Projet industriel

Robot navigateur

l'internaute virtuel

Master II Pro SIS GL

2008/2009

Équipe :

Grégory ANNE

Yosra BARATLI

Yoann JANSZEN

Papa Issa DIALLO

Encadrant :

Jean CAUSSANEL

Table des matières

1. Introduction

Les technologies web se développent et s'améliorent de jour en jour, en répondant aux différents besoins des propriétaires des sites web ainsi que ceux des internautes, surtout avec l'utilisation indispensable de l'internet dans tous les domaines.

A partir de là, plusieurs outils peuvent être profitables pour analyser la navigation et par la suite pour avoir des statistiques sur les pages et les liens les plus fréquentés par les internautes.

Dans le cadre des recherches en cours du Laboratoire des sciences de l'information et des systèmes (LSIS), notre projet industriel porte sur la création d'une application JAVA capable de parcourir de manière automatique un site web, et enregistrer les liens des pages parcourues..

Pour réaliser ce projet, il est nécessaire de récupérer le code source de la page affichée sur le navigateur, de parcourir ce code pour trouver tous les liens accessibles à partir de cette page et enfin de choisir un lien au hasard pour afficher la page correspondante et reprendre l'exécution de l'application jusqu'à ce qu'une condition d'arrêt soit vérifiée.

Quelles méthodes seront utilisées pour le parcours du code html d'une page web, et la recherche des différents types de liens contenus dans ce dernier?

Et quels outils et techniques seront utiles pour la réalisation de l'application ?

1.1 Sujet

- Proposé par : Jean Caussanel (Jean.Caussanel@univ-cezanne.fr)
- Pré requis : Bases de la programmation objet Java
- Sujet :

  Il s'agit d'écrire une application JAVA capable de parcourir de manière automatique un site Web. Les paramètres d'entrée seront : l'URL de la page d'accueil du site, le temps passé sur chaque page. L'application analyse chaque page ouverte pour en extraire tous les liens accessibles depuis celle ci. Une fonction de sélection est appliquée permettant de sélectionner un lien dans l'ensemble des liens trouvés. L'application va alors chercher la page suivante et recommence la boucle tant qu'une condition de fin n'est pas vérifiée. D'autres conditions ou paramètres seront définis dans la phase d'expression des besoins. L'application devra intégrer ou dialoguer avec un navigateur Web pour le recueil des pages web et la demande de chargement de la page suivante.

2. Phase de recherche

2.1 Les navigateurs

2.1.1 Piste 1 : navigateur minimal construit en swing

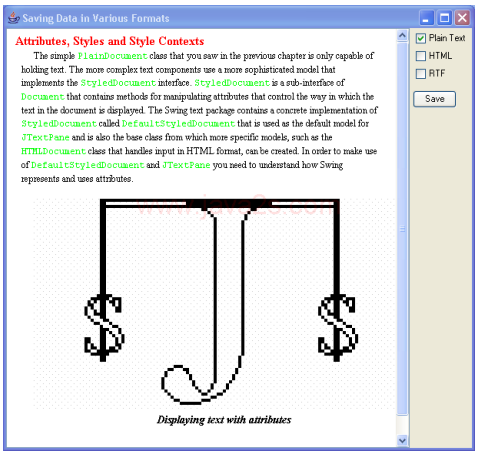

Illustration 1: Un exemple d'utilisation de javax.swing.jeditorpane

Les JEditorPane peuvent inclure de l'interprétation HTML construite à partir d'une URL, ils gèrent le CSS mais pas le JavaScript : sur beaucoup de sites, l'affichage serait très limité. Cette piste est donc abandonnée.

2.1.2 Piste 2 : navigateurs open source développés en java

Illustration 2: Le programme java JDIC_browser

- JDIC est une API Java dont les classes permettent de générer un petit navigateur Java. Le snippet de démonstration sur le site ne fonctionnait pas : une exception "*Exception in thread "EventThread java.lang.NullPointerException at org.jdesktop.jdic.browser.internal.MsgClient.<init>(Unknown Source)*" est affichée au lancement, ce qui empêche l'utilisation correcte de la GUI. Seul le code compilé est disponible, il est donc impossible de corriger. Il s'est avéré que JDIC n'était compatible qu'avec IE et Mozilla sur Windows. JDIC est obsolète, il supporte Mozilla et non pas Firefox.
- JRex : JRex est un module open-source développé dans le cadre du projet mozilla. Il s'agit d'un ensemble de packages java permettant notamment d'obtenir un composant graphique (JRexCanvas) contenant un navigateur web. Mais celui-ci n'a pas été mis à jour depuis trois ans, il ne supporte que Mozilla et non Firefox (même problème que JDIC).

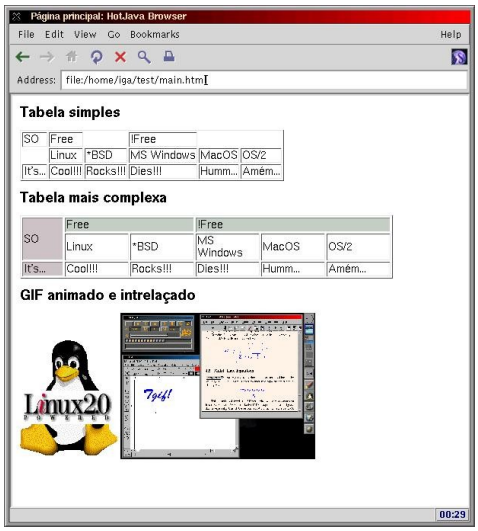

Illustration 3: navigateur HotJava

- HotJava : l'idée était de disposer d'un outil Sun Microsystems afin que l'adaptation au code Java soit optimale, malheureusement cet outil est si obsolète qu'un essai préalable du logiciel n'a même pas été possible, les sources ou les éléments nécessaires à l'installation sont introuvables. HotJava était à la base créé dans le but d'embarquer des navigateurs WAP sur les téléphones portables par exemple : HotJava présente de toute façon un navigateur extrêmement basique.
- Lobo Browser : il s'agit d'un projet Open Source cette fois actuel, qui inclut l'analyse du JavaScript. Après essai, ce navigateur n'est plus adapté aux dernières technologies, la plupart des sites incluent de l'Ajax et celui-ci n'est pas interprété par Lobo, le rendu de la plupart des sites s'avère donc très peu acceptable, de plus le fonctionnement du programme est très long, il utilise beaucoup trop de ressources.

Cette piste est aussi abandonnée.

2.1.3 Piste 3 : Mozilla Firefox Web Browser

Nous savons d'ores et déjà que Mozilla Firefox est capable de lire tout type de contenu sur le web, AJAX inclus. Il est possible de lancer mozilla ou de lui faire changer de page à partir de commandes shell. Cette piste est donc une première possibilité pour notre programme, il se peut que les traitements de code (pour trouver les différents liens soient faits d'un coté, pendant qu'une autre couche du programme se charge de l'affichage en envoyant ces commandes remote à Firefox.

2.1.4 Piste 4 : JavaXPCOM et événements de Navigateur

XPCOM (Cross-Platform Component Object Model) est un modèle libre de composants développé par la Fondation Mozilla.

Il s'agit d'une bibliothèque logicielle qui, schématiquement, permet de concevoir une application comme un ensemble de plugins. Ces composants peuvent être développés dans des langages distincts, auquel cas les communications entre ces composants sont assurées par la technologie XPCONNECT. Cette bibliothèque a servi, notamment, à développer Mozilla, Mozilla Firefox, Mozilla Thunderbird, Mozilla Composer, Nvu, Komodo…

JavaXPCOM autorise la communication entre java et xpcom, de telle sorte que des applications puissent accéder à des objets xpcom, et xpcom puisse accéder à des classes de java qui implémentent une interface xpcom. Avec JavaXPCOM un développeur peut communiquer avec xpcom ou intégrer Gecko(Navigateur de rendu Mozilla).

Javaxpcom est maintenant construit par défaut comme partie de xulrunner. **Pour l'utiliser il faut donc télécharger une version récente de xulrunner**.

Il peut être une solution pour les événements sur un navigateur : xpcom permet d'écouter le processus courant d'un navigateur Firefox pour savoir s'il est en cours de chargement par exemple. Peut-être est-il aussi possible d'avoir des écouteurs sur les différents hyperliens inclus dans une page web...

Pour intégrer mozilla dans une application java, il est nécessaire d'ajouter la librairie MozillaInterfaces.jar (on la trouve dans le répertoire xulrunner/sdk/lib/MozillaInterfaces.jar). Cette librairie fournie les interfaces nécessaires pour appeler les méthodes de XPCOM et amorcer mozilla. Pour l'intégrer on utilise la classe singleton Mozilla.

On peut passer des objets java aux méthodes de XPCOM, par exemple:

```
Mozilla mozilla = Mozilla.getInstance();
WindowCreator creator = new WindowCreator();  // implements nsIWindowCreator

nsIServiceManager serviceManager = mozilla.getServiceManager();

nsIWindowWatcher windowWatcher = (nsIWindowWatcher) serviceManager
  .getServiceManagerByContractID(NS_WINDOWWATCHER_CONTRACTID,
    nsIWindowWatcher.NS_IWINDOWWATCHER_IID);
windowWatcher.setWindowCreator(creator);
```

2.1.5 Piste 5 : Navigateur SWT basé sur moteur Gecko développé par Eclipse Foundation

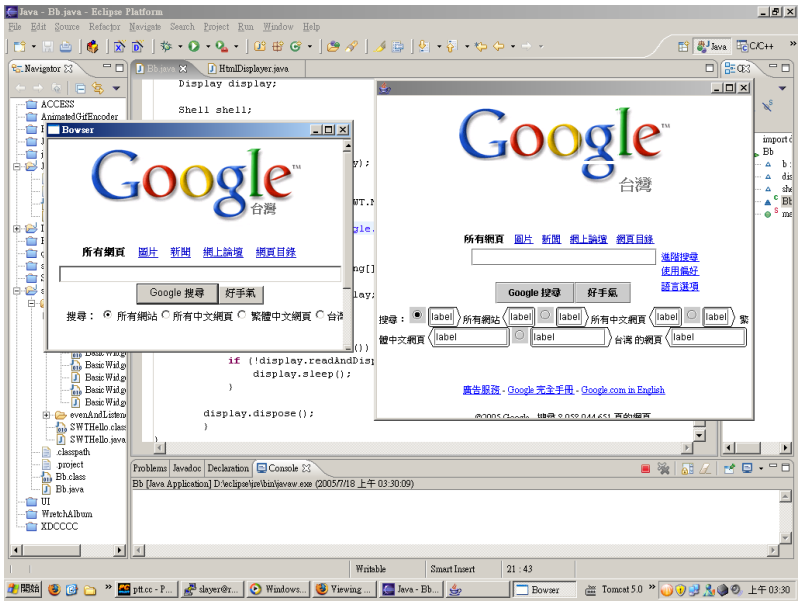

Illustration 4: L'utilisation du composant graphique browser de SWT

Standard Widget Toolkit (SWT) est une bibliothèque graphique libre pour Java, initiée par IBM. SWT n'est pas un standard Java reconnu par le JCP. Cette bibliothèque se compose d'une bibliothèque de composants graphiques (texte, label, bouton, panel), des utilitaires nécessaires pour développer une interface graphique en Java, et d'une implémentation native spécifique à chaque système d'exploitation qui sera utilisée à l'exécution du programme.

La deuxième partie de SWT n'est en fait qu'une réencapsulation des composants natifs de système (Win32 pour Windows, GTK ou Motif pour Linux). Plusieurs projets travaillent aujourd'hui sur une implémentation utilisant les composants de Swing.

L'environnement de développement libre Eclipse, sponsorisé lui aussi par IBM, repose sur cette architecture.

L'un des composants proposé par SWT est "browser". Celui-ci intègre un affichage de pages web basé sur le moteur Gecko. Et de plus, il est possible par ce composant de récupérer le document DOM d'une page web actuellement visitée (document DOM généré une fois le Javascript exécuté).

2.2 Autres technologies utiles au projet

2.2.1 JTidy

JTidy est une version Java de HTML Tidy, un vérificateur de syntaxe HTML. Tout comme son cousin non-Java, JTidy peut être utilisé comme un outil pour nettoyer le HTML mal formé ou contenant des erreurs. En outre, JTidy fournit une interface DOM aux documents chargés, ce qui permet d'utiliser JTidy comme un parser DOM de HTML.

Les outils que nous avions initialement à disposition en Java pour le parcours DOM (org.w3c.dom

ou même JDOM) ne peuvent générer un document DOM qu'à partir d'un code valide. Les documents HTML du web sont souvent générés par des AGL web qui produisent un bon nombre d'erreurs de syntaxe XML dans le code, c'est la raison pour laquelle nous avons généré nos documents DOM en deux temps grâce à JTidy. JTidy peut prendre en paramètre n'importe quel flux, issu d'une URL ou bien d'un code textuel, et générer à partir de ceci un document DOM (org.w3c.dom) valide : il est d'un grand intérêt dans la fermeture des balises non fermées par exemple.

2.2.2 org.w3c.dom

Document Object Model (DOM)

Plutôt que d'utiliser org.w3c.dom et JTidy, il aurait sûrement pu être possible d'utiliser les outils DOM fournis par XPCOM (package org.mozilla.interfaces), ce dernier fournissant même des outils spécifiques de récupération de balises HTML (méthode getLinks() par exemple). Cependant leur fonctionnement est sensiblement différent : contrairement à JTidy, mozilla.interfaces produit des documents DOM non pas à partir d'un flux mais à partir d'une instance de Browser (lorsque l'écouteur du Browser est en "completed" -affichage complet-) et du code contenu dans la fenêtre. L'utilisation de JTidy et la génération de documents DOM par buffers, ou directement par téléchargement à partir de l'adresse, s'est avérée beaucoup plus rapide. Dans la décomposition des frames par exemple, des générations de documents DOM sont imbriquées, l'utilisation simple de mozilla.interfaces aurait considérablement ralenti l'analyse de code. De plus, l'outil DOM est incomplet chez mozilla.interfaces : il manque certains éléments comme l'objet "Element" et des méthodes appendChild() ou removeChild() par exemple que nous utilisons.

Nous nous servons donc des méthodes de mozilla.interfaces uniquement pour la reconstitution de code après exécution de JavaScript.

3. Gestion de projet

3.1 Démarche entreprise

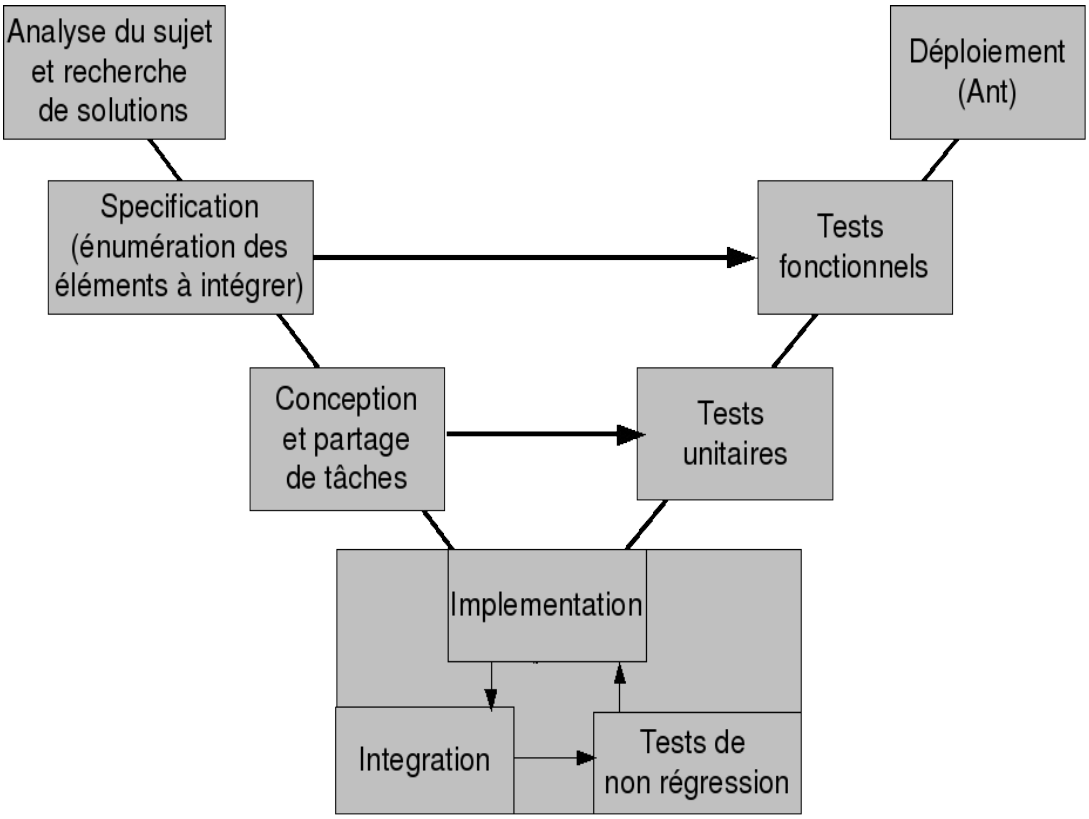

Illustration 5: Cycle en V

3.2 PERT

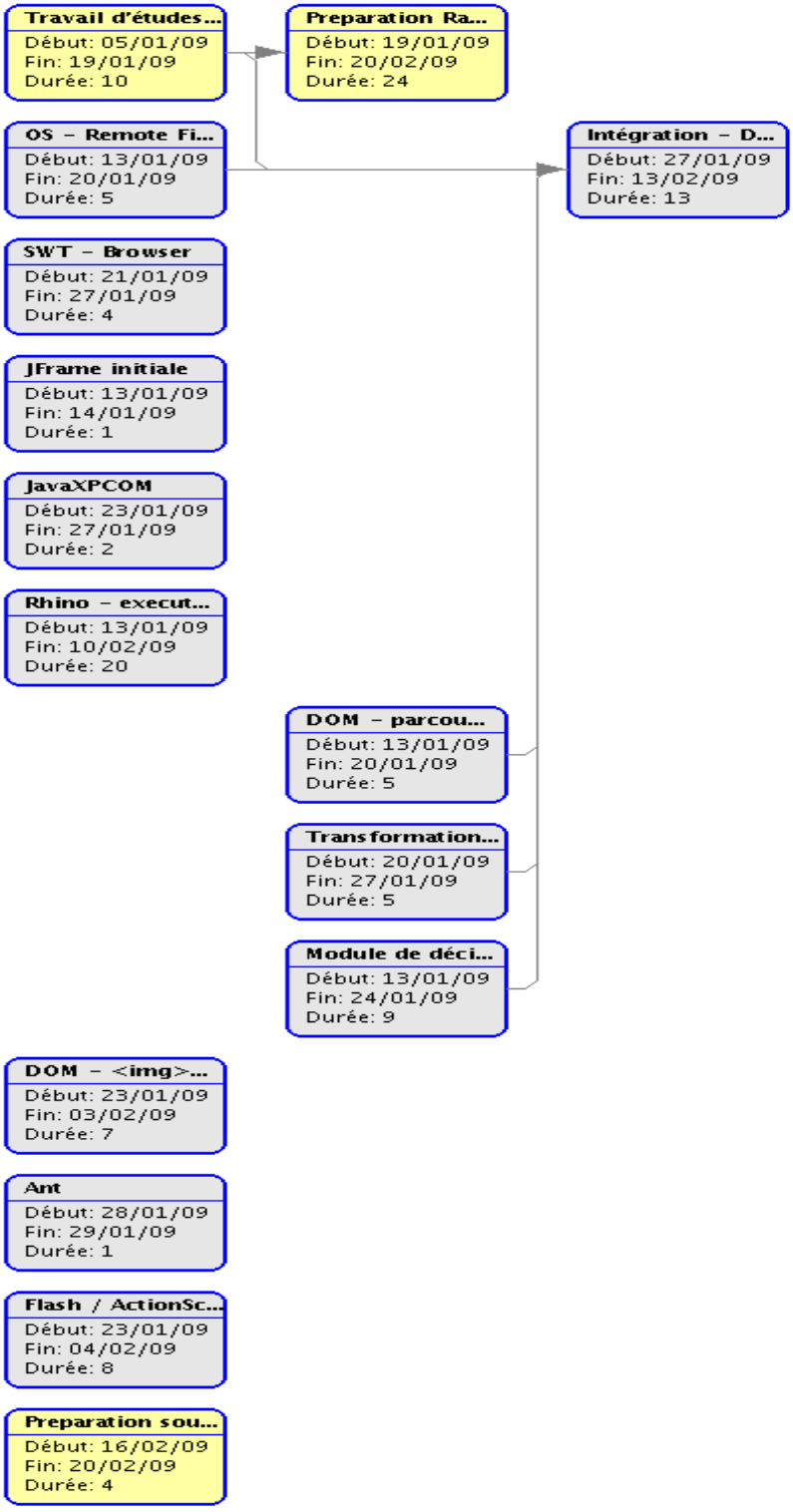

Illustration 6: Diagramme de PERT

3.3 Partage des tâches au fur et à mesure de l'avancement

No Tâche	Domaine	Description	Difficulté	Responsable de la tâche	Aboutissement
13 Janvier 2009					
1	Swing	Création d'une fenêtre d'accès au logiciel pour demander une première URL	*	Issa	100%
2	Systèmes d'exploitation - Shell	Création de scripts de contrôle de mozilla pour Linux, Windows, Solaris, MacOS	**	Yoann	100%
3	~~Rhino~~Gecko	Exécution de Javascript sur des contenus HTML avec les paramètres de l'URL	****	Yoann	100%
4	DOM	Parcours de code HTML et indexation de liens	***	Grégory	90%
5	Math.Random()	Gestion du modèle de décision, choix des pages à visiter selon critères	*	Grégory	100%
22 Janvier 2009					
6	SWT	Organiser le code de navigateur SWT pour l'adapter à nos besoins, essayer de naviguer entre boutons JavaScript au sein de cette sous-application, ajouter des onglets de navigation	**	Yoann	100%
7	Flash, ActionScript	les programmes flash embarqués sur une page web peuvent contenir des liens hypertexte, faire l'état des solutions possibles à cet effet, les mettre en œuvre	**	Issa	100%
8	JavaXPCOM	Faire le rapport des solutions JavaXPCOM et événements sur un navigateur	**	Issa	100%
9	DOM	récupérer les liens hypertexte présents dans les images mappées	*	Grégory	100%
10	DOM	récupérer les liens hypertexte présents dans les frame incluses en HTML transitionnel	***	Grégory	100%
11	JavaXPCOM	Partager les tâches concernant l'exécution de procédures JavaScript, mettre en œuvre les solutions trouvées	****	Yoann	100%
12	Ant	Faire un build.xml pour Ant correct	**	Grégory	100%
30 Janvier 2009					
13	Flash, ActionScript	Récupérer du code ActionScript à partir de l'URL d'un programme Flash, reconnaître dans ce code les boutons de l'application, savoir comment générer les événements correspondants à ces boutons	****	Issa	100%
14	URLChecker	Créer une procédure de tri des liens par serveur web pour pouvoir naviguer sur un site unique	*	Yoann	100%
6 Février 2009					
15	DOM	Récupérer les balises embed	**	Grégory	100%
16	Math.Random()	Adapter le module aux options	**	Grégory	100%

17	JUnit	Compléter les tests unitaires et de non régression	*	Grégory	100%
18	Wiki	Documenter les grands vides du wiki	*	Yoann, Issa, Grégory, Yosra	70%
13 Février 2009					
19	Présentation	Préparer les slides	*	Grégory, Yoann, Issa, Yosra	0%
20	Debugging	Dernières tentatives sur l'exécution d'URL de type javascript:uneFonction()	***	Issa	10%
21	Debugging	Reprendre les bugs rapportés en Bêta-test	**	Grégory	40%
22	Code	Être cohérent sur les accès (private, , protected, public)	**	Yoann	100%
23	UML	Schématiser nos spécification & conception	**	Issa, Yosra	50%
19 & 20 Février 2009					
24	Debugging	Corriger les bugs	**	Grégory, Yoann	50%
25	SWT	Remplacer le Thread.sleep() par un bouton de passage à la page suivante	**	Grégory, Yoann	100%

Tableau 1: Table de répartition des tâches.

3.4 GANTT / Planning

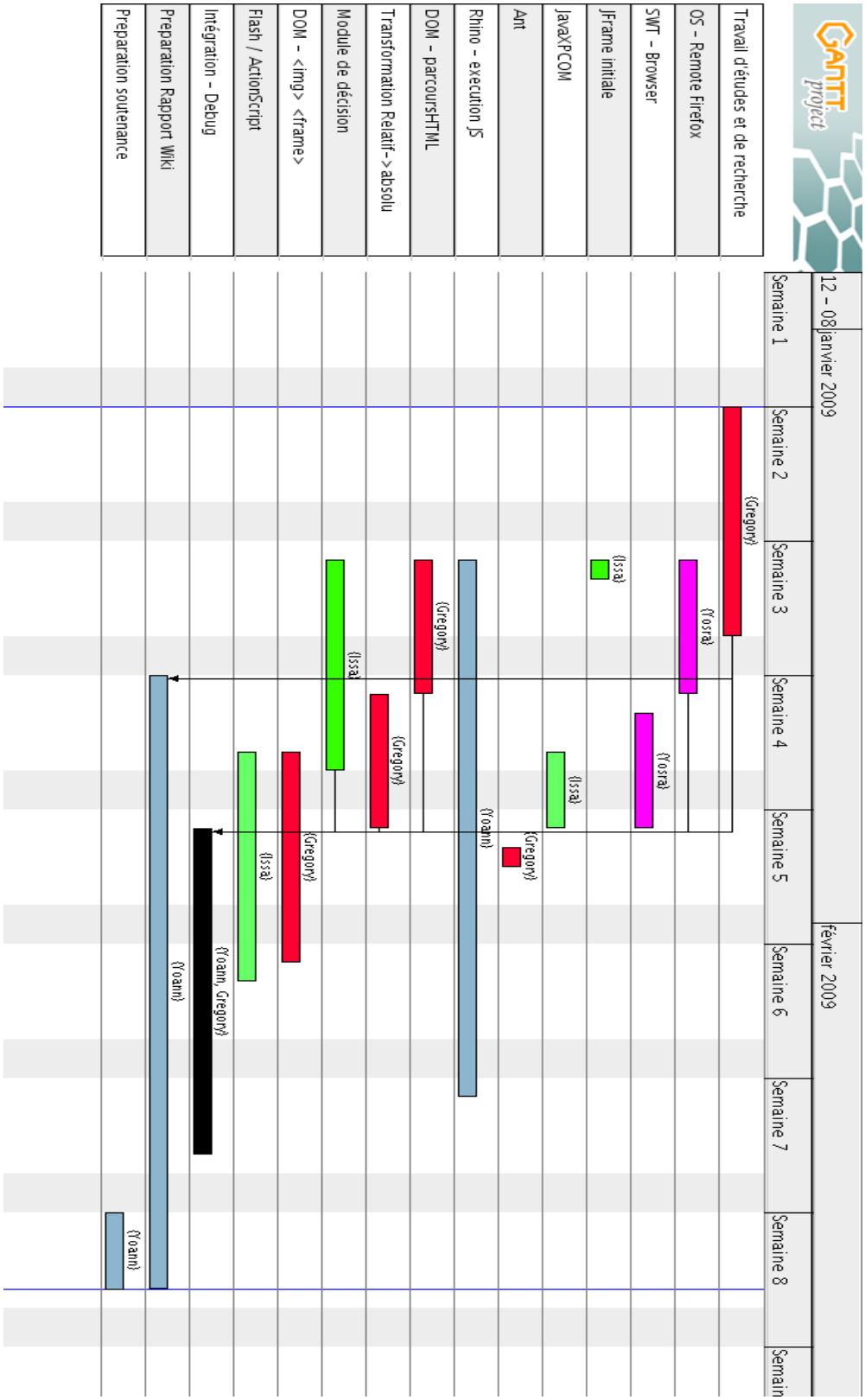

Illustration 7: Diagramme de Gantt.

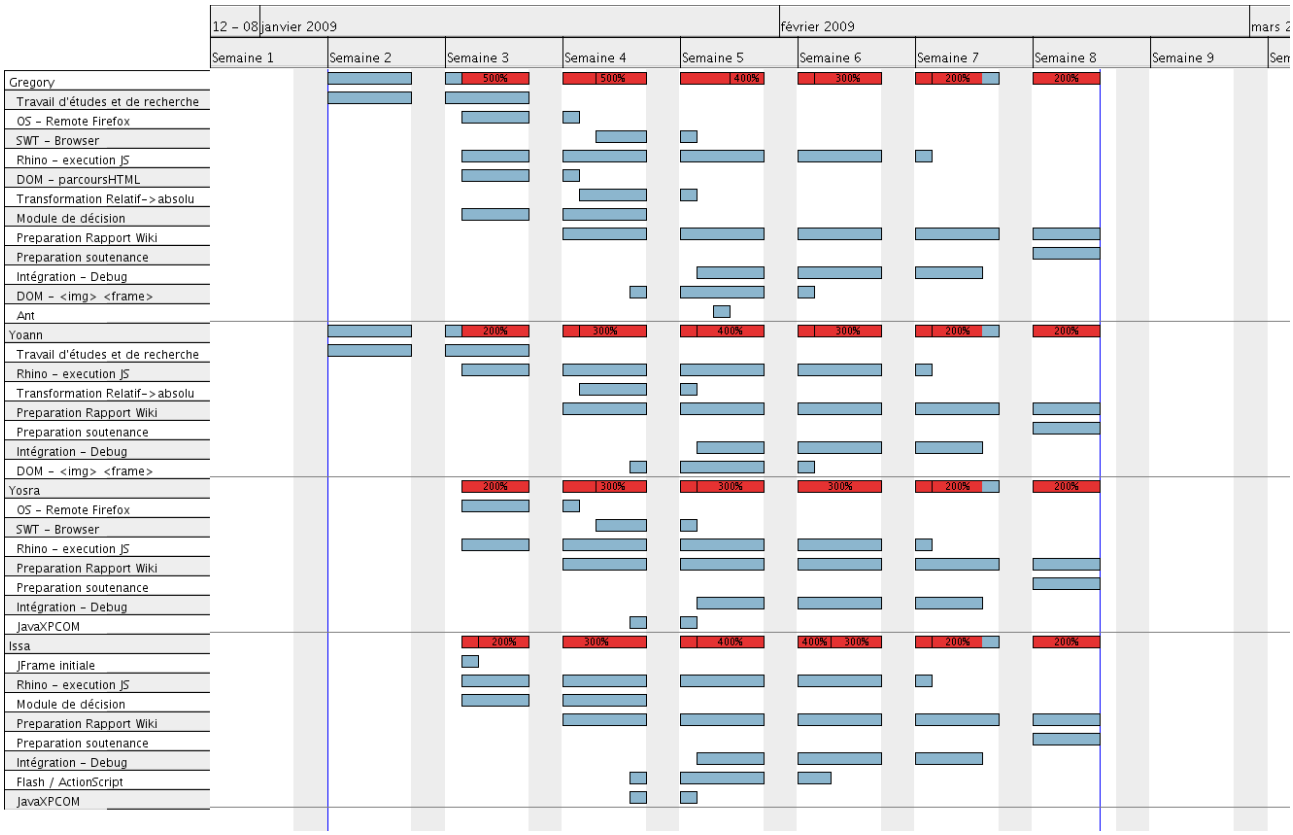

Illustration 8: Planning par ressources

3.5 Historique des versions, calendrier

(présentation du tableau au 10 Février 2009)

Version	code	Date de sortie	Nouveautés principales
RN 0.1	*Alpha*	13 Janvier 2009	- Premières analyses et spécifications - Un outil de documentation Wiki - Un module de décisions basique - Un script bash d'affichage de pages web sur mozilla firefox - Un prototype de parseur DOM pour balises a
RN 0.2	*Alpha*	22 Janvier 2009	- Un snippet SWT navigateur web - Un squelette de tests unitaires - Un squelette de gestion des logs - Un module de validation de documents DOM - Un module de contrôles de documents DOM (affichage)
RN 0.3	*Alpha*	30 Janvier 2009	- Un module de gestions d'URL relatives et absolues - Une première interface d'accueil
RN 0.4	*Alpha*	6 Février 2009	- Un module d'exécution de Javascript - Un module de récupération de code ActionScript - Un navigateur SWT adapté
RN 0.5	*Bêta*	13 Février 2009	- Un module de décomposition des frames et du HTML transitionnel - Un module de décisions complet - Un module de récupération de liens dans le code Flash - Les outils nécessaires à la génération de Javadoc et au déploiement du logiciel en exécutable.
RN 1.0	*Stable*	20 Février 2009	- Une documentation aboutie + Powerpoint - La correction des derniers bugs

Légende :

Version expirée	Version actuelle	Version en développement	Version planifiée

Tableau 2: Table de l'historique des versions de l'application.

3.6 Conventions sur la conduite du projet

- Indenter, Commenter son code dans l'optique javadoc (en français)
- Documenter ses parties, textuellement et schématiquement sur le wiki. (UML de rigueur lorsqu'il est possible) (en français)
- Donner des noms de variables et d'objets explicites même s'ils sont longs, suivant la norme fixée disponible en commentaire sur le fichier Main.java (en anglais)
- Ne jamais mettre d'accents, en français, même dans les commentaires.
- Utiliser les logs pour le suivi des exécutions, et junit pour les tests unitaires (une classe de test par package, un test unitaire par procédure).
- Respecter les deadlines du planning qui sont très largement respectables.

4. Objectifs

4.1 Compétences requises

- Notre contrainte de délais nous a poussé à mettre en place une gestion de projet. Nous avons tout de même cherché à ce que les tâches s'effectuent le plus spontanément possible. Pour cela le site communautaire que nous avons utilisé nous a été d'une grande utilité : nous avons utilisé un portail internet dans le cadre de notre gestion de projet, chacun pouvait y trouver des tâches, suivre l'avancement, y faire part de ses difficultés.
- Le site a été développé en Java, le volume de code est assez lourd, surtout que les librairies utilisées (parfois recompilées) sont nombreuses. Vu le volume de ce code, il a donc été utile dès le début d'établir des classes de tests unitaires, permettant en même temps les tests de non-régression. Il a de plus été utile de commenter le code afin que chacun puisse utiliser les parties qu'il trouvera nécessaire.
- Nos notions de spécification et conception ont aussi été nécessaires à la modélisation de ce système plutôt complexe.

4.2 Principales difficultés

4.2.1 Portabilité

- Les programmes développés en Java sont logiquement portables sur tous types de station. Certaines librairies sont cependant dépendantes de la plateforme, c'est le cas de SWT.

Leur comportement peut avoir de petites différences entre l'un ou l'autre système d'exploitation, certaines interfaces n'existent même pas sur l'un ou l'autre, ainsi dans le code il a été nécessaire de tester le type de plateforme avant de recourir à ces méthodes. Mais finalement le navigateur SWT est bien utilisable que ce soit sur Linux ou Windows.

- En revanche le navigateur Firefox fonctionne en remote uniquement par commandes, la récupération d'instances dans le navigateur en tant que programme installé indépendamment sur le système d'exploitation est impossible. Ces commandes sont actionnées par des scripts shell (bash pour Linux, MacOS et Solaris, batch pour Windows).
- Une autre de nos librairies : mozilla.interfaces (qui intègre les outils JavaXPCOM) doit dialoguer avec Xulrunner qui est cité dans la partie documentation de JavaXPCOM.

XULRunner est un logiciel libre qui sert comme environnement d'exécution d'applications XUL. Il permet donc de lancer des applications écrites en XUL sans avoir besoin d'installer Mozilla ou Firefox, les deux logiciels originaux interprétant le XUL. L'intérêt est de disposer d'une plate-forme multi-système d'exploitation, tournant aussi bien sous Windows, Mac OS ou Linux, et interprétant des programmes écrits sans précision de système d'exploitation, aptes grâce à XULRunner à fonctionner sur n'importe quel système supportant XULRunner. Si les programmes écrits en XUL n'ont pas besoins d'être adapté à un système d'exploitation particulier, **les versions de XULRunner doivent par contre être adaptées à chaque système d'exploitation**. XULRunner inclut le moteur de rendu Gecko ainsi que la majeure partie des API de Firefox (entrées/sorties, communication réseau, toolkit XUL, gestionnaire de thèmes et d'extensions, localisation, manipulation de fichiers XML, RDF etc.).

4.2.2 Traduction d'URL relatives en URL absolues

Quelque soit l'origine du code HTML (produit directement ou à partir de JavaScript ou PHP, ASP), les liens présents sur la page peuvent être fournis en URLs relatives. Ceci nous pose un problème dans une navigation par onglets (avec le navigateur Firefox ou le navigateur SWT) : puisque l'onglet est vide au départ, il faut traduire l'URL relative en absolue pour pouvoir lancer la page sans qu'il n'y ait d'environnement préexistant.

Une autre solution aurait pu être de remplir l'onglet par le contenu précédent, puis d'appliquer sur cet onglet l'URL relative. C'est cette méthode qui est utilisée lorsque sont actionnées les procédures javascript. Mais, cette méthode provoque un mauvais résultat à l'affichage (pas assez de rafraîchissements). De plus, si nous voulions conserver l'utilisation du navigateur Firefox, il nous fallait tout de même mettre en place cette méthode de résolution d'adresses relatives.

Cette traduction s'opère de la manière suivante :

-d'abord il faut savoir si le dernier élément de l'URL (dernier slash) est un nom de fichier ou de répertoire.

-ensuite il faut savoir si ce nom de fichier a son extension d'indiquée : (index ≠ index.html)

-ensuite il suffit de recouper les arborescences pour construire l'adresse souhaitée.

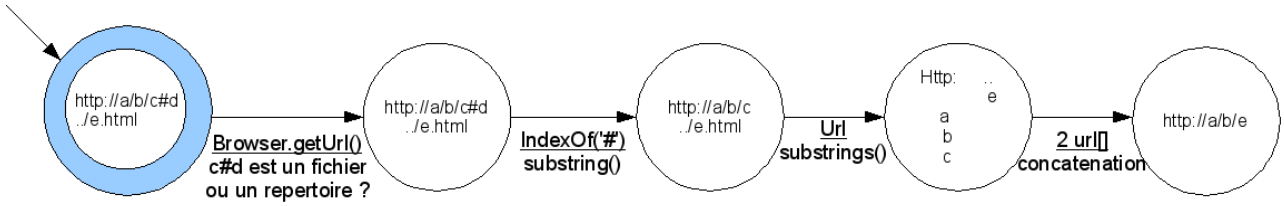

Illustration 9: Automate de l'algorithme de traduction d'URL.

5. Spécification

5.1 Modélisation de l'environnement

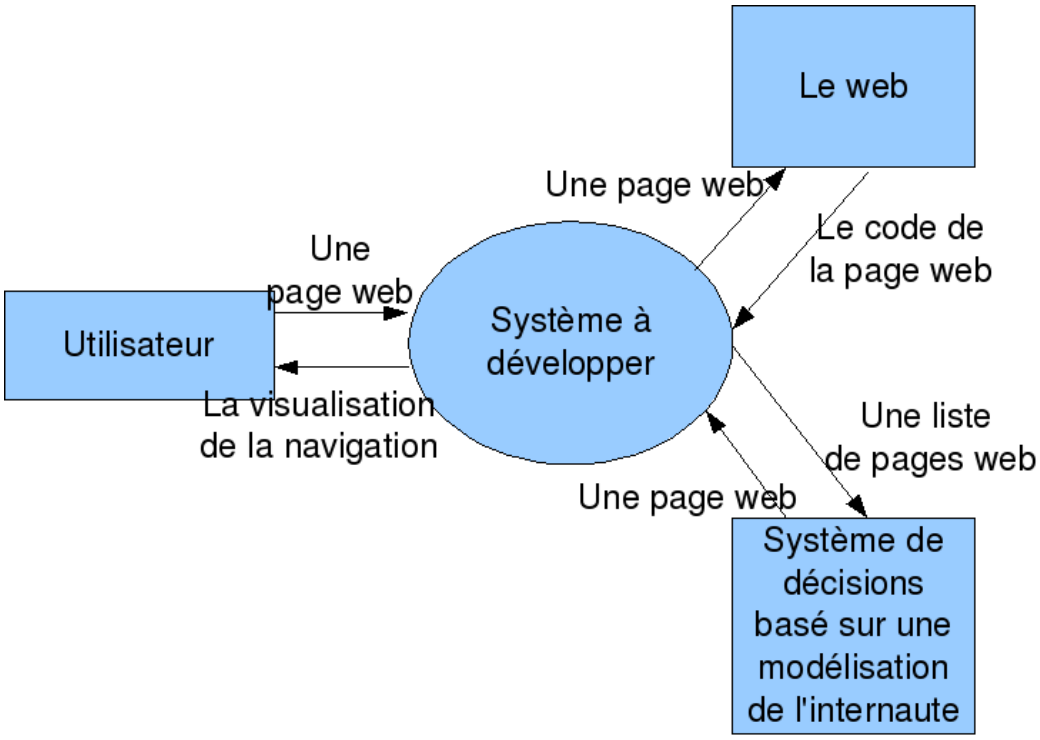

Illustration 10: Diagramme de modélisation de l'environnement de l'application

5.2 Vue en couches

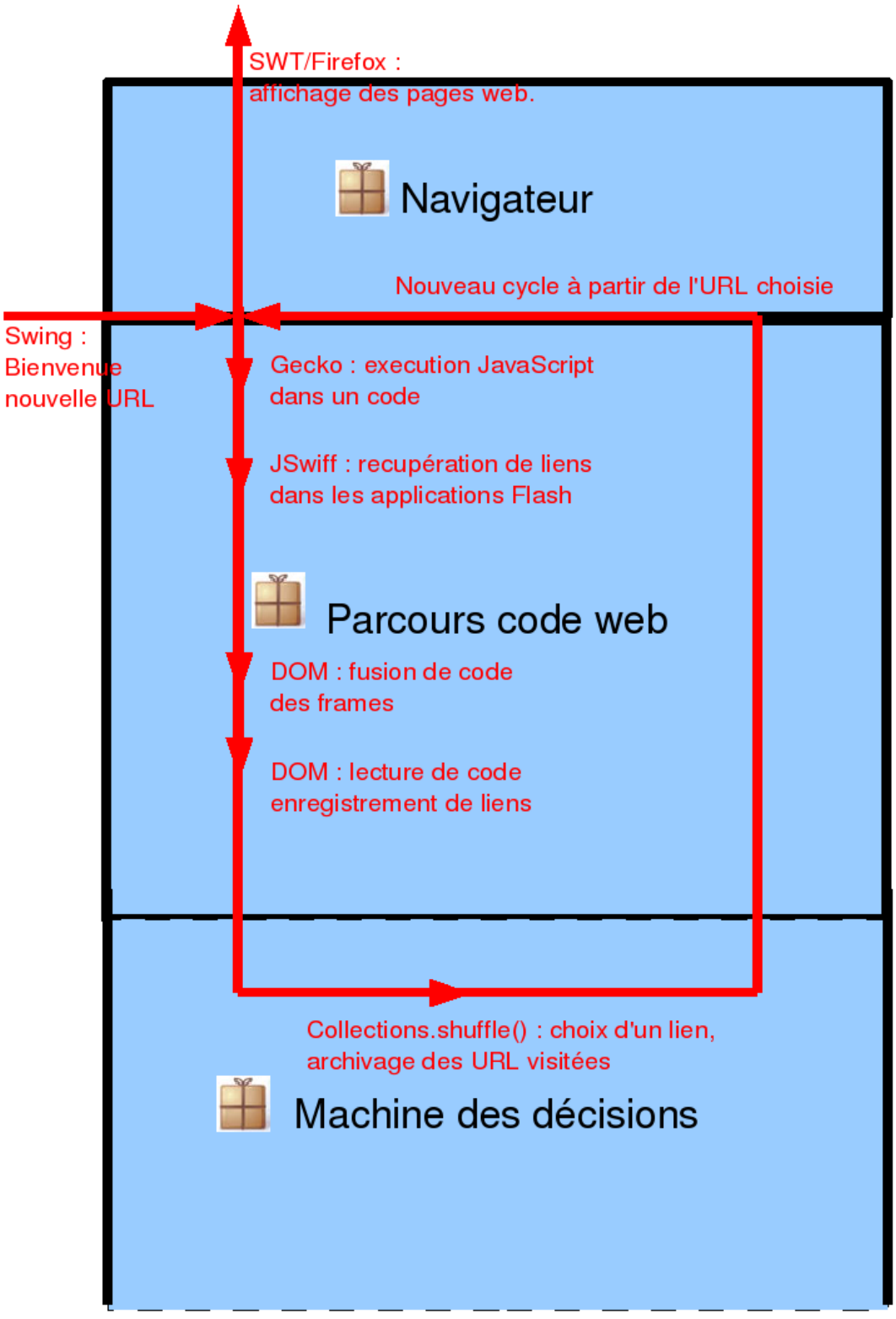

Illustration 11: Vue en couche du projet.

5.3 Les différentes structures dans lesquelles retrouver des liens hypertexte

5.3.1 Liens apparents dans le code

Pour la recherche de liens nous créons un document DOM pour récupérer les valeurs contenues dans les balises.

5.3.1.1 Balises a

Qu'ils soient présents de façon statique sur le code HTML ou générés depuis le serveur ou le navigateur, les liens hypertextes les plus usuels sont ceux présents dans les balises <a>. Il existe deux sortes de liens hypertexte pour les balises a :

- les ancres : ceux qui mènent à un élément différent sur la même page, on peut y trouver des chemins relatifs au sein du même site (ex :du texte)
- les liens hypertexte web : ceux qui conduisent à une autre page web (ex :du texte)

L'url vers lesquels mènent ces liens se trouvent toujours dans l'attribut "href" de cette balise.

5.3.1.2 Images mappées

On peut trouver sur certaines pages web des images dont seulement certains morceaux constituent un lien. Ce sont des images découpées selon leurs coordonnées : décrites dans le code HTML de la façon suivante :

```
     <map name="Map">
          <area shape="rect" coords="56,23,166,82"
href="http://www.exemple.com">
          <area shape="circle" coords="35,22,13" href="#">
          <area shape="poly" coords="57,101,66,100,57,84,65,96,68,101" href="#">
          ...
     </map>
```

5.3.1.3 Attributs 'action' d'une balise form

Cette balise est utile à l'exécution d'un script, produisant du code coté serveur, lorsque un formulaire est rempli. Il ne parait pas logique qu'un internaute automatique ait à remplir des formulaires : si on souhaite considérer ces liens d'enregistrement de formulaire, il faudra en même temps intégrer au robot un outil qui remplit les formulaires automatiquement avant de les valider. Cet outil est abandonné.

5.3.1.4 Balise 'input' de type Submit

Ces balises peuvent réclamer dans leur attribut onClick le lancement d'une procédure JavaScript, il est possible que cette procédure change l'aspect de la page web.

5.3.2 Liens à recomposer par un processus d'interprétation de langage

5.3.2.1 Code exécuté JavaScript

5.3.2.1.1 Problématique

Actuellement pour la recherche de liens nous récupérons le contenu brut de la page, le problème est que tout ce qui est généré dynamiquement est totalement ignoré. Qui plus est le Javascript devient au fil du temps de plus en plus utilisé notamment grâce à l'apparition d'Ajax. C'est pour cela que nous avons décidé d'exécuter le Javascript contenu dans la page web afin de disposer du plus de liens possibles.

5.3.2.1.2 Solutions examinées

- Moteur de script Rhino :

Rhino est un moteur JavaScript libre. Il est développé entièrement en Java par la Fondation Mozilla.Rhino, il peut être utilisé seul comme interpréteur, mais aussi intégré dans une application. De même, il est capable de fonctionner en mode interprété, tout comme en mode compilé. Il ne fournit en principe que les objets natifs définis par la spécification du langage (exemples : String, Date, etc.). N'étant lié à aucun document, on n'y trouvera donc pas les objets habituellement exposés par les moteurs JavaScript des différents navigateurs (exemples : window, document, alert(), getElementById...), ce qui rend Rhino inenvisageable pour notre utilisation car la plupart du temps les liens sont générés grâce a document.write() ou document.getElementById().innerHTML par exemple. De plus, lors de la rencontre de l'une de ces instructions non supportées, Rhino lève une exception ce qui rend impossible d'analyser le code javascript.

- La classe JSObject

La deuxième solution a était de faire communiquer une applet Java avec JavaScript afin de pouvoir lier le code javascript à une page web. La communication entre JavaScript et une applet Java est possible grâce à LiveConnect, qui est automatiquement fournit à partir de Netscape Navigator 3.0 et activé par la double activation de Java et JavaScript dans la configuration du navigateur. Une applet ne peut accéder à JavaScript et les objets que si elle importe le package netscape.javascript qui définit les classes JSObject et JSException. De l'autre côté, le document HTML intégrant cette applet doit autoriser celle-ci à communiquer avec JavaScript grâce au paramètre mayscript du tag <applet>. Par exemple :

```
<applet code="monApplet.class" width=100 height=100 mayscript></applet>
```

La classe Java JSObject permet alors d'accéder aux objets JavaScript de la page, à commencer par l'objet window, qui est récupérable à l'aide de la fonction getWindow() :

```
JSObject win = JSObject.getWindow (this);
```

Les méthodes JavaScript, qu'elles soient utilisateur ou système, peuvent alors être appelées de deux manières différentes:

- Soit via la méthode call() de la classe JSObject, en passant en paramètres le nom de la méthode et un tableau de ses arguments (ne pouvant pas être null mais pouvant contenir une chaîne vide) :

```
String arguments[] = {"premierArg", "secondArg"};
win.call ("nomMethode", arguments);
```

- Soit via la méthode eval() de cette classe, qui reçoit en paramètre une chaîne JavaScript à exécuter. Par exemple :

```
win.eval ("nomMethode(" + premierArg + "," + secondArg + ")");
```

Pour mettre cette solution en place, nous avions pensé à exécuter l'applet contenant le Javascript de la page dans le navigateur SWT (grâce à XulRunner), mais malheureusement ce navigateur ne supporte pas les applets ce qui nous a obligé à rechercher de nouvelles solutions.

- JavaXPCOM

C'est cette solution que nous avons finalement mis en place et qui nous a permis de récupérer le code de la page exécutée. JavaXPCOM permet la communication entre Java et XPCOM de telle façon qu'une application Java puisse accéder à des objets XPCOM, et que XPCOM puisse accéder à des classes Java qui implémentent une interface XPCOM. Avec JavaXPCOM, notre application peut dialoguer avec Gecko (le moteur de rendu de Mozilla) présent dans XULRunner et ainsi récupérer le code exécuté :

```
nsIWebBrowser webBrowser = (nsIWebBrowser)browser.getWebBrowser();
nsIDOMWindow window = webBrowser.getContentDOMWindow();
nsIDOMDocument document = window.getDocument();
```

Attention : Ce morceau de code ne fonctionne pas sur Linux :

ce code dialogue avec le programme Xulrunner dont les versions Linux et Windows sont différentes et sur Linux webBrowser.getContentDOMWindow(); renvoie null et provoque donc une NullPointerException. Lors de l'utilisation du programme sur Linux, on récupère donc le code d'une manière plus classique, ce code contenant les balises <script> non remplacées.

Sur Windows, il arrive aussi que l'on obtienne un comportement inattendu :

si le code javascript doit disposer de variables passées en GET ou en POST qui ne sont pas présentes, alors Xulrunner va renvoyer un DOMDocument vide où l'on ne trouvera aucun lien. C'est pourquoi nous avons dû compter les liens de chaque solution (JS et classique) et garder le maximum.

5.3.2.1.3 Distinction avec les liens de type javascript:uneFonction()

Il existe deux problèmes distincts avec le JavaScript : l'exécution des balises <script> d'un côté, et le changement de contenu d'une page web en exécutant une commande javascript:uneFonction() depuis un bouton ou un lien par exemple.

Cet autre problème n'a bien sûr pas été possible sur Linux non plus puisque si on change le contenu d'une page sans changer l'url, le seul moyen de récupérer ce contenu est d'utiliser la méthode JavaXPCOM.

5.3.2.2 Balises frame / iframe

Grâce à la technologie des frames (en français "cadres"), il est désormais possible d'afficher plusieurs pages HTML dans différentes zones (ou cadres). Les frames sont un processus d'inclusion de code HTML entre fichiers. Par ce processus, des contenus web sont donc affichés sans que le code soit directement accessible. Des liens peuvent se trouver dans ces cadres inclus, il a donc fallu

récupérer le code inséré afin de le parcourir à son tour et y trouver des liens.

La première possibilité est l'inclusion de frames de type block :

la page est constituée exclusivement d'inclusions, le code de la page contient :

```
<FRAMESET ROWS="20%,80%">
<FRAME SRC="frame1.htm" NAME="haut">
<FRAME SRC="frame2.htm" NAME="bas">
</FRAMESET>
```

Une seconde possibilité est l'inclusion inline de cadres (balise iframe) :

```
<IFRAME src="http://perso.wanadoo.fr/bernard.quevillier/toposnew/fondex03.htm"
width=600 height=200
 scrolling=auto frameborder=1 > </IFRAME>
```

Dans les deux cas, les frames peuvent elles-mêmes contenir des frames supplémentaires, il a donc dû être conçu un processus itératif d'inclusion du code dans le document original à partir de la source.

- Il faut récupérer la source
- L'inclure dans le document DOM original
- Supprimer la balise frame correspondante
- S'il y a encore des frames dans le code on reprend le processus.

5.3.2.3 Flash/ code ActionScript

Dans le travail de recherche, il a été établi que les liens hypertextes peuvent être présentes de différentes manières sur une page web. c'est à dire à partir de balise <a> ou à partir de JavaScript exécuté, ou dissimulés derrière des objets de types flash(swf qui sont sous formes de boutons ou sous formes de liens). Ici on s'intéresse à la manière dont on pourrait récupérer ces liens pour le cas du flash.

On s'est rendu compte que les objets de types flash sont appelés sur une page web à partir de balises '''**<object><embed>**'''.A l'intérieur de la balise '"<embed>"' on donne le lien vers la source de l'objet flash sous la forme d'une "url". Le travail étant donc de savoir une fois l'objet flash chargé, sous quelle forme les liens hypertextes étaient dissimulés avec l'objet flash.

La première approche est plutôt directe, c'est à dire que le lien est directement accéder à partir de balise <a> entre ces balises on met notre objet de type '''**swf**'''.

'''ex:'''

```
   <a href="http://www.google.com"><object classid="......>
   <param name="movie" value="/uploads/flash/preloader_basic.swf">
   <param name="quality" value="high">
   <param name="wmode" value="transparent">
   <param name="menu" value="false">
     <embed src="/uploads/flash/preloader_basic.swf" width="160px"
height="600px">
</embed>
  </object></a>
```

Cette méthode aurait était la plus simple pour nous car elle permettait de récupérer le lien à partir de

notre classe "'**CodeParser'**". Cependant, il apparaît que l'on ne peut pas utiliser cette méthode car obsolète et ne permettant pas d'accéder aux liens.

A partir de la, il nous fallait trouver une méthode pou récupérer les liens qui sont directement dissimulés derrière l'objet flash. En général dans le codage d'un objet flash,ces liens sont appelés grâce à la méthode "'**getURL("lien")'**" et souvent avant un appel d'action de type "'**on(release)'**" ou "'**on(press)'**".

voir ex:

// « on press »

on(press) getURL("http://www.toto.fr");

après recherche on a pu trouver deux « packages » qui permettent de parser ou manipuler des objets de type flash(swf).

5.3.2.3.1 JSwiff

L'objectif du projet Jswiff est de créer un framework(open source) pour la manipulation et la création de fichiers Flash qui supporte toutes les versions de flash existante.

Avec cet outil on peut :

-générer des objets de type flash.

-modifier des objets flash existant en utilisant l'outil de « parsing ».

-générer du XML à partir d'un objet Flash ou l'inverse.

La liste des packages présentes dans Jswiff est décrite ci-dessous.

« p »

Pour la partie qui nous intéresse lorsque l'on manipule un objet flash avec "'**Jswiff'**", les classes présentes dans cette librairie permettent de "parser" le code de l'objet flash et de représenter chaque élément du code par une variable ou une classe. Cela concerne les actions, les images, du Javascript ou du texte...

Ainsi dans le package «"'**com.jswiff.swfrecords'**" » il est répertorié sous forme de classes, tous les éléments susceptibles d'être rencontrés dans un objet flash.

Notre démarche a été de créer une classe"' **FLASHParser'**" qui fait appel à la classe "'**Transformer'**" présent dans le package « "'**com.jswiff.xml'**" ».

La classe "'**Transformer'**" possède une méthode "parseSWFDocument(InputStream)" qui permet de parcourir le document flash. Dans le champ "InputStream" nous mettons le chemin d'accès de notre flash.

Lorsque le document est parcouru, chaque action trouvé est ajouté à la classe correspondante par une méthode "add".

D'après la structure de "'**Jswiff'**", les actions de type "'**getURL'**" qui nous intéressent sont représentées sous forme de constantes ActionConstants."GETURL »".Et c'est dans la class "'**ActionBlock**" du package "'com.jswiff.swfrecords.actions'" que l'on a introduit un test pour les trouver.

Donc nous avons recompiler cette classe en lui rajoutant un test sur le type de " *ActionConstants"* rencontré et une nouvelle classe "'**FlashParserTools'**" qui permet de définir une liste à laquelle on

ajoute les liens à chaque fois qu'on les rencontre durant le test.

Ainsi nous allons disposer d'une liste à la fin du parcours contenant tous les liens dans le code de l'objet flash.

Lien utile: http://www.jswiff.com/index.jsp

5.3.2.3.2 Javaswf2

Javaswf2 met à disposition des méthodes pour générer et parcourir des contenues Flash4 Flash5.

'"Son Architecture"': Le code est basé principalement sur les interfaces qui sont définies dans le package « '"**com.anotherbigidea.flash.interfaces"'** » . Ces interfaces sont utilisées pour passer différentes informations sur le contenue du fichier flash swf. Le "parser" ou le "modifier" de swf implémente ces interfaces et contrôle les autres classes qui les implémente.

Il y a deux niveaux avec JavaSWF2:

"le format fichier":le niveau "format fichier" procure un accès à tous les détails sur le format fichier de swf (pour le parsing comme pour l'écriture) et est utile si vous avez des informations sur le format du fichier et que vous voulez implémenter rapidement un code de parsing ou de génération.

"Le modèle objet": Le niveau "modèle objet" est implémenté par les classes dans le package «'" **com.anotherbigidea.flash.movie"'** » et procure des moyens pour accéder et créer des contenus SWF sans connaître de détails sur le format du fichier.

Liens utiles:

http://www.anotherbigidea.com/javaswf/samples/DumpSWF.java

http://www.anotherbigidea.com/javaswf/samples/ExtractText.java

5.4 Diagramme de cas d'utilisation

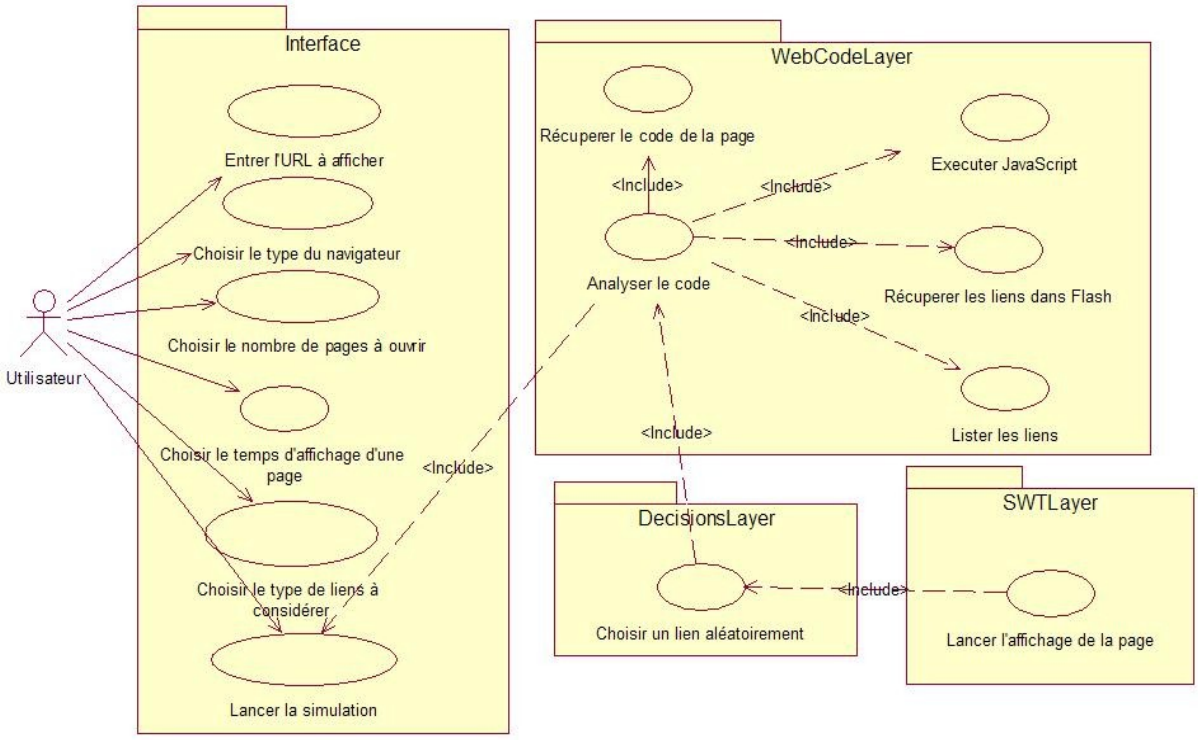

Illustration 12: Diagramme de cas d'utilisation

Ce diagramme de cas d'utilisation montre les différentes interactions qui peuvent exister entre l'utilisateur du logiciel et l'interface de l'application dans un premier temps.

Puis dans un deuxième temps, les interactions qui existent entre les différentes classes des trois packages de l'application « WebCodeLayer » , « DecisionsLayer », « SWTLayer ».

6. Conception détaillée

6.1 Conception IHM

Cette première interface a été dessinée sur KDevelop, AGL open source équivalent de WinDev.

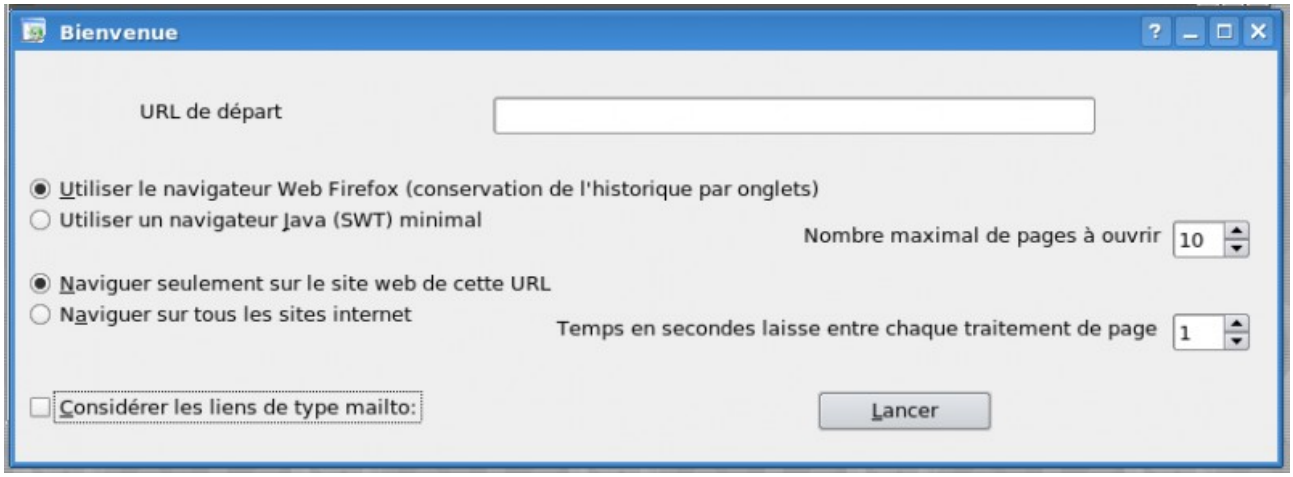

Illustration 13: Dessin d'interface d'accueil sur le programme.

6.2 Diagrammes de séquence

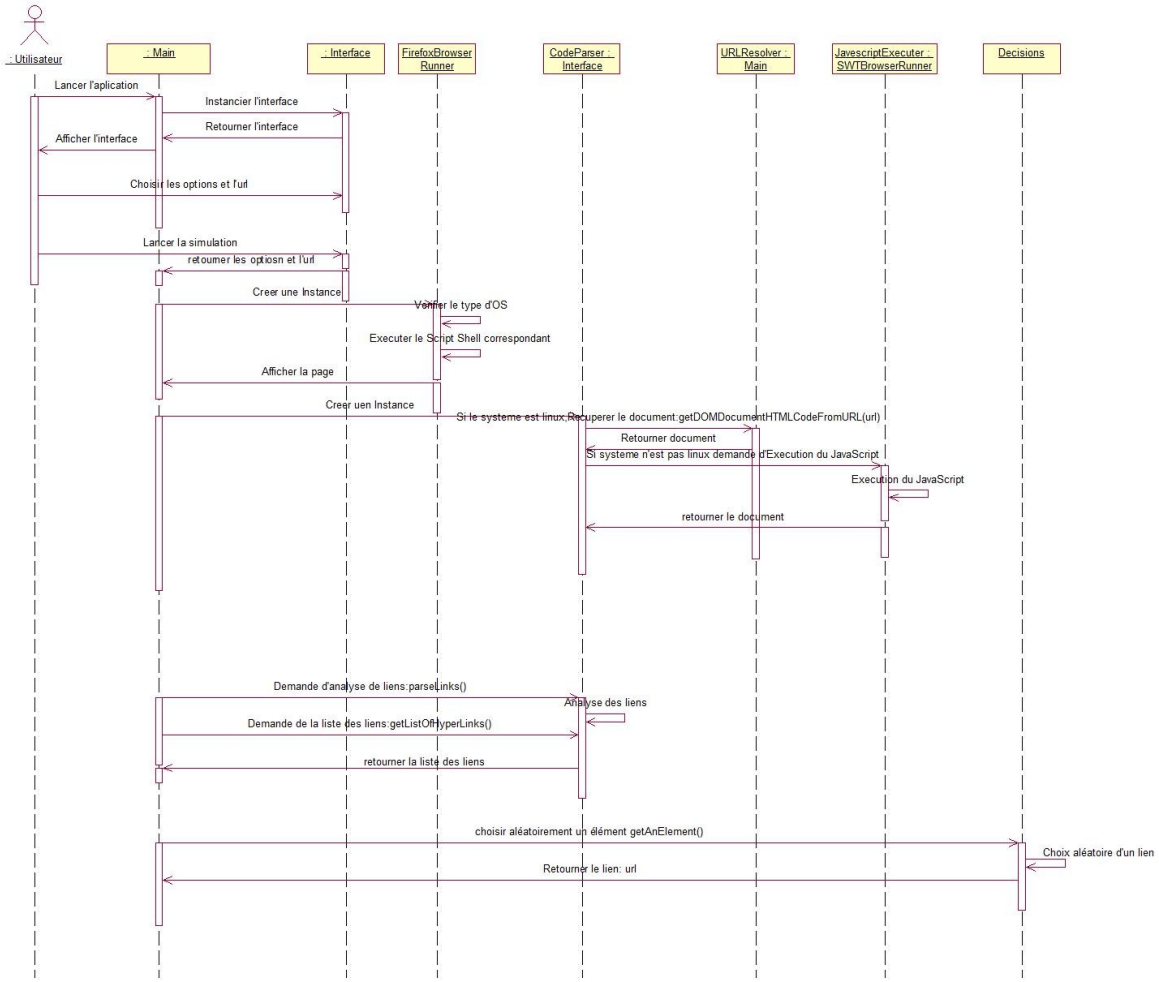

Illustration 14: Fonctionnement de l'application sur le mode Navigateur Firefox

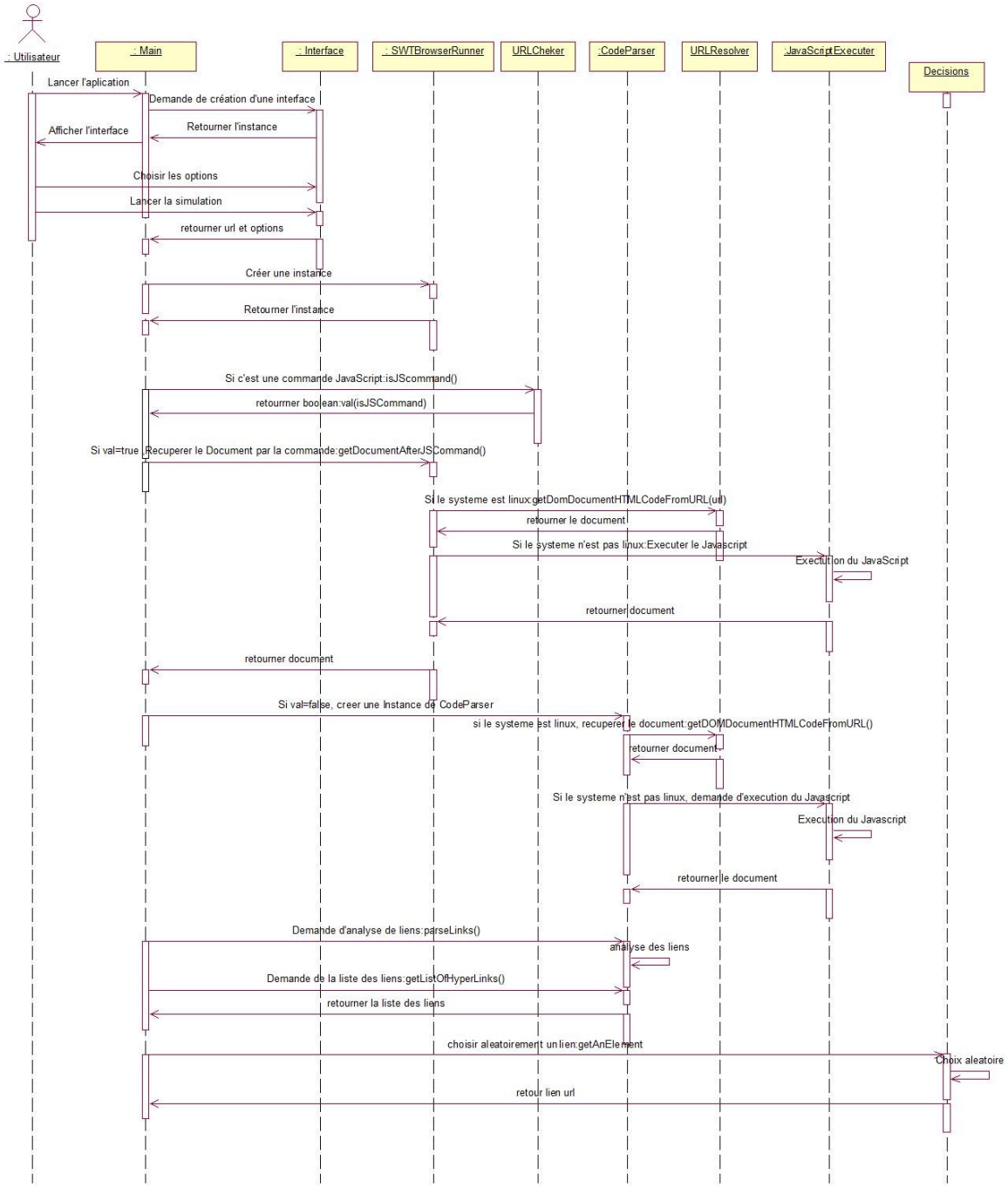

Illustration 15: Fonctionnement de l'application sur le mode navigateur SWT.

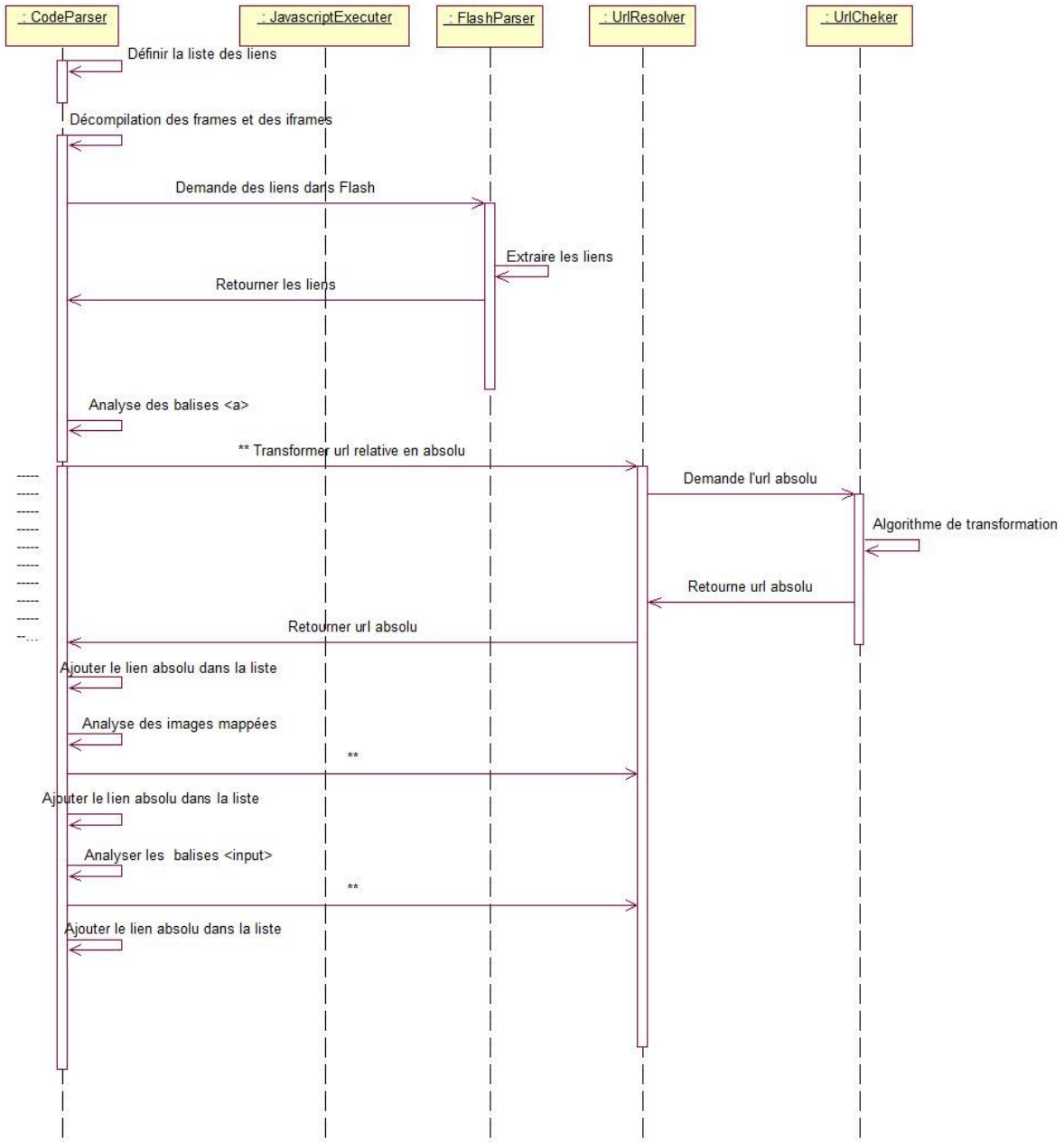

Illustration 16: Parcours du code et analyse de liens.

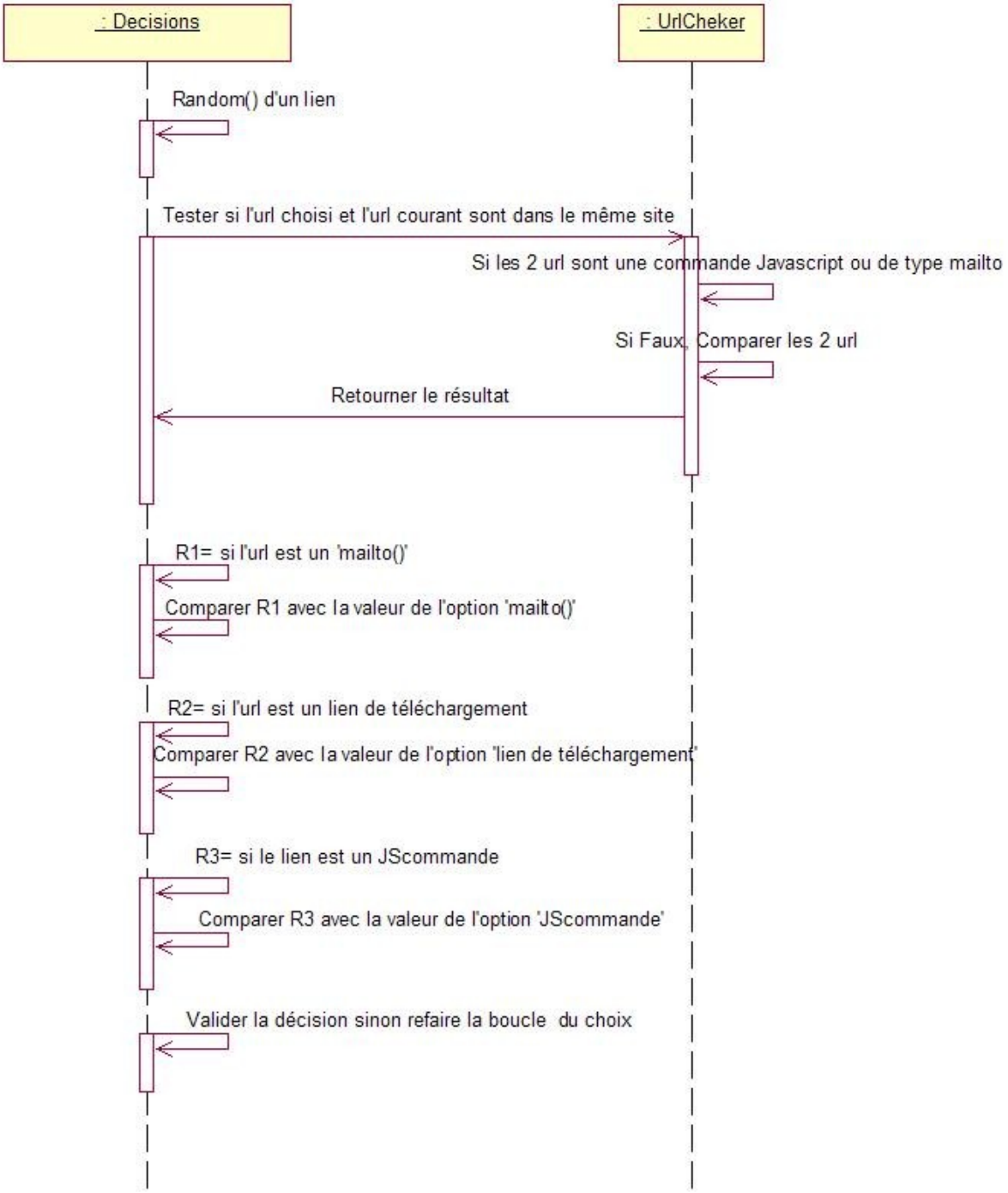

Illustration 17: Algorithme de décisions : choix de lien.

6.3 Diagramme d'activité

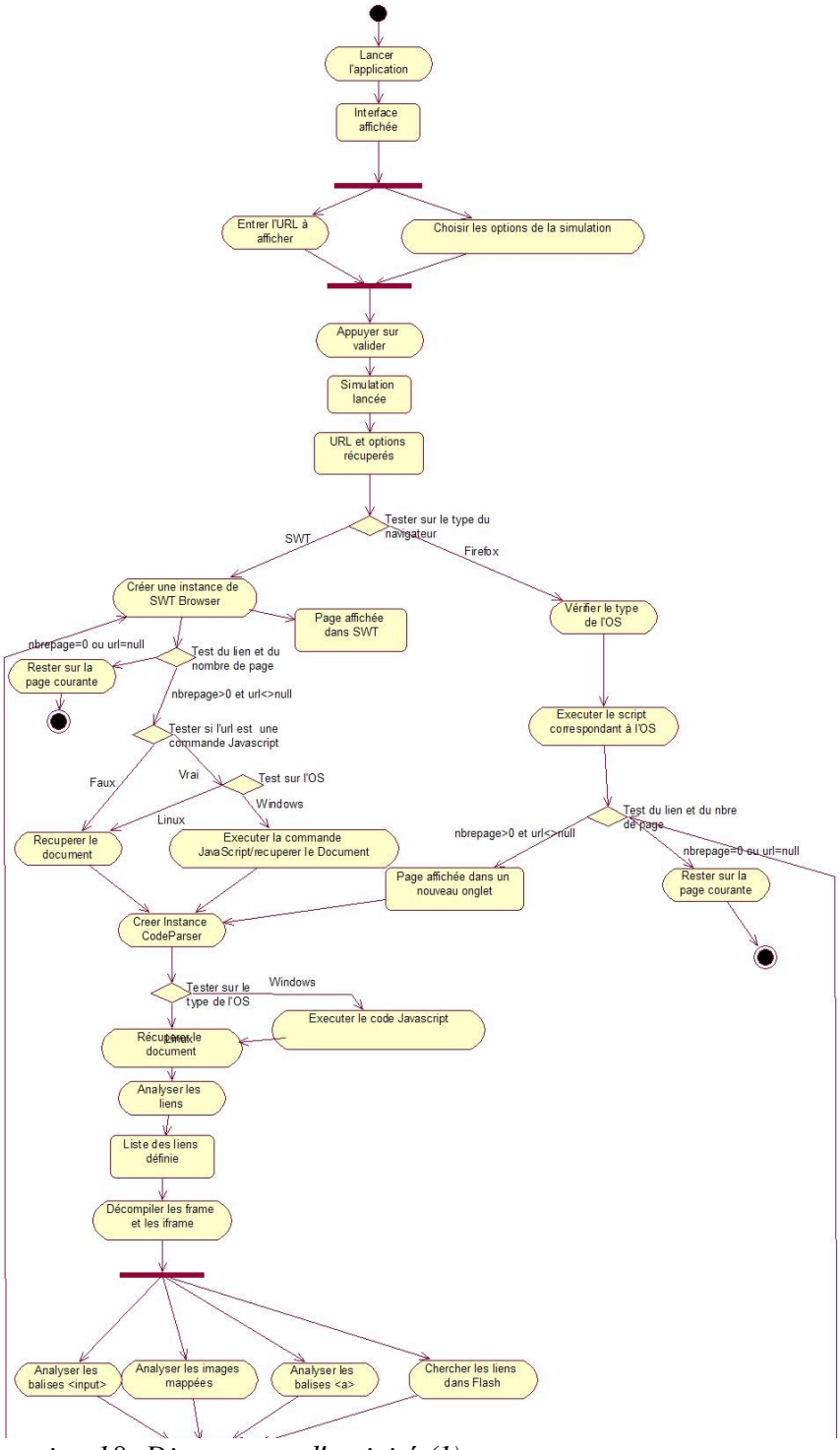

Illustration 18: Diagramme d'activité (1)

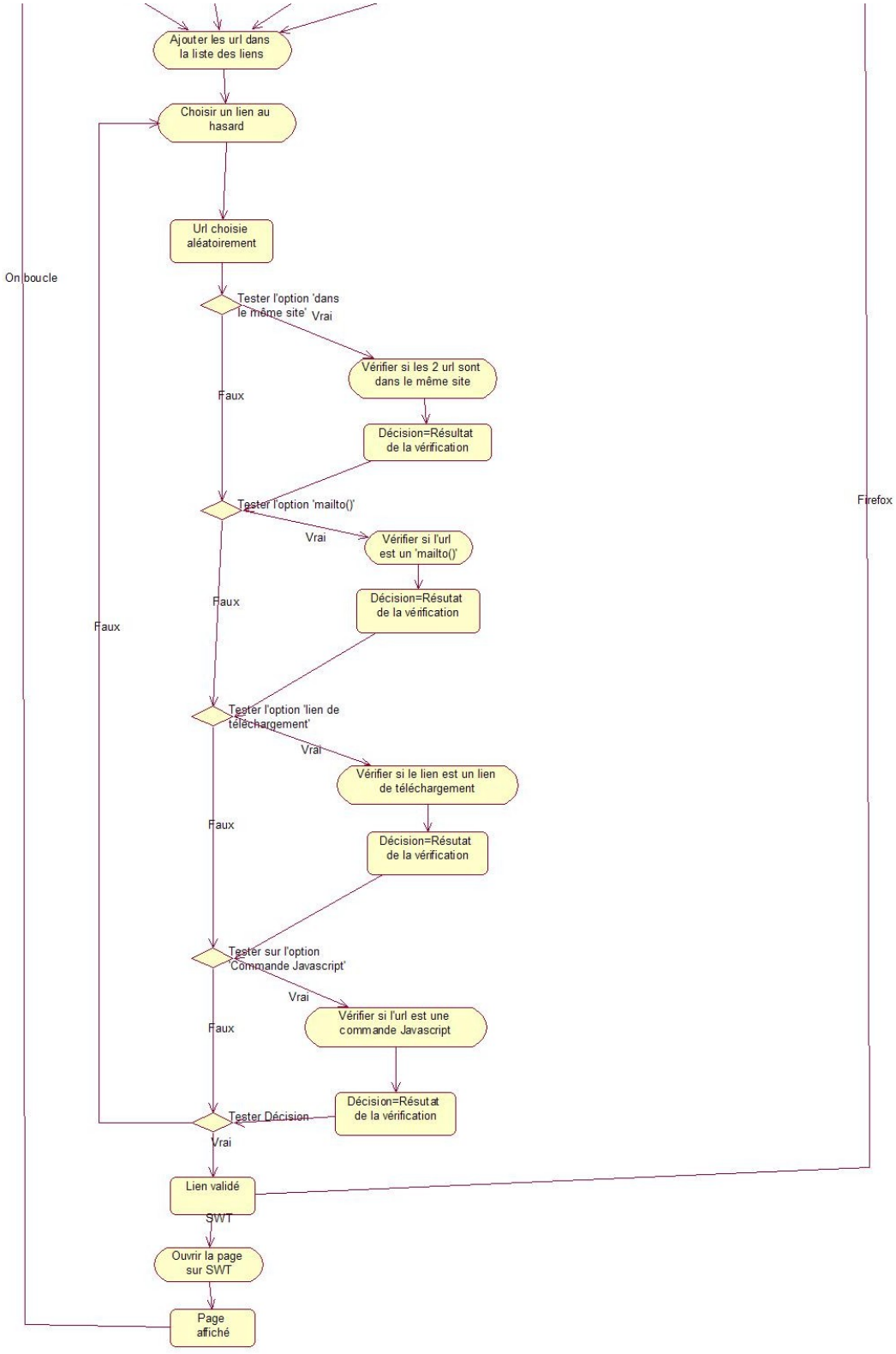

Illustration 19: Diagramme d'activité (2)

6.4Diagrammes de classes

6.4.1 Diagramme de Classe des packages.

Ce Diagramme montre les liens qui existent entre les différents packages du projet.

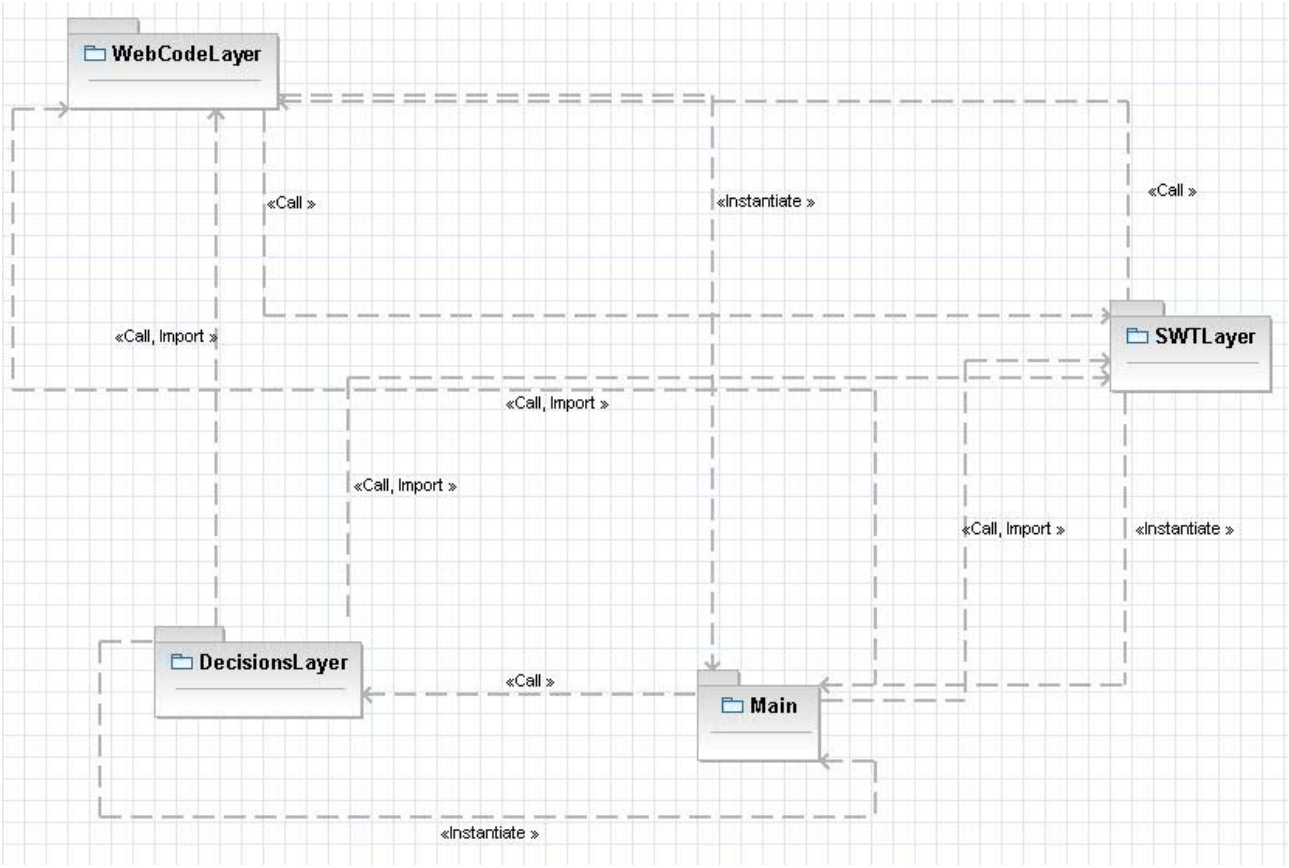

Illustration 20: Diagramme de classe des liens entre Packages

6.4.2 Diagramme de Classe Package DecisionsLayer

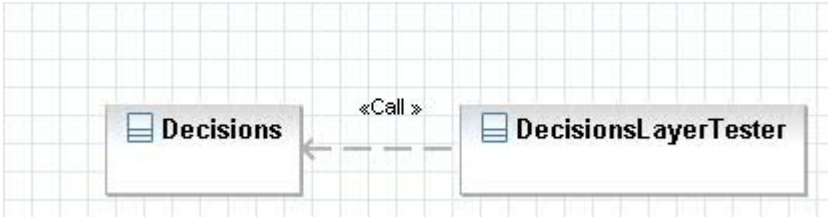

Illustration 21: Diagramme de classe du Package DecisionsLayer

6.4.3 Diagramme de Classe Package SWTLayer

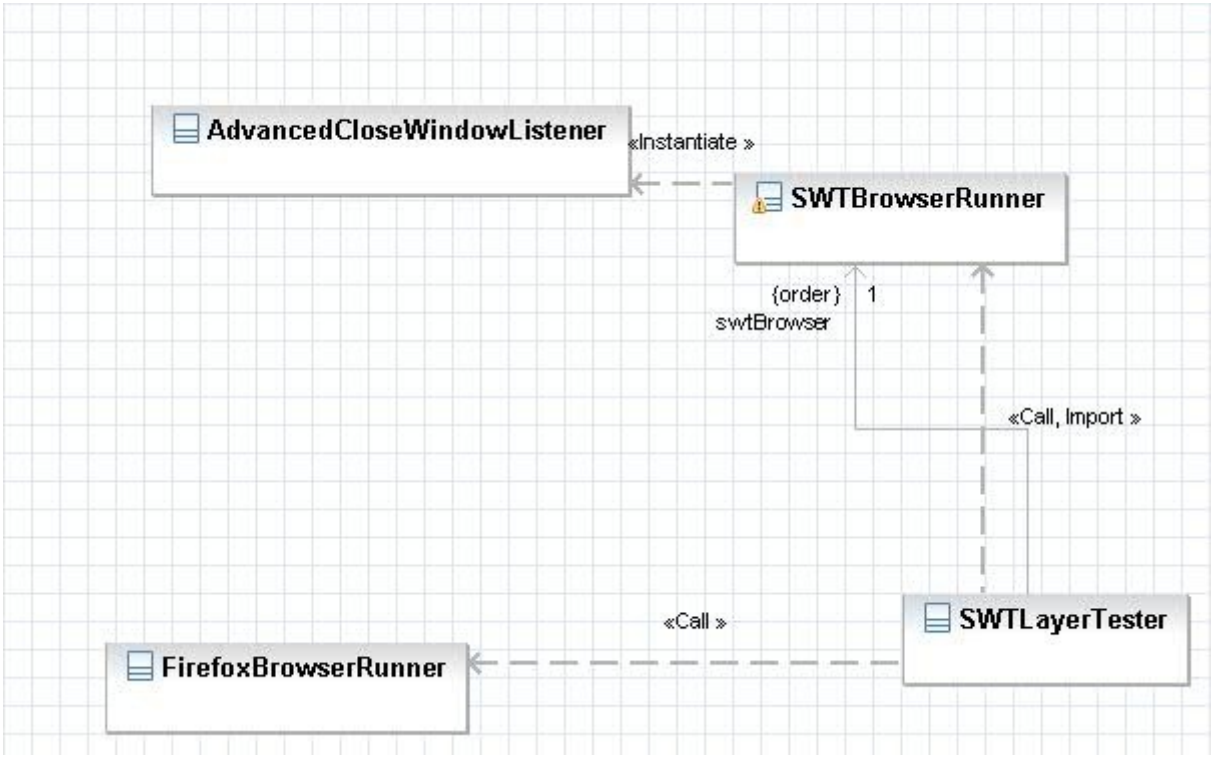

Illustration 22: Diagramme de classe du Package SWTLayer

6.4.4 Diagramme de Classe Package WebCodeLayer

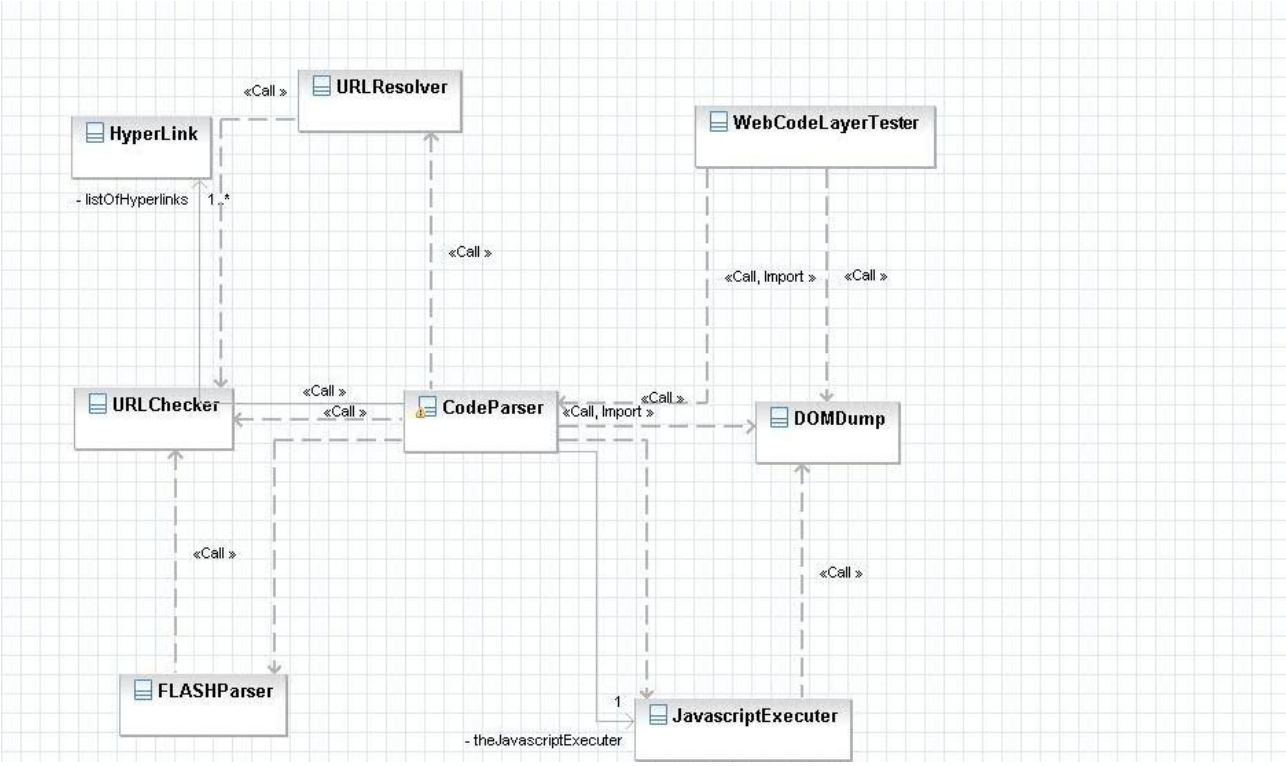

Illustration 23: Diagramme de classe du Package WebCodeLayer

6.4.5 Diagramme de Classe Package Main

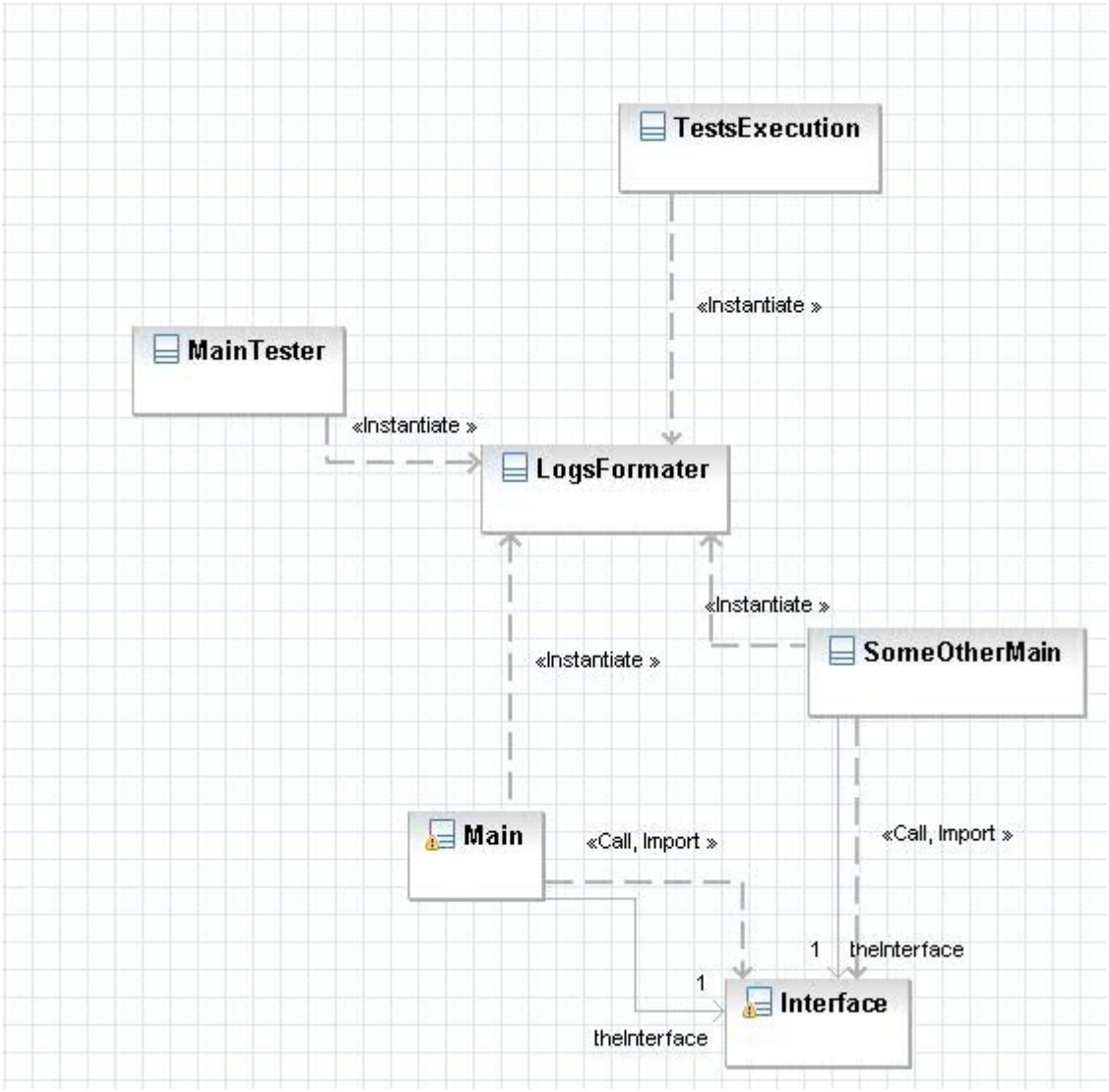

Illustration 24: Diagramme de classe du Package Main

6.4.6 Composants des différentes Classes Utilisées

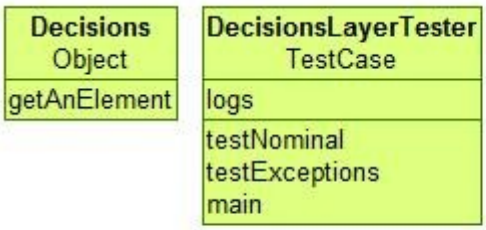

Illustration 25: Package DecisionsLayer

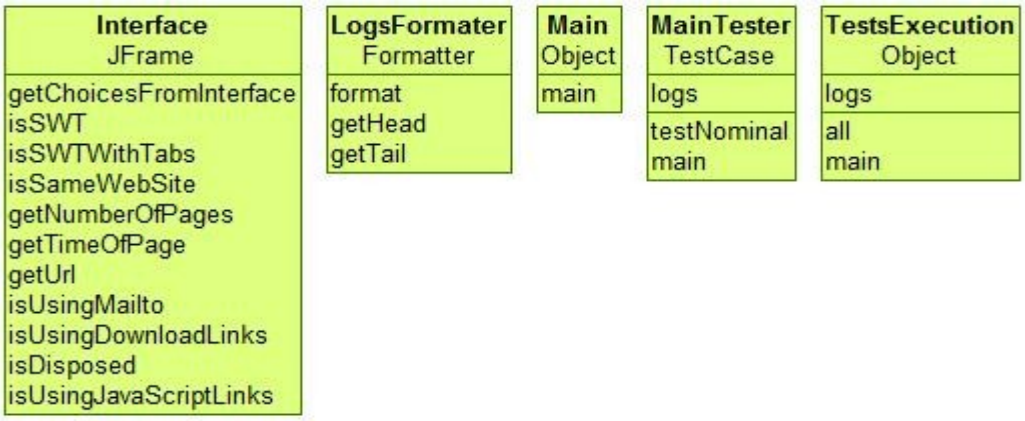

Illustration 26: Package Main

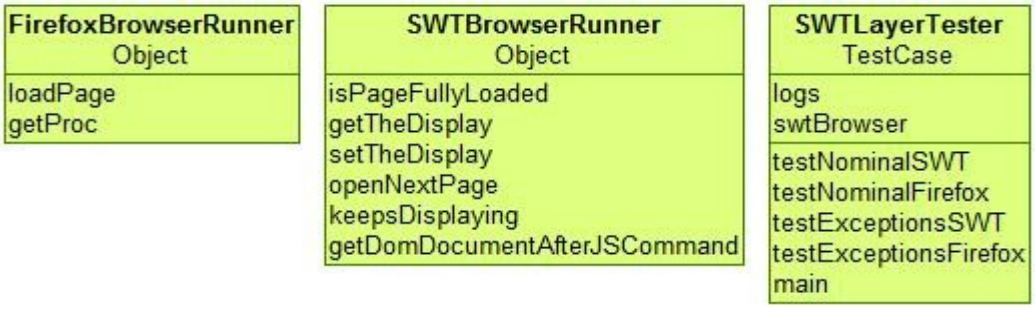

Illustration 27: Package SWTLayer

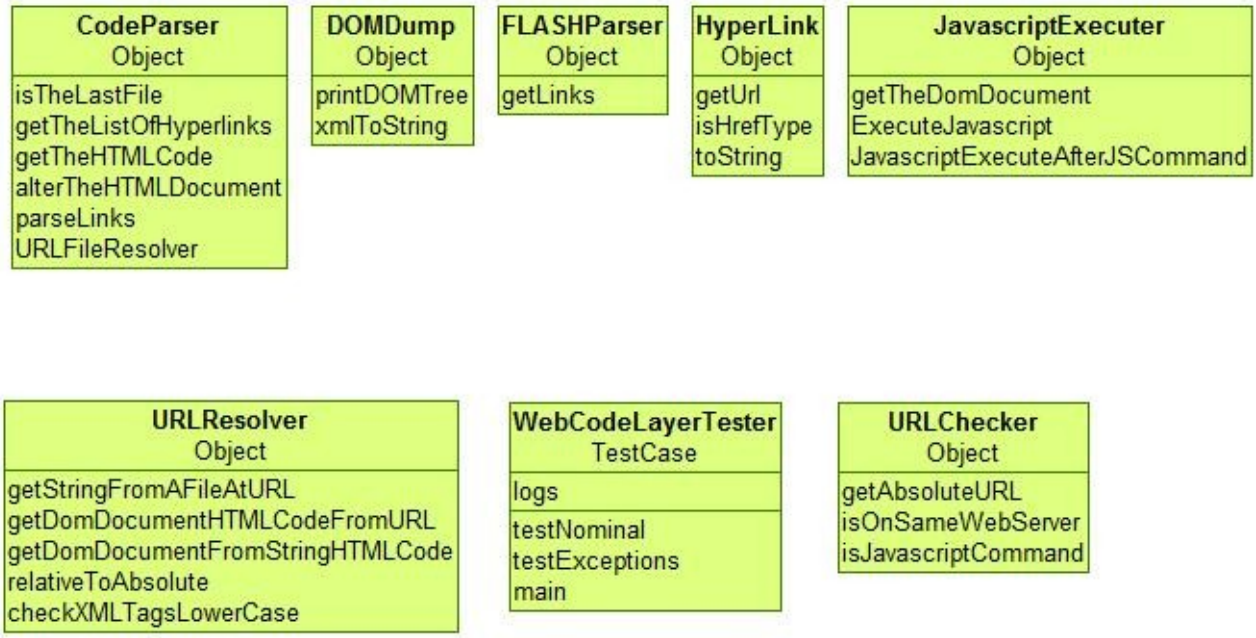

Illustration 28: Package WebCodeLayer

7. Déploiement du projet

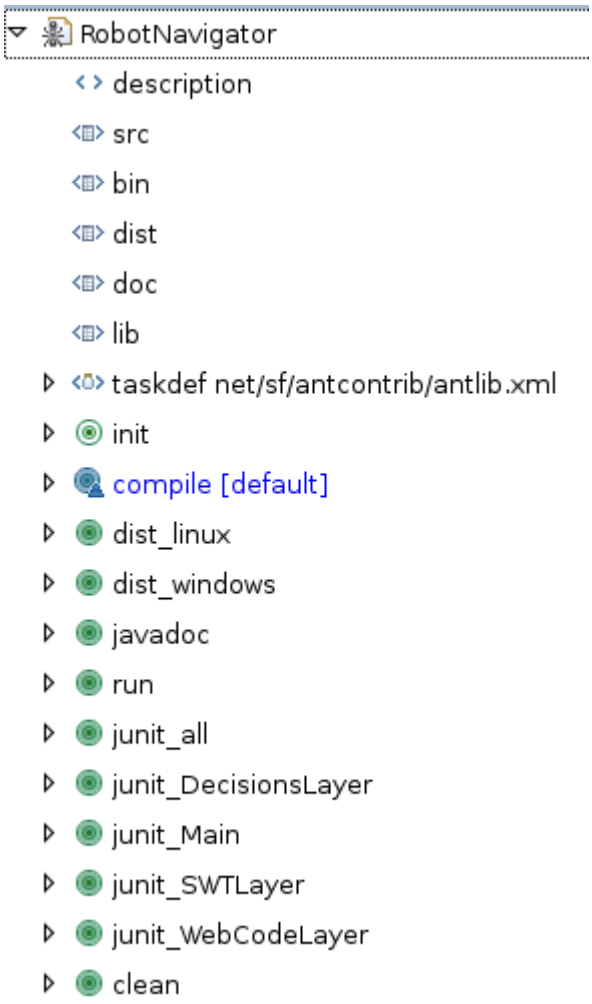

Illustration 29: cibles ant

- Autour de ce projet est disponible en plus de la présente documentation, une javadoc qui détaille le code source
- Deux versions du logiciel sont disponibles : une version Linux et une version Windows (certaines des bibliothèques utilisées changeant en fonction de la plateforme)
- Pour chacune de ces versions il existe un exécutable .jar standalone.
- La qualité du projet est mise à l'épreuve par les cibles JUnit. En cas d'anomalie dans les processus du système il est normalement possible de poursuivre l'exécution du programme.

7.1 Configuration requise

Le programme est multi-plateforme mais nécessite la pré-installation d'au moins un outil pour la navigation web.

- Si vous souhaitez utiliser le mode "navigateur Firefox", il est alors nécessaire d'avoir installé sur votre machine Mozilla Firefox version "2.0 Bon Écho" ou supérieure
- Si vous souhaitez utiliser le mode "navigateur SWT", il est alors nécessaire d'avoir installé sur votre machine Xulrunner version 1.8 ou supérieure.

Tous les outils utilisés pour la documentation, dans le code ainsi que ces outils nécessaires à l'exécution du programme sont sous licence publique générale GNU.

7.2 Javadoc

La javadoc de notre code est disponible sur http://.

7.3 Contrôle qualité

7.3.1 Qualité structurelle

Le but de ce projet était de créer un outil pluggable sur un autre programme. Comme il est énoncé dans le sujet, les décisions aléatoires à prendre à chaque changement de page ne sont qu'un substitut provisoire au module final. Il a donc été nécessaire de présenter un programme pouvant fonctionner par couches : Les packages Java correspondant aux différentes couches de l'application peuvent fonctionner indépendamment.

7.3.2 Qualité fonctionnelle

Élément à tester	URL	Résultat
Scénario nominal		
page web HTML statique	http://englishblazere.free.fr/moncv	OK
Scénarii d'exception		
page web contenant une base de données	http://karine.anne.free.fr	OK
page web contenant une image mappée	http://tecfa.unige.ch/guides/htmlman/imagemap-ex.html	OK
page web contenant une frame	http://karine.anne.free.fr/workspace/IA/doc/	OK
page web contenant des programmes Flash	http://www.imaginaweb.ch/_a/gina/v2/free-flash/game.html	OK
page web contenant des éléments générés par du JavaScript, AJAX	http://karine.anne.free.fr/workspace/web2/homePage.html	OK
page web contenant des commandes Javascript changeant l'aspect de la page sans changer l'URL	http://195.221.220.118/~anne/homePage.php	50%

Tableau 3: Table de la qualité fonctionnelle

7.3.3 Bêta Test : Rapport de bugs

Programmes exécutables :

- version Linux
- version Windows

info < warning < severe :

Code	Niveau	Description	Résultat
01	info	Les url ne portant pas le préfixe http:/ provoquent une erreur	OK
02	warning	boolean URLFileResolver(String aUrl) est parfois sensiblement trop lent.	
03	severe	Sous windows et SWT, les frames sont parsées anormalement.	OK
04	warning	lors de l'exécution de commandes javascript (url: javascript:uneFonction()), l'affichage se produit anormalement : on ne voit que l'exécution de la dernière commande	
05	warning	Impossible de reprendre l'exécution de javascript une fois les frames récupérées.	
06	info	Si Xulrunner n'est pas installé : Could not instantiate Browser: No more handles [Could not detect registered XULRunner to use] + Exception Il faut un pop-up expliquant la configuration requise.	
07	severe	Sur Windows (à cause de Javascriptexecuter), les URL http://www.lsis.org et http://www.lephoceen.fr ne fonctionnent pas.	OK
08	severe	Le JDK de Yoann est défaillant.	OK
09	info	Lorsqu'on traverse des liens morts durant l'exécution, l'utilisateur n'en est pas informé	OK
10	info	Si Mozilla Firefox n'est pas installé, rien n'indique qu'il doit l'être.	

Tableau 4: Table des erreurs

7.4 Démonstration illustrée

Illustration 30: interface d'accueil sur le programme

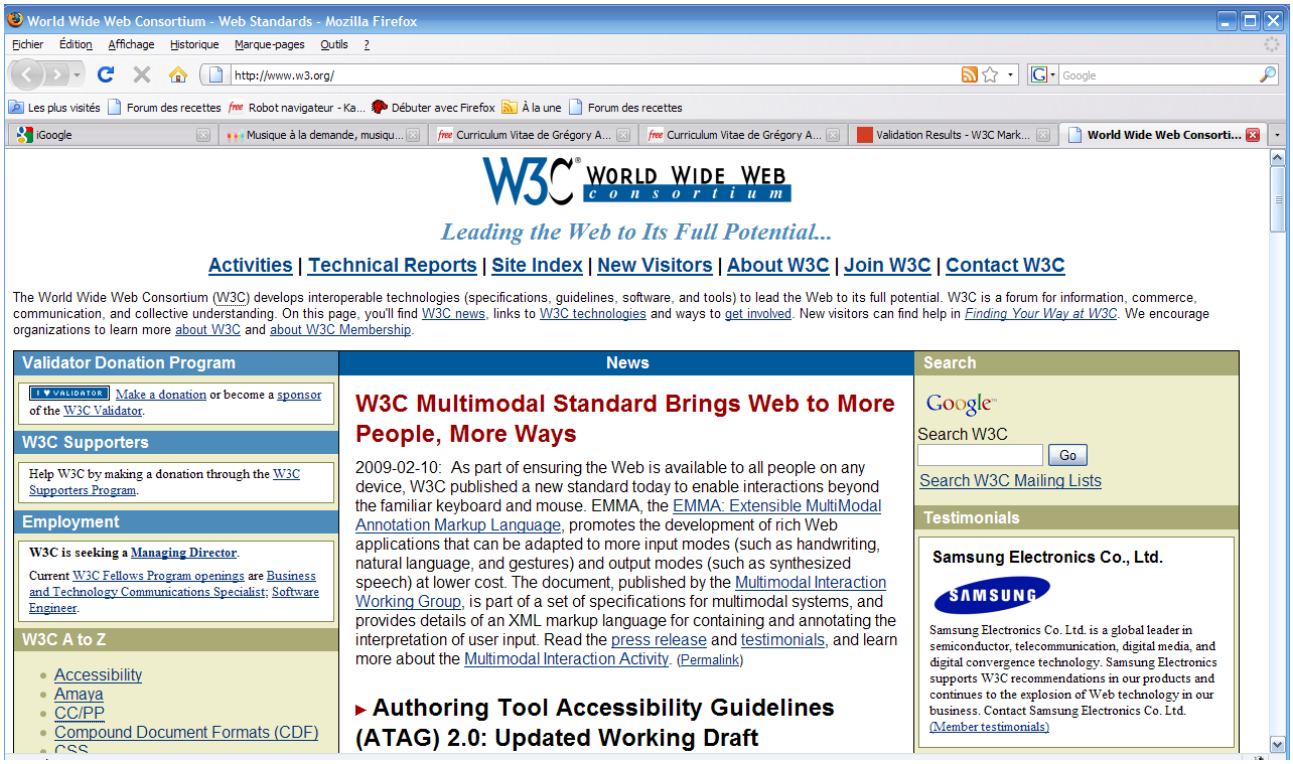

Illustration 31: Mode Firefox

Illustration 32: Mode SWT sans onglets

Illustration 33: Mode SWT avec onglets

8. Bilan du projet

8.1 Fiche de Génie Logiciel

ce script a été élaboré pour le décompte des lignes de code :

```
echo "nombre de lignes en tout : "
tr "\t" "\n" < $1 | tr -s "\n" | tr -s "\t" | wc -l
echo "nombre de lignes de code : "
cat $1 | sed  's!/\*.*\*/!!' | sed  '/\/\*/,/\*\//d' | sed '/^\ *\/\//d' | sed
'/^\ *$/d'  | wc -l
echo "nombre de lignes de commentaires : "
expr `tr "\t" "\n" < $1 | tr -s "\n" | tr -s "\t" | wc -l` - `cat $1 | sed
's!/\*.*\*/!!' | sed  '/\/\*/,/\*\//d' | sed '/^\ *\/\//d' | sed '/^\ *$/d'  |
wc -l`
```

Etapes		Nombre de lignes effectives	Temps passe -chacun- (J)	Ressources (H)	JH
Analyse			1	4	4
Conception			1	1	1
Codage	Decisions.java	99	0,5	1	0,5
	DecisionsLayerTester.java	118	1	1	1
	Interface.java	332	2	1	2
	LogsFormater.java	60	0,5	1	0,5
	Main.java	153	1	1	1
	MainTester.java	42	0,5	1	0,5
	TestsExecution.java	34	0,5	1	0,5
	FirefoxBrowserRunner.java	58	1	1	1
	SWTBrowserRunner.java	216	4	1	4
	firefox.sh	15	1,5	1	1,5
	firefoxBrowser.bat	2	0,5	1	0,5
	SWTLayerTester.java	134	0,5	1	0,5
	CodeParser.java	295	8	1	8
	HyperLink.java	34	0,5	1	0,5
	URLChecker.java	170	1	1	1
	DOMDump.java	111	0,5	1	0,5
	JavascriptExecuter.java	126	4	1	4
	URLResolver.java	97	1	1	1
	FLASHParser.java	40	0,5	1	0,5
	WebCodeLayerTester.java	210	1	1	1
Mise au point	Decisions.java		1	2	2
	DecisionsLayerTester.java		0,5	1	0,5
	Interface.java		0,33	3	1
	LogsFormater.java		0,5	1	0,5
	Main.java		4	2	8
	MainTester.java		0,5	1	0,5
	TestsExecution.java		0,5	1	0,5
	FirefoxBrowserRunner.java		0,5	2	1
	SWTBrowserRunner.java		4	2	8
	firefox.sh		1	1	1
	firefoxBrowser.bat		0,5	1	0,5
	SWTLayerTester.java		0,5	1	0,5
	CodeParser.java		5	3	15
	HyperLink.java		0,5	1	0,5
	URLChecker.java		8,5	2	17
	DOMDump.java		0,5	2	1
	JavascriptExecuter.java		6	2	12
	URLResolver.java		1	2	2
	FLASHParser.java		2,5	2	5
	WebCodeLayerTester.java		0,5	1	0,5
Contrôle Qualité			1,5	3	4,5
Documentation			7	4	28
TOTAUX		2346	78,83	4	144,5

Calcul des jours (JH / H) : 36,13 jours
Ratio de productivité : 64,94 lignes / jour
29,76 lignes / jour / personne

* les « lignes de codes effectives » sont les nombres de ligne du fichier, lignes de commentaires une fois soustraites.

Illustration 34: Calcul de la productivité

8.2 Conclusions

Bien que de nombreuses améliorations soient encore possibles, le projet est arrivé à un résultat intéressant, qui remplit les objectifs fixés. Mettre en œuvre le projet dans toutes les différentes étapes est quelque chose de très motivant et le travail engendré est proportionnel à cette motivation.
La réalisation de ce projet nous a permis de nous familiariser avec plusieurs outils et technologies de programmation Java et web.

Ce projet a été établi dans le cadre de recherches du LSIS sur la manière dont naviguent les internautes, c'est donc ce qui correspond à notre couche de décisions qui va être transformé en un algorithme complet statistique du comportement de l'internaute. Il serait bien sûr intéressant pour nous d'être témoins de la mise en œuvre de ce projet.

Index des illustrations

Index des tables