Maîtrise d'Algorithmique et programmation

Plugin Eclipse pour O'Caml

version 2

Guillaume CURAT **Sylvain Le Ligné**

Encadrant : Emmanuel Chailloux

Sommaire

Présentation du sujet

Description générale

Eclipse est un outil pour construire des outils de développement. Une première version d'environnement pour O'Caml a été réalisée l'an passé. Ce projet est en prolongation. Il devra être porté pour la dernière version d'Eclipse et améliorer l'existant pour l'édition structurée en y intégrant l'information de typage issue de la compilation d'un module. Il tentera d'intégrer de nouveaux outils existants de profilage et de mise au point. L'implémentation pourra être réalisée directement en Java ou en O'Caml en utilisant un IDL Java-O'Caml.

Travail à réaliser

Faire le portage de la version 1 du plug-in pour la dernière version d'Eclipse. Améliorer la version 1 du plug-in en particulier un niveau de l'édition structurée. Intégrer l'information de typage engendrée par le compilateur ocamlc avec l'option –dtypes pour l'afficher dans l'éditeur. Ajouter les informations de profilage pour la version byte-code. Tester l'intégration du debugger pour la version Unix.

Outils de développement pour Objective Caml existants

La première partie de notre travail a été de faire une analyse des outils de développement pour Ocaml disponibles sous les différentes plateformes du marché (Linux, Windows, Mac OS X).

- **Linux**
  - L'éditeur de texte Emacs est le principal environnement de développement sous linux. Il dispose d'un support natif pour éditer des fichiers caml. De plus il dispose d'un plugin plus complet nommé Touareg. Cet

environnement permet d'avoir une coloration syntaxique des mots clés du langage Caml, d'un raccourci à la documentation, d'exécuter un toplevel et de lancer des commandes.
    - Un outil de listage des modules est maintenant fournit avec la distribution d'Ocaml : Ocamlbrowser.
    - Pour facilité la compilation de projet, il existe un utilitaire nommé Ocamake. Ce dernier permet de créer un makefile facilement.

- **Mac OS X**
    - Depuis la version 10, le système d'exploitation d'Apple est basé sur un noyau Unix.. Donc tous les outils (ou presque) disponibles sous Linux le sont aussi sous Mac OS X. Cependant l'installation de ces outils demande de bien connaître le monde Unix. Il existe un environnement de développement spécifique au Mac : CocOCaml.
      Ce dernier s'apparente à un toplevel fenêtré.

- **Windows**
    - Sous ce système d'exploitation, il n'existe presque pas d'outils. Il y a un toplevel fenêtré (fourni avec la distribution d'Ocaml pour Windows).

Conclusion de l'analyse :

Il existe très peu d'outils sous Windows, aucun adaptés au développement d'un projet en Ocaml. Il n'existe pas d'outils de compilation comme Make. Sous linux, il y a beaucoup d'outils, mais qui ne sont pas liés entre eux. De plus ils ne sont pas toujours très conviviaux. Quant à Mac OS X, si on ne maîtrise pas les bases Unix, on se retrouve dans le même cas qu'un développeur sous Windows.

Objective Caml étant un langage de programmation portable, il serait bien d'avoir aussi un environnement de développement commun à tous les systèmes d'exploitation.

La solution envisagée est le développement d'un plugin pour Eclipse permettant d'avoir un environnement de développement supportant le langage Ocaml.

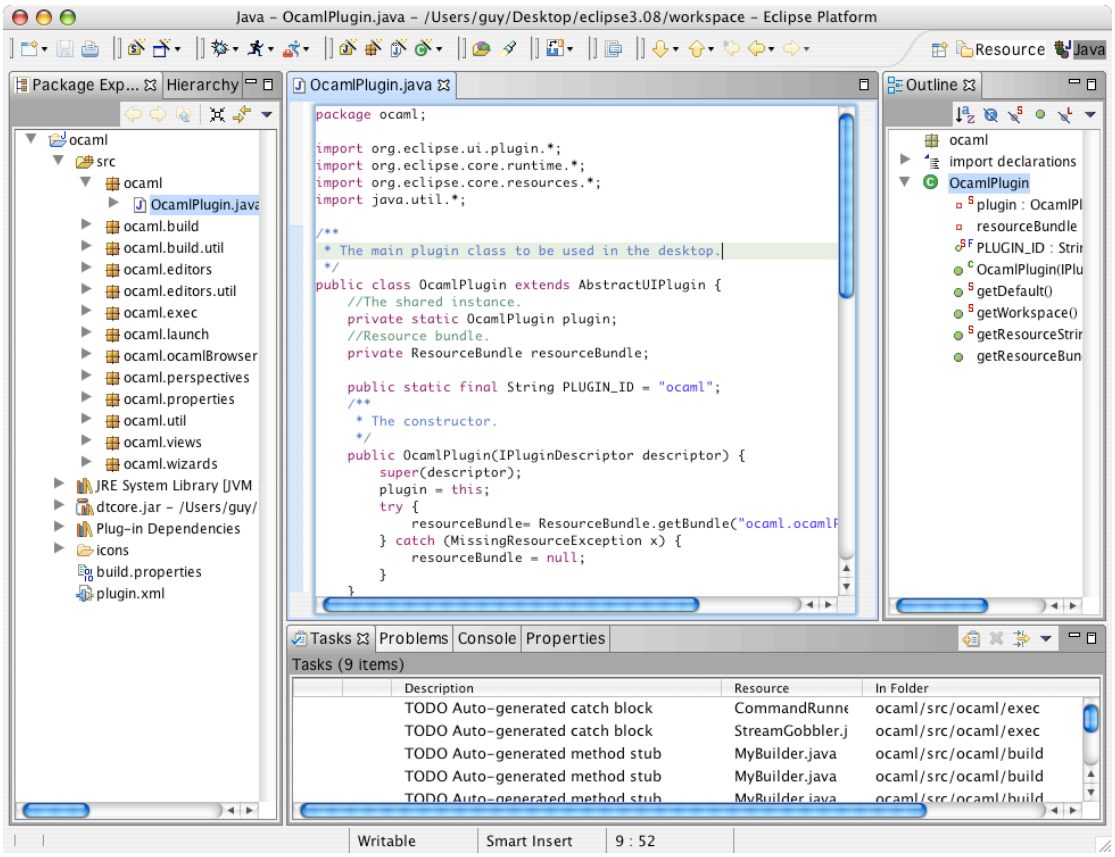
Présentation d'Eclipse

La plateforme Eclipse est un IDE (integrated development environment) qui peut être utilisé pour créer diverses applications comme des sites web, des programmes Java, des programmes C++. Eclipse est donc un IDE très généraliste.
Eclipse a été entièrement écrit en langage Java, ce qui a permis un déploiement sur la plupart des systèmes d'exploitation : Windows, Unix, MacOSX. Eclipse a pour vocation d'intégrer facilement des concepts et des outils sous la même plateforme à fin de fournir un environnement de développement performant, complet, modulaire et gratuit.

Donc Eclipse est une plateforme qui est utilisable par des utilisateurs quelconques sur différentes plateformes. Le langage utilisé permet une utilisation similaire sur les différents

systèmes d'exploitation avec une interface d'utilisation strictement identique. Eclipse est un IDE très efficace grâce à des outils de formatage et d'aide à la programmation. Mais sa force réside en l'intégration d'outils extérieurs. Ainsi il est possible sous Eclipse d'utiliser un bash, gérer des bases de données ou de modéliser un projet. Cela simplifie et accélère donc le travail des utilisateurs d'une telle plateforme.

La figure 1 montre l'interface de développement d'un projet Java. L'environnement est composé d'un ensemble de vues (Package Explorer, Outline, Tasks, Problems, Console et Properties) et d'éditeurs de fichiers (éditeur java). On appelle une perspective le regroupement d'un ensemble de vues et d'éditeurs.

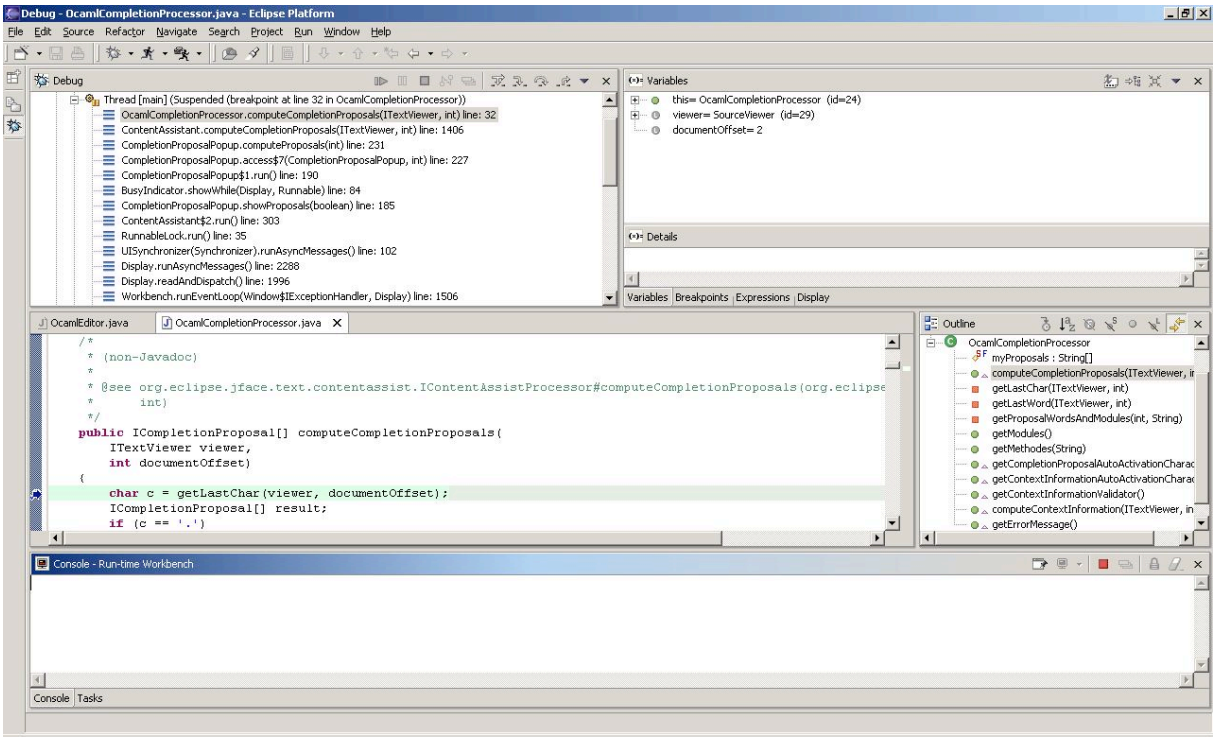


Interface Eclipse pour le développement en Java (Figure 1)

Voici un certain nombre de fonctionnalités offertes avec Eclipse pour le langage Java:

- Gestion de projet

  - Gestion des projets par des vues et des perspectives. Par exemple, le développement d'un projet et le débuggage du projet seront fait par 2 perspectives différentes. De même pour faciliter la lecture du code d'une classe, une vue nommée « outline » permet de voir la signature de toutes les méthodes de la classe et ses données.



Interface Eclipse pour débugguer un projet

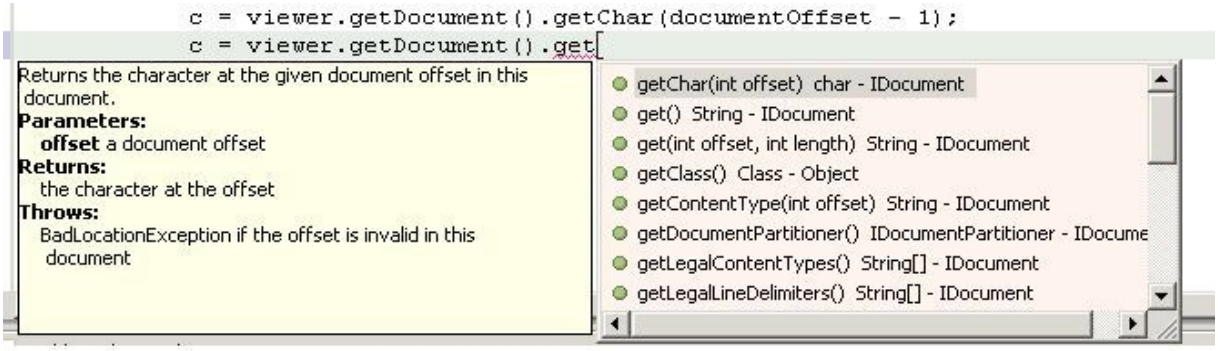
- Gestion de la documentation : la documentation d'un projet peut se faire grâce à la « javadoc », un outil java de documentation.

- Gestion des options de compilation pour un projet : il est possible de créer des dépendances entre les différents projets. La création de jar est facile à mettre en place.

Le développement d'un projet est donc très aisé grâce à tous les outils mis à disposition. De même l'organisation d'un projet est simplifiée et le travail en équipe devient moins contraignant.

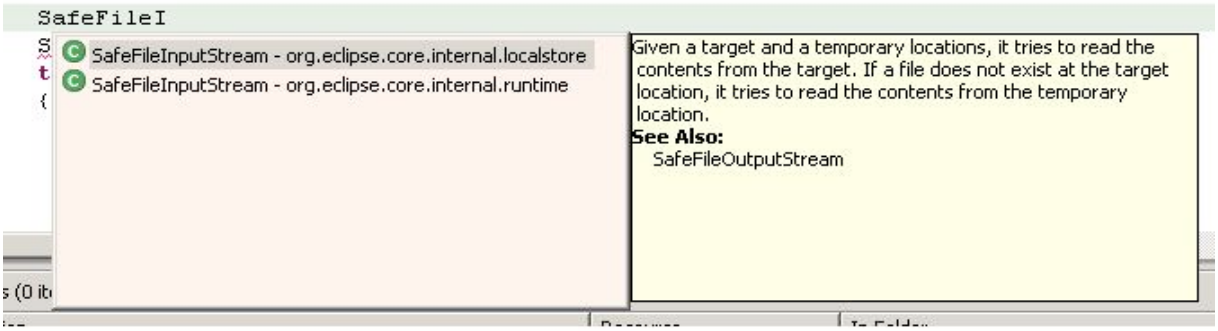
➢ Développement rapide

- Complétion : lors de la frappe au clavier, une liste de mots clé apparaît. Ces mots sont des classes ou des méthodes selon le contexte d'apparition. Ainsi la consultation de la documentation d'une classe ne devient plus systématique. Dans cette liste, sont aussi proposées les classes et méthodes du projet courant. De plus, pour chaque ligne de la liste, l'affichage de la « javadoc » si elle existe se fait. Cela permet donc de se passer de la consultation de la documentation de chaque méthode lors de son utilisation. Il y a ici un réel gain de temps.



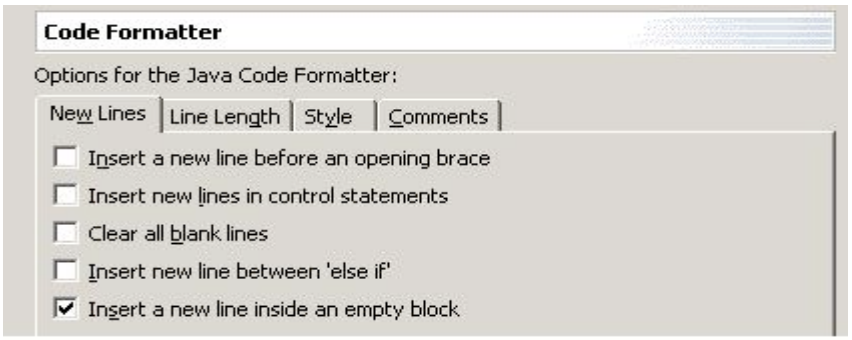
Complétion

- Importation automatique de librairies : lorsqu'une classe est utilisée son importation se fait automatiquement. Si un conflit existe, alors la complétion se charge de déterminer la classe concernée et fait l'importation.



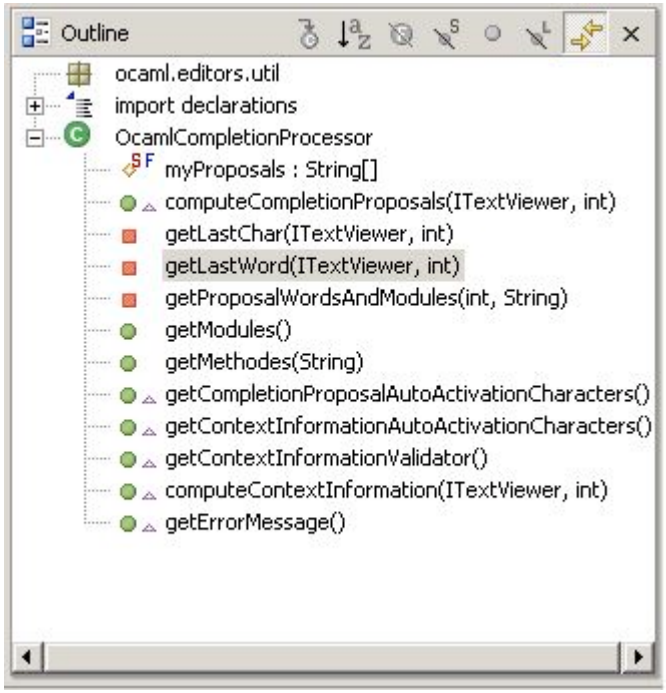
Importation de d'une classe avec conflit (2 classes avec le même nom)

- Indentation automatique paramétrable : il est possible de paramétrer l'indentation que l'on désire appliquer à un fichier. Cette option d'indentation n'est pas obligatoirement utilisable. Le texte ne se formate que si l'utilisateur le demande explicitement.



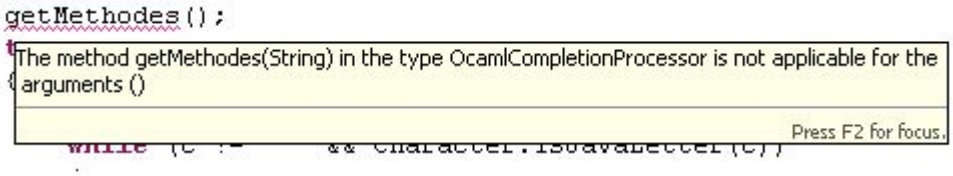
Option de formatage du code

- Signature des méthodes dans des vues : la vue outline permet de voir toutes les méthodes du fichier en cours d'édition. Toutes les informations y sont présentes : la signature et le fait qu'elle soit public, protected ou private.



La vue outline

- Vérificateur de code au cours de la frappe : la frappe au clavier est constamment vérifiée. Ainsi une faute de frappe ou un oubli de fermeture de parenthèse est immédiatement identifié. De même, quelques règles de vérification sémantique sont faites telles que la vérification des types paramètres lors de l'appel d'une méthode ou la non initialisation d'une variable.



Méthode appelée sans les bons arguments

Toutes ces petites fonctionnalités ne sont pas grand-chose en elle-même mais à l'utilisation, l'utilisateur se rend vite compte que le gain de temps est énorme, notamment au moment de la compilation. Avec Eclipse, lorsqu'une classe est finie d'être codé, elle est déjà compilée. Ensuite il est possible de la tester avec JUnit qui permet de créer des classes de test unitaire.

Eclipse est donc une plateforme complète qui est en constante évolution grâce à sa modularité et à sa politique d'ouverture sur des produits existants.

La plateforme Eclipse a été bâtie autour d'un mécanisme d'intégration et de lancement de modules appelés plugins. Ces derniers vont composer l'ensemble des outils et des fonctions de l'environnement de développement.

Ainsi pour ajouter de nouvelles fonctionnalités à Eclipse comme le support d'un nouveau langage (OCaml), il suffit de développer un plugin.
Eclipse permet de développer des plugins au sein même de son environnement.

La plateforme Eclipse : présentation technique

La plateforme Eclipse a été conçue pour satisfaire les points suivants :

- Fournir un environnement pour le développement d'applications.
- Support pour manipuler des contenus très différents (HTML, Java, C, JSP, XML…)
- Faciliter l'intégration d'outils existants et variés.
- Support pour le développement d'applications graphiques ou non.
- La plateforme est disponible pour de nombreux systèmes d'exploitation (Windows, Unix, MacOSX).

Le principal rôle de la plateforme Eclipse est de fournir un ensemble d'outils définis par des mécanismes et des règles qui constituent l'environnement de développement. Ces mécanismes sont décrits grâce à des API (Application Programming Interface), classes et méthodes. La plateforme fournit également un puissant framework qui facilite le développement de nouveaux outils.

La figure 1 montre les principaux composants et APIs de la plateforme Eclipse.

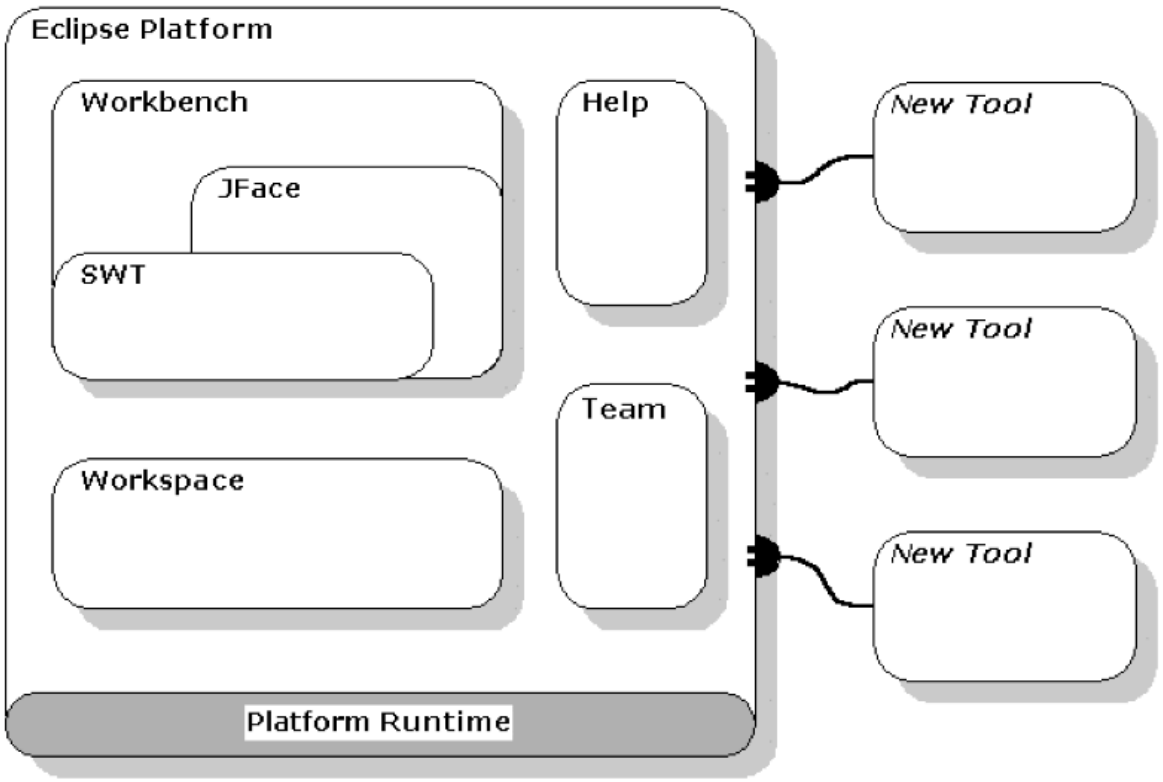


Figure 1 : Architecture de la plateforme Eclipse.
(Source : eclipse.org , eclipse-overview)

La plateforme runtime et l'arichitecture Plug-ins

Un plug-in est le plus petit élément de la plateforme Eclipse qui peut être développé et distribué. Dans la majorité des cas, un nouvel outil est développé sous la forme d'un seul plug-in alors que les fonctionnalités d'un outil complexe seront réparties sur plusieurs plug-ins. Toutes les fonctionnalités d'Eclipse sont gérées par des plug-ins sauf pour le noyau de la plateforme appelé *plateforme runtime*.

Les plug-ins sont entièrement codés en Java. Un plug-in classique est constitué de code Java dans une librairie JAR, de fichiers en lecture seule et d'autres ressources comme des images.
Certains plug-ins ne contiennent pas du tout de code. Le plug-in gérant l'aide en ligne en est un exemple. Il existe aussi un mécanisme qui permet à un plug-in d'être défini par un ensemble de fragments de plug-in. Ceci est souvent utilisé pour le support multi langue des plug-ins.

Chaque plug-in est associé à un fichier décrivant les interconnexions avec les autres plug-ins. Ce fichier est appelé *manifest file*. Le model d'interconnexions est simple : un plug-in déclare un certain nombre de points d'extensions (*extension points*), et de liens vers d'autres points d'extensions. Le fichier manifeste est un fichier XML.

Un point d'extension d'un plug-in peut être étendu par d'autres plug-ins. Par exemple, le *workbench* plug-in déclare un point d'extension pour les préférences utilisateurs. Tout autre plug-in a la possibilité d'avoir ses propres préférences utilisateurs en définissant extension du point d'extension du *workbench* plug-in.

Un point d'extension peut avoir une API correspondante. D'autres plug-ins peuvent contribuer à l'implémentation de cette interface via des extensions des points d'extensions. Tout plug-in peut définir de nouveaux points d'extensions et fournir une nouvelle API.

Au lancement, la plateforme *Runtime* recherche l'ensemble des plug-ins disponibles, lie les fichiers manifestes et construit une structure de dépendance entre les plug-ins, appelée base de registre des plug-ins. La plateforme associe chaque nom des déclarations d'extension avec les points d'extensions correspondants déclarés. Tous les problèmes détectés, comme l'impossibilité d'associer une déclaration à un point d'extension, sont enregistrés dans un fichier log. Les plug-ins ne peuvent pas être ajoutés après le lancement de la plateforme.

Voici un exemple de fichier manifest d'un plug-in qui définit un nouvel éditeur de texte pour les fichiers d'extension «txt ».

```
<plugin
  id="MyEditor2"
  name="MyEditor Plug-in"
  version="2.0.0"
  provider-name=""
  class="MyEditor.MyEditorPlugin">

   <extension
      point="org.eclipse.ui.editors">
    <editor
```

```
        name="MyEditor"
        icon="icons/sample.gif"
        extensions="txt"
        contributorClass="org.eclipse.ui.editors.text.TextEditorActionContributor"
        class="org.eclipse.ui.editors.text.TextEditor"
        id="org.eclipse.ui.editors.text.texteditor">
    </editor>
  </extension>
</plugin>
```

Attribut	Description
point="org.eclipse.ui.editors"	Point d'extension pour enregistrer l'éditeur
name="MyEditor"	Nom de l'éditeur
icon="icons/sample.gif"	L'icône qui sera associée à chaque fichier ouvert avec l'éditeur.
extensions="txt"	Extension de fichier associée à l'éditeur
contributorClass="org.eclipse.ui.editors.text.TextEditorActionContributor"	Définition des actions du menu.
class="org.eclipse.ui.editors.text.TextEditor"	Classe principale de l'éditeur. Ici, on utilise l'éditeur de texte par défaut d'Eclipse.
id="org.eclipse.ui.editors.text.texteditor">	Identificateur unique de l'éditeur.

Ce système de plug-ins est utilisé pour partitionner la plateforme Eclipse elle-même. Donc on dispose d'un plug-in gérant l'espace de travail (*workspace*), un autre pour l'environnement graphique (*workbench*)… Même le noyau d'Eclipse a son propre plug-in. Ainsi pour lancer Eclipse sans interface graphique, il suffit juste de supprimer le plug-in *workbench* et les autres dépendants de ce dernier.

Workspaces

On appelle *workspaces* les espaces de travail. Les différents outils introduits dans la plateforme Eclipse interagissant avec les fichiers de l'espace de travail de l'utilisateur. Le *workspace* regroupe un ensemble de projets qui ont un emplacement propre dans le système de fichiers. Chaque projet regroupe un ensemble de fichiers.

Eclipse introduit un mécanisme de *markers* qui permet d'associer une valeur à une ressource. Ainsi on peut spécifier la nature d'un projet (projet java, c…). De même on pourra spécifier le compilateur à utiliser pour un type de ressource. Les plug-ins peuvent bien sûr créer de nouveaux markers pour fournir de nouvelles fonctionnalités.

Toutes les opérations sur le *workspace* sont sauvegardées. Ainsi lors d'une nouvelle session, on retrouvera l'état du *workspace* comme on l'avait laissé auparavant.

Workbench et UI Toolkits

L'interface utilisateur (*UI*) de la plateforme Eclipse est gérée par une structure appelée *workbench.* Cette dernière apporte tous les outils nécessaires à la personnalisation de l'interface utilisateur. L'API du *workbench* et son implémentation repose sur deux *toolkits* :

- **SWT** (Standard Widget Toolkit) est une bibliothèque graphique libre pour Java, créée par IBM. Cette bibliothèque se compose d'une bibliothèque de composants graphiques (texte, label, bouton, panel…), de tous les 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. Par contre son API est entièrement indépendante du système d'exploitation.

- **JFace** fournit un ensemble de classes pour gérer facilement de nombreuses tâches sur les interfaces graphiques. L'implémentation et l'API de JFace sont indépendants du système de fenêtres graphiques. De plus JFace a été conçu pour fonctionner avec SWT. C'est cet outil qui va gérer toutes les fenêtres, et vues d'Eclipse.

Structure du Plug-In OCAML

Le plug-in ocaml est en fait composé de deux plug-ins distincts. Le premier est commun à toutes les plateformes. C'est le plug-in « ocaml » qui contient toutes les fonctionnalités visuelles du plug-in : éditeur et vues (ocaml browser, console, …). Le deuxième plug-in ocamlCompiler possède plusieurs versions différentes. Ce deuxième plug-in contient le compilateur caml et toutes les librairies standards associées. Il y a donc une version du plugin par plateforme : Windows, Mac et Linux. Une quatrième version a été ajoutée pour les développeurs « confirmés » où il faut définir soit même le chemin du compilateur et des outils utiles pour le fonctionnement du plug-in. Nous avons décidé d'intégrer le compilateur à un plug-in car nous utilisons des options du compilateur qui ne sont apparues qu'à partir de la version 3.07 d'Ocaml. Donc pour éviter des problèmes d'utilisation du plug-in, il nous est apparu évident d'intégrer le compilateur. Ainsi, nous savons quelle version du compilateur Ocaml est utilisée. Par la suite, il a été utile de faire un plugin ne contenant pas de compilateur, mais où il était possible de choisir son propre compilateur.

Les deux plug-in sont donc liés : le plug-in ocaml va chercher tous les chemins d'accès aux librairies ou au compilateur (ou autres outils) dans le plug-in ocamlCompiler via des méthodes spécifiques.

Il est à remarquer que si le plug-in ocamlCompiler sans compilateur est utilisé, il faut que les chemins d'accès ne contiennent pas d'espaces ou de caractères spéciaux.

Par la suite, toute la description se portera sur le plug-in ocaml uniquement. En effet, c'est dans ce plug-in que l'essentiel du travail a été effectué. Le plug-in ocamlCompiler n'est qu'une interface pour obtenir des chemins.

Fonctionnalités du plug-in

Perspective et Vues

Une perspective regroupe un ensemble de fenêtres appelées vues. La perspective ocaml fournit quatre vues :

- OCaml Browser : listage des modules caml disponibles.
- L'éditeur de fichier
- Outline : listage des fonctions définies dans l'éditeur.
- OCaml Console: zone d'affichage de messages.
- Navigator : navigateur de fichiers.
- Problem : regroupe toutes les erreurs de compilation.

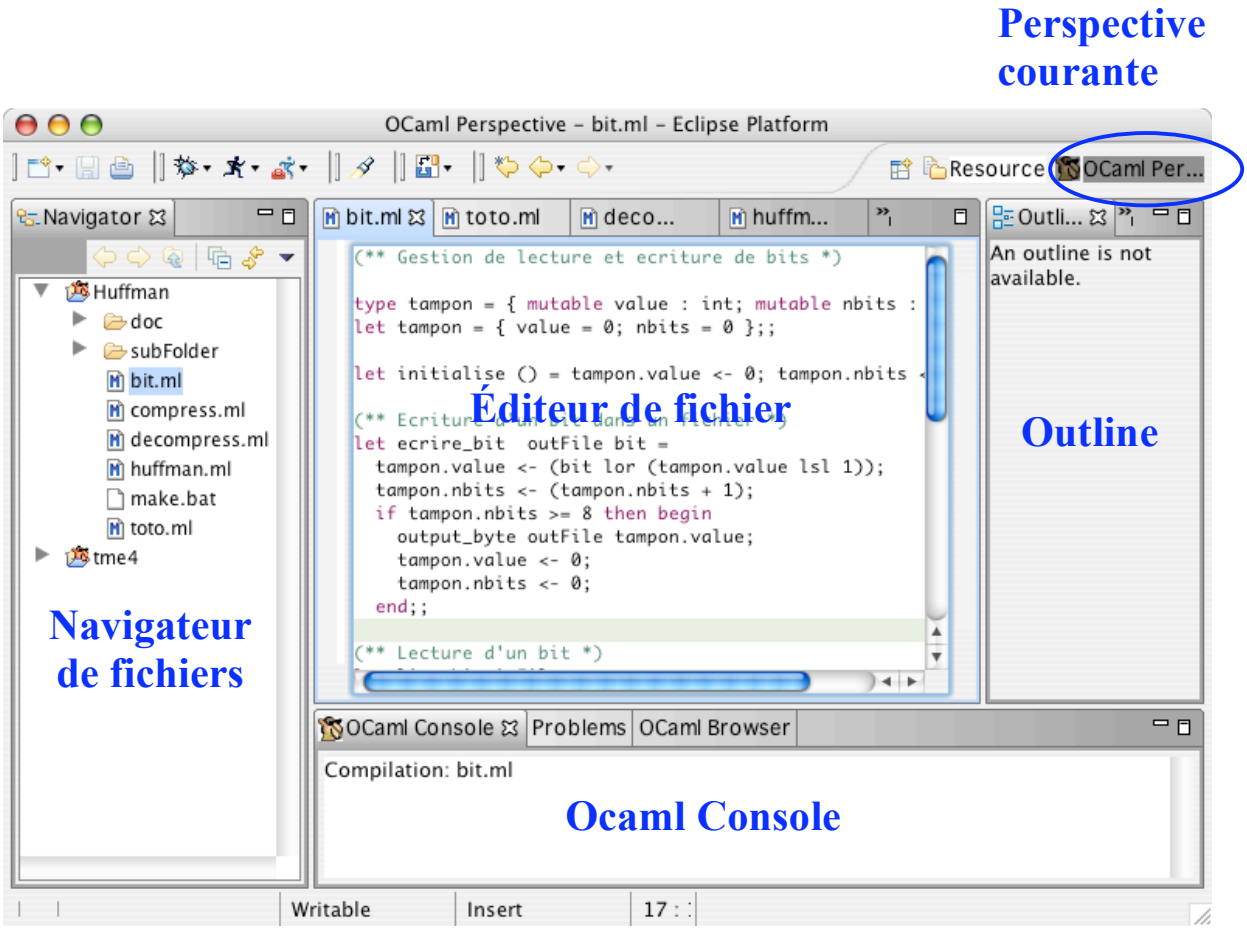


Figure 2 : Les différentes vues offertes par la perspective Ocaml.

OCaml Browser

La vue OcamlBrowser est une réplique de l'outil du même nom fourni avec la distribution d'OCaml. La version caml de cet explorateur de modules étant impossible à intégrer sous Eclipse, une version entièrement écrite en java a été développée (voir figure 7). Lors de la création de la vue, on récupère la liste des fichiers d'extension « mli » dans le chemin d'inclusion des librairies ocaml du plugin ocamlCompiler. Ainsi on obtient la liste des modules disponibles. Pour chaque module, on parcourt le contenu du fichier pour récupérer le nom des méthodes associées. Enfin on peut remplir les différents champs de la vue... La liste des modules est la liste des librairies standard de caml. Lors de la sélection du module, le fichier correspondant portant l'extension « mli » est parsé pour récupérer la signature de la méthode, de l'exception ou du type avec le commentaire correspondant. Le graphisme de la fenêtre a été créé entièrement grâce à l'API JFace..

Classes utilisées :

ocaml.ocamlBrowser.OcamlBrowser
ocaml.views.OcamlBrowserView

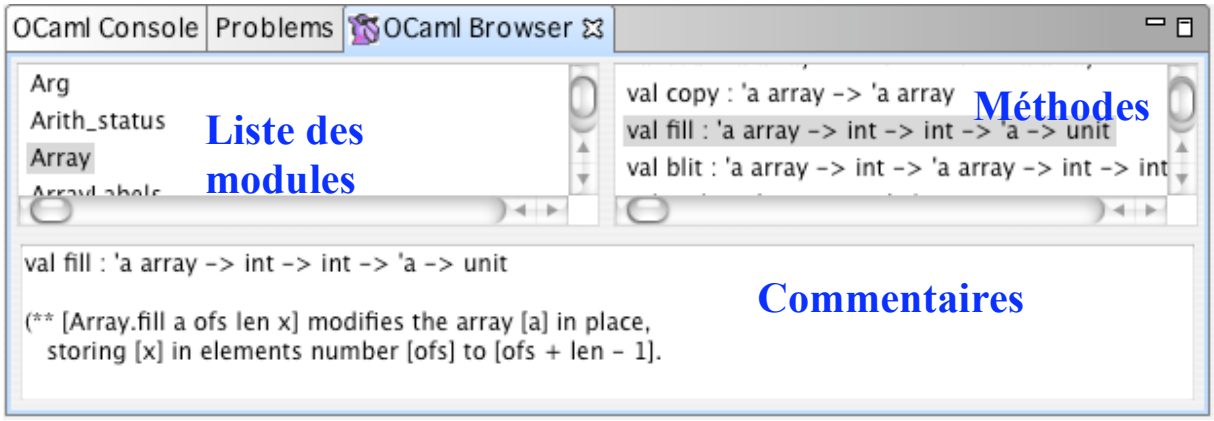


Figure 7 : OcamlBrowser

Éditeur de fichiers caml

Les fichiers caml (dont l'extension est ml, mli, yacc, mly) sont éditables grâce à un éditeur de texte qui a été défini dans la classe ocaml.editors.OcamlEditor. Cette classe étend la classe org.eclipse.ui.editor.text.TextEditor qui décrit l'éditeur de texte standard offert par Eclipse.

L'éditeur de fichiers caml offre les options suivantes :

- **Coloration syntaxique des mots du langage**. Les mots clés et constantes du langage sont définis dans l'interface ocaml.editors.util.ILanguageWords. La classe ocaml.editors.util.OcamlRuleScanner définit les règles du parser.

  Classes utilisées :
  ocaml.editors.util.OcamlRuleScanner
  ocaml.editors.util.ILanguageWords

- **Complétion :** lors de l'appui des touches ctrl+espace, un menu apparaît proposant de compléter le texte en cours de frappe. Les mots clés du langage sont proposés dans le menu déroulant ainsi que les modules ou méthodes associées disponibles selon le contexte. La liste des modules contient les modules de la bibliothèque standard ainsi que les modules du projet courant qui ont été compilés et dont la création du fichier mli s'est bien déroulée. La récupération des modules et le parsing pour retrouver la liste des méthodes du module correspondant se fait de la même façon que pour la vue OcamlBrowser. La figure 3 montre un exemple de complétion.

  Classes utilisées :
  ocaml.editors.util.OcamlCompletionProcessor
  ocaml.editors.util.OcamlProposal

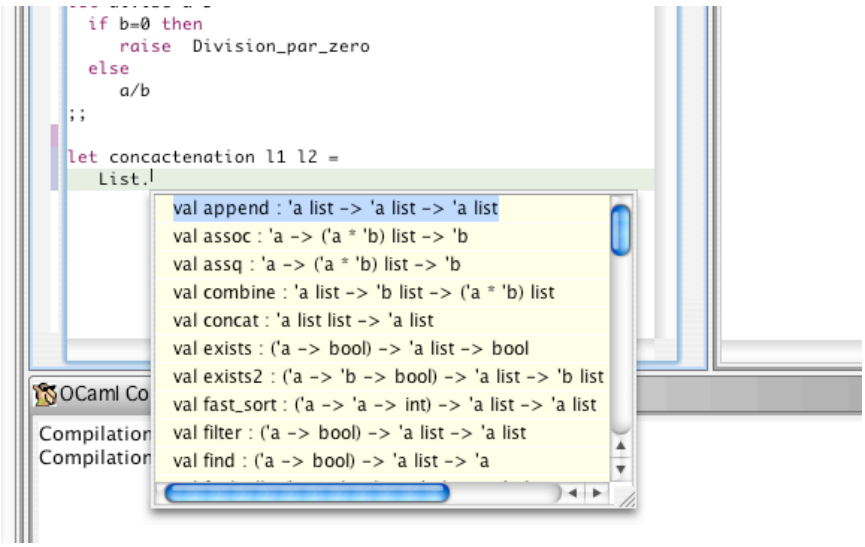


Figure 3 : la complétion

- **Information sur le type des expressions :** lorsque l'on passe la souris au-dessus d'un mot du code source, on fait apparaître le type de l'expression. Ce dernier est calculé lors de la compilation avec l'option –dtypes. Cette option de compilation étant disponible depuis la version 3.07 d'OCaml, il est indispensable d'installer la version 3.07 ou ultérieur d'OCaml sur la machine pour avoir l'information sur le type des expressions. La compilation avec l'option –dtypes crée un fichier d'extension *annot* qu'il suffit d'analyser pour obtenir l'information de typage pour les expressions du fichier. Il est à

remarquer que l'option -dtypes est pour certaines expressions peu précises sur leur type.(match par exemple). Lorsque le fichier est modifié, les informations sur les types n'apparaissent plus. En effet, si le fichier change, les types peuvent être conduits à évoluer aussi. Nous avons donc décidé de supprimer cette information pour ne pas afficher de mauvaises informations sur le type des expressions.

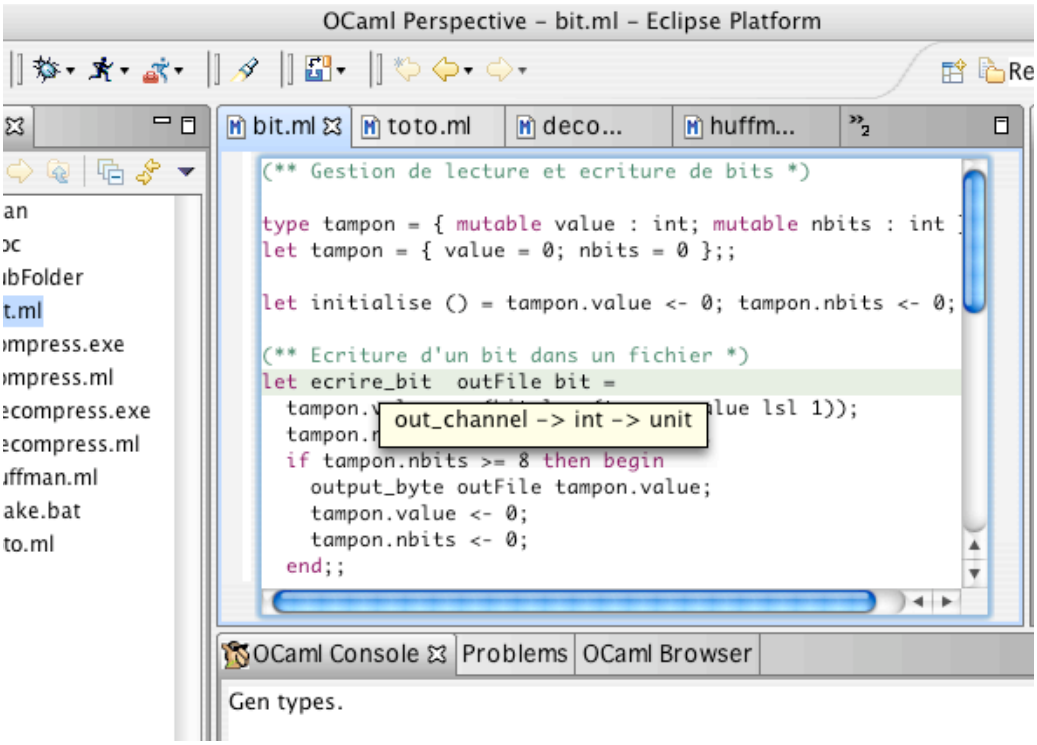


Figure 4 : affichage des types

- **OCaml menu** : lors d'un clic sur le bouton droit de la souris sur le texte en cours d'édition, un menu déroulant apparaît. Ce menu permet de compiler le fichier courant, compiler l'ensemble du projet, générer les types pour le fichier (option présentée précédemment), générer la documentation ou bien nettoyer le projet. De plus chacune des actions est associée à un raccourci clavier comme le montre la figure 5. La génération de la documentation se fait sur le projet, dans un répertoire nommé « doc ». Cette documentation est générée soit au format html , soit au format LaTex. Le choix du format de la documentation se fait dans les propriétés du projet. Le nettoyage du projet supprime tous les fichiers qui ont été créés lors de la compilation (exe, .mli, …).

  Classes utilisées :

  ocaml.editors.actions.CleanProjectAction
  ocaml.editors.actions.CompileProjectAction
  ocaml.editors.actions.CompileFileAction
  ocaml.editors.actions.GenDocAction
  ocaml.editors.actions.GenTypesAction

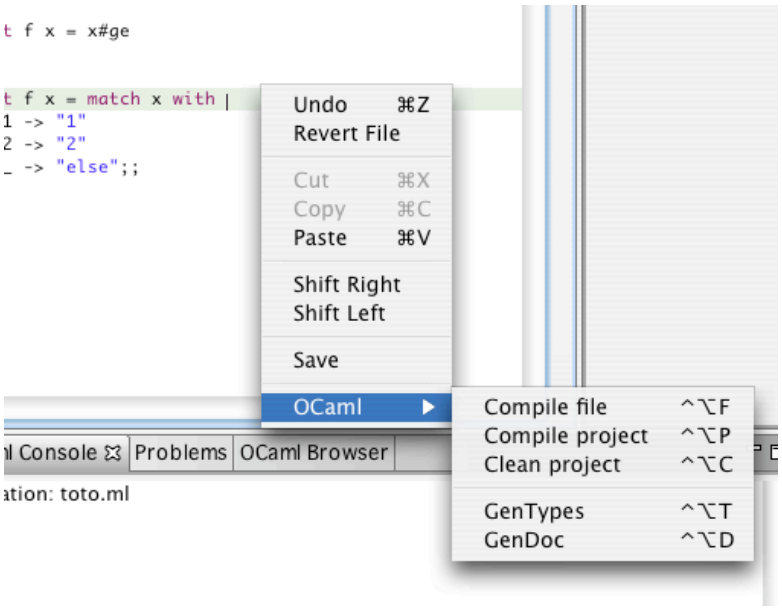


Figure 5 : menu OCaml de l'éditeur de fichier.

Outline

La vue outline représente un résumé des fonctions, types, constructeurs déclarés dans le fichier en cours d'édition. Tous ces éléments sont récupérés lors de la compilation du fichier avec l'option –dprasetree. La compilation d'un fichier avec l'option –dparsetree retourne un arbre syntaxique qu'il ne reste plus qu'à parcourir et analyser pour y afficher les informations.

Pour chaque élément de l'arbre syntaxique, on associe une icône :


Représente la racine de l'arbre.


Déclaration d'un nouveau type caml. (type)


Déclaration des exceptions. (exception)

Utilisation d'une librairie. (open)


Déclaration d'une valeur. (let)

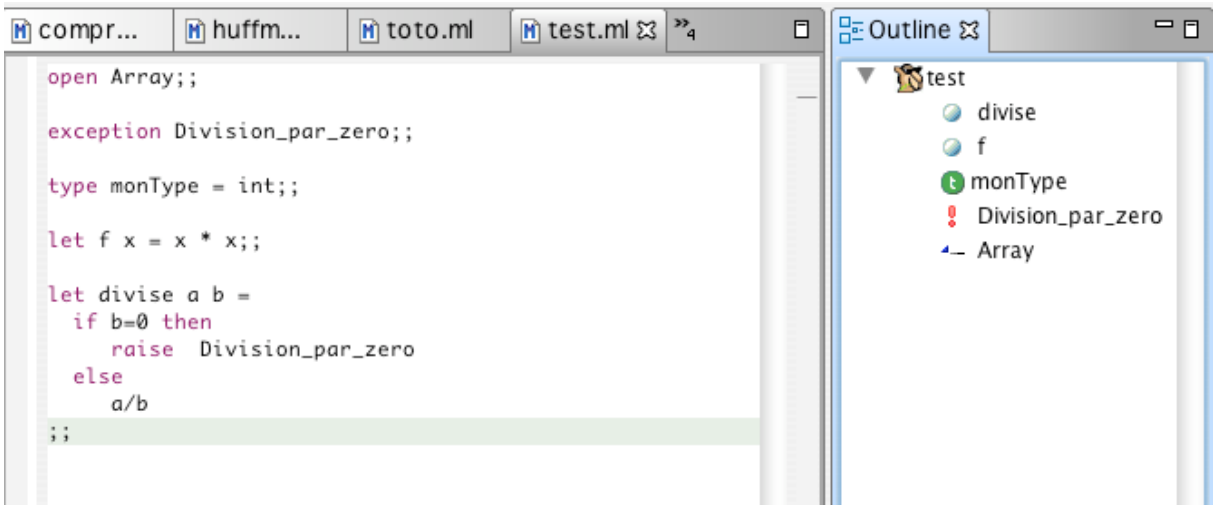


Figure 6 : la vue outline.

Classes utilisées :

ocaml.views.OcamlOutlineControl
ocaml.views.util.*

OCaml Console

La vue OCaml Console est une zone de texte qui affiche tous les messages liés à la compilation et à l'exécution de commandes (voir figure 6). Cette vue n'est qu'une redirection des messages des différentes commandes (ocamlc, ocamldoc…). Cette vue permet de vérifier d'éventuelles erreurs retournées par les commandes. Elle s'avère très utile pour vérifier le bon fonctionnement des autres vues ou de l 'éditeur.

Classes utilisées :

ocaml.views.OcamlConsole

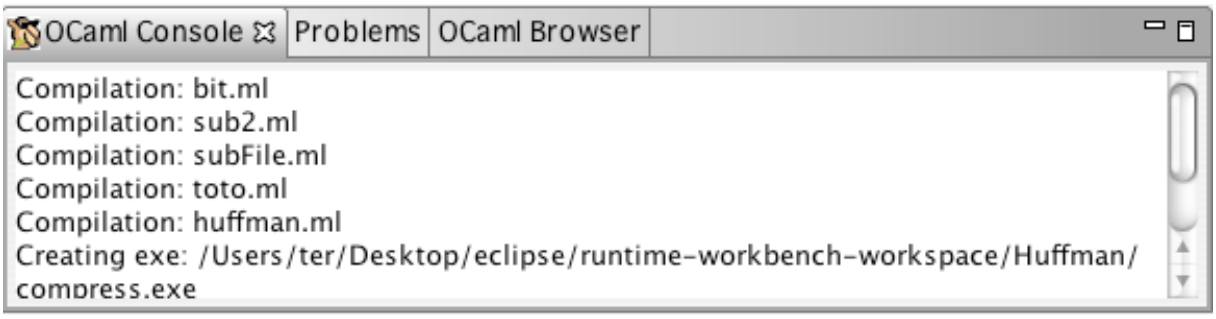


Figure 7 : la vue OCaml Console

Navigator

Cette vue correspond à un explorateur de fichiers. Les projets caml sont représentés par un dossier avec un chameau  . Ceci est géré à l'aide d'un marqueur qui spécifie la nature d'un projet (voir la classe ocaml.natures.OcamlNature). Pour chaque fichier caml de types différents, on associe une icône. L'association icône-fichier se fait dans le fichier *Manifest* (plugin.xml).

De plus la coloration rouge autour d'une icône montre qu'un fichier a été généré lors de la compilation.
Ainsi, on peut repérer facilement les fichiers propres à un projet et ceux qui sont générés.

Pour obtenir cette coloration, on a procédé de la manière suivante : lors de la phase de compilation, pour chaque fichier généré on modifie la valeur de son attribut de génération à vrai. Dans le fichier *manifest*, on déclare une association entre une icône rouge et les fichiers dont l'attribut de génération vaut vrai.

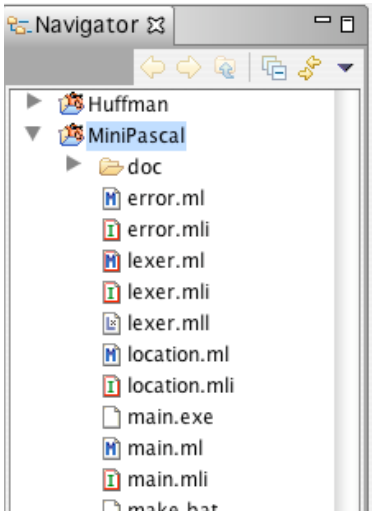


Figure 8 : l'explorateur de fichiers

Problem

La vue *problem* regroupe les erreurs et avertissements engendrés lors de la compilation. Il y a un lien entre le message d'erreur ou d'avertissement et l'emplacement de l'erreur dans le fichier. Le compilateur s'arrête à la première erreur trouvée dans un fichier. Il ne peut donc y avoir qu'un seul lien d'erreur par fichier. Mais il peut y avoir plusieurs avertissements comme le montre la figure 9.

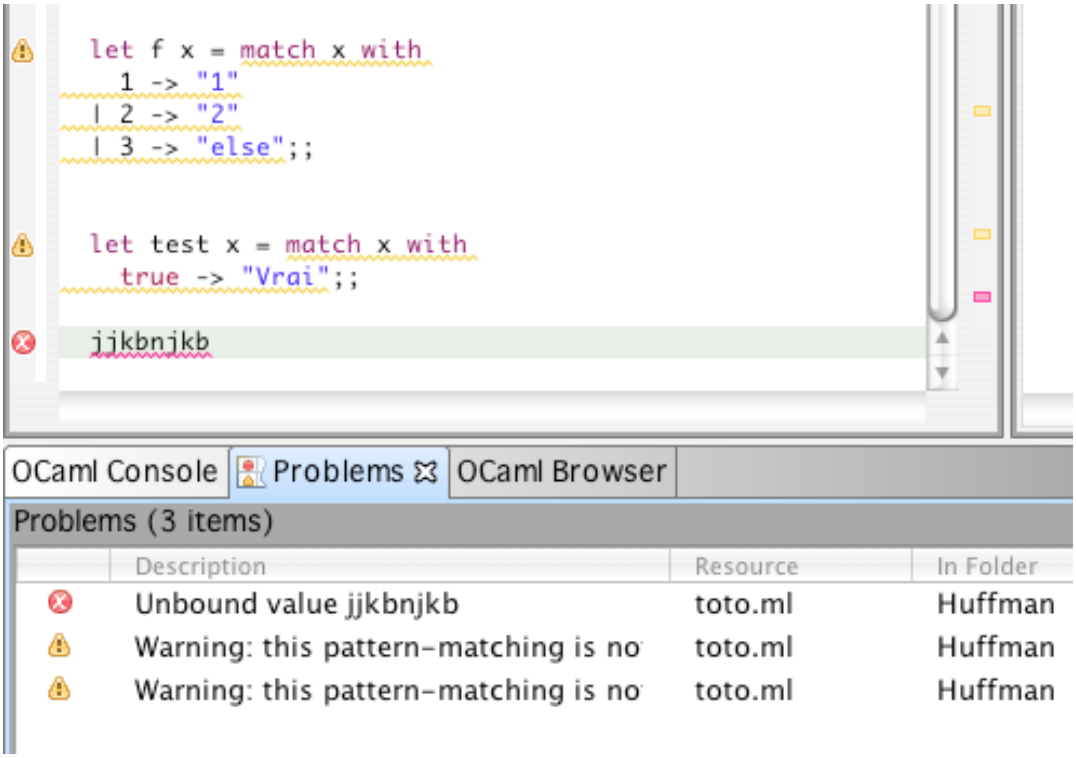


Figure 9 : la vue Problem.

Compilateur

Le processus de compilation d'un projet caml qui a été implémenté sous Eclipse gère les modules, interfaces, parseurs et lexeurs. Il existe deux processus de compilation : la compilation d'un fichier et la compilation d'un projet. Ils supportent tous deux une arborescence de fichiers sur plusieurs niveaux.

- La compilation d'un fichier

  Ce processus demande de calculer toutes les dépendances du fichier à compiler. Ceci est réalisé grâce à la commande *ocamldep*. Une fois les dépendances directes trouvées, il faut réitérer récursivement le processus avec chacune des dépendances. Ainsi on construit un arbre de dépendance pour le fichier à compiler.

  Donc pour compiler un fichier, il suffit de parcourir son graphe de dépendance et compiler les fichiers dans l'ordre du parcours du graphe.

- La compilation d'un projet

  Pour compiler l'ensemble d'un projet, il suffit de créer un graphe de dépendance pour tous les fichiers contenus dans le projet, puis de parcourir le graphe pour compiler les tous les fichiers du projet.

Lors de la compilation d'un fichier, si le fichier a déjà été compilé et que le source n'a pas été modifié depuis, alors on ne recompile pas le fichier. Ceci a pour but d'optimiser la compilation.

De plus l'utilisateur a le choix entre compiler en byte-code ou en natif. Cette option se fait grâce à l'interface des propriétés du projet.

La compilation est impossible s'il y a une détection de cycle dans le calcul des dépendances. Dans ce cas-là, un message apparaît dans la console caml pour prévenir l'utilisateur.

Classes utilisées :

ocaml.build.OcamlBuilder
ocaml.build.util.Deps
ocaml.build.util.Cycle

Wizards

Pour faciliter de nombreuses actions, Eclipse offre un système d'assistants (*wizards*). Pour gérer la création de fichiers et de projets caml, on a implémenté de nouveaux assistants. Ces derniers sont sous la forme d'une suite de boîtes de dialogues offrant toutes les options nécessaires. (voir figure 8)

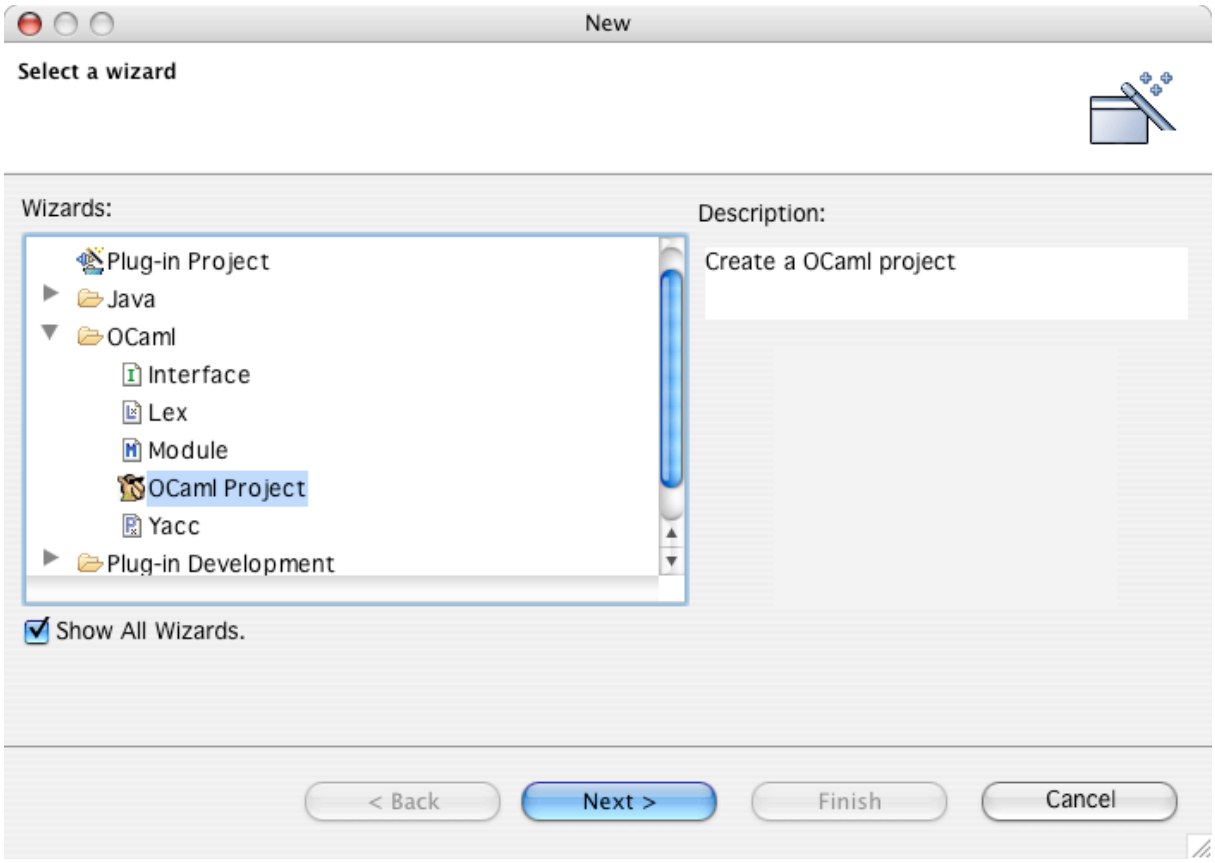


Figure 10 : listage de tous les wizards disponibles.

Sept nouveaux *wizards* ont été ajoutés répartis en deux catégories :

- Wizards de création de projets-fichiers qui étendent la classe org.eclipse.ui.wizards.newresource.BasicNewResourceWizard
- Wizrads d'importation et d'exportation de projets-fichers qui étendent respectivement la classe org.eclipse.ui.wizards.datatransfer.FileSystemImportWizard et org.eclipse.ui.wizards.datatransfer.FileSystemExportWizard

Classes utilisées :

Ocaml.wizards.OcamlExportWizard
Ocaml.wizards.OcamlImportWizard
Ocaml.wizards.OcamlNewInterfaceWizard
Ocaml.wizards.OcamlNewModuleWizard
Ocaml.wizards.OcamlNewProjectWizard
Ocaml.wizards.OcamlNewLexWizard
Ocaml.wizards.OcamlNewYaccWizard

Préférences et Propriétés

La fenêtre de préférences correspond à une fenêtre regroupant l'ensemble des propriétés d'un plug-in. Cette fenêtre est atteignable par le menu « Windows - Preferences ».

Dans le cas de l'utilisation du plug-in ocamlCompiler qui ne contient pas le compilateur nativement, il faut spécifier le chemin du compilateur ocaml et des outils utilisés par le plug-in. Lors de l'utilisation du plug-in ocamlCompiler contenant un compilateur, il n'est pas possible de renseigner les chemins d'accès.

On peut aussi spécifier les couleurs à utiliser dans l'éditeur de fichier caml. Pour que la modification des couleurs soit effective, il faut redémarrer Eclipse. En effet, celui-ci ne propose pas le changement dynamique de ses éditeurs.

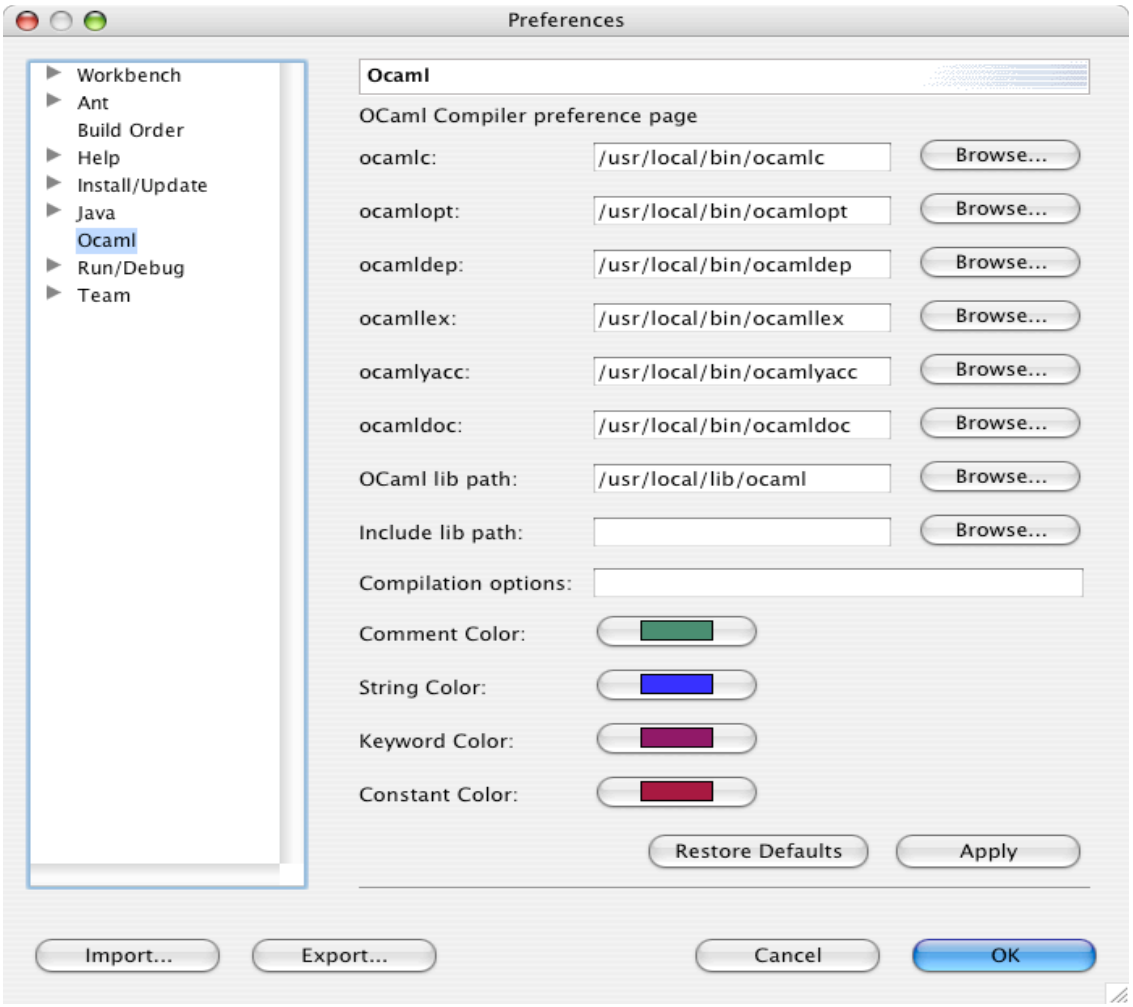


Figure 10 : préférences du plug-in ocamlCompiler version multiplateforme

Il est possible de spécifier un certain nombre de propriétés pour un fichier ou un projet. Pour cela il suffit de cliquer sur le bouton droit de la souris sur le projet ou le fichier et choisir « propriété » dans le menu.

Pour le projet, apparaissent les fichiers du projet qui peuvent devenir des exécutables lors de la compilation du projet. Il est aussi possible de définir le format de la documentation générée (html ou latex) et de définir le type du code généré lors de la compilation (Byte-code ou natif). L'option auto compilation permet de définir si lors de la sauvegarde d'un fichier, une compilation est faite automatiquement.

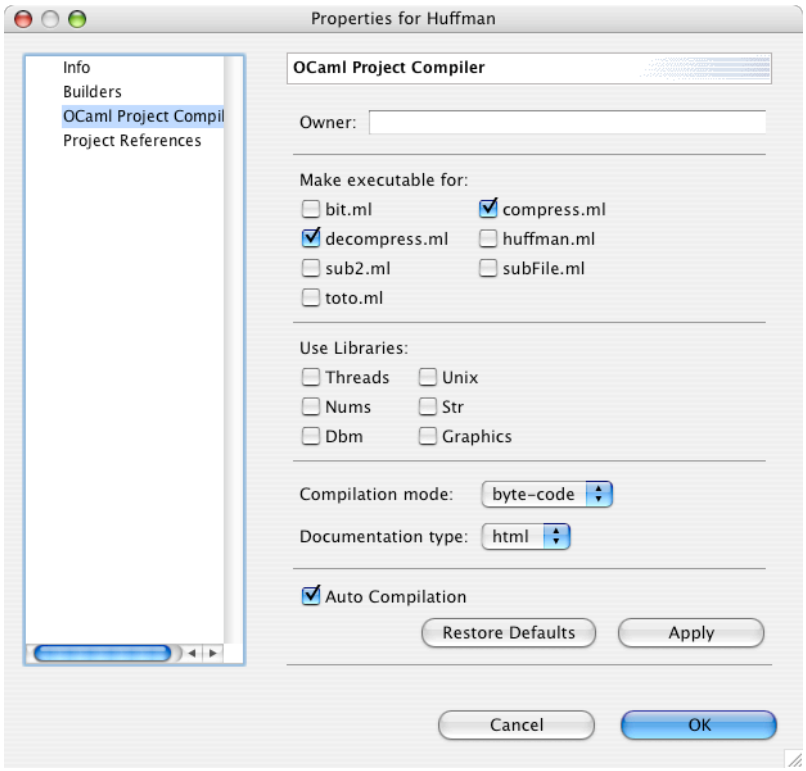


Figure 12 : propriétés d'un projet

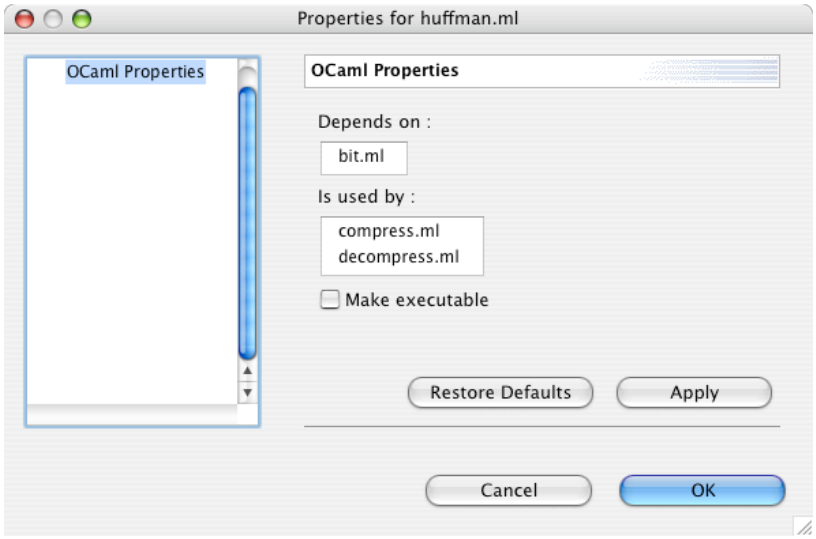


Figure 13 : propriétés d'un fichier

Importation / Exportation de projets

Pour importer un projet, il faut tout d'abord créer un nouveau projet caml puis aller dans le menu « File/import… », choisir le type OCaml project (figure 14). Ensuite il faut spécifier les fichiers que l'on veut importer et dans quel projet caml on veut les importer (figure 15).

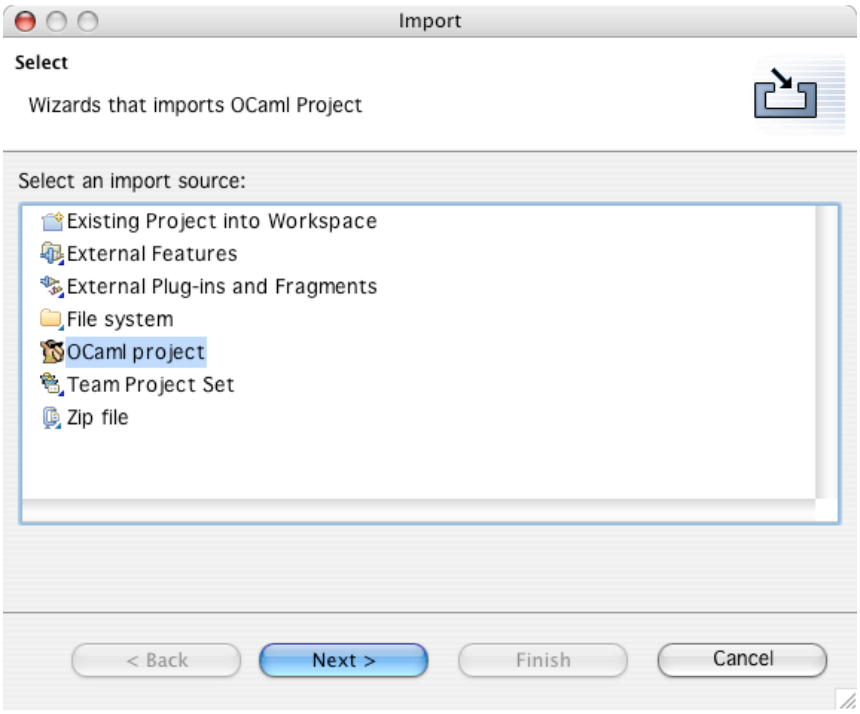


Figure 14 : importation d'un projet caml sous Eclipse.

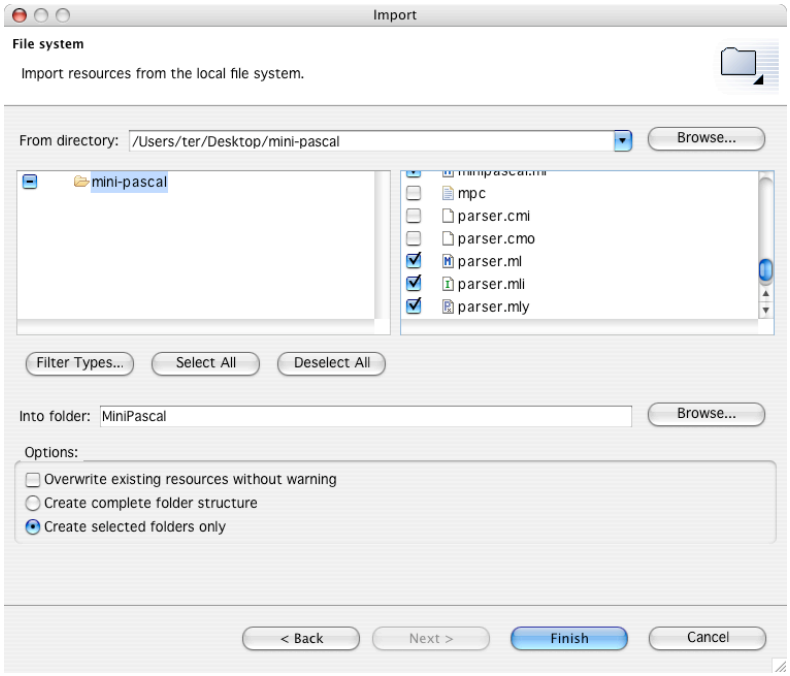


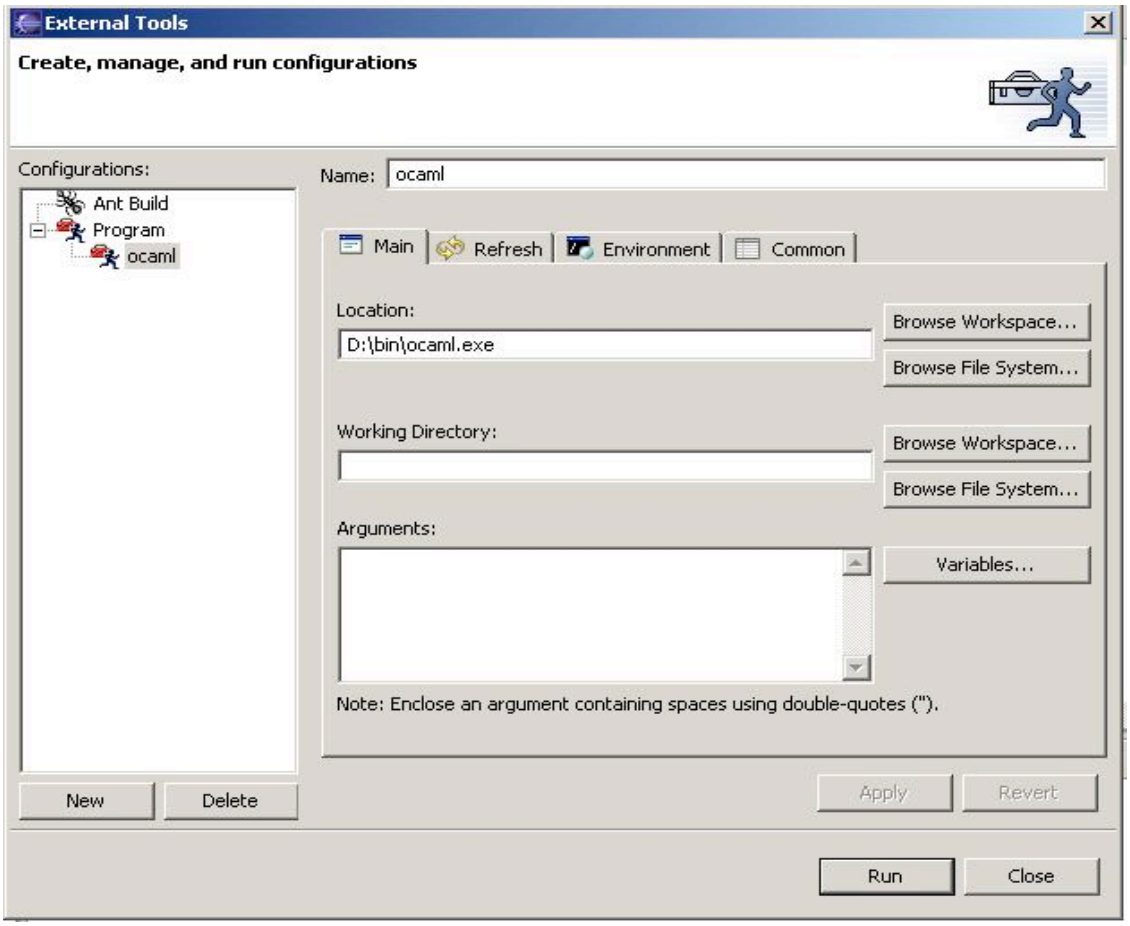
Figure 15 : importation d'un projet caml sous Eclipse.

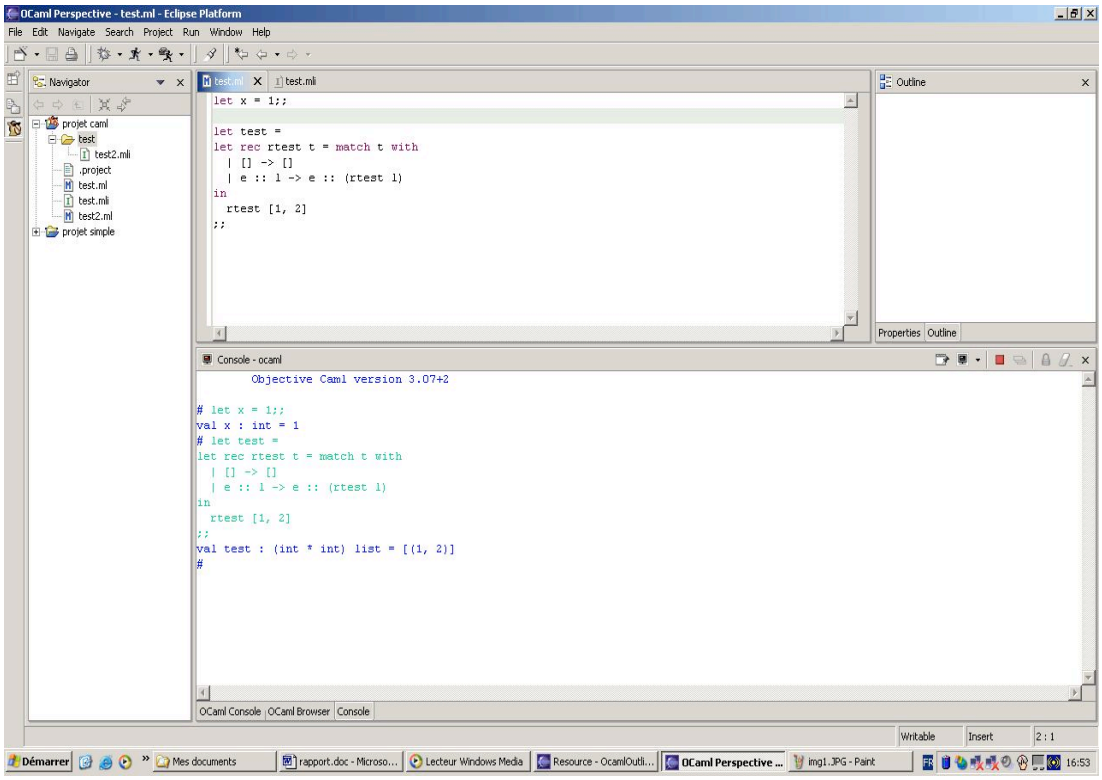
L'exportation se fait de la même façon que l'importation (menu « File/Export… »). A part qu'Eclipse compile l'ensemble du projet pour créer un fichier script *make.bat* pour compiler le projet facilement en dehors d'Eclipse.

Fonctionnalités utiles d'Eclipse

Eclipse permet de lancer des outils externes dans une fenêtre propre, au sein de son interface. Par exemple il est possible de lancer un shell dans une fenêtre et de travailler dessus. Cela est très utile car un utilisateur peut avoir ses outils dans son environnement intégré sans efforts. Il est donc possible d'intégrer très facilement un top-level pour exécuter son fichier. Par exemple, nous pouvons intégrer ocaml à l'environnement d'Eclipse.

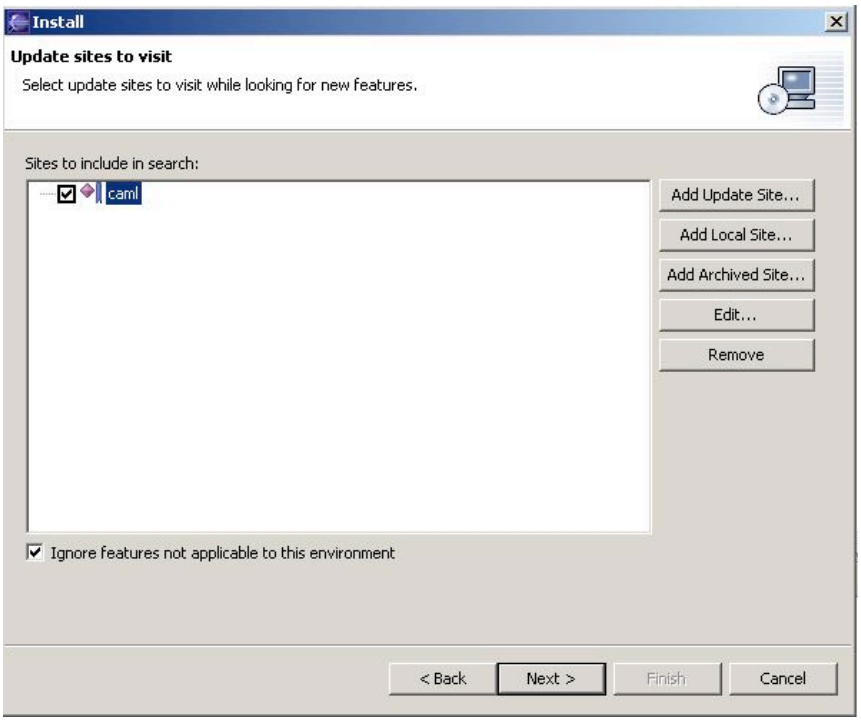
Il faut donc aller dans le menu « run – external tools ». Ensuite, il faut créer une nouvelle configuration de programme et mettre le chemin de ocaml. Il n'y a plus qu'à lancer la nouvelle configuration. Certains problèmes de rafraîchissement existent sous Mac.





La mise à jour de plug-in est un outil proposé par Eclipse. Ainsi la récupération des nouvelles versions des plug-ins est très facile. En effet, pour les personnes qui proposent le plug-ins, ils doivent le mettre en téléchargement sur un site web. Via l'interface d'Eclipse, il est alors aisé d'avoir cette nouvelle version.

Pour la mise à jour, il faut aller dans le menu « help – Software updates – find and install ». Alors il suffit de suivre le wizard proposé.



Ajouter une fonctionnalité : exemple de la complétion

Pour ajouter une fonctionnalité, il faut d'abord savoir quelles vues vont être modifiées ou crées. La complétion de notre éditeur doit proposer à l'utilisateur une liste de mots clé. Cette liste doit contenir la liste des modules de base du langage caml, la liste des mots du langage et la liste des modules du projet dont les fichiers d'interface ont été créés. Lorsque ces fonctionnalités sont définies, nous devons définir quelles sont les classes qui vont être utilisées. Il est évident que la vue de l'éditeur va évoluer. En effet, l'événement qui va faire apparaître la liste de propositions doit être capté par l'éditeur. Les informations de la liste sont trouvées grâce au plug-in « OcamlCompiler ». Ce module permet de trouver les informations sur les librairies, le compilateur et les outils associés. La classe qui permet de gérer la complétion doit implémenter l'interface « IContentAssistProcessor ». Cette interface déclare deux méthodes essentielles :

- **« computeCompletionProposals »** qui retourne une liste de propositions qui sont du type « ICompletionProposal ». Nous devons créer une classe qui va implémenter l'interface « ICompletionProposal ». Nous allons décrire un des procédés pour trouver une des listes de mots.
  Par exemple, si la liste des modules standard doit être affiché, pour récupérer cette liste, nous devons faire appel à la classe « OcamlCompiler » qui connaît tous les chemins. On utilise alors la méthode static « getLibPath() » qui retourne la localisation des librairies standard OCAML. Il suffit alors de prendre tous les fichiers avec l'extension « mli » et de retourner leur nom privé de l'extension. Pour que la liste se réduise au fur et à mesure de la frappe, il faut trouver la séquence de caractères entrées précédemment. Ensuite, une simple épuration suffit pour faire se réduire la liste. L'interaction avec l'éditeur se fait automatiquement.
  Pour l'affichage des modules du projet, la méthode « getMlMliFiles() » de la classe « OcamlProject » sera utilisée car elle renvoie la liste des fichiers avec les extensions « ml » et « mli » du projet en prenant en compte des sous dossiers.

- **GetCompletionProposalAutoActivationCharacters** qui retourne la liste des caractères qui déclenchent l'affichage de la liste.

Chaque élément de la liste est une instance d'une classe qui implémente l'interface « ICompletionProposal ». Cette classe contient les informations d'une ligne : le texte à afficher dans la liste de proposition, le texte à remplacer, l'image à associer à la ligne, …. La méthode « apply » est la méthode appelée lorsque la ligne est sélectionnée dans la liste. Cette méthode est importante car il est important de faire en sorte de ne pas réécrire le début du mot. L'éditeur est passé en paramètre de la fonction ainsi que l'emplacement de la prochaine lettre à écrire dans la page. Avec un petit calcul, nous pouvons écrire le mot que nous désirons en écrasant les lettres précédentes.

Pour prendre en compte une combinaison de touches pour faire apparaître la liste de proposition, il est impossible de passer par la méthode décrite précédemment. Par exemple pour faire afficher la liste lorsque l'on tape « ctrl – espace », alors il faut capter l'événement au niveau de l'éditeur. L'éditeur doit alors implémenter l'interface « KeyListener ». Dans

cette interface, la méthode « keyReleased () » nous intéresse tout particulièrement. C'est grâce à cette méthode que l'on peut capter que deux touches ont été sélectionnées en même temps. Il suffit ensuite de faire un masque pour savoir si c'est bien la séquence de touche qui nous intéresse. Si c'est le cas, alors on peut appeler la méthode « doOperation () » implémenté de « ItextOperationTarget » par Eclipse. Cet appel se fait avec le code « 13 » pour signaler que l'opération à effectuer est une complétion. Alors, le mécanisme de complétion décrit précédemment peut se mettre en place.

Ceci décrit brièvement la description de l'intégration de la complétion au sein du projet. Le travail le plus conséquent est la recherche d'information car les quelques interfaces et classes présentées ici sont noyées dans un flot d'informations conséquent.

Comparaison du plug-in V1.0 au plug-in V2.0

Le plug-in que nous avons repris ne marchait pas en grande partie à cause du changement de version d'Eclipse. Nous avons donc dû refaire en grande partie le plug-in. Mais nous nous sommes tout de même inspirés fortement de certaines parties comme la compilation ou le Outline. Des fonctionnalités ont été rajoutées par rapport à la première version : la complétion est plus complète, le ocaml browser a été intégré, la génération de type, ….

Le tableau suivant résume les fonctionnalités créées et compare les deux versions du plug-in.

Fonctionnalités	**Plug-in Version 1**	**Plug-in Version 2**
Éditeur		
Coloration syntaxique	✔	✔
Complétion		✔
Affichage des types		✔
Exécution d'actions		✔
Gestion de projet		
Création projet/fichier caml	✔	✔
Import/Export		✔
Menu popup actions		✔
Gestion des arborescences		✔
Génération de la documentation		✔
Préférences	✔	✔
Compilation		
Fichier ml	✔	✔
Fichier lexer/parser		✔
Fichier gestion des dépendances	✔	✔
Gestion des erreurs	✔	✔
Mode decompilation	✔	✔
Outline	partiel	✔
OcamlBrowser		✔

Installation du plug-in

Pour installer le plug-in, il suffit de copier le dossier « ocaml » et « ocamlCompiler » dans le dossier « plugins » d'Eclipse. Pour le dossier « ocamlCompiler », il faut choisir la version sans le compilateur caml intégré (donc indépendant de la plateforme) ou avec le compilateur (dépendant de la plateforme). Finalement, il ne reste plus qu'à exécuter Eclipse pour finaliser l'installation. Pour la version sans compilateur, on doit spécifier les différents chemins du compilateur dans le menu « Windows » -> « Preferences » -> « Ocaml ».

Méthodologie et organisation du travail

Quatre étapes se sont distinguées dans le déroulement du TER :

1. La première étape consista à la prise en main de la plateforme Eclipse et à la compréhension de ses mécanismes de fonctionnement. Ce fut un long travail qui nous a pris environ 2 mois car la documentation d'Eclipse est peu claire et très complexe. Nous avons étudié la conception et la création de petits plug-ins grâce aux tutoriaux offerts par Eclipse.
2. Ensuite nous avons analysé le travail qui a été réalisé lors de la première version d'Eclipse.
3. Pour faire le portage du premier plug-in sous la version 3 d'Eclipse, nous avons déterminé les différences entre les versions 2 et 3 d'Eclipse. Nous avons constaté qu'il n'était pas possible de reprendre l'architecture du premier plug-in car il a eu trop de modifications faites entre deux versions d'Eclipse.
4. La partie d'analyse étant terminé, nous avons pu commencer à coder. Ce fut une longue tâche à réaliser. Notre plug-in se résume à environ 6000 lignes de code java réparties sur 60 fichiers.

Conclusion

Ce plug-in est utilisable pour faire l'ensemble des projets de cette année sur toutes les plateformes : linux (tests en salle info), Windows ou Mac. De plus, il apporte certaine facilité de développement grâce à des outils comme la complétion ou la visualisation des types pour chaque expression. Ce plug-in peut ainsi apporter un apport précieux au développeur débutant le langage OCAML. Cet outil est donc un outil fini. Mais de nombreuses améliorations peuvent êtres apportées comme le formatage ou le refactoring (le renommage d'une déclaration entraîne le renommage de toutes ses utilisations). D'autres outils comme une interface interactive pour faire des applications graphiques peuvent aussi apporter beaucoup au plug-in. L'intégration d'une fenêtre de débuggage peut aussi être très utile et agréable lorsque l'on travaille sous linux ou Mac. En effet, Windows ne possède pas de débuggeur caml.

L'essentiel du travail a été la découverte et la compréhension de la structure d'Eclipse. En effet, le développement de plug-in n'est pas aisé à cause d'un cruel manque de documentation. L'exploration se fait donc en aveugle et fait perdre beaucoup de temps.

Bibliographie

Ocaml

- *Développemnt d'applications avec Objective Caml* par E.Chailloux et P.Manoury et B.Pagano, édition O'Reilly.

Eclipse

- Projet Eclipse : http://www.eclipse.org
- TER : plugin ocaml pour Eclipse version 1 par Alexendre Deckner, Nicolas Deckner et Léonard Dallot.
- Creating Eclipse plugins : http://devresource.hp.com/drc/technical_articles/ePlugIn/index.jsp
- SWT : http://www.swt-designer.com

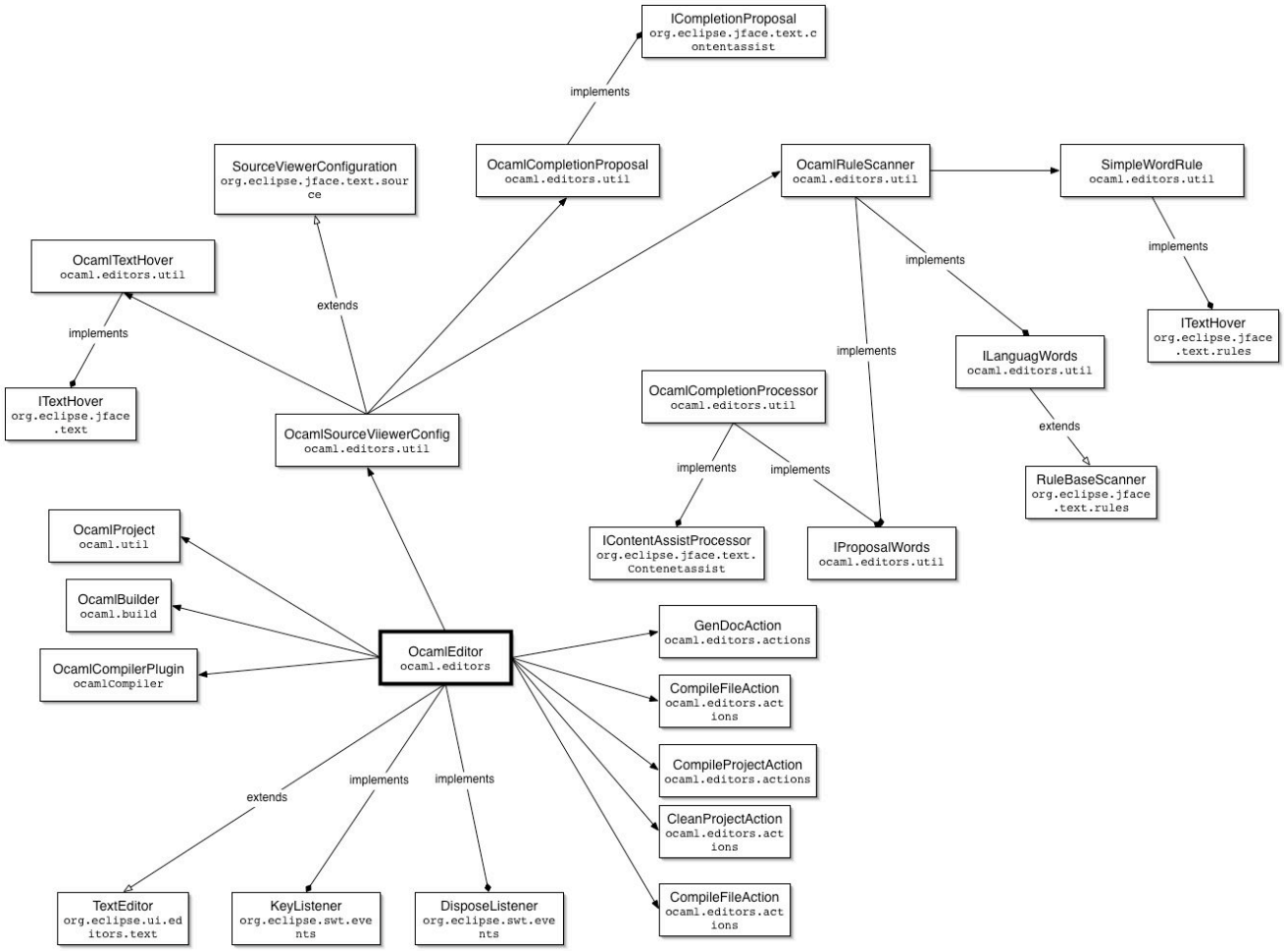
Java

- Sun Microsystems Java : http://java.sun.com

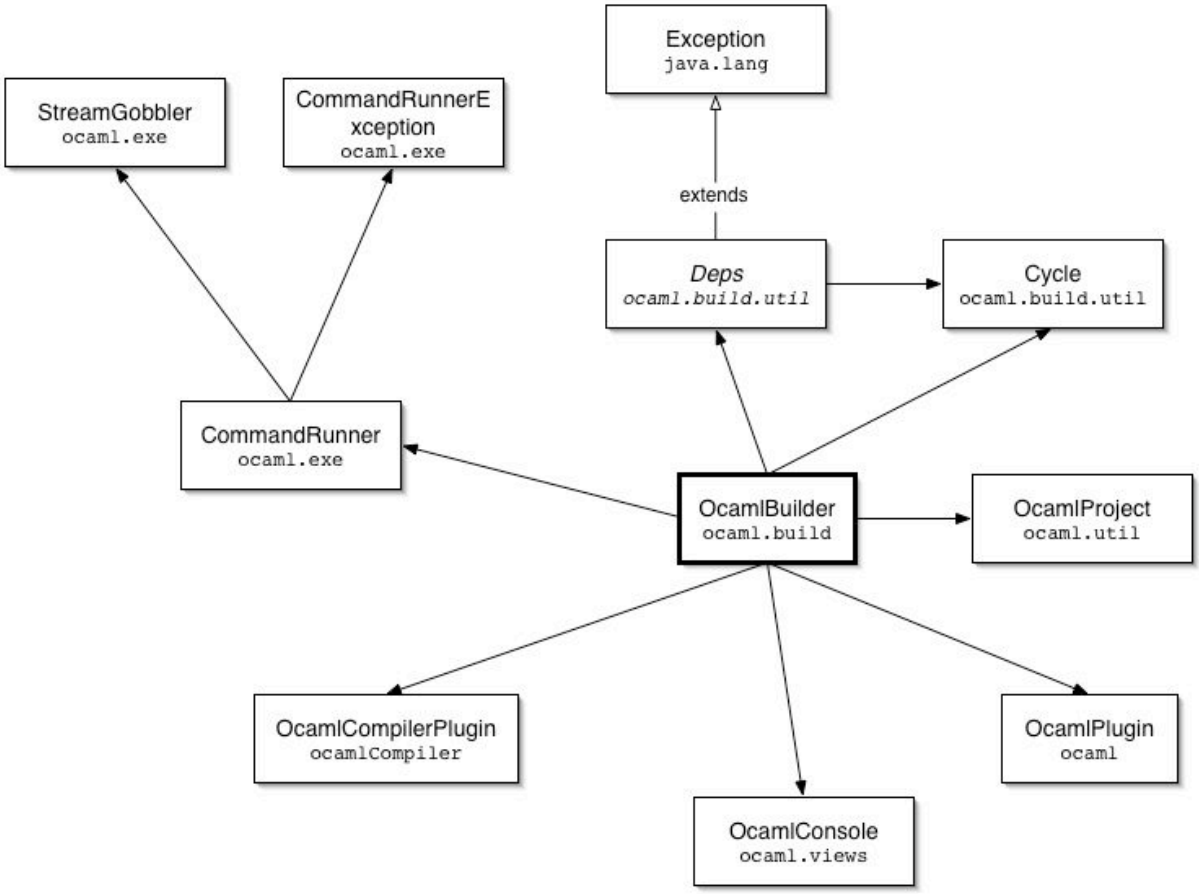
Annexes

Graphes de dépendance de classes

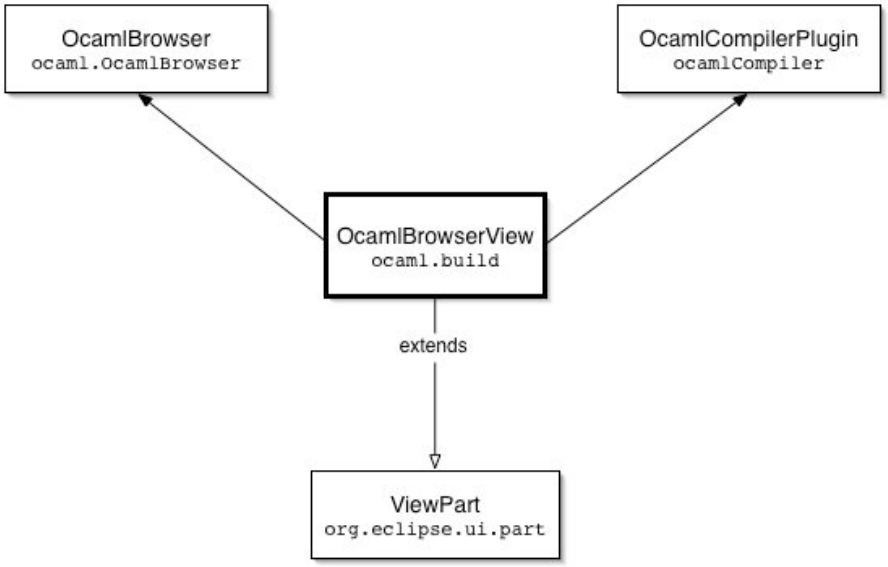
OcamlEditor



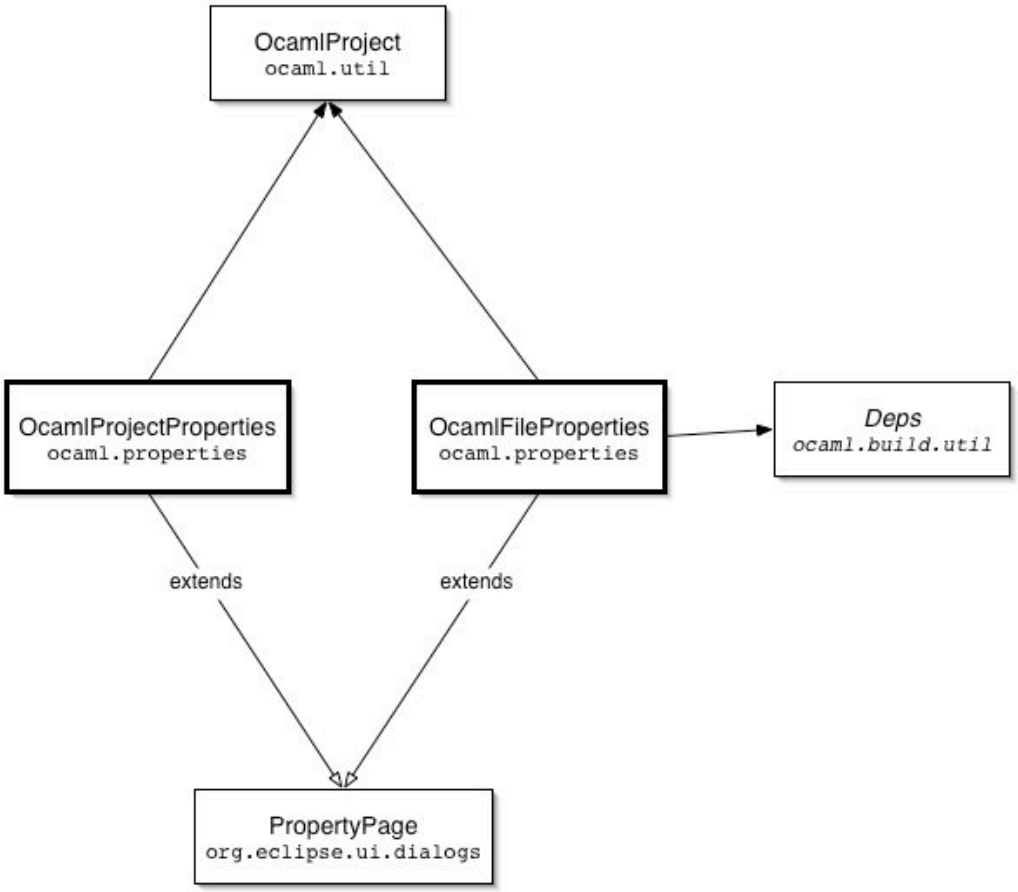
OcamlBuilder



OcamlBrowserView



OcamlFileProperties et OcamlProjectProperties



Wizards