s'exécuteront une première fois sans tenir compte de son expression conditionnelle.

Appel de fonction :

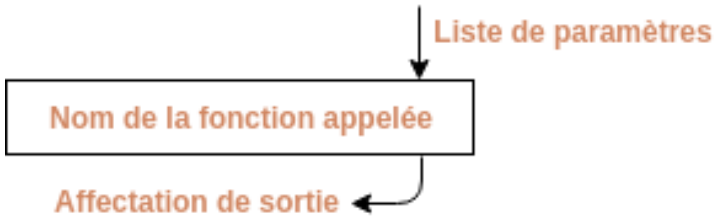


Illustration 12: Représentation d'un appel de fonction en pseudo-code selon la charte graphique de la HERS.

Un appel de fonction peut s'exprimer en une seule ligne de pseudo-code. Il représente l'invocation du code contenu dans une fonction désignée et l'assignation dans une variable de la valeur renvoyée par cette fonction. Toutefois, Il faut que cette fonction ai été déclarée et définie au préalable.
Aussi, pour que l'appel à la fonction puisse fonctionner, il faudra lui fournir une liste de paramètres correspondant aux exigences de la fonction appelée.

Appel de procédure :

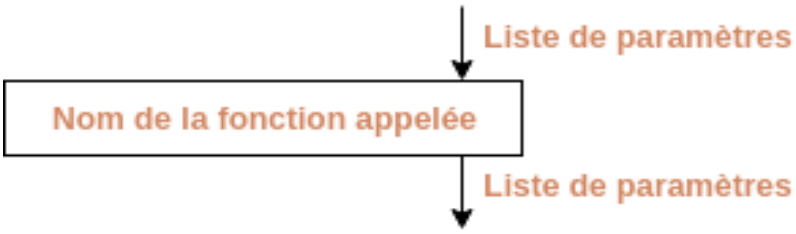


Illustration 13: Représentation d'un appel de procédure en pseudo-code selon la charte graphique de la HERS.

Un appel de procédure peut s'exprimer en une seule ligne de pseudo-code. Il représente l'invocation du code contenu dans une procédure désignée. Toutefois, il faut que cette procédure ai été déclarée et définie au préalable.
Aussi, pour que l'appel à la procédure puisse fonctionner, il faudra lui fournir une liste de paramètres correspondant aux exigences de la procédure appelée.
Une procédure peut renvoyer plusieurs données, représentées à droite de la flèche sortante. Ce sont les paramètres en sortie et en entrée-sortie de la procédure. La flèche en sortie n'est pas une affectation, elle indique juste que ces paramètres ont été *passés par référence* et que, donc, l'appel de la procédure aura des effets de bord.

Déclaration de fonction :

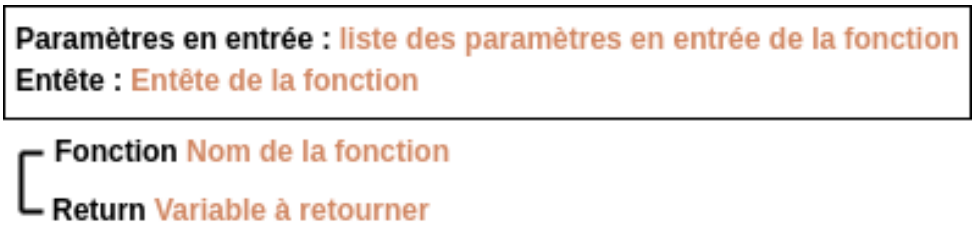


Illustration 14: Représentation d'une déclaration de fonction en pseudo-code selon la charte graphique de la HERS.

Une déclaration de fonction s'exprime sur plusieurs lignes de pseudo-code. Elle enclave d'autres lignes qui forment alors le corps de la fonction déclarée.
Une fonction ne peut retourner qu'une seule valeur, indiquée à la fin de la déclaration de la fonction.
Une fonction peut accepter plusieurs paramètres en entrée, obligatoires pour faire fonctionner la fonction. Ces paramètres peuvent être modifié au sein de la fonction, mais cela n'aura aucun impacte sur leur situation en dehors de celle-ci.
L'entête de la déclaration d'une fonction est composée du nom de la fonction suivi de la liste des paramètres en entrée et s'achève avec le type de la variable de retour.
Exemple : FonctionAddition (nb1:int, nb2:int) : int

Déclaration de procédure :

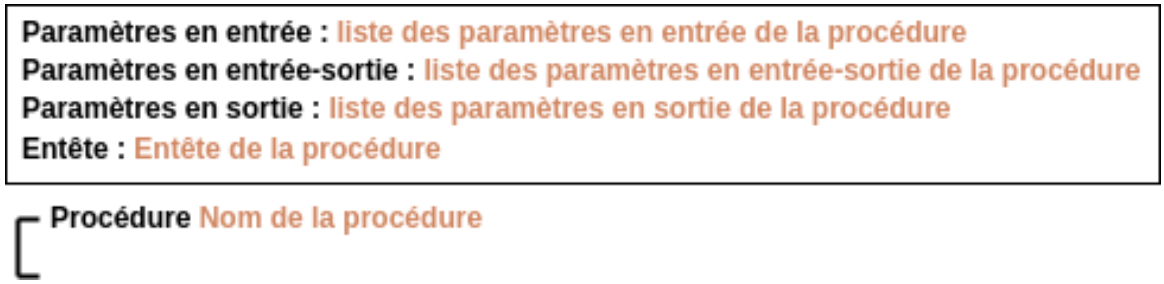


Illustration 15: Représentation d'une déclaration de procédure en pseudo-code selon la charte graphique de la HERS.

Une déclaration de procédure s'exprime sur plusieurs lignes de pseudo-code. Elle enclave d'autres lignes qui forment alors le corps de la procédure déclarée.
Une procédure peut accepter plusieurs paramètres en entrée et en entrée-sortie, obligatoires pour faire fonctionner la procédure.
Les paramètres en entrée peuvent être modifié au sein de la fonction, mais cela n'aura aucun impacte sur leur situation en dehors de celle-ci.
Les paramètres en entrée-sortie, quant à eux, si ils sont modifiés au sein de la procédure, seront impactés en dehors de celle-ci. On dit qu'ils sont *passés par référence*.
Les paramètres en sortie sont des paramètres que l'on fournit à la procédure pour qu'elle les remplisse de valeurs de retour.
L'entête de la déclaration d'une procédure est composée du nom de la procédure suivi de la liste des paramètres en entrée, en entrée-sortie et en sortie.
Exemple : ProcédureRechercheParce (numéro:string, var ligne:int, colonne:int, tableau:array[1..20,1..20] of string)

Déclaration de record :

Illustration 16: Représentation d'une déclaration de record en pseudo-code selon la charte graphique de la HERS.

Une déclaration de record s'exprime sur plusieurs lignes de pseudo-code. Elle enclave d'autres lignes qui forment alors le corps du record déclaré.
La déclaration d'un record est assimilable à la création d'un type composite que l'on pourra utiliser dans notre code par le suite.
On peut interpréter un record comme regroupement de plusieurs variables sous un seul label.
Exemple :

```
---- Type Adresse : record
|       numéro : string
|       rue : string
|       codePostal : string
|       ville : string
|       pays : string
---- End
```

Ce projet vise également à fournir une expérience plus profonde et plus interactive de l'apprentissage de la programmation.

5.1.5. Exigences du logiciel

Exigences fonctionnelles

- Le logiciel permettra aux étudiants et aux enseignants d'enchâsser des composants pseudo-code, les uns à côté des autres ou les uns dans les autres, afin de composer des documents pseudo-code. (Must have)(Accompli)
- Le logiciel permettra aux étudiants et aux enseignants de sauvegarder des documents pseudo-code produits avec l'application. (Must have)(Accompli)
- Le logiciel permettra aux étudiants et aux enseignants de charger, de manière sûre, des documents pseudo-code produits par soi ou par autrui. (Must have)(Accompli)
- Le logiciel permettra aux étudiants et aux enseignants de générer des documents PDF à partir de documents pseudo-code produits avec l'application. (Must have)
- Le logiciel permettra aux étudiants et aux enseignants d'imprimer des documents pseudo-code produits avec l'application. (Must have)
- Le logiciel permettra aux étudiants et aux enseignants d'insérer des commentaires au sein de leurs documents pseudo-code pour en augmenter l'expressivité. (Nice to have) (Accompli)
- Le logiciel permettra aux étudiants et aux enseignants d'exécuter des documents pseudo-code en temps réel pour en observer, plus précisément, le fonctionnement. (Nice to have)
- Le logiciel permettra aux enseignants d'inscrire des corrections facilement distinguables sur les travaux pseudo-code des étudiants. (Nice to have)

Exigences non-fonctionnelles

- Les éléments pseudo-code mis à disposition dans le logiciel doivent correspondre au sous-ensemble de pseudo-code utilisé dans le cadre du cours de « *principe de programmation* » (ce sous-ensemble est illustré au point 5.1.4). Ces éléments doivent aussi respecter la mise en forme convenue à la HERS : indentations, accolades, etc.
- Le logiciel évoluera dans le temps, il faut donc prendre ses précautions quant à la rétro-compatibilité. À ce propos, le chargement de fichiers sauvegardés sera, probablement, le principal point de friction.
- Le chargement et l'exécution en temps réel de documents issus de sources tierces doit se faire de manière sûre. C'est-à-dire sans que cela ne puisse servir de vecteur pour attaquer l'utilisateur d'une manière quelconque : attaquer le système de l'utilisateur, tromper ou incommoder l'utilisateur, etc.

6. Analyse

Cette application s'éloigne de celles habituellement traitées en informatique de gestion. En effet, elle ne vise pas à modéliser un système du monde réel (logique métier), à le stocker (base de données), à le consulter et le faire évoluer à l'aide d'interfaces utilisateur (Gui) plus ou moins prédéterminées.

Cette application met principalement l'accent sur l'affichage et l'interactivité (Front-End). Elle ne comporte pas, à proprement parler, de logique métier, puisque sa logique se résume à des composants graphiques qui s'emboîtent et interagissent de façon visuelle. Aussi, comme dans bon nombre d'applications similaires, cette application utilise des fichiers XML pour sauvegarder les documents réalisés par l'utilisateur.

6.1. Contours et formes de l'application

De toutes les préoccupations qu'un développeur peut avoir lorsqu'il développe un éditeur graphique (éditeur de texte, éditeur d'image, etc), je pense que celle d'offrir une expérience visuelle ergonomique et cohérente devrait être l'une des plus importantes.

N'étant pas graphiste ou développeur Front-End de formation, la réalisation d'une interface graphique esthétique, dynamique et ergonomique se révèle être, pour moi, un défi inédit. L'interface graphique de l'application, ici développée, est minimaliste et se veut autant exploitable sur PC que sur dispositif tactile.

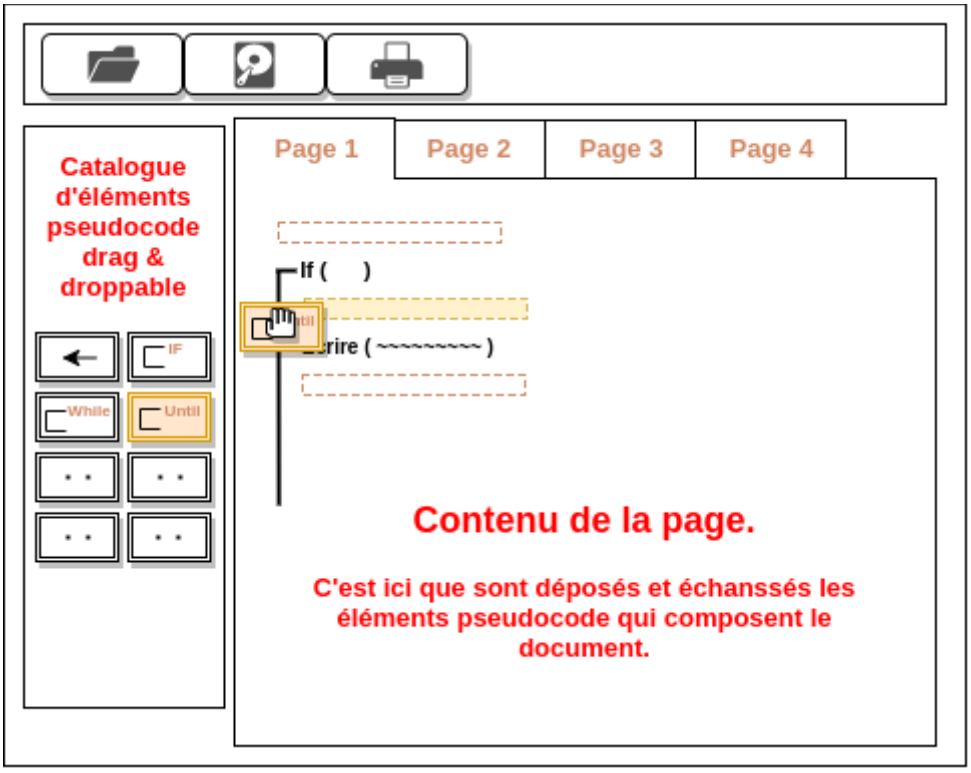


Illustration 17: Maquette de l'interface utilisateur.

6.2. Comportement des éléments pseudo-code

Les éléments pseudo-code et leur assemblage représentent, bien évidement, le corps de l'application. Ils sont les données et la logique de cette dernière. À ce titre, ils méritent une explication particulière.

6.2.1. Structure du pseudo-code et des éléments pseudo-code

En regardant un document pseudo-code, un développeur peut faire une rapide constatation : c'est une structure en arbre n-aire.

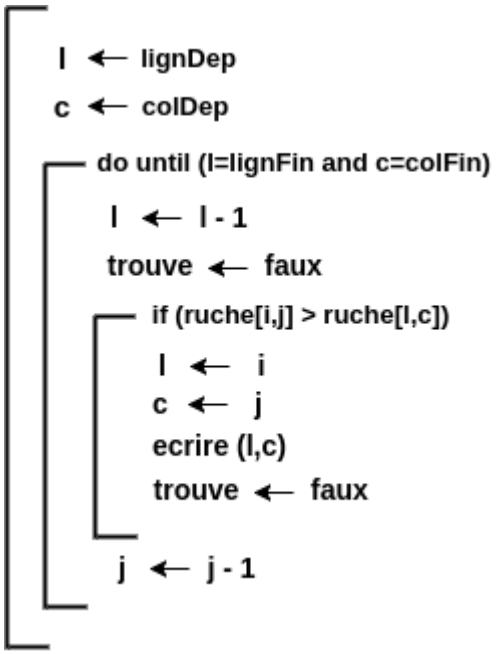


Illustration 18: Un extrait de pseudo-code quelconque.

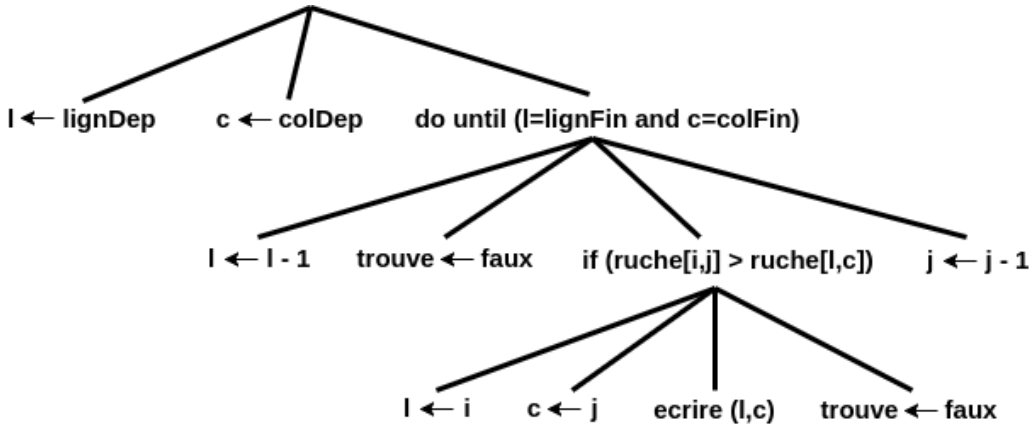


Illustration 19: La représentation en arbre n-aire du pseudo-code présenté dans l'illustration 18.

Cependant, une autre constatation peut être faite : certains éléments peuvent avoir des enfants et d'autres pas. Par exemple, l'assignation *I ← I - 1* n'aura jamais d'enfants.

Ainsi, en pseudo-code, il existe deux catégories d'éléments qui sont de natures fixes : des éléments qui se comportent exclusivement comme les *branches* d'un arbre et des éléments qui se comportent exclusivement comme les *feuilles* d'un arbre. Un développeur habile reconnaîtra en ces mots la définition d'un *composite pattern*.

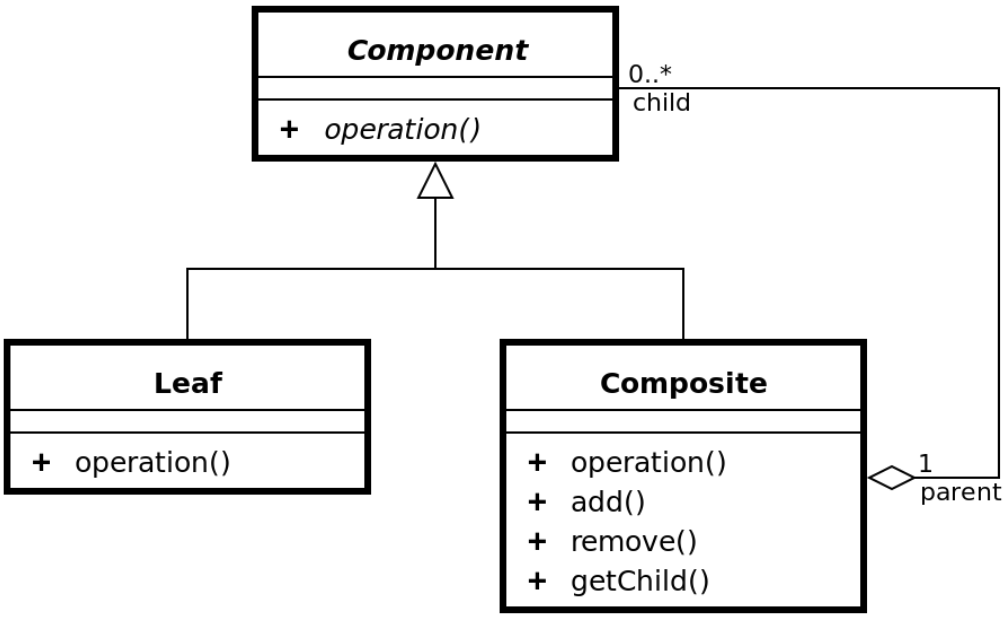


Illustration 20: Schéma UML du composite pattern (Source : Wikipedia.org, License : domaine public)

Dès lors, il nous est possible de classer les différents éléments du sous-ensemble de pseudo-code que nous utilisons (voir au point 5.1.4) entre ces deux catégories.

Éléments *inclusifs* (branches)		**Éléments *plats* (feuilles)**	
> Boucle <While>	> Boucle <Until>	> Affectation	> Lecture
> Boucle <Do While>	> Boucle <Do Until>	> Écriture	> Appel de fonction
> Déclaration de fonction	> Déclaration de procédure		> Appel de procédure
> Condition <If>	> Déclaration de record		

Cependant, Il est un élément que nous n'avons pas encore classé : la condition <If-Else>. Cette dernière est problématique, car bien qu'elle ressemble aux autres éléments inclusifs, elle diverge de ceux-ci par le fait qu'elle accueille, en son sein, deux listes d'enfants distinctes au lieu d'une seule.

Illustration 21: Représentation pseudo-code d'un élément <If-Else> avec mise en évidence de ses deux listes d'enfants

L'élément <If-Else> ne respecte donc ni le modèle d'*arbre n-aire*, ni le *composite pattern*. Pire encore, les éléments enfants se retrouvent dans une situation où ils possèdent deux parents simultanément : l'élément <If-Else> et le racine du sous-groupe auquel ils appartiennent.

Bien qu'il soit possible d'implémenter l'élément <If-Else> de la façon indiquée précédemment. Il serait préférable de garder les choses simples (*principe KISS*) et facilement maintenables. Ainsi, deux solutions me sont venues à l'esprit :

1. **Créer un élément inclusif <Else> et le coller à un élément <If> à l'aide d'un élément *encapsulateur* <If-Else>.**

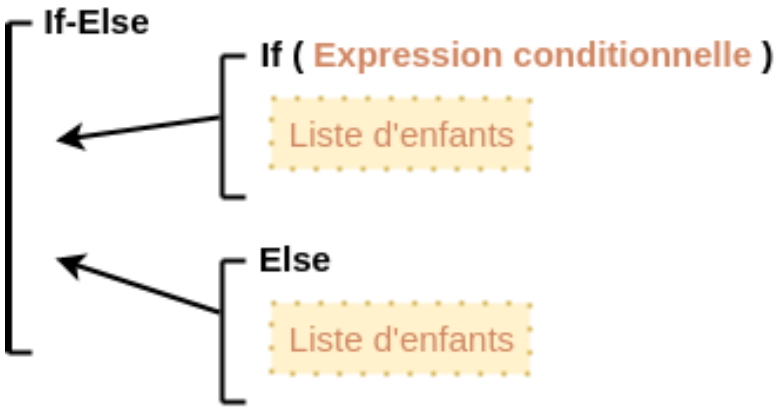


Illustration 22: Schéma explicatif de la solution encapsulatrice au problème de l'élément <If-Else>

Dans cette solution, le sens inclusif de l'élément <Else> reste conservé. Ainsi, il sera plus simple de l'interpréter, si nous le devons par la suite. Par exemple, pour traduire le pseudo-code dans un autre langage.

2. **Créer un élément *plat* <Else> et s'en servir comme séparateur dans un <If> re-labellé en <If-Else>.**

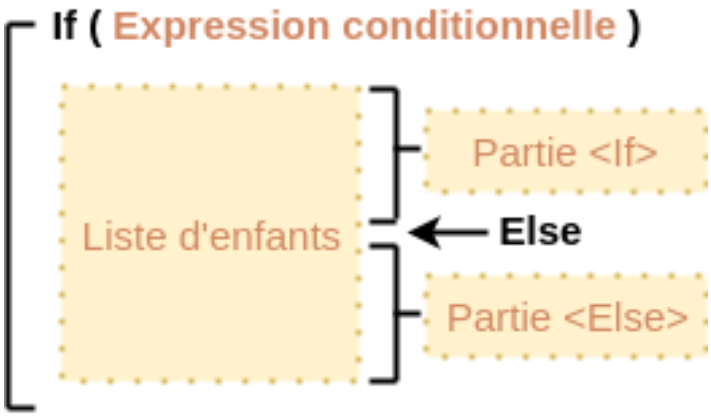


Illustration 23: Schéma explicatif de la solution avec séparateur au problème de l'élément <If-Else>

Cette solution exploite une astuce visuelle très simple à mettre en œuvre. Cependant, elle fait perdre à l'élément <Else> son sens inclusif et pourrait rendre l'ajout de futures fonctionnalités à l'application, plus délicat.

6.2.2. Interactions entre les éléments pseudo-code

Pour former un document pseudo-code, il faut enchâsser des éléments pseudo-code les uns à la suite des autres et les uns dans les autres. La mécanique la plus *user friendly* pour y parvenir serait un système de glisser-déposer (*drag & drop)* avec des *points d'insertion* (ou *drop zones*).

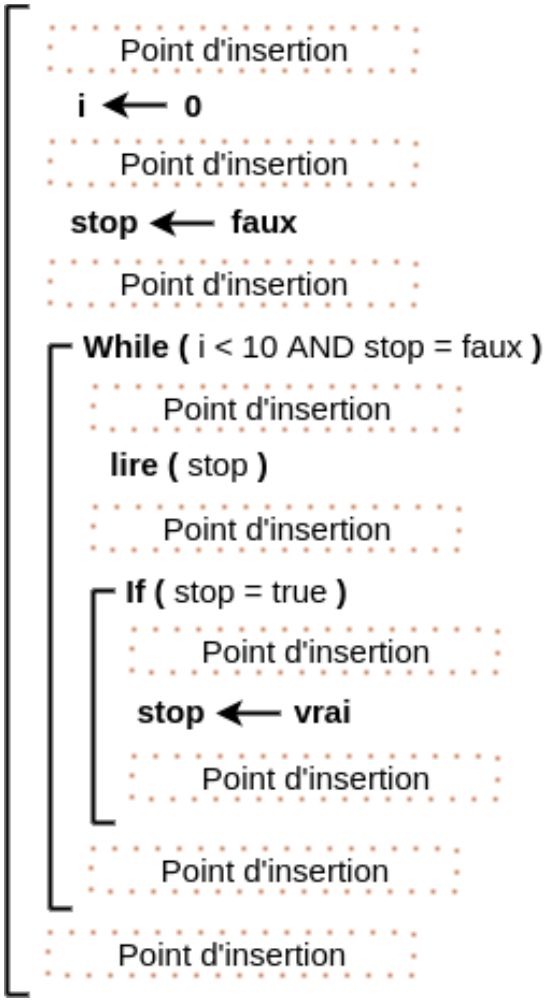


Illustration 24: Un exemple de pseudo-code où les points d'insertion de chaque élément sont visibles

Il faut que n'importe quel élément puisse s'insérer avant ou après un autre élément. Mais il faut, également, que n'importe quel élément puisse s'insérer dans un élément *inclusif*, même lorsque celui-ci est vide.

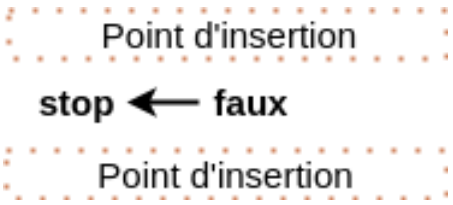


Illustration 25: Exemple de représentation d'un élément plat avec ses points d'insertion visibles

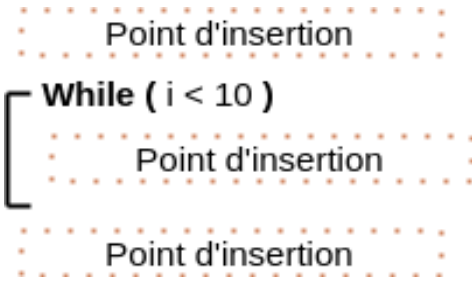


Illustration 26: Exemple de représentation d'un élément inclusif avec ses point d'insertion visibles

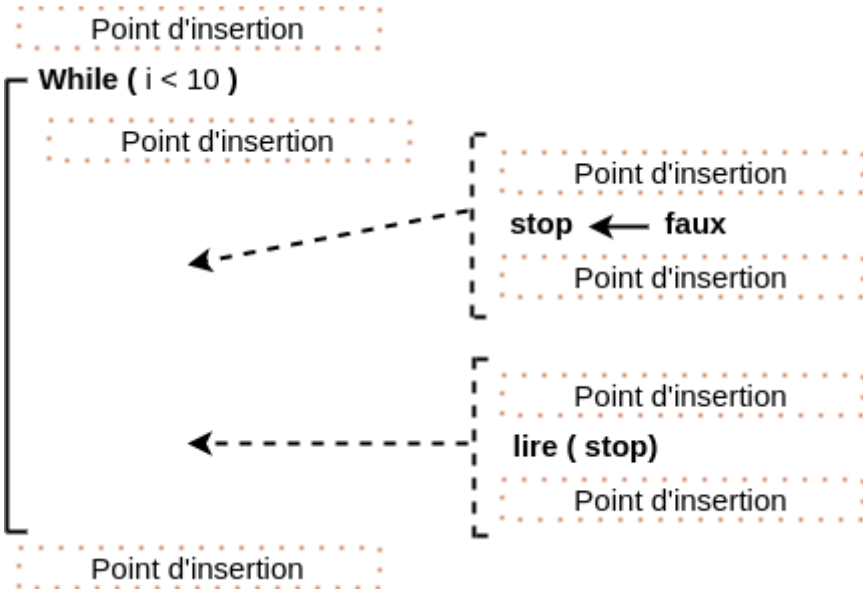


Illustration 27: Schéma explicatif du problème de redondance des points d'insertion lors de l'insertion d'éléments

Cependant, on observe rapidement un problème de répétition. Plusieurs points d'insertion se touchent et remplissent, alors, exactement le même rôle :

- le premier point d'insertion à l'intérieur du <While> et le point d'insertion avant le « *Stop ← faux* » (point d'insertion *pré-élément*) ;
- le point d'insertion après le « *Stop ←faux* » (point d'insertion *post-élément*) et le point d'insertion avant le « *lire(stop)* » (point d'insertion *pré-élément*) ;
- le point d'insertion avant le <While> (point d'insertion *pré-élément*) qui rentrera en conflit avec l'élément qui sera au dessus de lui ou qui le contiendra.

Ce problème peut être résolu en ne gardant que les points d'insertion post-élément, ainsi que le premier point d'insertion interne de tout élément inclusif.

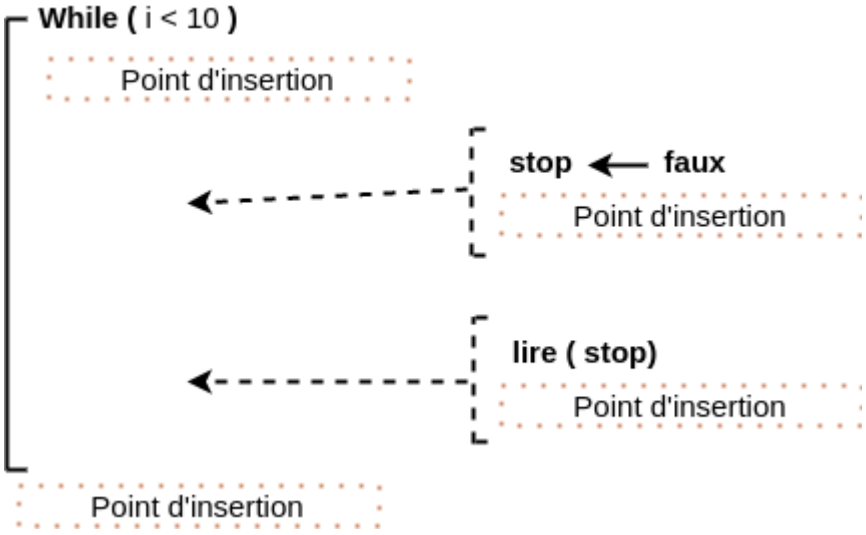


Illustration 28: Schéma explicatif de la résolution du problème de redondance des points d'insertion lors de l'insertion d'éléments

Il est intéressant de noter qu'insérer un élément A après un élément B revient à insérer l'élément A dans le même parent que l'élément B à le position après l'élément B. Cela nous permettra de généraliser le processus d'insertion des éléments.

Un autre point important concernant les interactions entre éléments pseudo-code, est le filtrage qui a lieu au cours de l'insertion d'un élément pseudo-code. En effet, n'importe quel élément pseudo-code ne peut pas être inséré au sein de n'importe quel autre élément pseudo-code.

Par exemple, il n'y a pas de sens à insérer une lecture au sein de la déclaration d'un record.

Illustration 29: Schéma illustrant le besoin d'un filtrage lors des insertions d'éléments

Chaque élément pseudo-code possède ses propres règles en matière de filtrage. Voici une liste exposant les filtres de chaque élément pseudo-code.

Affectation :

Illustration 30: Réédition de l'illustration 3

L'affectation n'admet aucun enfant.

Lecture :

Illustration 31: Réédition de l'illustration 4

La lecture n'admet aucun enfant.

Écriture :

Illustration 32: Réédition de l'illustration 5

L'écriture n'admet aucun enfant.

Condition <If> :

Illustration 33: Réédition de l'illustration 6

La condition <If> accepte les enfants suivants : déclarations de variables ; affectations ; lectures ; écritures ; <If> ; <If-Else> ; <While> ; <Until> ; <Do While> ; <Do Until> ; appels de fonction ; appels de procédure.

Condition <If-Else> :

Illustration 34: Réédition de l'illustration 7

La condition <If-Else> accepte les enfants suivants : déclarations de variables ; affectations ; lectures ; écritures ; <If> ; <If-Else> ; <While> ; <Until> ; <Do While> ; <Do Until> ; appels de fonction ; appels de procédure.

Boucle <While> :

Illustration 35: Réédition de l'illustration 8

La boucle <While> accepte les enfants suivants : déclarations de variables ; affectations ; lectures ; écritures ; <If> ; <If-Else> ; <While> ; <Until> ; <Do While> ; <Do Until> ; appels de fonctions ; appels de procédures.

Boucle <Until> :

Illustration 36: Réédition de l'illustration 9

La boucle <Until> accepte les enfants suivants : déclarations de variables ; affectations ; lectures ; écritures ; <If> ; <If-Else> ; <While> ; <Until> ; <Do While> ; <Do until> ; appels de fonction ; appels de procédure.

Boucle <Do While> :

Illustration 37: Réédition de l'illustration 10

La boucle <Do While> accepte les enfants suivants : déclarations de variables ; affectations ; lectures ; écritures ; <If> ; <If-Else> ; <While> ; <Until> ; <Do While> ; <Do Until> ; appels de fonction ; appels de procédure.

Boucle <Do Until> :

Illustration 38: Réédition de l'illustration 11

La boucle <Do Until> accepte les enfants suivants : déclarations de variables ; affectations ; lectures ; écritures ; <If> ; <If-Else> ; <While> ; <Until> ; <Do While> ; <Do Until> ; appels de fonction ; appels de procédure.

Appel de fonction :

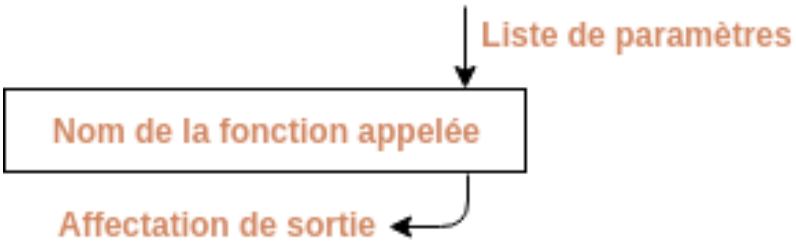


Illustration 39: Réédition de l'illustration 12

L'appel de fonction n'admet aucun enfant.

Appel de procédure :

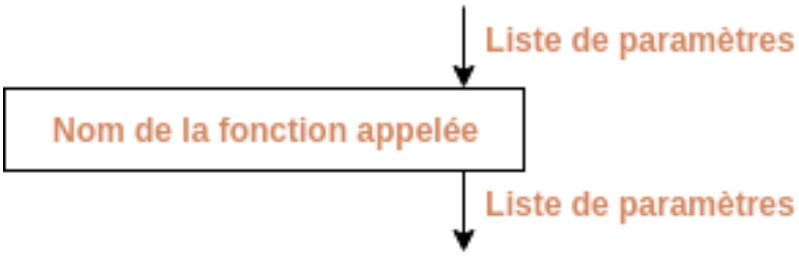


Illustration 40: Réédition de l'illustration 13

L'appel de procédure n'admet aucun enfant.

Déclaration de fonction :

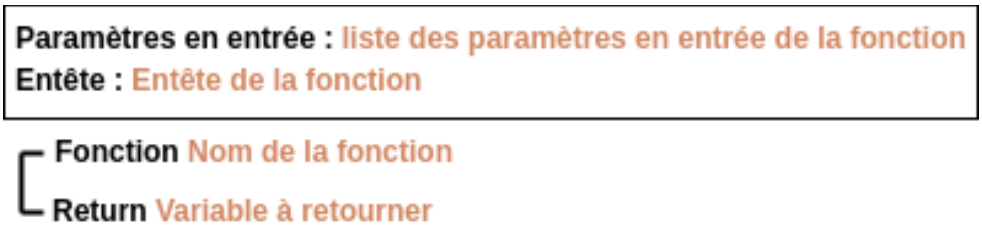


Illustration 41: Réédition de l'illustration 14

La déclaration de fonction accepte les enfants suivants : déclarations de variables ; affectations ; lectures ; écritures ; <If> ; <If-Else> ; <While> ; <Until> ; <Do While> ; <Do Until> ; appels de fonction ; appels de procédure.

Déclaration de procédure :

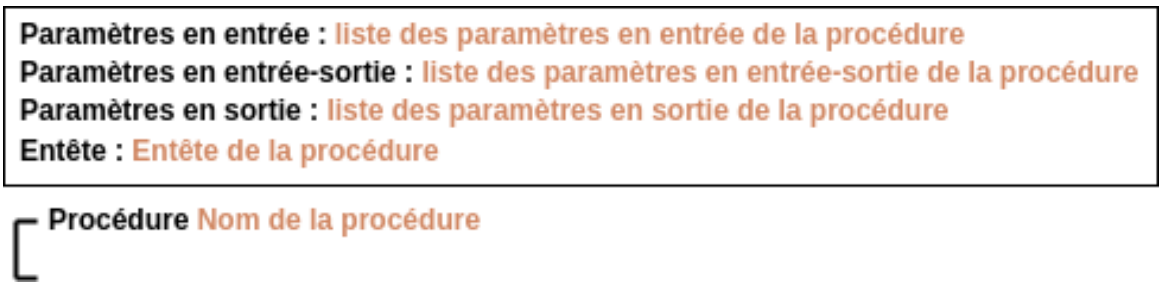


Illustration 42: Réédition de l'illustration 15

La déclaration de procédure accepte les enfants suivants : déclarations de variables ; affectations ; lectures ; écritures ; <If> ; <If-Else> ; <While> ; <Until> ; <Do While> ; <Do Until> ; appels de fonction ; appels de procédure.

Déclaration de record :

Illustration 43: Réédition de l'illustration 16

La déclaration de record accepte les enfants suivants : *déclarations de variables*.

NB : Le sous-ensemble de pseudo-code utilisé à la HERS n'inclus pas les déclarations de variables. Mais nous les utiliserons tout de même, ici, pour une raison de simplicité.

6.3. Fonctionnalités implémentées

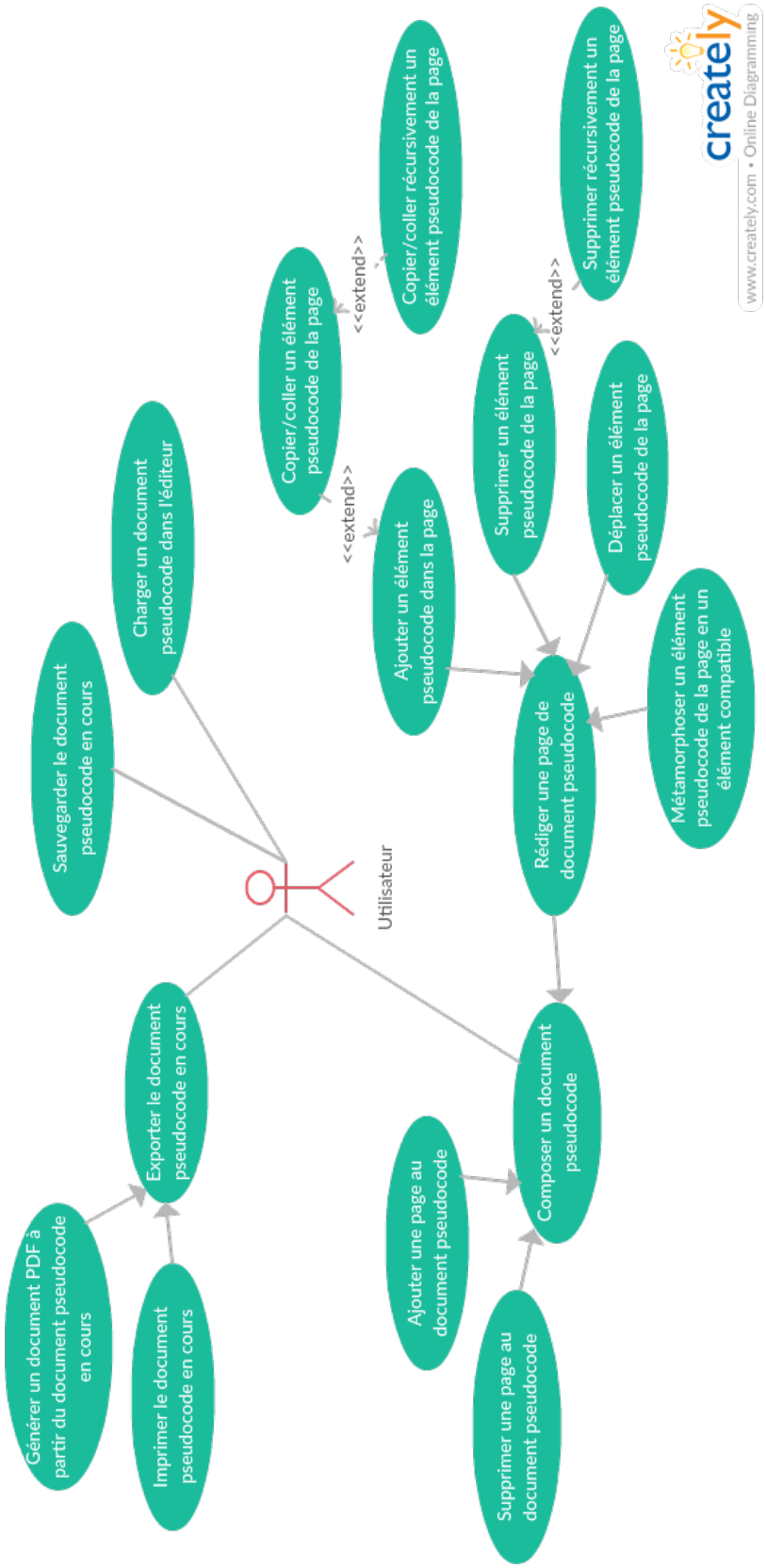


Illustration 44: Diagramme des cas d'utilisation de notre application.

7. Recherches et choix technologiques

Le développement de l'application a été très mouvementé. En effet, cette dernière ne ressemble que très peu au type d'applications habituellement abordé à la Haute École Robert Schuman. De plus, elle est orientée vers une programmation Front-End alors que ma formation tend plutôt vers le Back-End.

De très nombreuses technologies ont été testées pour parvenir au résultat escompté.

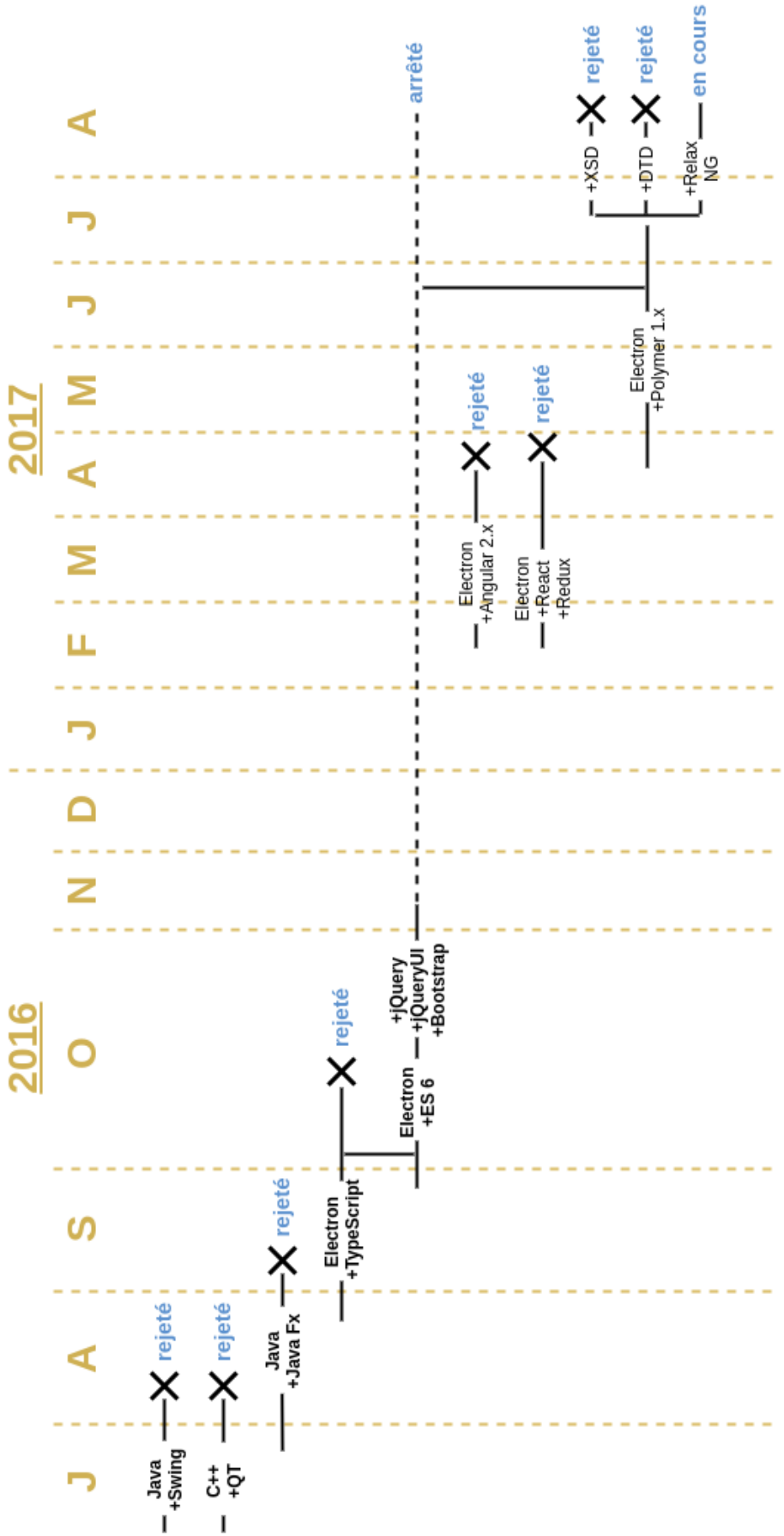


Illustration 45: Diagramme retraçant l'évolution des choix technologiques

7.1. Java + Swing

Java est un langage de programmation multiparadigme, mais principalement orienté objet, développé par Oracle. Il est utilisé pour réaliser des applications pour desktop, systèmes embarqués, mobile et web.

Les programmes Java sont multiplate-formes, car exécutés dans une machine virtuelle (Java Virtual Machine). Ils peuvent fonctionner sur une large gamme de systèmes d'exploitation, mais aussi sur différentes architectures matérielles.

Swing est une bibliothèque graphique Java qui permet de créer des interfaces graphiques identiques sur différents systèmes d'exploitation. Cependant, elle n'offre que peu de possibilités de mise en page.

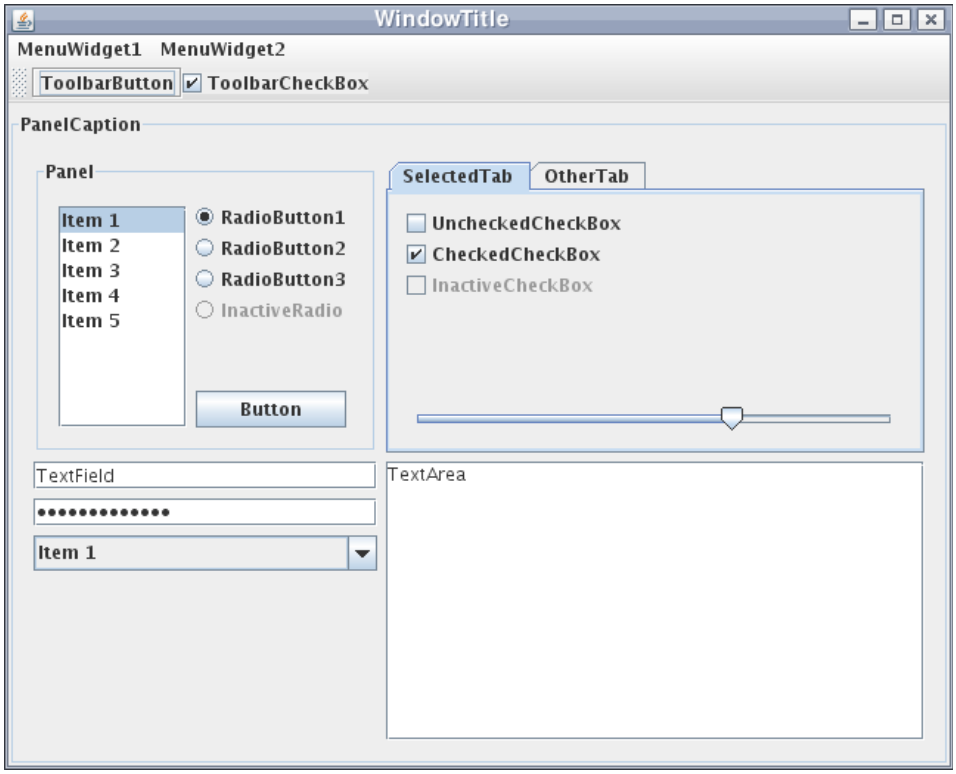


Illustration 46: Exemple d'interface graphique réalisée avec Swing. (Source : wikipedia.org, License : GPL)

À cause des limitations de mise en page de Swing, j'ai décidé de ne pas l'utiliser pour ce projet.

7.2. C++ + Qt

C++ est un langage de programmation libre, multiparadigme, mais principalement orienté objet. Il est utilisé pour réaliser des applications pour desktop, systèmes embarqués, mobile et web.

Qt est une API orientée objet et mais aussi un framework. Il fournit une large bibliothèque de composants graphiques, ainsi que des classes génériques pour la gestion des fils d'exécution

(Thread), des connexions réseau, de l'analyse XML, etc. Qt facilite aussi la structuration des applications grâce, par exemple, au mécanisme des signaux et slots. Aussi, divers plugins peuvent étendre les capacités de Qt.

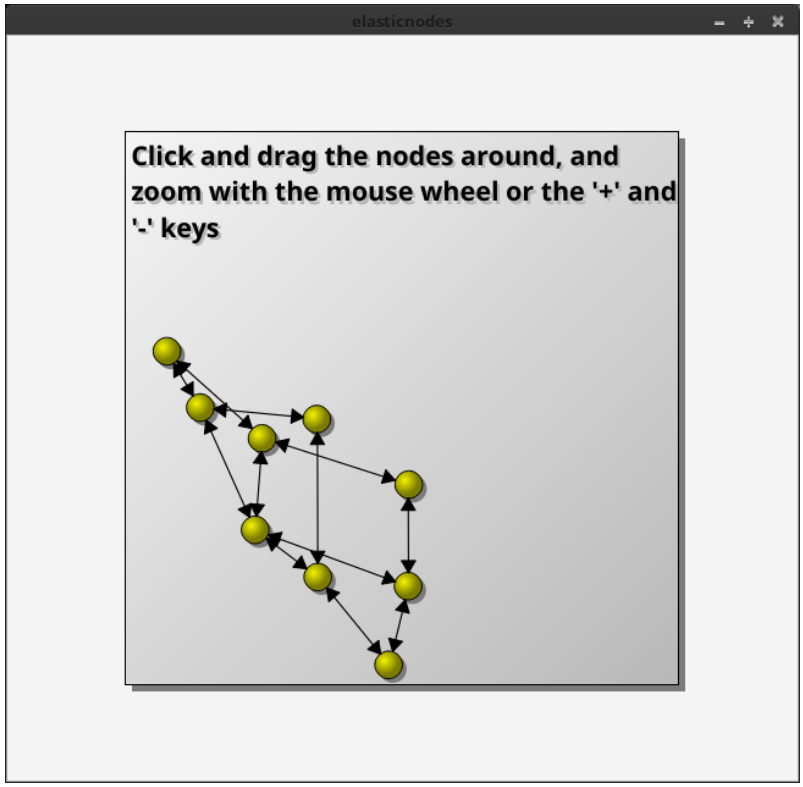


Illustration 47: Exemple d'interface graphique réalisée avec Qt. (Source : Elastic Nodes - Qt Creator, License : LGPL)

Les programmes C++ couplés à Qt sont compilables vers plusieurs plate-formes à partir d'un code source unique et sans modifications.

Je n'ai finalement pas utilisé C++ et Qt pour ce projet, parce que le C++ est un langage qui me met mal à l'aise.

7.3. Java + JavaFX

JavaFX est une bibliothèque graphique Java qui a pour objectif de remplacer Swing. Elle permet, tout comme Swing, de créer des interfaces graphiques identiques sur une multitude de plate-formes différentes. JavaFX amène un grand lot de nouveautés par rapport à son précurseur, comme, par exemple, le support intégré des appareils tactiles ou la mise en forme à l'aide de CSS.

Illustration 48: Exemple d'interface graphique réalisée avec JavaFX. (Source : Colorful Circles - OracleDoc, License : Oracle legal terms for screenshots)

En réalisant un prototype du projet sur *JavaFX*, je me suis heurté à divers problèmes, dont un particulièrement intriguant.

- Alors que j'utilisais la mise en forme *CSS*, certains éléments se déplaçaient soudainement en groupes lorsqu'ils étaient survolés par le curseur. Cependant, rien dans le code *CSS* ou dans le code Java n'impliquait un tel comportement.
- La notion de <Hover> semble ne pas exister en JavaFX. Toutefois, il existe les évènements <Mouse_Entered>/ <Mouse_Entered_Target> et <Mouse_Exited>/<Mouse_Exited_Targe> qui peuvent être utilisé pour simuler un <Hover>. Cependant, les choses deviennent particulièrement compliquées lorsque les éléments sont contenus les uns dans les autres. Il est également possible d'émettre manuellement des évènements, mais la procédure est compliquée.

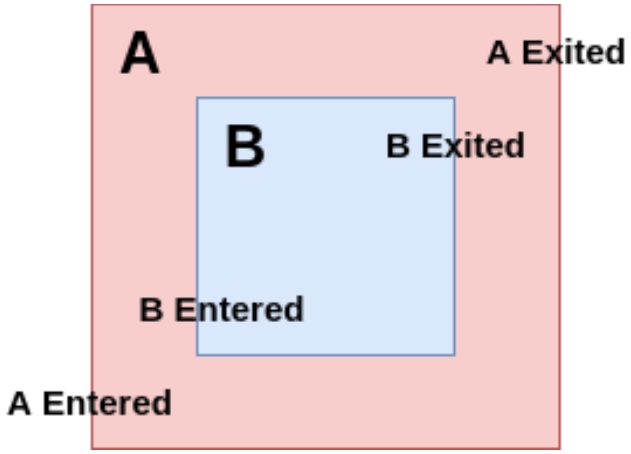


Illustration 49: Schéma explicatif du problème des évènements <Mouse_Entered> et <Mouse_Exited> lors dans le cas d'éléments imbriqués.

- La palette des fonctionnalités CSS utilisables en JavaFX est plus restreinte que celles offertes dans les navigateurs web.

  Tout bien considéré, j'ai décidé de ne pas utiliser JavaFX pour réaliser ce projet.

7.4. Electron + TypeScript

Electron est un framework open-source qui permet de développer des applications multiplate-formes en utilisant les technologies Chromium (Front-End) et Node.js (Back-End). Les applications Electron peuvent être écrites à l'aide de n'importe quelle technologie appartenant au vaste écosystème issu de la trinité HTML, CSS et JavaScript.

Bien qu'étant une technologies très récente, Electron est à la base de nombreuses applications populaires telles que Atom, Visual Studio Code, Discord, WordPress Desktop, WhatsApp Desktop ou Slack. Il faut toutefois noter qu'il est impossible de masquer le code source d'un programme écrit avec Electron.

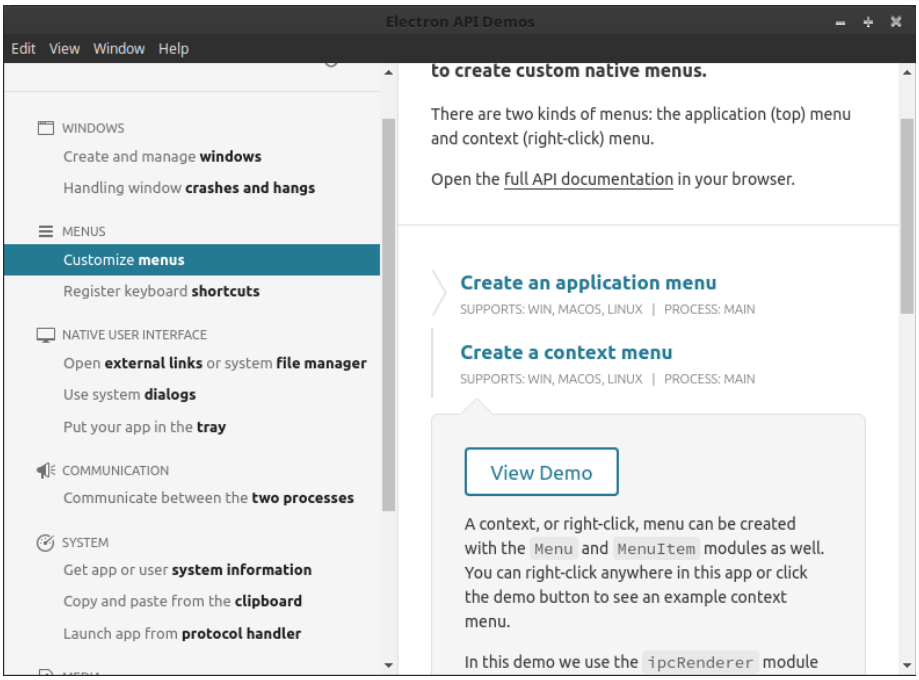


Illustration 50: Exemple d'interface graphique réalisable sous Electron. (Source : Electron API Demos - Github, License : MIT)

TypeScript est une surcouche de JavaScript développée par Microsoft. Son intérêt principal est d'apporter le typage statique des variables et des fonctions, la création de classes et d'interfaces et l'importation de modules, sous la formes auxquels les langages de programmation orientés objet traditionnels nous ont habitués. Les codes TypeScript ne sont jamais utilisés tels quels, ils doivent être transcompilé vers l'ECMAScript.

Dans sa version 1.8.10 (celle que j'ai testé), TypeScript souffrait encore de quelques lacunes qui pouvait désorienter les débutants :

- il n’était pas possible d’aligner plus de deux mots-clés. Ainsi, par exemple, il n’était pas possible d’écrire la déclaration suivante : « *public abstract class »* ;
- il n’était pas possible d’écrire des énumérations de string ;
- etc.

Aussi, il est important de noter qu’il n’est pas possible, en TypeScript, d’utiliser une bibliothèque JavaScript importée sans fournir son *fichier de déclarations* permettant de l’interfacer avec TypeScript. De nombreux fichiers de déclarations sont disponible sur le GitHub de DefinitelyTyped.

Tout compte fait, pour un débutant en technologies web, TypeScript n’apporte que peu de choses réellement utiles comparativement à la complexité et à la confusion qu’il peut amener. C’est pour ces raisons que j’ai décidé de ne pas utiliser TypeScript dans ce projet.

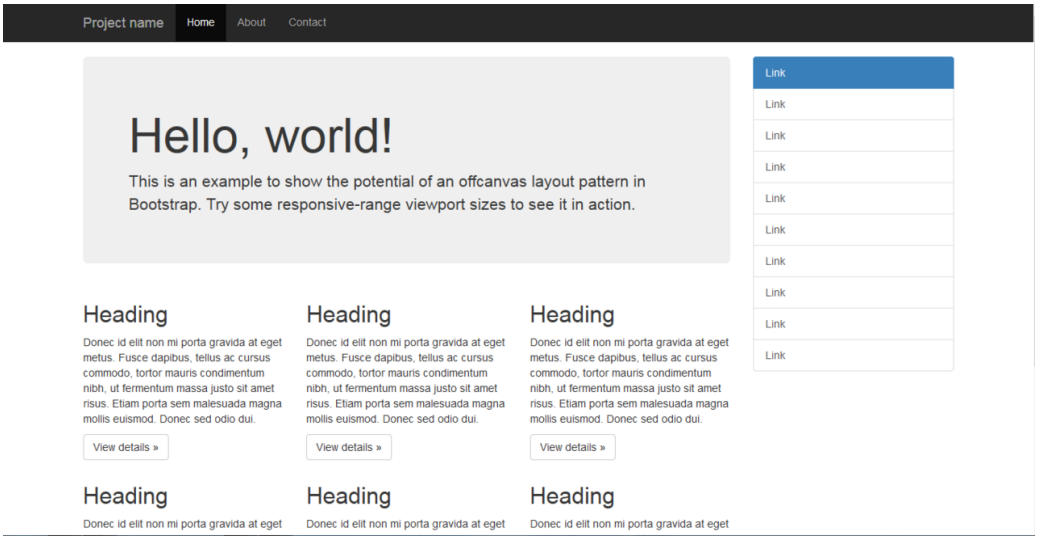
7.5. Electron + ES 6 + jQuery + jQueryUI + Bootstrap

ES6 ou ECMAScript 6 (ou ECMAScript 2015) est une standardisation du langage JavaScript. Javascript est un langage multiparadigme, mais principalement orienté prototype. Il est le langage par excellence utilisé pour donner de l’interactivité aux pages web côté client. Cependant, il est également utilisé par certaines bases de données ou pour écrire des applications exécutées du côté serveur.

jQuery est un bibliothèque Front-End open-source JavaScript. Elle a pour objectif, entre autres, de simplifier le manipulation du DOM et du CSS, de simplifier l’Ajax, d’apporter un lot de nouveaux effets visuels, etc.

jQueryUI est une collection open-source de composants graphiques, d’effets visuels et de thèmes.

Bootstrap est un framework open-source qui permet d’améliorer la mise en page du contenu des pages web. Il apporte un système de mise en page qui s’adapte au support sur lequel la page est visionnée. Il fournit également une grande collection de composants graphiques, d’éléments typographiques, etc.



Hello, world!

This is an example to show the potential of an offcanvas layout pattern in Bootstrap. Try some responsive-range viewport sizes to see it in action.

Heading

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

Heading

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

Heading

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

Heading

Donec id elit non mi porta gravida at eget

Heading

Donec id elit non mi porta gravida at eget

Heading

Donec id elit non mi porta gravida at eget

Illustration 51: Exemple d’interface graphique réalisable sous Jquery et Bootstrap. (Source : Wikipedia.org, License : Creative Common 3.0)

J’ai utilisé ces technologies pour réaliser la première implémentation de ce projet.

7.6. Electron + Angular 2.x

Angular 2.x est un framework Front-End JavaScript pour créer des applications web développé par Google. Il permet de créer des applications web monopage en utilisant les architectures MVC et MVVM. Il a pour objectif :

- de séparer le fonctionnement de l’interface graphique et la logique applicative ;
- de découpler le côté client de l’application et le côté serveur de l’application pour faciliter un développement en parallèle des deux ;
- d’amener une structure de base pour construire les applications.

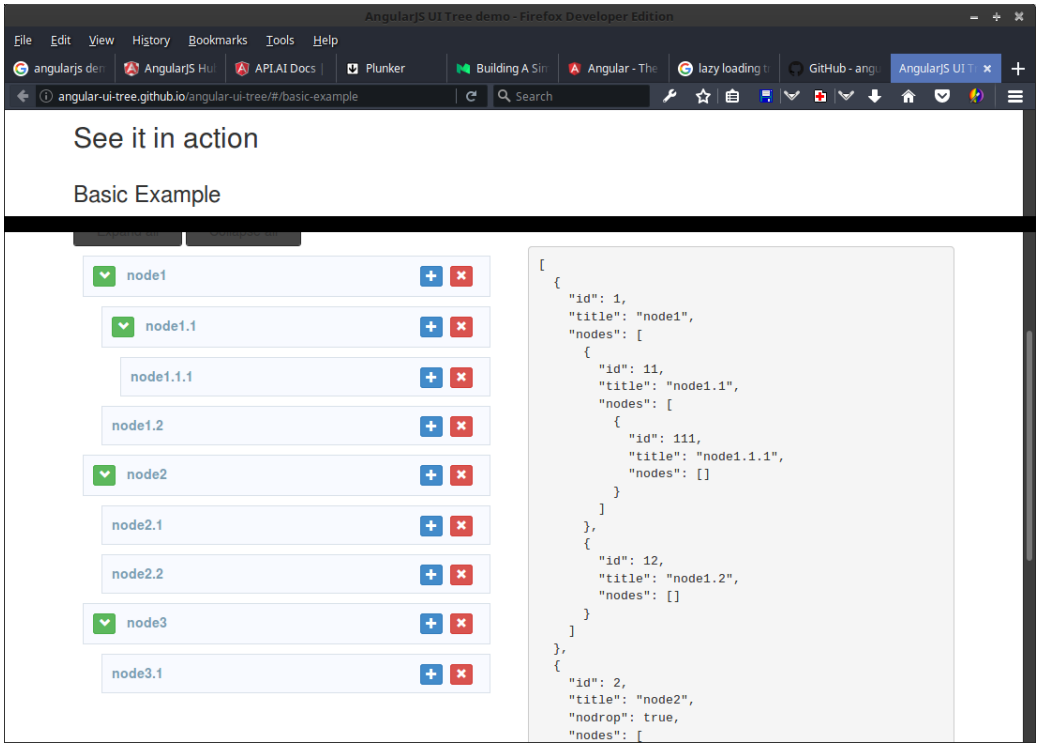


Illustration 52: Exemple d'interface graphique réalisable avec Angular. (Source : Github.com/angular-ui-tree/angular-ui-tree, License : MIT)

Après avoir appris AngularJs (Angualr 1.x) et Angular 2.x, j'ai décidé de ne pas les utiliser car ils ne sont pas conçus pour réaliser le type d'application que je voudrais faire :

- Angular isole le modèle métier et s'occupe d'automatiser l'affichage des données selon des règles prédéfinies. il utilise des vues dites *déclaratives*. Dans mon cas, cette automatisation manque de flexibilité. En effet, Angular propose un nombre limité de composants graphiques qui exigent des sources de données sous un format prédéfini.
- Dans mon application la logique métier se mélange avec l'interface graphique, ce qui est contraire aux objectifs d'Angular. En certains endroits de mon application, il n'est pas possible de séparer les deux. Exemple : le contenu admissible dans un élément pseudo-code (voir point 6.2.2) est, en un sens, à la fois une donnée métier et à la fois un élément important du fonctionnement de l'interface graphique.

7.7. Electron + React.js + Redux

React.js est une bibliothèque open-source JavaScript créée par Facebook. Elle a pour but de faciliter la création d'applications web monopage. Contrairement à Angular, React ne s'occupe que d'assurer le lien et la cohérence entre la vue et les données. Pour ce faire, il utilise des vues *déclaratives* et travaille avec une technologie appelée *DOM virtuel* qui lui confère d'excellente performances.

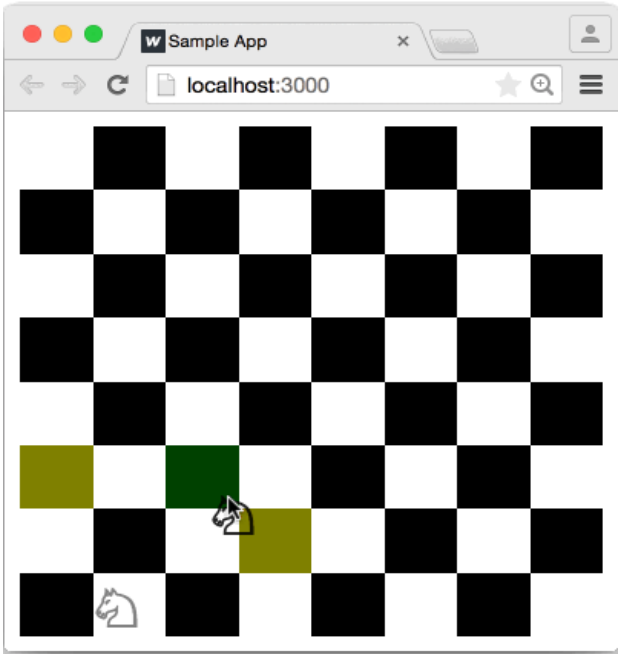


Illustration 53: Exemple d'interface graphique réalisable avec React. (Source : Github.com/react-dnd/react-dnd, License : MIT)

Redux est une bibliothèque open-source JavaScript. Son objectif est de traiter le modèle d'une application comme une suite d'états. Ainsi, Redux permet, entre autres, de rapidement et facilement revenir à un état précédant des données de l'application. Redux est très léger et son fonctionnement rappel celui de Git : il ne mémorise que les modifications, qu'il stocke dans un arbre.

J'ai voulu utiliser React.js et Redux comme alternative à Angular et pour pouvoir facilement implémenter une fonctionnalité « Retour en arrière ». Après avoir appris React.js, j'ai essayé de faire fonctionner un *boilerplate*[1] React/Electron, mais je n'y suis pas parvenu. J'ai donc décidé de ne pas utiliser React et Redux.

7.8. Electron + Polymer 1.x + Relax NG

Polymer 1.x est une bibliothèque open-source JavaScript développée par Google. Elle a pour objectif de faciliter la création d'applications web à l'aide de composants web. Pour ce faire, elle apporte un degré d'abstraction facilitant l'usage de la technologie des composants web.

Polymer est une bibliothèque centrée sur le DOM, à l'opposé d'Angular et de React.js qui utilisent des vues déclaratives.

[1] Boilerplate : Une base de code et de configurations réutilisable. Les templates de projets peuvent être considérés comme des boilerplates.

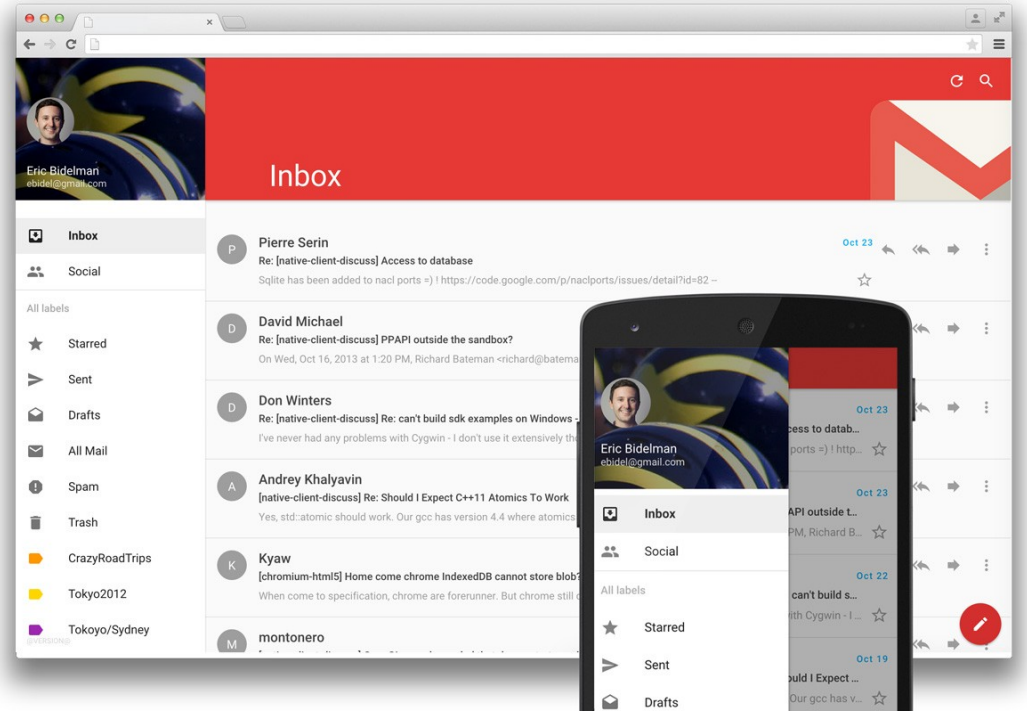


Illustration 54: Exemple d'interface graphique réalisable avec Polymer. (Source : Github.com/ebidel/polymer-gmail, License : Apache 2.0)

La technologie des composants web est une spécification du W3C qui permet la création de composants encapsulés réutilisables. Quatre particularités sont à noter les concernant.

- Pour chaque nouveau composant web créé, une nouvelle balise HTML est créée pour pouvoir l'invoquer.
- Le contenu HTML d'un composant web est défini à l'aide d'un *Template HTML*.
- Le DOM et le CSS d'un composant web sont masqués et protégés derrière sa balise HTML et ne sont altérables que via une interface exposée par le composant .
- L'importation HTML sert à importer les composants web au sein de la page HTML et permet aussi d'importer et d'employer des composants web dans d'autres composants web.

Relax NG (ou Regular Language for XML Next Generation) est un langage de description de documents XML. Il sert à énumérer un ensemble de contraintes qu'un document XML doit respecter pour être considéré comme valide. La validation XML peut être très utile pour vérifier un document XML avant de l'importer. Relax NG permet d'apporter des solutions là où XSD et DTD sont débordés.

J'ai décidé d'utiliser Polymer car dans la version du projet écrite avec ECMAScript 6 et jQuery, j'ai rencontré plusieurs difficultés :

- je n'arrivais pas à séparer, de manière propre, le code orienté vue et le code orienté métier ;
- la complexité de mon code a soudainement grimpé, rendant la maintenabilité du code difficile.

Aussi, j'ai préféré Polymer 1.x à Polymer 2.x pour sa maturité et parce qu'il n'utilise pas TypeScript.

8. Développement

Comme expliqué aux points 7.5 et 7.8, deux combinaisons de technologies ont été retenues pour réaliser ce projet. La combinaison Electron – jQuery – jQueryUI – Bootstrap a été la première développée. Mais à cause des problèmes énoncés au point 7.8, la réalisation de cette version a été mise en pause à un stade avancé de son développement. Ensuite, le développement de la combinaison Electron – Polymer 1.x – Relax NG a débuté avec comme objectif de remplacer la première version à l'aide de technologies plus adaptées et tout en évitant de reproduire les erreurs réalisées dans la précédente version.

Bien que j'aie déjà expliqué les raisons qui m'ont poussé à recommencer le développement avec de nouvelles technologies, j'aimerai expliquer en détails :

- comment ces deux versions ont été implémentées ;
- en quoi ces dernières diffèrent l'une de l'autre ;
- quelles ont étés mes erreurs ;
- qu'ai-je appris au cours du développement.

8.1. L'implémentation Electron – jQuery – jQueryUI – Bootstrap

La première implémentation a été réalisée dans un optique profondément orientée objet. Les visés de la première implémentation sont bien plus grandes que celle de la deuxième implémentation. En plus de pouvoir composer des documents pseudo-code, les sauvegarder, les charger et les imprimer, la deuxième implémentation visait également à pouvoir vérifier en temps réel le pseudo-code saisi, à traduire les documents pseudo-code dans un langage de programmation et à les exécuter. Ainsi, on notera que cette version est composée d'un très grand nombre de classes très spécifiques en comparaison de la deuxième version.

8.1.1. Le placement des points d'insertions

Nous allons reprendre ici l'analyse commencée aux points 6.2.1 et 6.2.2. Comme énoncé précédemment, un document pseudo-code est assimilable à un arbre n-aire, voire à un composite pattern. Cependant, il est un élément qui n'adhère pas à ce modèle : l'élément pseudo-code <If-Else>. Dans cette implémentation, j'ai utilisé la première solution proposée au point 6.2.1 : créer un élément inclusif <Else> et le coller à un élément <If> à l'aide d'un élément encapsulateur <If-Else>.

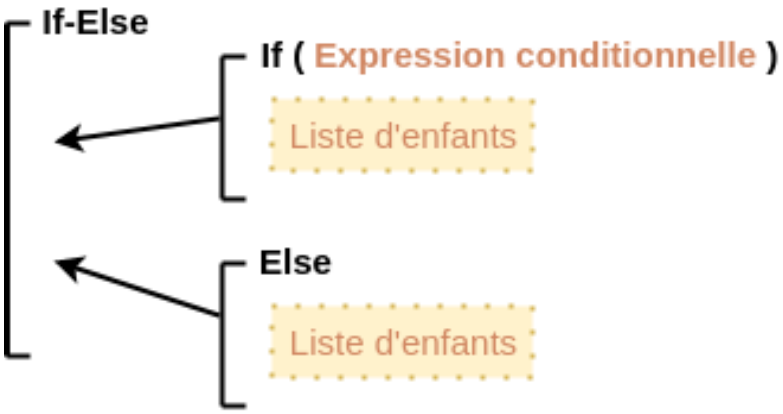


Illustration 55: Réédition de l'illustration 22.

Si nous continuons notre analyse et plaçons les points d'insertion des éléments (comme expliqué au point 6.2.2), nous pouvons immédiatement remarquer un problème.

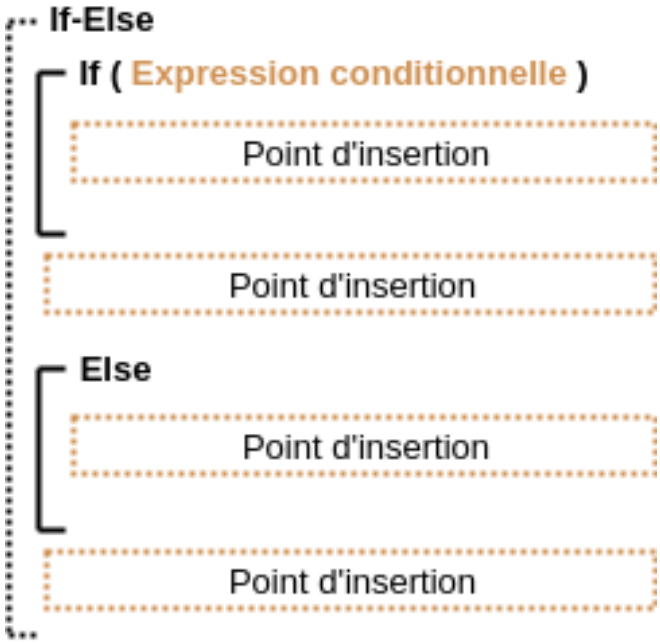


Illustration 56: Schéma explicatif du problème de brisure du <If-Else> par un point d'insertion.

Des éléments peuvent être insérés à des positions où rien ne devrait être mis, comme par exemple entre l'élément <If> et l'élément <Else>. Le placement des points d'insertion *post-élément* rend possible une *brisure* du <If-Else>. Afin de pallier à ce problème, il nous faut imaginer une nouvelle manière d'incorporer les points d'insertion *post-élément*.

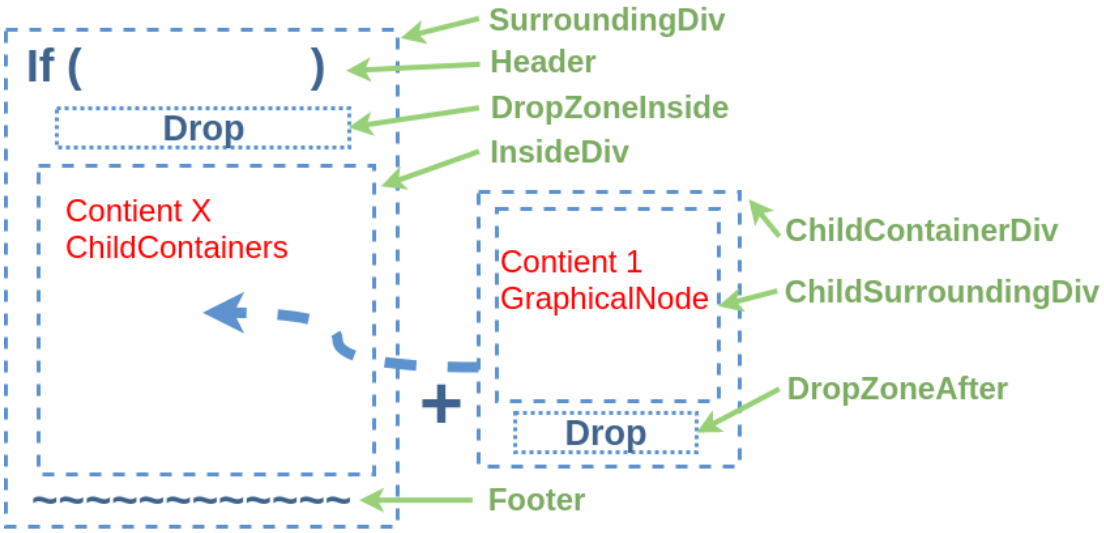


Illustration 57: Schéma explicatif de la structure d'un élément et de l'enchâssement de ses enfants en son sein, à l'aide de conteneurs.

La solution que j'ai imaginé consiste à déléguer la création des points d'insertion *post-élément* aux éléments inclusifs parents. Ainsi, quand un élément inclusif se voit ajouter un enfant, il crée une boîte (un *ChildContainer*), puis y enferme l'enfant et finalement l'ajoute à sa liste d'enfants (dans son *insideDiv*).

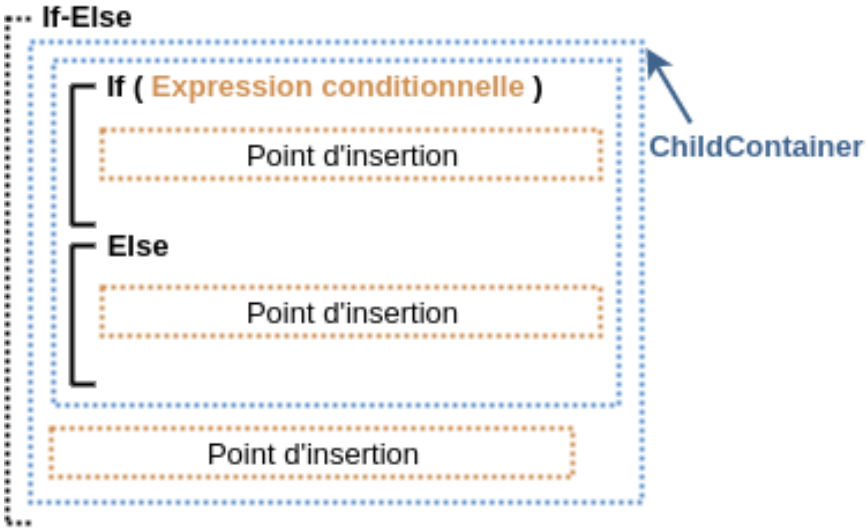


Illustration 58: Représentation du système de conteneurs appliqué au cas du <If-Else>.

8.1.2. L'architecture

Il est important de noter que la norme de schématisation UML est, à l'origine, destinée à schématiser des systèmes orientés objet et qu'elle n'est donc pas particulièrement adaptée pour schématiser des système orientés prototype (JavaScript) ou des systèmes utilisant le DOM.

Ainsi, on se retrouve, dans le cas d'un système orienté prototype, à pouvoir écrire dans le code de l'application des opérations qui ne sont pas modélisables en UML et dans le cas d'un système utilisant le DOM, à ne pas pouvoir représenter toute la *logique graphique* car une grande

partie de celle-ci est effectuée de manière opaque par le DOM. Le diagramme présenté ci-après pourrait différer quelque peu du code JavaScript réellement écrit dans notre implémentation.

N’ayant pas d’expérience dans la création de collections de composants graphiques, je me suis inspiré, tant bien que mal, de la bibliothèque Swing de Java.

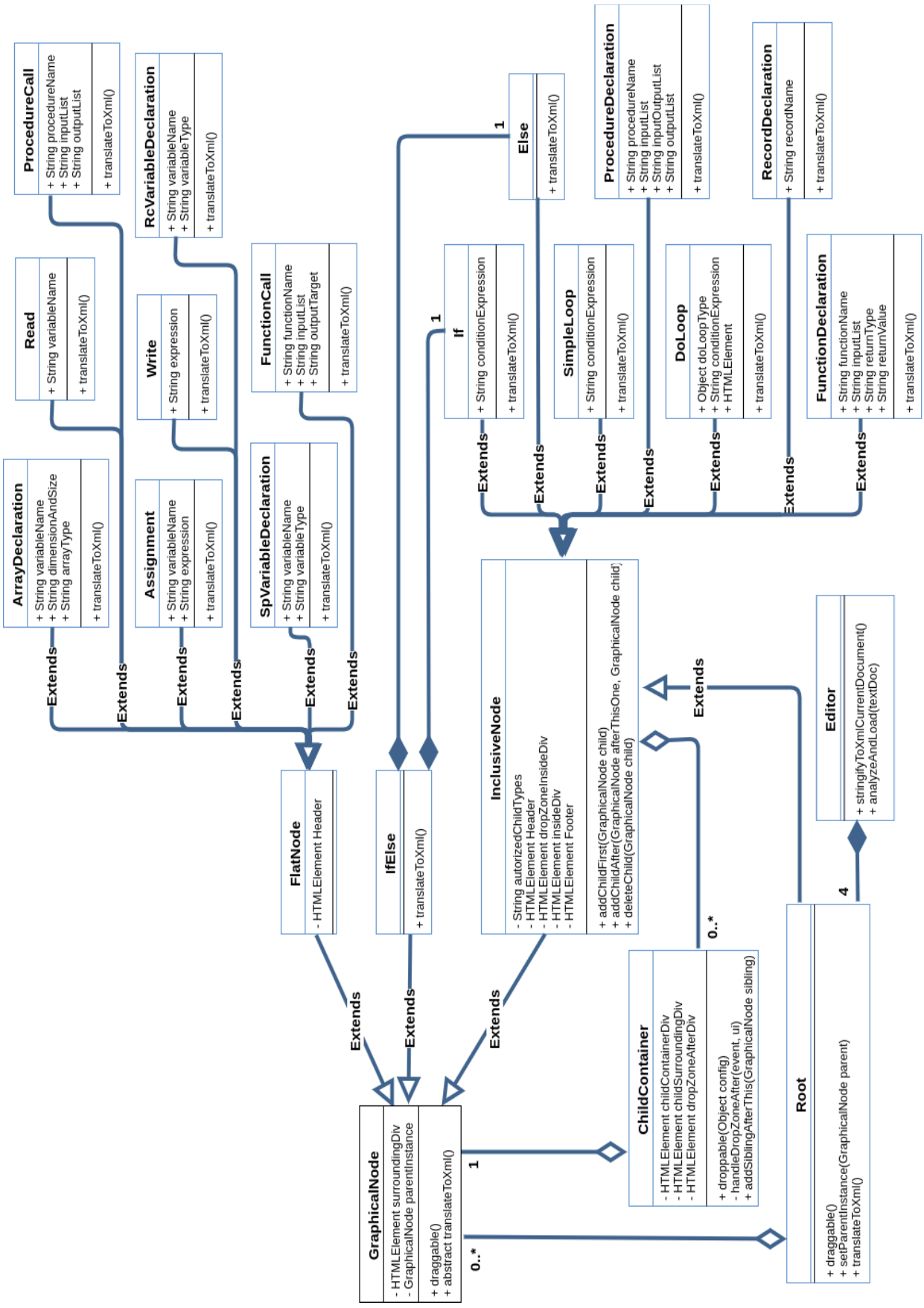


Illustration 59: Diagramme des classes de l'implémentation Electron-jQuery-jQueryUI-Bootstrap

Dans ce diagramme, on observe plusieurs éléments intéressants qui méritent d'être abordés.

- La classe nommée *Editor* :

La classe Editor s'occupe de contenir tout l'éditeur de pseudo-code et de proposer une façade au reste de l'application pour pouvoir l'utiliser. De plus, c'est au sein de cette classe que sont commandées les fonctionnalités de sauvegarde et de chargement des données.

La classe Editor héberge quatre pages de pseudo-code qui ont chacune leur propre racine : *main page* ; *records page* ; *functions page* ; *procedures page*. Chacune de ces pages a son propre rôle dans le document pseudo-code : *main page* est la page d'entrée du code, *records page* contient les déclarations des records, *functions page* contient les déclarations des fonctions et *procedures page* contient les déclarations des procédures.

- Le *composite pattern* énoncé précédemment dans l'analyse :

La structure en *composite pattern* du pseudo-code est ici représentée par le trio des classes abstraites *GraphicalNode*, *FlatNode* et *InclusiveNode*. La classe abstraite *FlatNode* représente la classe que les *feuilles* du composite pattern doivent étendre et la classe abstraite *InclusiveNode* représente la classe que les *branches* du composite pattern doivent étendre.

À ce trio s'ajoute la classe *ChildContainer* issue de la résolution du problème des placements des points d'insertion. Cette classe gère les actions effectuées sur les points d'insertion et invoque les méthodes appropriées en conséquence des actions qu'elle subie.

- La classe *Root* :

La classe *Root* hérite de la classe abstraite *InclusiveNode*, mais elle diffère des autres types de *branches* de l'arbre pseudo-code puisque celle-ci ne doit pas être déplaçable, ni pouvoir se voir attribuer un parent. Ainsi, la classe Root override les méthodes *draggable()* et *setParentInstance(GaphicalNode parent)* pour les neutraliser. Si cela nous permet d'économiser du code, cela induit aussi un profond problème de logique dans l'héritage de notre application. Nous en reparlerons dans la deuxième implémentation.

- La problématique classe *If-Else* :

Nous avons créé une classe *If-Else* qui rassemble les classes *If* et *Else* et ce dans l'optique de réutiliser un maximum de code. Cependant cette classe *If-Else* ne s'inscrit pas, contrairement à ses enfants, dans les carcans de la classe abstraite *InclusiveNode*, mais toutefois hérite de la classe abstraite *GraphicalNode* puisqu'il partage avec elle sa structure et sa capacité d'être déplacée par mécanisme de drag & drop.

Si je qualifie ici, encore une fois, le *If-Else* de problématique, c'est parce qu'il nous empêche de placer l'attribut *HTMLElement Header* dans la classe abstraite *GraphicalNode*. En effet, cet attribut est nécessaire dans les classes abstraites *FlatNode* et *InclusiveNode*, mais pas dans la classe *If-Else*. Le soucis est que si nous nous servons de l'héritage pour régler ce genre de problème, alors nous risquerons de faire perdre son sens à la hiérarchie de nos classes.

8.1.3. La sauvegarde des données

Pour enregistrer les documents pseudo-code générés à partir de ce programme, j'ai décidé d'utiliser le format XML. En effet, le XML sait parfaitement représenter des données structurées sous la forme d'un arbre n-aire.

L'objectif, ici, est donc de traduire une structure en mémoire de type arbre n-aire en une représentation XML textuelle. Pour ce faire, j'utilise la bibliothèque *xmlbuilder* et je parcours mes données en mémoire. Je vais stocker mes données de la même manière que l'on pourrait observer dans un document HTML : j'adopte le parcours en profondeur de type préfixé.

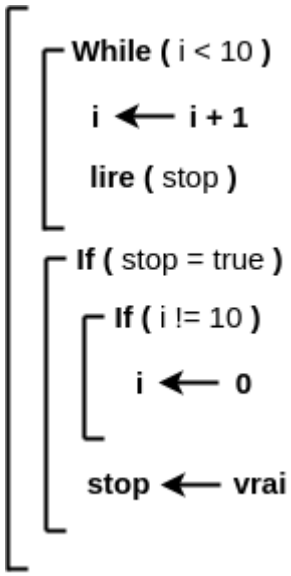


Illustration 60: Un exemple de pseudo-code pour illustrer la fonctionnalité d'enregistrement

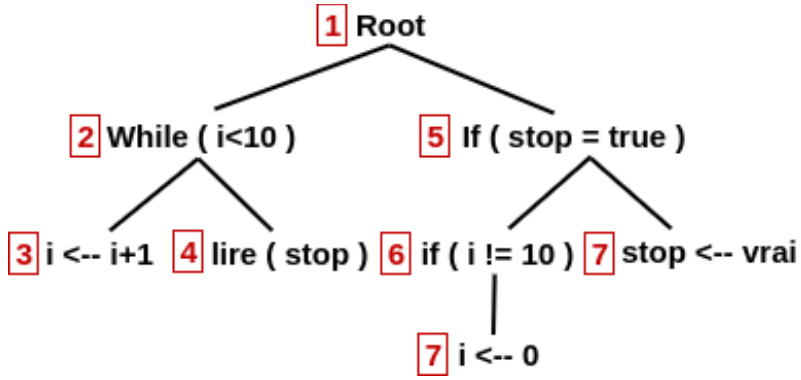


Illustration 61: L'arbre découlant du pseudo-code illustratif

```
<root>
   <while conditionexpression="i<10" >
      <assignment variablename="i" expression="i+1">
      <read variablename="stop">
   <if conditionexpression="stop=true" >
      <if conditionexpression="i!=10" >
         <assignment variablename="i" expression="0">
      <assignment variablename="stop" expression="vrai">
</root>
```

Illustration 62: Le code XML découlant du pseudo-code illustratif via le parcours de l'arbre.

Voici ce à quoi pourrait ressembler le code utilisé pour parcourir l'arbre n-aire du pseudo-code et composer le document XML. Ce code est distribué dans toutes les différentes classes qui constituent des nœuds potentiels du document pseudo-code et est adapté selon les besoins de chacune.

```
TranslateToXML() {
   Var xmlNode = CreateXMLNode('this.name')

   XmlNode.addAttributToNode('att1Name', this.att1)
   XmlNode.addAttributToNode('att2Name', this.att2)
   …

   ForEach(child in childList) {
      Var childXML = child.translateToXML()
      XmlNode.importDocument(childXML)
   }
}
```

Text 1: Un algorithme pseudo-code qui traduit en XML un arbre qu'il parcourt de façon infixée.

8.1.4. Le chargement des données

Charger les données issues d'un enregistrement XML consiste à faire exactement l'inverse de la procédure d'enregistrement. Ici, nous lisons un fichier XML, nous soumettons ses données à un parser (j'utilise la bibliothèque DOMParser) puis traduisons l'arbre XML obtenu en un arbre pseudo-code utilisant nos classes.

Alors que le parsing de l'arbre XML est délégué à une bibliothèque extérieure, la traduction de l'arbre, quant à elle, est de notre propre cru. Nous partons d'une situation où nous possédons un nœud XML non typé et nous devons retrouver son type parmi la liste des classes à notre disposition.

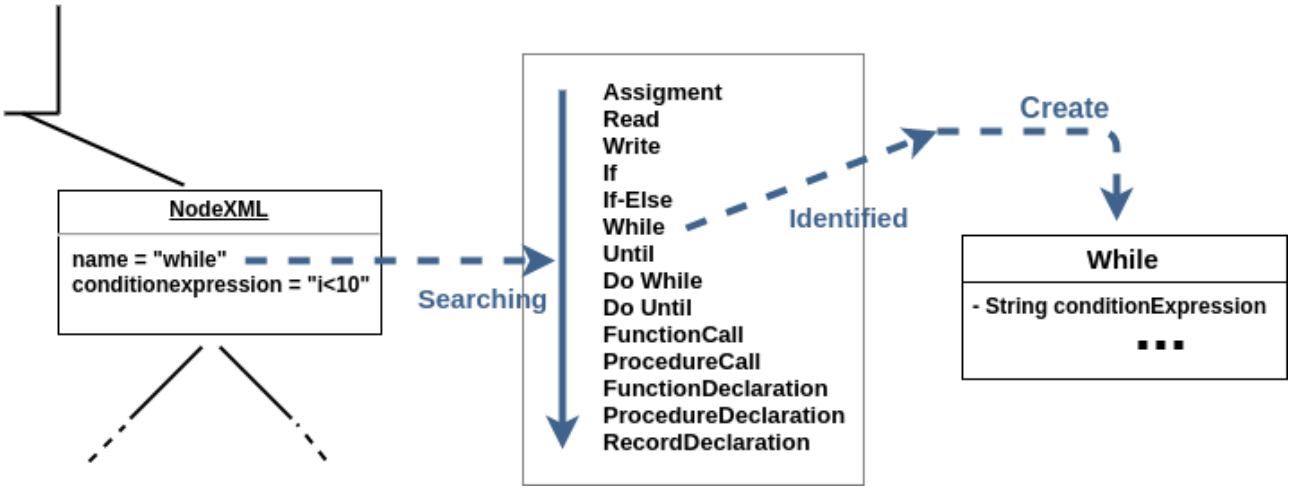


Illustration 63: Schéma illustrant le parcours d'une liste pour trouver le bon interpréteur permettant de traduire un nœud XML en classe.

Dans cette illustration, les processus d'identification et de traduction sont centralisés au sein de la même classe. Cependant, il serait plus approprié en terme de structuration et d'encapsulation de mettre directement dans chacune des classes concernées, le code pour identifier et traduire un élément XML en instance de classe.

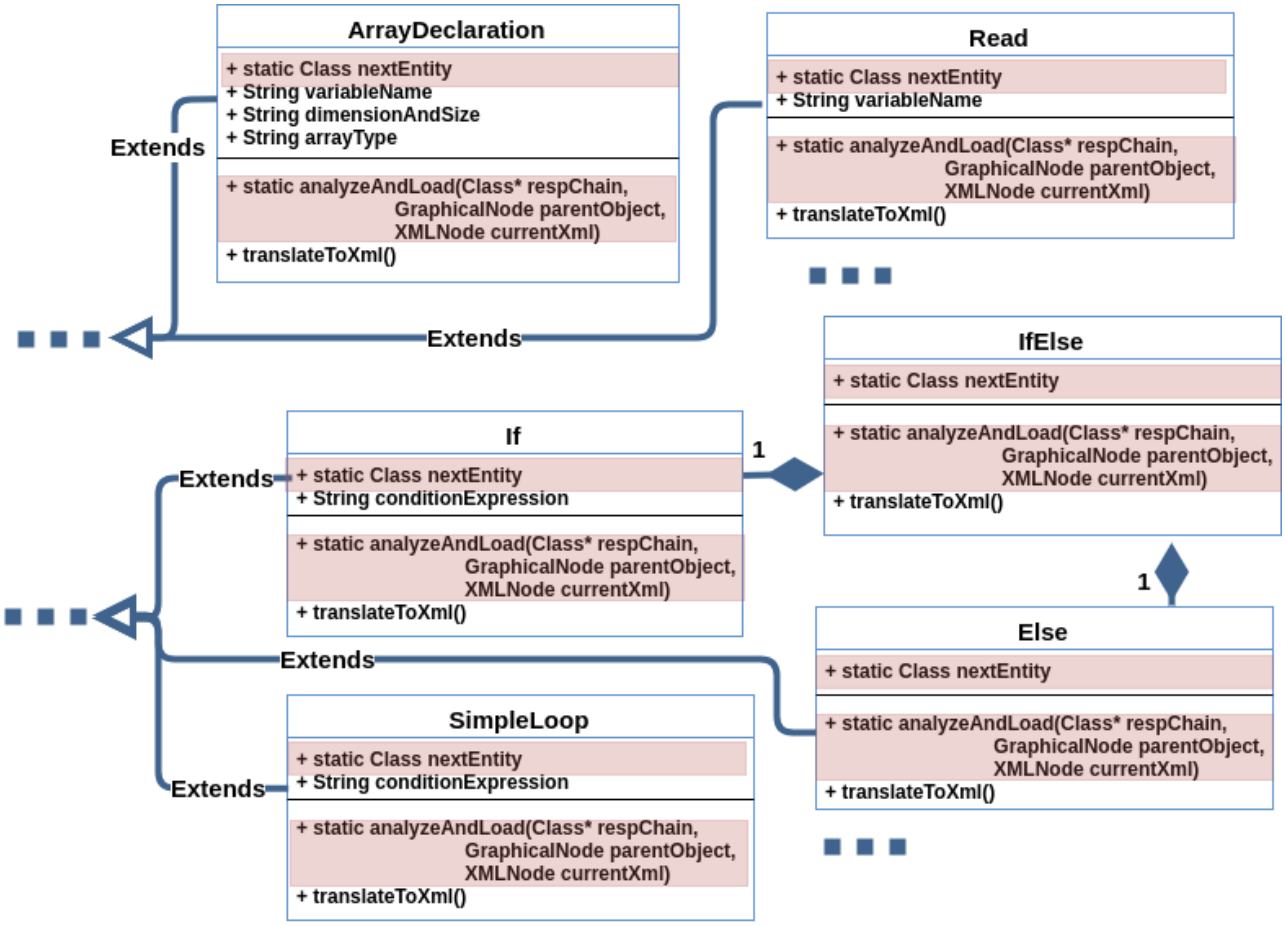


Illustration 64: Un schéma illustrant quelques classes reprises de l'illustration 59, mais qui ont été modifiées pour pouvoir traduire le XML et former une chaîne de responsabilité.

Nous observons l'ajout d'un attribut statique *nextEntity* de type *Class* et d'une méthode statique nommée *analyzeAndLoad*. Ils n'étaient pas présent dans le diagramme des classes de l'implémentation (point 8.1.2) par mesure de simplification. Cet attribut et cette méthode forment à eux deux un mécanisme proche du pattern *chaîne de responsabilité*.

Ainsi, les classes sont chaînées les unes aux autres via leur attribut statique *nextEntity* et plutôt que de centraliser la reconnaissance des éléments XML dans une seule classe, nous la distribuons sur une chaîne composée de nos classes spécialisées. L'élément XML à traduire est envoyé à cette chaîne, analysé par chacun des maillons et si l'un d'eux le reconnaît, il se charge de son traitement.

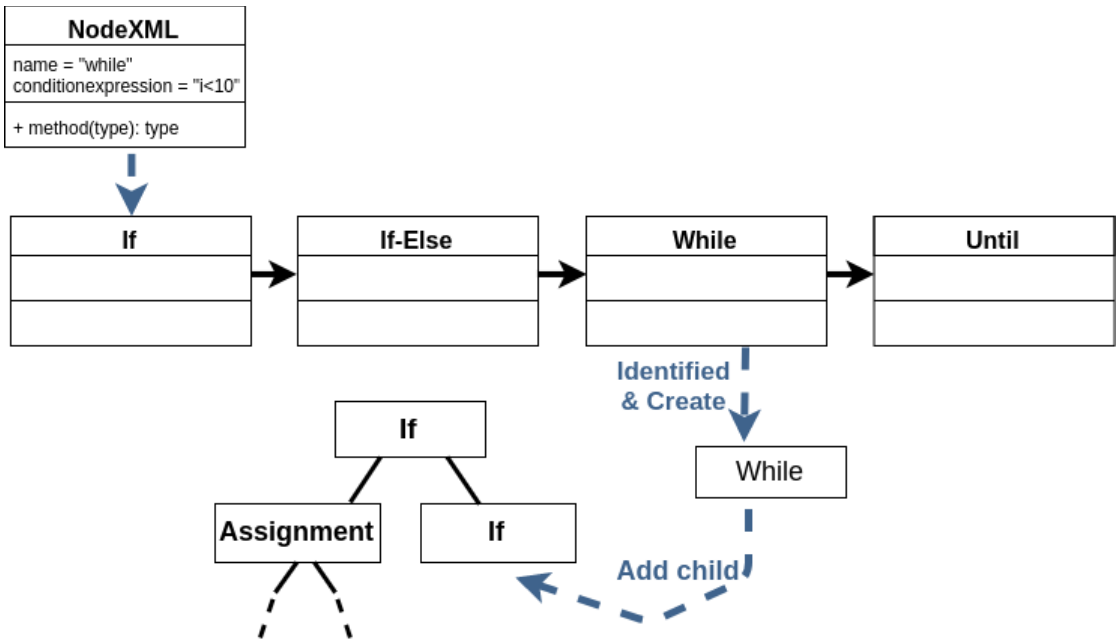


Illustration 65: Schéma illustrant le parcours d'une chaîne de classes pour trouver le bon interpréteur permettant de traduire un nœud XML en classe.

En conséquence, nous obtenons une architecture qui permet l'ajout et la suppression facile de nouvelles classes d'éléments pseudo-code.

Aussi, il est important de noter que l'opération de traduction de la source XML est une opération double. En effet, si lors de la traduction un problème survient, alors une exception est levée. Par conséquence, l'opération de chargement du fichier est annulée et l'utilisateur reçoit une alerte lui indiquant que le document XML utilisé est incorrect selon les critères de l'application. À titre d'exemple, on pourra noter les cas où :

- le nœud ne parvient pas à être identifié : toute la chaîne a été parcourue ;
- tous les attributs demandés par la classe pour effectuer la traduction n'ont pas été trouvés ;
- un élément n'appartenant pas à la liste d'acceptation d'un élément *branche* a été trouvé parmi les enfants de l'élément *branche*.

8.2. L'implémentation Electron – Polymer 1.x – Relax NG

La deuxième implémentation a été réalisée avec la technologie des *composants web* et dans une visée moins ambitieuse que la première implémentation. Ses objectifs étaient de réduire la complexité de l'application, d'insoler efficacement les manipulations du DOM du reste de la logique applicative, d'améliorer sensiblement le visuel de l'application et de s'essayer à une programmation plus flexible.

Cette implémentation prend en charge la composition de documents pseudo-code, la sauvegarde, le chargement et l'impression. Elle apporte également de nombreuses micro-fonctionnalités, des améliorations ergonomiques et le support des dispositifs tactiles.

8.2.1. Le placement des points d'insertion

Comme nous l'avons fait dans la première implémentation, nous repartons ici de l'analyse commencée aux points 6.2.1 et 6.2.2. Pour résoudre le problème du <If-Else>, je me suis permis d'utiliser, dans cette implémentation, la deuxième solution proposée au point 6.2.2. En effet, puisque la technologie Polymer 1.x m'offre un meilleur contrôle sur le HTML et le CSS et que les visées de cette version sont moindres que celles de la première implémentation, cette solution apparaît comme plus simple et largement suffisante.

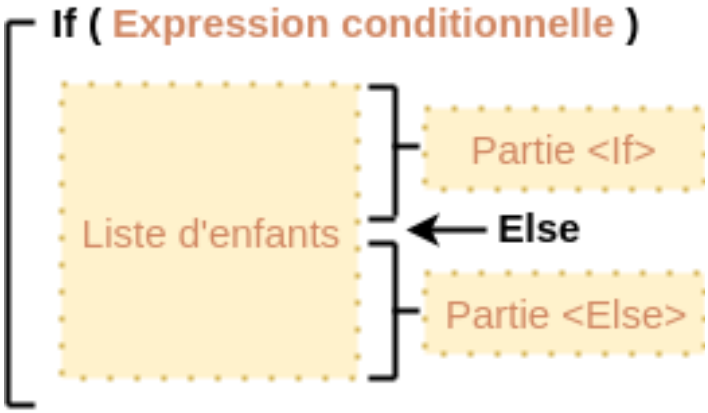


Illustration 66: Réédition de l'illustration 23.

8.2.2. La technologie Polymer 1.x

Dans cette section, je vais essayer de brièvement expliquer le fonctionnement de Polymer 1.x, car cela nous sera nécessaire pour la suite. Après cette explication, vous comprendrez ce que Polymer peut apporter à vos développements web.

8.2.2.1. Introduction

Avant de commencer, rappelons que Polymer est basé sur la technologie des composants web. En fait, Polymer propose les fonctionnalités des composants web en y ajoutant quelques abstractions et polyfills[2] pour en faciliter l'utilisation. Pour une question de facilité et malgré que cela soit technologiquement incorrect, à partir de maintenant nous n'utiliserons plus que le terme Polymer pour parler de cette technologie.

Polymer nous permet de créer nos propres balises HTML. Derrière une balise créée avec Polymer se cache énormément de choses. Ainsi, il nous est possible de créer un micro-monde HTML totalement isolé du reste du document HTML et réutilisable à souhait. Ce micro-monde HTML est composé de trois composantes principales :

- un template HTML, qui est un assemblage complètement invisibles et inaccessibles depuis l'extérieur du micro-monde de balises HTML et de styles CSS ;
- une logique JavaScript pour pouvoir animer le HTML et le CSS du micro-monde ;

[2] Polyfill : Un polyfill est un moyen de fournir le support pour une fonctionnalité encore non supportée en ne se basant que sur des capacités préexistantes.

- une interface, à implémenter soi-même, qui permet d'influencer de manière limitées le micro-monde depuis l'extérieur.

Code :

```
<dom-module id="custom-element">
  <template>
    <style>
      .frame {
        border: 1px solid black;
      }
    <style>
    <div class="frame">
      <p>Hi, I'm {{name}}</p>
      <img src="redTriangle.png" on-tap="clickTriangle"/>
    </div>
  </template>
  <script>
    Polymer({
      is: "custom-element",
      properties: {
        name: {
          type: String,
          value: "Specify your name"
        }
      },
      clickTriangle: function(e){
        console.log("Clicked");
      }
    });
  </script>
</dom-module>
```

Text 2: Un code Polymer illustrant la définition d'un composant web.

Résultat visuel :

Illustration 67: Un aperçu du résultat généré par le précédant code Polymer définissant un composant web.

Balise HTML correspondante :

```
<custom-element>
</custom-element>
```

Dans cette exemple nous pouvons apercevoir :

- la déclaration d'une nouvelle balise HTML à l'aide de la balise **<dom-module id="custom-element"></dom-module>** et de la propriété JavaScript **is:"custom-element"** ;
- le template HTML/CSS, qui se situe au sein de la paire de balises HTML **<template></template> ;**
- une partie de l'interface exposée au monde extérieur, ici représentée par le contenu de la propriété JavaScript **properties: {}**. Il s'agit des attributs que l'on doit préciser dans la balise d'invocation de notre élément. Exemple : **<custom-element name="Tomy"></custom-element>** ;

- un mécanisme de *data-binding* utilisable entre les **properties: { }** et le **<template></template>**, ici représenté sous la forme d'un binding double-sens ** Hi, I'm {{name}} ** ;
- La logique JavaScript stockée dans des méthodes personnalisées comme **clickTriangle: function(e){ }** ou dans des fonctions arbitraires, mises à notre disposition, comme **attached: function(){ }**.

8.2.2.2. Cycle de vie

Il existe plusieurs évènements importants dans la vie d'un composant web. À titre non exhaustif, nous pourrions, par exemple, parler du moment où le composant est créé ou encore le moment où le composant est ajouté ou retiré du DOM du document HTML. Tous ces moments sont regroupés au sein du cycle de vie du composant web.

Pour le développeur, la face visible de ce cycle de vie se présente sous la forme d'une série de fonctions arbitraires appelées automatiquement par le système à des moments précis (fonctions callback). Ces fonctions callback font parties des fonctions arbitraires dont nous avions déjà parlées dans la logique JavaScript. Voici la liste des plus fréquemment rencontrées d'entre elles.

- Created : elle survient quand le composant a été créé, mais n'est pas encore initialisé.
- Ready : elle survient quand le composant a été créé et initialisé.
- Attached : elle survient quand le composant a été attaché à un document.
- Detached : elle survient quand le composant a été détaché du document.

```
<dom-module id=”custom-element”>
   <template>
        <p>Hi, everybody</p>
   </template>
   <script>
      Polymer({
         is: ”custom-element”,
         created: function(e){
            console.log(”Component created”);
         },
         ready: function(e){
            console.log(”Component ready”);
         }
         attached: function(e){
            console.log(”Component attached”);
         }
         detached: function(e){
            console.log(”Component detached”);
         }
      });
   </script>
</dom-module>
```

Text 3: Un code Polymer illustrant quelques fonctions du cycle de vie d'un composant web.

8.2.2.3. Composition et héritage

Polymer 1.x propose un approche très intéressante pour étendre les composants et pour factoriser le code.

Premièrement, dans Polymer 1.x, on peut composer dynamiquement des composants en les emboîtant entre eux dynamiquement via le HTML. En effet, il possible dans le template HTML de directement utiliser un composant après l’avoir importé avec la balise HTML **<link rel=”import” href=” ”>**. Mais, il est également possible de créer des points d’ancrage (*placeholder*) qui nous permettront d’ajouter dynamiquement des éléments comme enfants de notre composant et ce à des endroits précis de notre composant. Pour ajouter un point d’ancrage, il faut utiliser la balise **<content></content>**.

Code :

```
<link rel=”import” href=”custom-element”>

<dom-module id=”custom-compo-element”>
   <template>
      <custom-element name=”Harry”></custom-element>
      <content id=”lightdom”></content>
   </template>
   <script>
      Polymer({
         is: ”custom-compo-element”,
      });
   </script>
</dom-module>
```

Text 4: Un code Polymer illustrant le fonctionnement des points d'ancrage.

Exemple d’utilisation :

```
<custom-compo-element>
   <h1>Hello World</h1>
   <custom-element name=”Eric”>
   </custom-element>
</custom-compo-element>
```

Résultat :

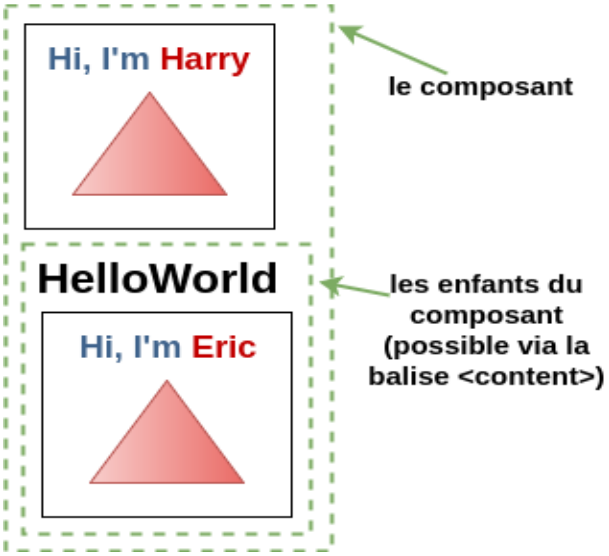


Illustration 68: Un aperçu d'un résultat possible obtenu avec le précédant code Polymer illustrant les points d'ancrage.

Les points d’ancrage (**<content></content>**) peuvent être filtrants grâce à l’attribut **select**. Cet attribut accepte un sélecteur CSS. Par exemple, le point d’ancrage **<content select=”h1, .bleu”></content>** n’acceptera que des éléments de type *<h1>* et des éléments ayant la classe *bleu*.

Je tiens à préciser un petit point de vocabulaire. La partie d’un composant directement inscrite dans son **<template>** s’appelle le *shadow DOM* car elle est invisible et inatteignable depuis l’extérieur du composant. Quand à la partie amenée par les balises **<content>**, elle s’appelle le *light DOM* car elle est visible et atteignable depuis l’extérieur du composant.

Deuxièmement, concernant la logique JavaScript, Polymer 1.x intègre et encourage l'utilisation du mécanisme d'héritage prototypal, particulièrement sous sa forme concaténative.

Javascript, qui est un langage orienté prototype, n'utilise pas de classes, il leur préfère les prototypes. En Javascript tout est objet, y compris les prototypes. Les prototypes doivent être vu comme des objets qui servent de références. On peut les cloner, on peut les modifier en temps réel et on peut aussi en hériter.

Polymer, tout comme JavaScript, effectue l'héritage à l'aide d'une chaîne de prototypes. Une chaîne de prototypes consiste en une suite de prototypes qui se référencent les uns à la suite des autres. Ainsi, les attributs et les méthodes des différents prototypes sont mis en commun et fusionnés dans la chaîne. Il existe une notion d'*overriding* dans la chaîne de prototypes puisque les attributs et méthodes définis en début de chaîne écrasent ceux de mêmes noms définis en fin de chaîne.

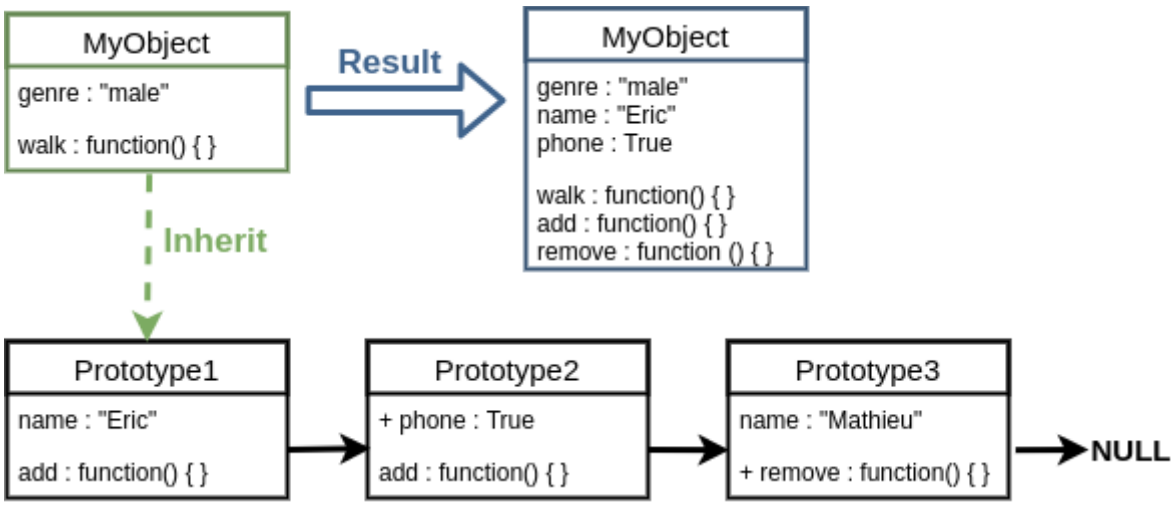


Illustration 69: Un schéma illustrant un héritage prototypal.

Nous avions dit précédemment qu'il existait des méthodes arbitraires (point 8.2.2.2), dont les méthodes du cycle de vie. Il faut savoir que Polymer traite ces méthodes différemment des autres. En effet, puisqu'elles possèdent les mêmes noms dans tous composants, elles sont promptes à être en écrasées en cas d'héritage. Ainsi Polymer préférera, en cas d'héritage, additionner leurs contenus plutôt que de les écraser par overriding.

Dans l'héritage orienté objet classique, une classe qui hérite se voit placée dans une hiérarchie rigide de classes. Elle obtient de ses parents l'ensemble de leurs attributs et de leurs méthodes. En contrepartie, cette classe sera considérée, à l'avenir, comme un sous-type de ses parents. Ainsi, il sera possible de l'utiliser là où ses parents auraient pu être utilisés (polymorphisme).

L'héritage concaténatif appartient au monde des langages faiblement typés. Quand un objet hérite concaténativement, il additionne à ses propres attributs et méthodes, ceux de sa chaîne d'héritage prototypal. En conséquence, l'objet n'est pas un sous-type des prototypes de sa chaîne de prototypes, mais le résultat de leur addition. L'héritage concaténatif est beaucoup moins rigide que l'héritage orienté objet classique et nous permet, avec une grande liberté, de composer nos objets en réutilisant des prototypes déjà créés.

Dans l’exemple suivant, l’héritage prototypal peut apporter une solution plus simple que l’héritage orienté objet classique.

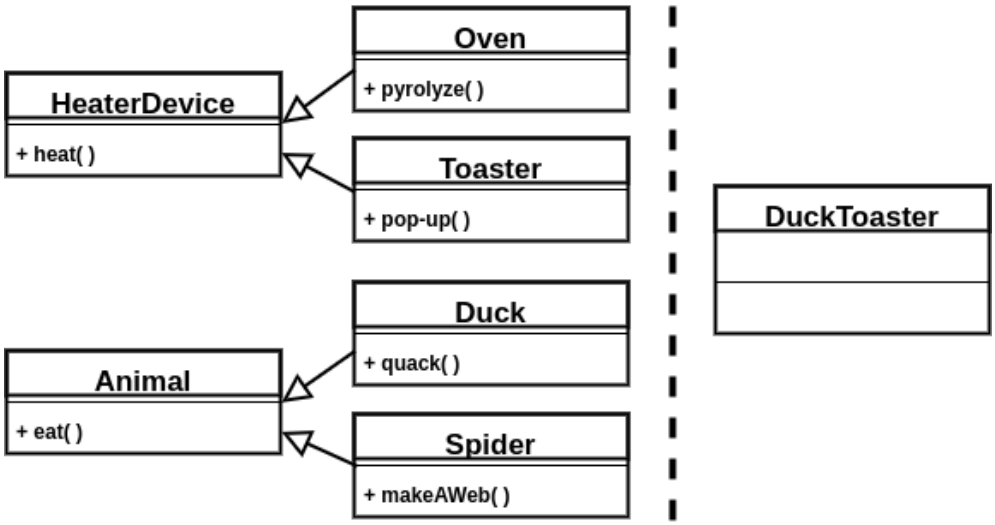


Illustration 70: Schéma d'une situation problématique pour l'héritage orienté objet.

Nous voudrions créer la classe *DuckToaster* qui représente un grille pain en forme de canard qui émet le son « quack » lorsque les tartines sont grillées. L’objectif est de parvenir à une solution sans réécrire plusieurs fois le même code et sans faire perdre son sens à la hiérarchie des classes.

En orienté objet classique, la meilleur solution serait de faire hériter *DuckToaster* de *Toaster* et de *Duck* (héritage multiple). Cependant, l’héritage multiple n’est pas supporté par tous les langages orientés objet et la méthode *eat( )* de la classe *Animal* sera héritée par *DuckToaster* alors qu’elle est, ici, indésirable : les grilles pain ne mangent pas. Pour se débarrasser de la méthode *eat( )*, il faudra réécrire par dessus et ainsi endommagé le sens de la hiérarchie des classes de notre exemple.

Avec l’héritage concaténatif, l’approche est toute autre. Au lieu de créer des classes qui sont des sous-types les unes des autres, nous définissons des comportements que nous voudrions réutiliser à différents endroits. Ainsi, lorsque nous créons un objet qui a besoin de ces comportements, nous les importons et les assemblons au sein de cet objet grâce à l’héritage concaténatif.

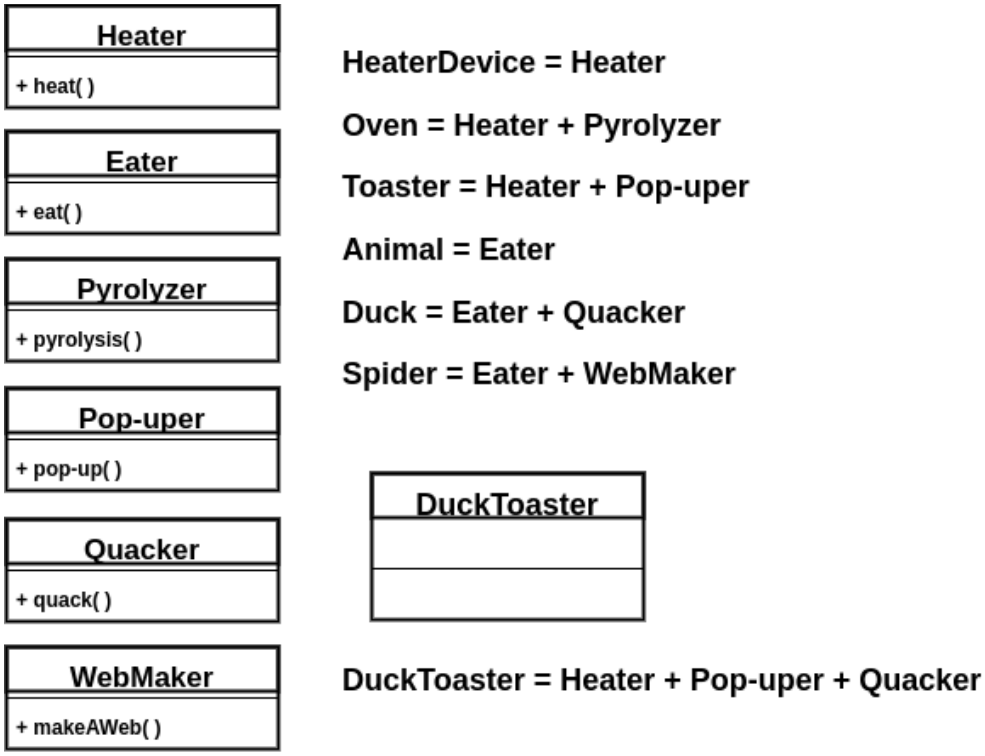


Illustration 71: Schéma illustrant la même situation que l'illustration 70, mais utilisant l'héritage concaténatif.

8.2.3. L’architecture

Pour cette deuxième implémentation, j’ai principalement utilisé l’héritage concaténatif pour structurer et factoriser le code. En effet, la composition via le HTML ne se prêtait pas à la situation.

Il n’existe apparemment pas de normalisation (UML, etc) pour représenter une application Polymer. Cependant, j’ai tout de même tenté de réaliser un diagramme schématisant mon application. Ce diagramme doit être interprété de la manière suivante.

- En haut, on trouve des composants Polymer spécialisés.

  Ils contiennent une représentation HTML, du code JavaScript et tout ce qu’il faut pour instancier un composant. Ce sont les éléments que nous utilisons pour composer du pseudo-code dans notre éditeur.

- En bas, on observe des comportements spécifiques.

  Ils ne contiennent que des méthodes JavaScript que nous voudrions isoler pour pouvoir les réutiliser dans plusieurs composants.

- Enfin, on remarque que les composants pointent sur des chaînes de prototypes.

  Les composants sont dotés des comportements contenu dans leur chaîne d’héritage.

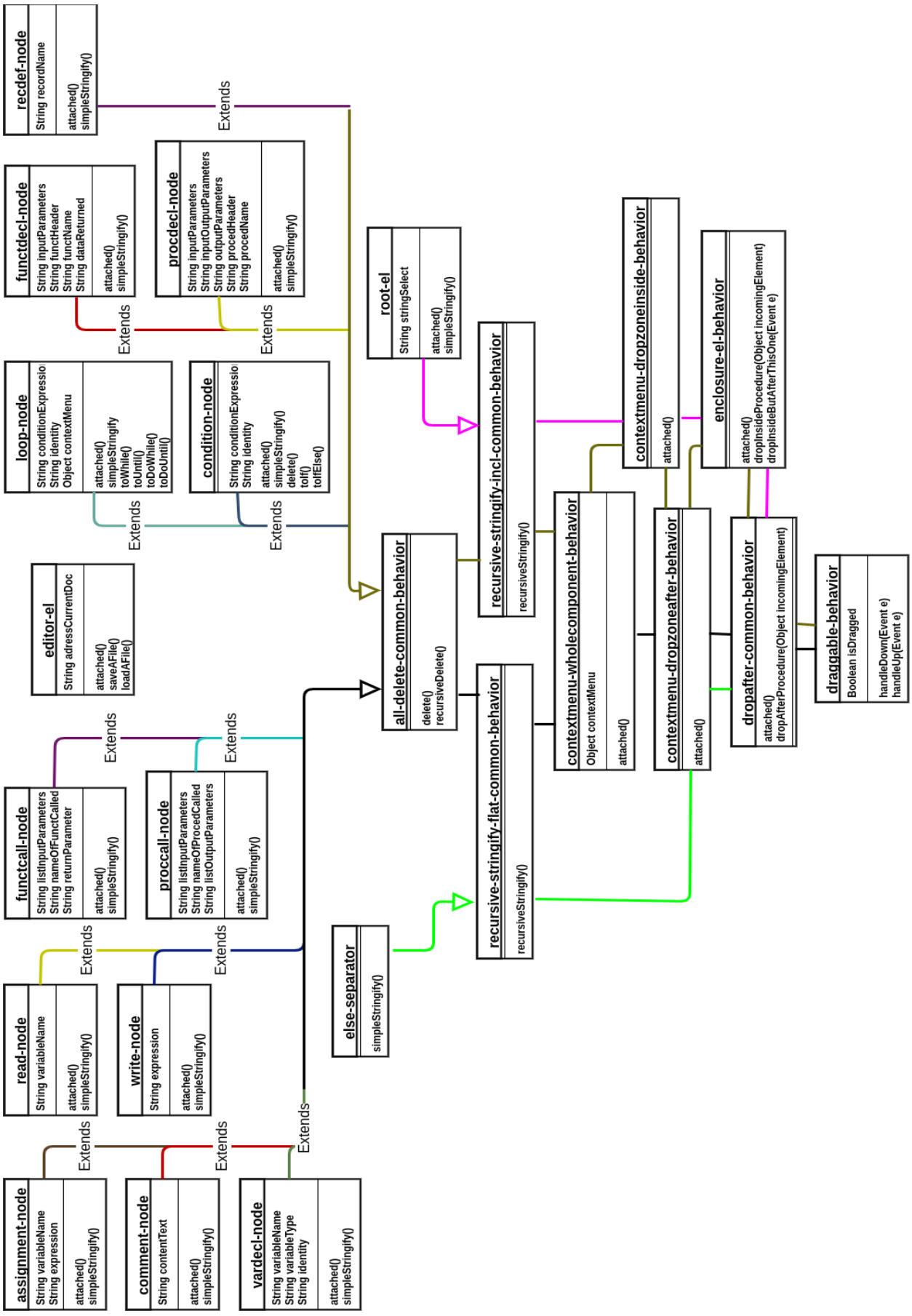


Illustration 72: Schéma illustrant l'implémentation de l'application avec Polymer

Pour être capable d'appréhender ce schéma, il nous faut absolument connaître le rôle de chacun des comportements utilisés dans l'application.

All-delete-common-behavior :

Le comportement *all-delete-common-behavior* apporte au composant qui en hérite les méthodes qui lui permettent de :

- supprimer le nœud qui lui correspond dans l'arbre pseudo-code, sans supprimer ses enfants : delete() ;
- supprimer le nœud qui lui correspond dans l'arbre pseudo-code en supprimant également ses enfants : recursiveDelete().

Recursive-stringify-flat-common-behavior :

Le comportement *recursive-stringify-flat-common-behavior* apporte au composant qui en hérite la méthode qui lui permet de se traduire avec son état interne en une balise HTML. Ce comportement vise les éléments plats et utilise la méthode simpleStringify( ) du composant qui en hérite.

Recursive-stringify-incl-common-behavior :

Le comportement *recursive-stringify-incl-common-behavior* apporte au composant qui en hérite la méthode qui lui permet de se traduire avec son état interne et tous ses enfants en une suite de balises HTML imbriquées, d'une manière telle qu'elles reproduisent le sous arbre pseudo-code dont le composant héritant est la racine. Pour y parvenir, la méthode effectue des appels récursifs sur les enfants du composant.

Contextmenu-wholecomponent-behavior :

Le comportement *contextmenu-wholecomponent-behavior* apporte au composant qui en hérite la base du menu contextuel utilisé par tous les nœuds de l'arbre pseudo-code. La base du menu contextuel est composée des fonctionnalités de copie, copie récursive, suppression et suppression récursive.

Contextmenu-dropzoneinside-behavior :

Le comportement *contextmenu-dropzoneinside-behavior* apporte au composant qui en hérite un menu contextuel avec la fonctionnalité de collage pour le point d'insertion interne au composant. Ce comportement vise les éléments inclusifs.

Contextmenu-dropzoneafter-behavior :

Le comportement *contextmenu-dropzoneafter-behavior* apporte au composant qui en hérite un menu contextuel avec la fonctionnalité de collage pour le point d'insertion post-élément.

Enclosure-el-behavior :

Le comportement *enclosure-el-behavior* apporte au composant qui en hérite les méthodes nécessaires pour ajouter un enfant au composant à une position déterminée. Ce comportement vise les éléments inclusifs.

Dropafter-common-behavior :

Le comportement *dropafter-common-behavior* apporte au composant qui en hérite la méthode nécessaire pour déclencher l'ajout d'un enfant post-élément. Cet méthode dépendant des méthodes du comportement *enclosure-el-behavior*.

Draggable-behavior :

Le comportement *draggable-behavior* apporte au composant qui en hérite les méthodes nécessaires pour permettre à l'élément d'être déplacé par drag&drop et de déclencher les méthodes adéquates selon le point d'insertion sur lequel il est lâché.

Maintenant, il nous est possible de pleinement comprendre le précédant schéma et nous y remarquons cinq groupes de composants.

- Le groupe des composants plats simples :

  Ce groupe est constitué des composants *assignment-node*, *comment-node*, *vardecl-node, read-node*, *write-node, functcall-node* et *proccall-node*. Tous les membres de ce groupe sont des *feuilles* de l'arbre pseudo-code. Leurs chaînes d'héritage indiquent :

  - qu'ils sont déplaçables par drag&drop ;
  - qu'ils est possible de placer un élément après eux;
  - qu'ils possèdent la base du menu contextuel qui leur donne accès aux fonctionnalités de suppression et de copie ;
  - qu'ils peuvent se traduire avec leur état interne en une balise HTML.

- Le composant *else-separator* :

  Le composant *else-separator* est, rappelons-le, le petit séparateur *else* qui permet de créer un composant *if-else* à partir d'un composant *if*. On remarque que sa chaîne d'héritage est très proche de celle du groupe des composants plats simples. Toutefois, elle s'en différencie car :

  - le séparateur *else* n'est pas directement supprimable pour une question cohésion ;
  - le séparateur *else* n'a pas de menu contextuel car il ne peut pas être supprimé directement et qu'il n'y aurait pas de sens à le cloner seul ;
  - le séparateur *else* ne peut pas être déplacé par drag&drop.

- Le composant *editor-el* :

  Le composant *editor-el* ne fait pas partie de l'arbre pseudo-code. Il n'a besoin d'aucun des comportements mis en commun par les nœuds de l'arbre.

- Le groupe des composants inclusifs simples :

  Ce groupe est constitué des composants *loop-node*, *condition-node*, *functdecl-node*, *procdecl-node* et *recdef-node*. Tous les membres de ce groupe sont des *branches* de l'arbre pseudo-code. Leurs chaînes d'héritage indiquent :

  - qu'ils sont déplaçables par drag&drop ;
  - qu'ils est possible de placer un élément après eux ;

- qu'ils est possible de placer des éléments à l'intérieur d'eux ;
- qu'ils possède la base du menu contextuel qui leur donne accès aux fonctionnalités de suppression et de copie ;
- qu'ils peuvent se traduire avec leur état interne en une balise HTML ;
- qu'ils peuvent se traduire avec leur état interne ainsi que tous leurs enfants en un série de balise HTML reflétant le sous-arbre dont ils sont la racine.

■ Le composant *root-el* :

Le composant *root-el*, racine de l'arbre pseudo-code, a une chaîne d'héritage très proche de celle du groupe des composants inclusifs simples. Cependant elle en diffère car :

- elle n'autorise pas le suppression de la racine ;
- elle n'apporte pas la base du menu contextuel à la racine, la privant des fonctionnalités de copie de suppression ;
- elle ne permet pas que l'on puisse insérer des éléments après la racine.

8.2.4. La sauvegarde des données

Les données de l'application sont, comme dans la précédente implémentation, sauvegardées dans un fichier XML. Cependant, ici, nous n'employons pas de bibliothèque pour traduire l'arbre pseudo-code en mémoire en un arbre XML. En effet, nous effectuons, nous-même, la traduction de l'arbre en mémoire en une chaîne de caractères respectant la syntaxe XHTML. Pour ce faire, nous parcourons récursivement notre arbre de composants en mémoire et demandons à chacun d'entre eux de se traduire en une chaîne de caractères XHTML. Les chaînes sont assemblées entre elles au fur et à mesure.

Le langage XHTML impose quelques règles à nos chaînes de caractères.

- Les éléments vides, comme <*img*>, par exemple, doivent se terminer en balise fermante (*/>)* ou être utilisés sous leur forme à deux balises (< ></ >).
- Les noms des balises et des attributs doivent être exclusivement en minuscules.
- Les caractères < et *&* doivent être représentés sous leur forme d'entité HTML (*&l* et *&amp*). Cela inclus aussi les séparateurs utilisés dans une URL.
- Les valeurs des attributs doivent être correctement enfermées entre guillemets.

Il existe aussi des règles relatives aux scripts, mais elles ne nous seront pas nécessaires dans notre cas.

```
<assignment-node
        variable-name=”i”
        expression=’1”>
</assignment-node>
<loop-node
        identity=”while”
        condition expression=”i < 10”>

    <assignment-node
            variable-name=”i”
           expression=”i + 1”>
    </assignment-node>

</loop-node>
```

Text 5: Un extrait de XHTML illustrant la syntaxe des fichiers de sauvegarde de notre application.

8.2.5. Le chargement des données

Nous n’allons pas avoir besoin d’écrire beaucoup de code pour charger en mémoire et afficher nos données sauvegardées. En effet, grâce à Polymer et à la technologie des composants web, nos balises HTML sauvegardées sont connues de l’interpréteur HTML. Il nous suffit donc de les lui présenter pour qu’elles soient automatiquement converties en composant HTML. Ainsi, il ne nous reste plus qu’à les ajouter aux bons emplacements dans le DOM de notre application.

Toutefois, pour des raisons de sécurité, nous allons quand même vérifier le contenu des documents XML avant de les charger. Pour ce faire, il nous faut utiliser un validateur XML : un programme qui détermine si un fichier XML donné respecte une liste de critères donnée.

J’ai essayé trois langages de validation différents : DTD, XSD, Relax NG.

8.2.5.1. Document Type Definition (DTD)

Le langage de description DTD s’intéresse à décrire des modèles d’éléments XML et leur contenu. En DTD, Il faut décrire les attributs, les enfants possibles et les cardinalités pour chaque type de balises que l’on accepte dans notre document. Une description DTD est, quelque peu, analogue à une modélisation de classes en programmation orientée objet : on y décrit des types d’éléments et la façon dont ils s’organisent entre eux.

Les nuances n’ont pas leur place dans une description DTD : un élément ne peut pas exclusivement accepter un certain type d’éléments dans une situation A, puis n’accepter qu’un autre type d’éléments dans une situation B.

Malheureusement nous sommes confronté à cette situation dans notre cas. En effet, la racine d’une page dédiée à la description de records et la racine d’une page dédiée à la description de fonctions sont toutes les deux issues du même type de balises : *Root-el*. Mais nous voudrions qu’elles n’acceptent pas les même types d’enfants.

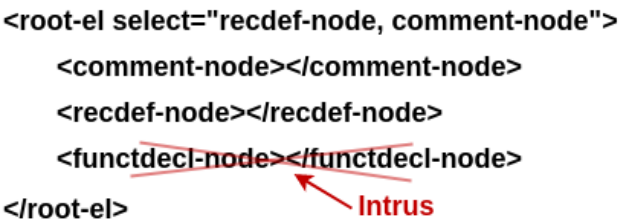


Illustration 73: Un code HTML qui illustre le problème de différenciation de la balise Root-el du point de vue d'une racine de page de déclarations de records.

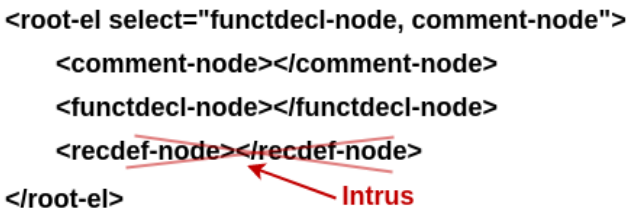


Illustration 74: Un code HTML qui illustre le problème de différenciation de la balise Root-el du point de vue d'une racine de page de déclarations de fonctions.

Une page HTML classique est un bon exemple de document vérifiable avec une description DTD. En effet, le HTML traditionnel a des règles générales qui ne varient pas selon le contexte. Par exemple : une balise aura toujours les mêmes attributs et acceptera toujours les mêmes types d'enfants peu importe la situation.

8.2.5.2. XML Schema Definition (XSD)

Le langage de description XSD s'intéresse a décrire avec précision une hiérarchie plus ou moins immuable de données. Le XSD pourrait, par exemple, servir à vérifier des fichiers de configuration. En effet, XSD permet de garantir la présence d'une donnée précise à un endroit précis. Il offre aussi une certaine flexibilité en permettant de décrire des listes d'éléments désordonnées ou la possible absence d'un élément. Cependant, il n'est pas capable de décrire des assemblages pleinement dynamiques d'éléments.

L'arbre pseudo-code que nous voulons vérifier est pleinement dynamique : n'importe quel élément parmi une liste peut apparaître dans n'importe quel ordre, un nombre de fois variant entre 0 et n fois. Ces éléments, à leur tour, peuvent contenir eux aussi une liste dynamique d'éléments. XSD ne peut pas décrire une telle structure.

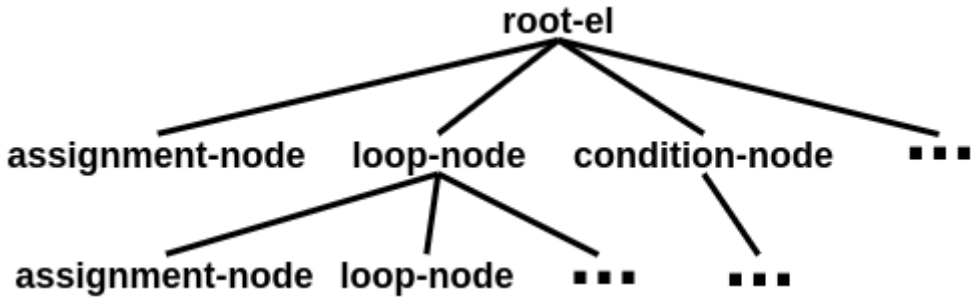


Illustration 75: Un exemple de structure dynamique impossible à décrire avec XSD

8.2.5.3. Relax NG

Relax NG combine les capacités du DTD et du XSD, il peut décrire une structure précise et prédéfinie, mais aussi décrire un comportement type pour une classe d'éléments.

Ainsi, j'ai décidé d'effectuer les validations des fichiers XML chargés dans l'application à l'aide de Relax NG.

En utilisant un validateur XML et un langage de validation, nous économisons beaucoup de code, nous réduisons significativement la complexité de l'application et nous découplons l'opération de *traduction* du XML de celle de la vérification.

Ainsi, il nous est possible, sans trop d'encombrement, de conserver les fichiers de validation pour chaque version de notre application et ainsi pouvoir, dans le futur, identifier avec certitude des fichiers créés avec d'anciennes versions de l'application. Si il nous est possible d'identifier la version du programme sous laquelle ces fichiers ont été créés, alors il nous sera probablement aussi possible de mettre à jour ces fichiers vers la dernière norme de sauvegarde de l'application. La rétrocompatibilité des fichiers pourrait être, en conséquence, assurée.

9. Captures d'écran de l'application

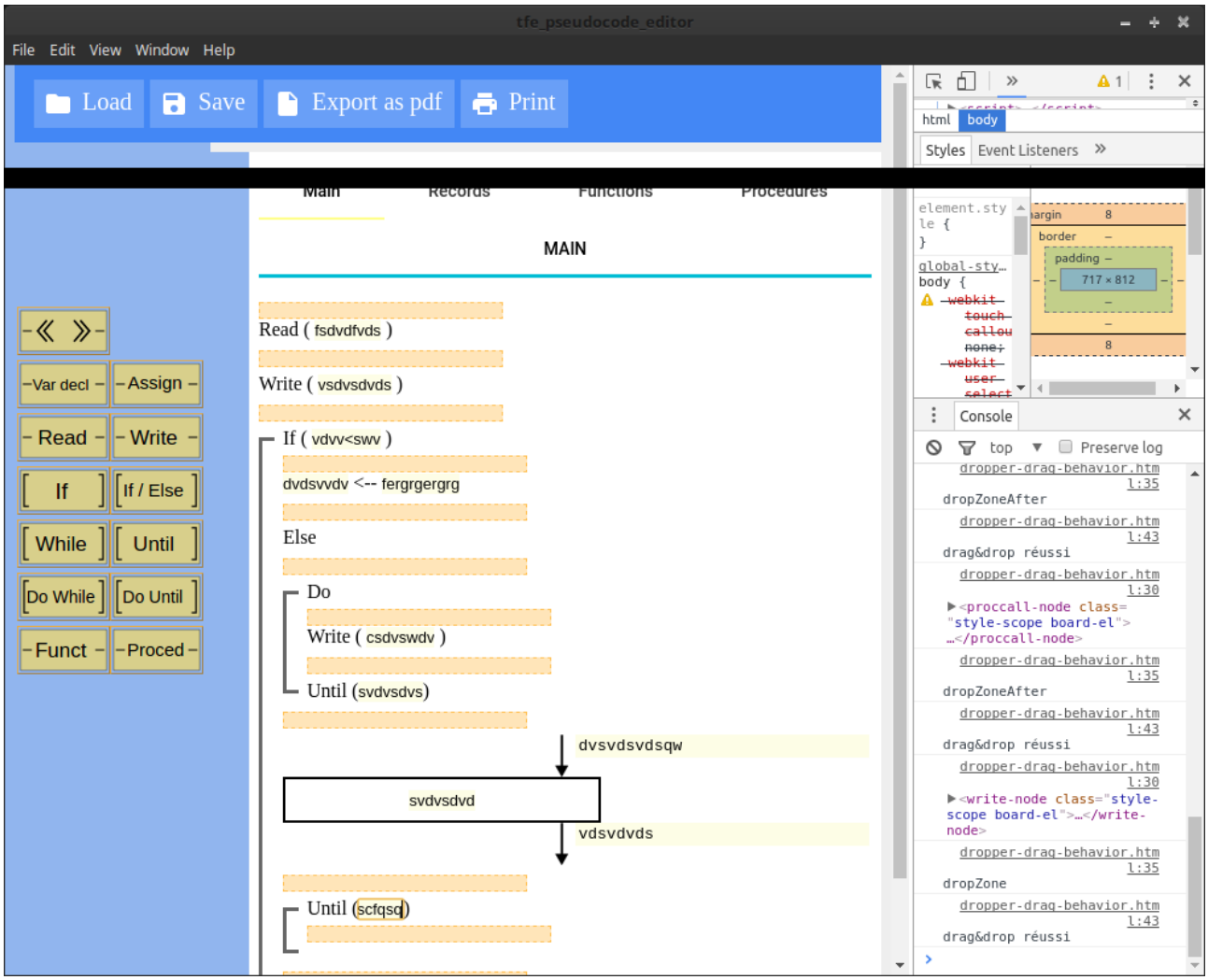


Illustration 76: Capture d'écran de la page principale.

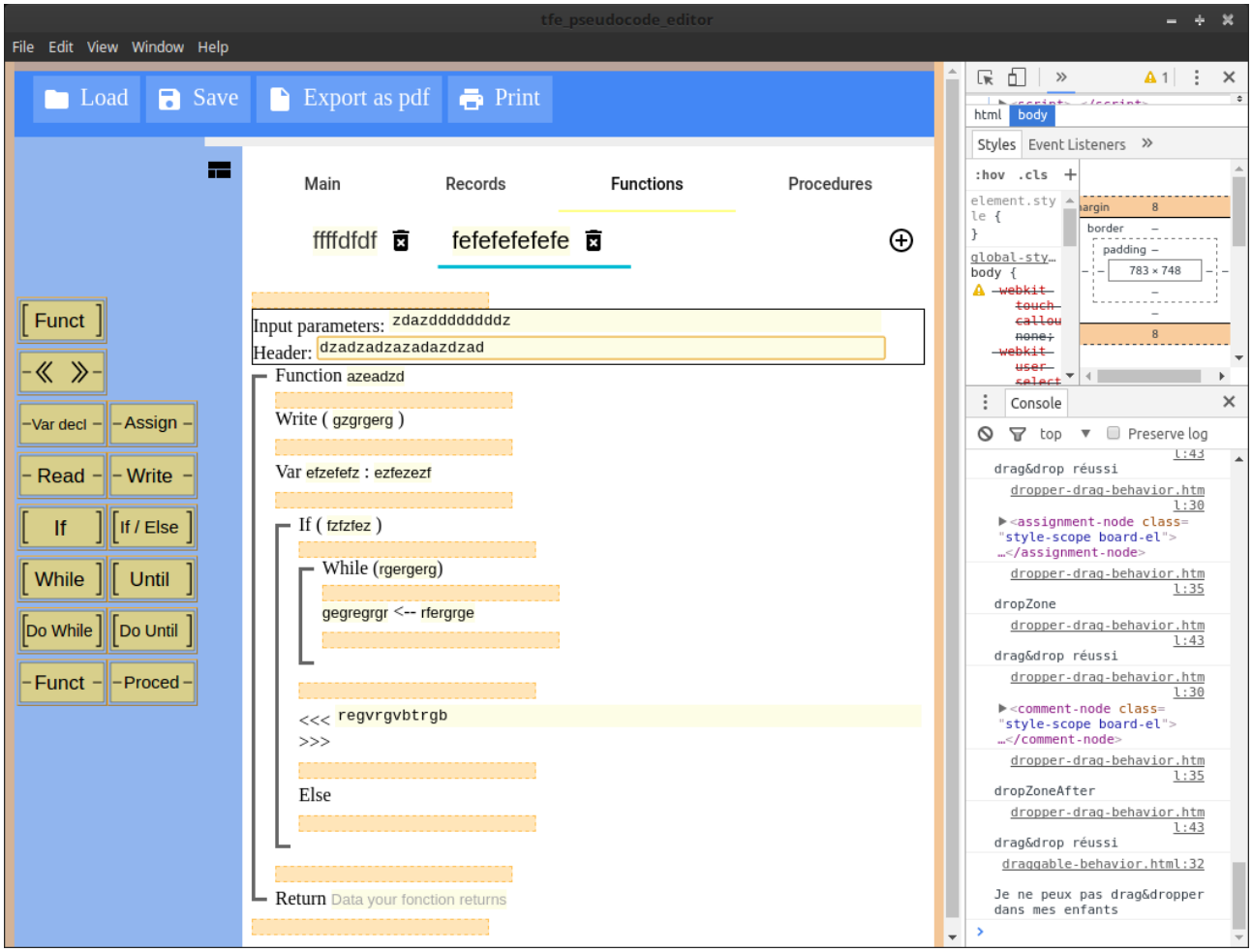


Illustration 77: Capture d'écran d'une page de définition de fonctions.

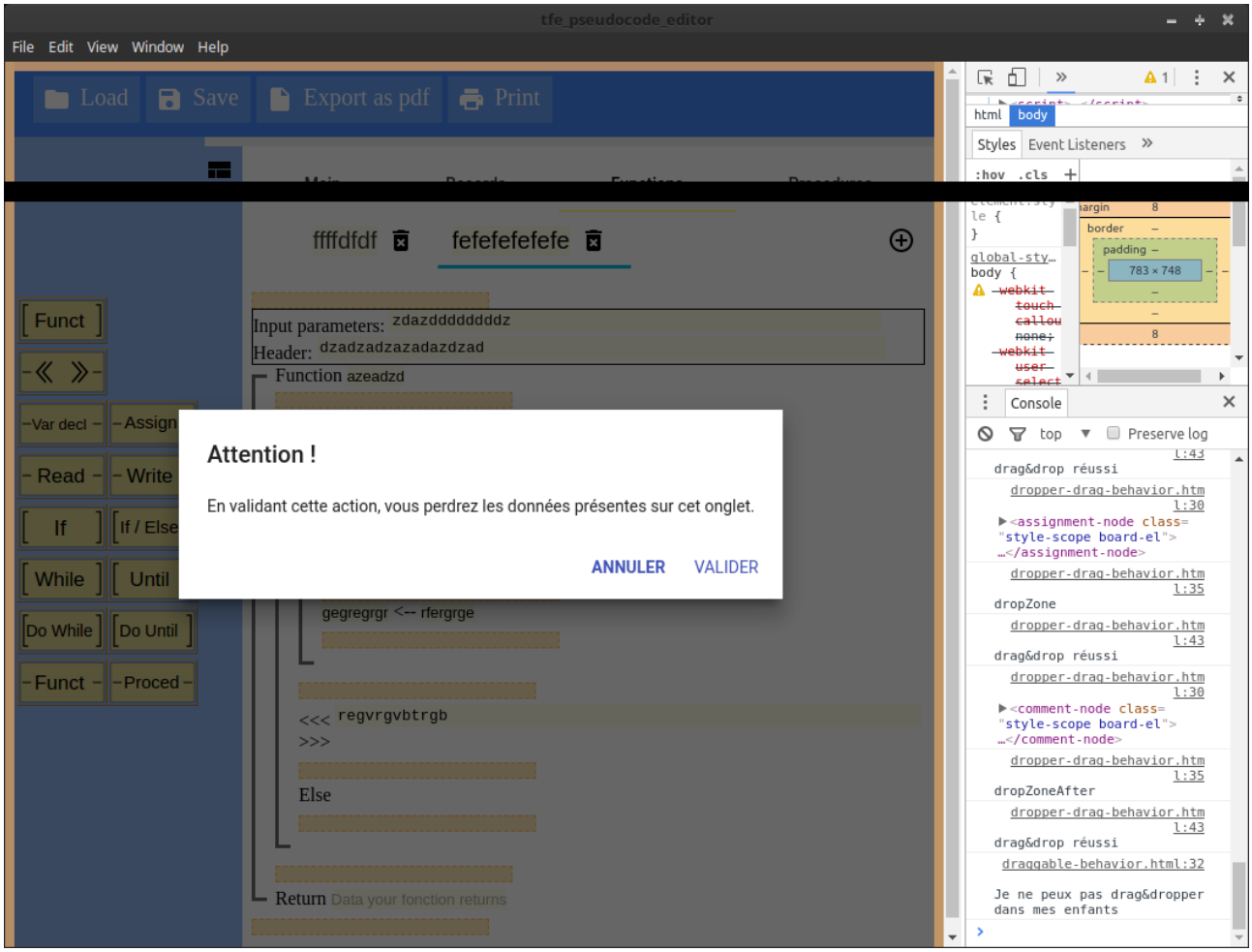


Illustration 78: Capture d'écran de la demande de confirmation pour supprimer une page de définition de fonctions.

10. Conclusion

Le développement de cette application a été difficile car je n'y étais pas préparé. Je pense que résumé mon travail n'apporterait pas de valeur ajoutée à cette conclusion, alors je vais plutôt lister ici les leçons que j'ai tirées de ce travail.

Premièrement, avant de démarrer un quelconque projet, il est toujours bon de s'informer sur les solutions qui existent déjà et de consulter l'avis de personnes plus qualifiées que nous dans ce domaine. Cela permet d'anticiper les problèmes et d'économiser beaucoup de temps.

Pour ma part, je n'avais pas conscience de la complexité et de l'exotisme du projet dans lequel je me lançais. Je ne connaissais rien à la programmation Front-End et au JavaScript. J'ai donc perdu beaucoup de temps à apprendre et à tester.

Deuxième, je pense que nous verrons de plus en plus d'applications web apparaître et peut-être aussi de plus en plus d'outils pour les réaliser. Chacun de ces outils ayant ses spécificités, je pense qu'il est important de développer une culture générale des outils pour savoir quel outil utiliser dans quelle situation.

Bibliographie :

Polymer Team, Documentation Polymer 1.x, in https://www.polymer-project.org/, dernière consultation le 24 août 2017.

CodeCademy, Tutoriaux React & Angular, in https://www.codecademy.com/, dernière consultation juin 2017.

Haute École Robert Schuman, Catégorie paramédical, in http://hers.be/index.php/paramedical, dernière consultation 15 août 2017.

Haute École Robert Schuman, Catégorie paramédical, in http://hers.be/index.php/economique, denière consultation 15 août 2017.

Wikipedia.org, Java (programming langage), in https://en.wikipedia.org/wiki/Java_%28programming_language%29, dernière consultation 16 août 2017.

Wikipedia.org, Swing (Java), in https://en.wikipedia.org/wiki/Swing_(Java), dernière consultation 16 août 2017.

Wikipedia.org, C++, in https://en.wikipedia.org/wiki/C%2B%2B, dernière consultation 16 août 2017.

Wikipedia.org, Qt (software), in https://en.wikipedia.org/wiki/Qt_(software), dernière consultation 16 août 2017.

Wikipedia.org, JavaFx, in https://en.wikipedia.org/wiki/JavaFX, dernière consultation 16 août 2017.

Wikipedia.org, Electron (software framework), in https://en.wikipedia.org/wiki/Electron_(software_framework), dernière consultation 16 août 2017.

Wikipedia.org, TypeScript, in https://en.wikipedia.org/wiki/TypeScript, dernière consultation 16 août 2017.

Wikipedia.org, ESCMAScript, in https://en.wikipedia.org/wiki/ECMAScript, dernière consultation 16 août 2017.

Wikipedia.org, jQuery, in https://en.wikipedia.org/wiki/JQuery, dernière consultation 16 août 2017.

Wikipedia.org, jQuery UI, in https://en.wikipedia.org/wiki/JQuery_UI, dernière consultation 16 août 2017.

Wikipedia.org, Bootstrap (front-end framework), in https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework), dernière consultation 16 août 2017.

Wikipedia.org, Angular (application platform), in https://en.wikipedia.org/wiki/Angular_(application_platform), dernière consultation 16 août 2017.

Wikipedia.org, React (JavaScript library), in https://en.wikipedia.org/wiki/React_(JavaScript_library), dernière consultation 16 août 2017.

Wikipedia.org, Polymer (library), in https://en.wikipedia.org/wiki/Polymer_(library), dernière consultation 16 août 2017.

Wikipedia.org, Relax NG, in https://fr.wikipedia.org/wiki/Relax_NG, dernière consultation 16 août 2017.