Designing a Development Environment for Logic and Multi-Paradigm Programming

Giulio Piancastelli[1] and Enrico Denti[2]

[1] DEIS – Alma Mater Studiorum – Università di Bologna
via Venezia 52, I-47023 Cesena, FC, Italy
[2] DEIS – Alma Mater Studiorum – Università di Bologna
v.le Risorgimento 2, I-40136 Bologna, BO, Italy

Abstract. The Eclipse platform has been extended to provide integrated development environments for many different languages and systems. Declarative programming, however, and in particular logic languages, has still to benefit from the state-of-the-art Eclipse infrastructure supporting a huge number of development activities. We set out to design an environment for logic programming built around tuProlog, a Java-based light-weight Prolog engine that provides seamless integration into the platform and promotes an innovative form of interaction with the interpreter, allowing a multiplicity of independently configurable instances to be exploited within the same project. Moreover, we use the Java/Prolog multi-paradigm capabilities of tuProlog as a case study for discussing the integration of two different programming languages in a single environment. Finally, we compare our environment to some related projects in the logic programming context.

1 Motivation and Background

Eclipse is well-known as an open and extensible integrated development environment (IDE) [1] supporting many languages and systems with varying degree of quality and completeness—from Java and C/C++ with official Eclipse packaging and download, to scripting languages such as Python and Web programming languages such as PHP, up to niche languages such as Fortran, directly incubated within the Eclipse Foundation. However, languages inspired to the declarative programming paradigm, and in particular logic programming languages such as Prolog, have still to benefit from the extensive support given by the Eclipse platform to build full-fledged open source development environments.

Indeed, among the major Prolog implementations [2], the interaction with the interpreter is typically reduced to a command line prompt in a text-based console, with no support for writing simple programs or more complex software; when an editor is available, it often suffers from the many limitations and shortcomings of in-house applications, each being rebuilt from scratch and following its own conventions; useful and innovative tools are provided only in few rare cases. Instead, building an IDE on top of Eclipse would mean not only to exploit

a state-of-the-art infrastructure for the support of a huge number of programming activities, but also to follow a series of user interface standards and usage workflow guidelines that should ease the impact of the tool on developers and make them feel more comfortable working with it.

The purpose of our research is to build a development environment for both logic and multi-paradigm logic/object-oriented programming on top of the Eclipse platform. To this aim, we mean to exploit tuProlog,[3] an open source light-weight Prolog engine, as a case study to analyse and evaluate the many peculiarities of the integration of a logic-based language and support within Eclipse. We chose tuProlog both because it is written in Java, thus representing the easiest choice for platform integration, and for its engineering properties, which seemed ideal to explore innovative forms of interaction with the interpreter: in particular, *minimality* and *configurability* [3] let developers spawn a multiplicity of tuProlog engines, each with its own set of libraries and theories, within the same Prolog project, with the aim of exploring, tracing, or comparing different programming solutions. Moreover, multi-paradigm Java/Prolog support on the tuProlog side [4] calls for an adequate counterpart on the development environment side, introducing further requirements that a high quality instrument such as the JDT (Java Development Tools) plug-in helped satisfy.

In this paper, we discuss the features, the objectives, and the design of such a logic-based multi-paradigm IDE: we first present the design of the whole environment supporting multi-paradigm Java/Prolog programming (Sect. 2), then illustrate the different parts composing the current prototype for the logic side of the IDE, emphasising the innovative way of interacting with multiple tuProlog engines embedded in Eclipse (Sect. 3); finally, we briefly discuss some related projects (Sect. 4), and draw some conclusions for future work (Sect. 5).

2 An Environment for Multi-Paradigm Programming

A programming paradigm is a style to represent and organize computations. For example, logic programming [5] realises computations as deductions over a knowledge base formed by facts (also known as extensional knowledge) and rules (intensional knowledge) representing relations on which queries can be issued; object-oriented programming [6] organises computational activities into hierarchies of entities (objects) which encapsulate state and methods to manipulate it. Multi-paradigm programming is the integration of two or more paradigms in a unique model [7] that may be realised within a single language by following a multi-style approach, or within a system by following a multi-language approach.

tuProlog supports multi-paradigm programming by integrating the logic and object-oriented paradigms of Prolog and Java at two different levels [4]: at the system level, where Java and Prolog can be combined to build tuProlog libraries, which can therefore be designed taking the best of both worlds; and at the application level, where bidirectional Java/Prolog integration can enable both

[3] `http://tuprolog.alice.unibo.it`

Java items to be effectively accessed from a Prolog program and Prolog engines to be dynamically exploited from a Java program.

Multi-paradigm programming achieved by integrating two or more languages needs to be directly supported in its own right by a development environment, next to tools provided for the combined languages on their own. In the case of tuProlog, the integration at the application level is managed by the environments dedicated to each language, granting access to Java resources on the Prolog side and to Prolog engines on the Java side. Instead, the combination of Java and Prolog at the system level needs to be explicitly supported as a multi-paradigm programming feature in an IDE, supporting the creation of libraries and making them available within the environment. Thus, an IDE designed to enable multi-paradigm Java/Prolog programming based on tuProlog has to supply tools for: *(i)* Java development; *(ii)* Prolog development; *(iii)* supporting the integration of the two languages by exploiting the tuProlog library mechanism.

The Java Development Tools (JDT) available within the Eclipse platform already deliver a set of high quality instruments that help developers deal with the many activities of Java programming. On the Prolog side, the objective is to provide a comparable set of tools for tasks related to logic programming. Such a set would include wizards for the creation of the basic building blocks of Prolog software: new projects, as containers of all the resources needed for a certain application; new logic theories/modules, and new predicates that would also possibly need a visibility indicator in order to be imported/exported. Modules and theories need to be navigated with ease independently from the amount of code they contain: for this purpose, an outline view should provide a list of defined predicates (further subdivided in facts and rules) next to another view integrating cross-reference functionalities, so as to allow developers to reach the clauses of a predicate used in a theory but possibly defined in a different module.

Editing Prolog programs should be supported by syntax highlighting, and by a code completion feature triggered to automatically enter the invocation of a predicate and step through the insertion of parameters by means of a template. General useful mechanisms for code editing include checking for predicates defined in some module but not used anywhere in the application, and for invoked predicates that have never been defined, alongside automatic importing management for any given theory. Syntax errors and possible suggestions for corrections should be indicated on the fly, while the developer is typing code in the editor. Among syntax warnings, specific logic programming issues should be considered, such as the use of singleton variables in the definition of a predicate clause.

More advanced features include refactoring and debugging. Refactoring is a technique for improving code design without altering its external behaviour; such technique is a preliminary step to take before performing changes required to add new functionalities or improve efficiency [8]. From the object-oriented field, this practice recently came to prominence for Prolog programming [9]. Similarly to the JDT, Prolog development tools should include a facility to automate code refactoring, from simple renaming to predicate extraction, up to more complex transformations involving cut, if-then-else, and unification. Debugging should

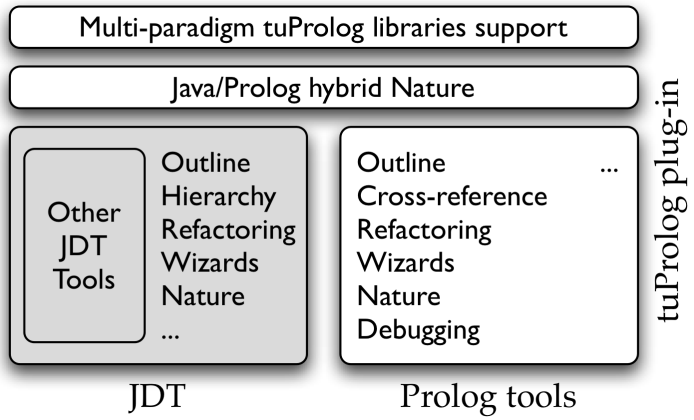

Fig. 1. The design of Java/Prolog multi-paradigm environment based on the Eclipse platform. Components of the final tuProlog plug-in are underlined by white background.

exploit the instruments supplied by the Eclipse platform to improve the current state of inspection and manipulation of the demonstration process, also allowing dynamic configuration of the tool, possibly by means of a special logic theory.

The Java/Prolog multi-paradigm programming support has to be placed on top of the JDT and the above-cited set of Prolog development tools. The integration between the Java side and the Prolog side of the environment needs to be first performed at the *nature* level. A nature is used to configure the capabilities of a project, typically to associate behaviour and functionalities in the form of builders that process modified files at specific times [10]. The Eclipse concept of natures is the straightforward counterpart of the multiple traits of a programming system involving more than one paradigm. Accordingly, next to the Java and Prolog nature in a multi-paradigm project, we need to provide an additional nature to carry out tasks belonging to the special hybrid quality of the project. In the case of tuProlog projects, one such task is to make the Java libraries under development immediately available on the Prolog programming side, without forcing the user to explicitly deal with classpath settings. Multi-paradigm support is further pursued by wizards and templates for creating new Java/Prolog libraries and new predicates, functors, directives, operators within the library. Moreover, an outline view of a hybrid library should show its contents as a list of logic programming elements rather than Java fields and methods.

The abstract design of the integrated development environment is presented in Fig. 1. The software shall be delivered in two different but integrated sets of plug-ins: the Prolog development tools form an environment for Prolog programming on their own, and should be made available both as an Eclipse plug-in and as a Rich Client Platform application, in order to deploy only the minimal set

of Eclipse features to users interested only in Prolog applications; the *tuProlog plug-in*, comprising the Prolog tools and the multi-paradigm Java/Prolog functionalities based on tuProlog, has to be deployed only as a platform extension, so as to exploit the JDT already bundled with the standard Eclipse download.

3 The tuProlog Plug-In for Eclipse

The current prototype of the tuProlog plug-in for Eclipse is composed of a single feature comprising three different components:

- `alice.tuprolog` makes tuProlog available as a software library within the platform, as suggested by Eclipse plug-in development best practices [10]
- `alice.tuprologx.eclipse` contains everything related to the interaction between tuProlog and Eclipse, including the user interface
- `alice.tuprologx.eclipse.doc` integrates documentation, in the form of a getting started guide for the plug-in and relevant parts of the tuProlog guide, with the Eclipse help system

The `alice.tuprologx.eclipse` component realises platform integration by the typical elements taken from the Eclipse framework: it provides a *perspective*, two *wizards* to help creating projects and logic theories that can be also invoked from buttons in a *toolbar*, a *nature* and a *builder* associated to Prolog projects, an *editor* with syntax highlighting and a basic form of syntax checking for the Prolog language, *launch configurations* to execute queries, *property pages* to change editor options and manage the possibly many tuProlog engines associated with a single project. The component also provides four *views* to display information related to Prolog development and the execution of Prolog programs:

- the *Clause* view accesses the AST of the Prolog program in the selected editor buffer to show its outline in terms of clauses identified by their head;
- the *Theory* view displays at a glance the content of all the tuProlog engines associated with the current project in terms of logic theories;
- the *Query* view provides the history of the logic queries asked within the context of the current project;
- the *tuProlog Console* view allows to select one of the possibly many tuProlog engines that have been asked a query and see the solution, output text, and tracing information derived by standard predicates such as `spy/0`.

3.1 Interaction with tuProlog Engines

The structure of Eclipse development environments and the typical approach to Prolog programming suggest a well-defined workflow to follow within the tuProlog plug-in. First, developers are expected to open the tuProlog perspective, so that tools such as views and buttons are displayed in the most suitable arrangement. Then, by using the available wizards, developers can create one or more Prolog projects, and any number of logic theories that will be immediately

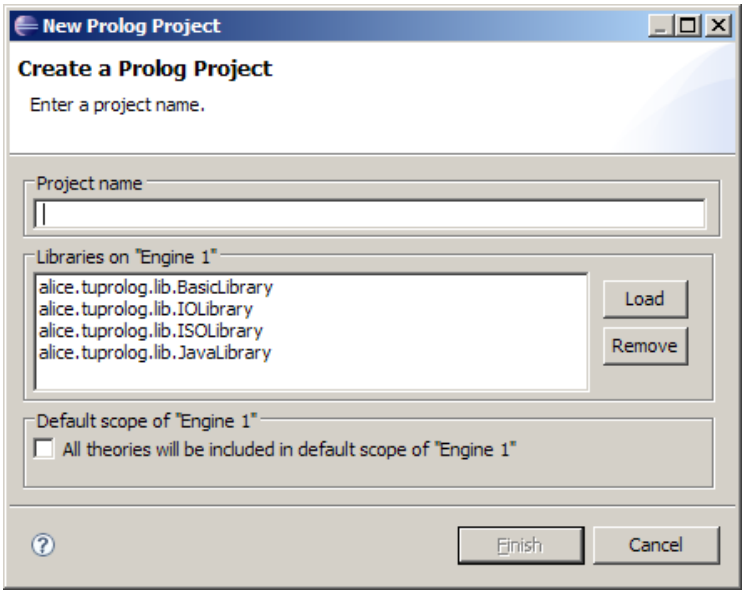

Fig. 2. The Prolog project wizard available in the tuProlog Eclipse plug-in.

opened in the provided Prolog editor. The Prolog project wizard, depicted in Fig. 2, is also the first place where the configuration of tuProlog engines occurs.

Any Prolog project comes with an associated tuProlog engine; thus, the project wizard offers an interface to configure the engine in terms of *libraries* and *theories*. Libraries group related predicates, operators, functors, and directives into stable sets of functionalities that can be dynamically loaded or unloaded at any time during the life of an engine. tuProlog comes already equipped with four libraries, providing the basic characteristics needed for Prolog programs execution, input/output operations, as well as features dictated by the ISO standard [11], and support for multi-paradigm Java/Prolog programming. Any of these pre-loaded libraries can be unloaded so as to configure a lighter engine; conversely, any new library can be loaded into it, provided it has been made available on the classpath. With respect to logic theories, the only configuration option at project creation time is to decide whether all the theories contained in the project need to be fed to the engine as an effect of the builder activities.

After creating a Prolog project and logic theories, developers may add additional tuProlog engines to the project and act on their configuration settings. These operations are performed through the proper page in the Prolog project properties, shown in Fig. 3. It is possible to create a new engine by providing a unique name to identify it; unlike the first tuProlog engine in a project, subsequently added engines do not come with the four libraries available in the distribution pre-loaded by default. Then, as for any other engine in a Prolog project, it is possible to manipulate the libraries configuration by loading and

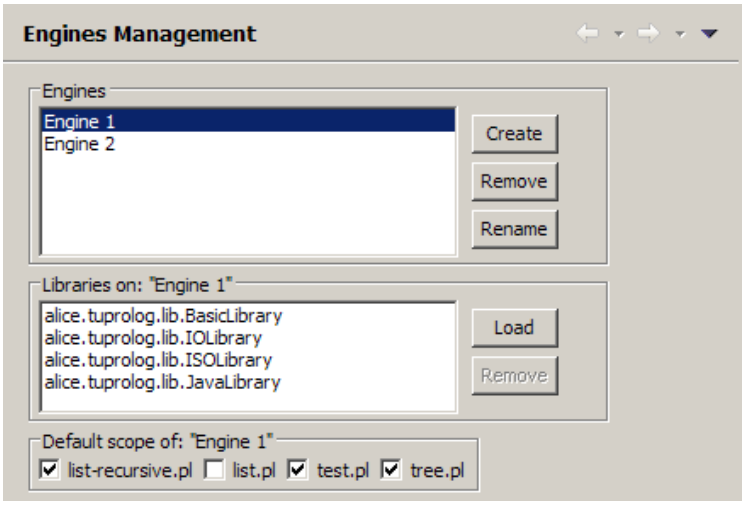

Fig. 3. The Prolog project property page where engine configuration takes place.

unloading, and to set the *default scope* in terms of logic theories contained in the project—that is, to select those theories that need to be automatically loaded into the engine at project building time. Any tuProlog engine in a project can be renamed or removed, provided that the new name does not clash with existing engines, and that at any time at least one engine is active within the project.

The last step in the development workflow is to issue queries to the whole set (or to a subset) of the tuProlog engines defined in the project. Queries are entered through the launch configuration dialog supplied by Eclipse, that the tuProlog plug-in extends to introduce its own launch configuration type (Fig. 4). First, the project where the query execution will take place needs to be selected. Then, the scope of the query, in terms of engines and theories against which it will be asked, has to be delimited: by default, each tuProlog engine uses the set of theories chosen in the project properties; however, an alternative, temporary scope can be defined by selecting again the theories to be used as the knowledge base only for this particular query, (Fig. 5). Finally, the Prolog query can be entered by means of a small dialog window (Fig. 6) that can detect Prolog syntax errors and avoid submission if the query is not written correctly. By extending launch configurations to execute Prolog queries, Eclipse facilities can be exploited such as the Run button and dropdown menu to re-run the last query or run a query from the history of recently launched elements.

4 Discussion

tuProlog itself comes distributed with two user interfaces: a command-line console and a simple GUI based on the Swing graphical toolkit. The GUI works on a single tuProlog instance and, despite offering an uncommon table-oriented

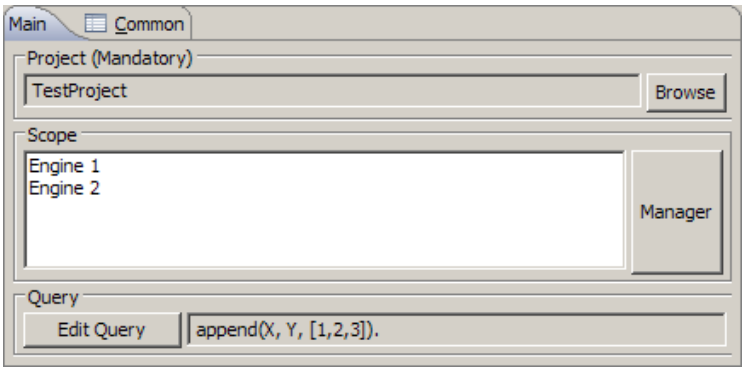

Fig. 4. The Launch Configuration to set up to execute a query within a Prolog project.

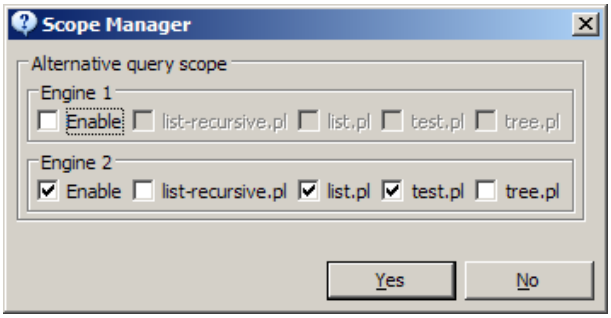

Fig. 5. Through the Scope Manager dialog a temporary query scope can be set for a Prolog engine, different from the default provided at the time of engine creation.

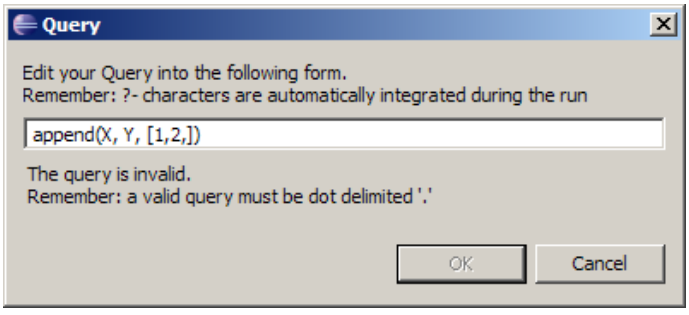

Fig. 6. The Query editor is able to automatically check for Prolog syntax errors and avoid submission if the query is not written correctly.

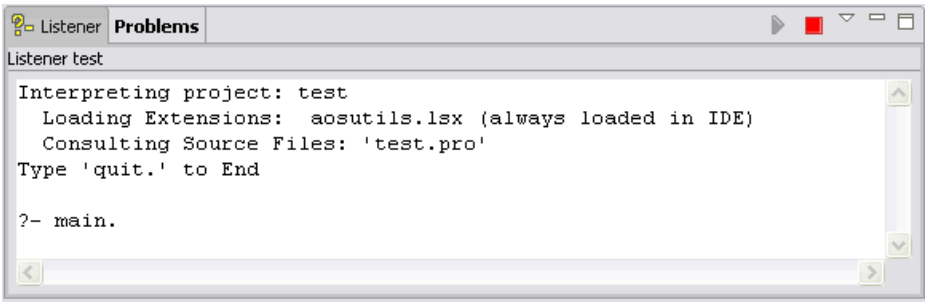

Fig. 7. The command-line interface for engine interaction within Eclipse delivered by the Amzi! Prolog integrated development environment.

display of query solutions, it is oriented to developers that wish to immediately experiment Prolog programming after downloading the interpreter. The GUI is designed to edit a bunch of Prolog files in the same window, but it has no notion of projects, thus being unsuitable for medium- to large-size applications. Besides these two interfaces, it must be noted that a plug-in based on the NetBeans platform [12] has been under development for a couple of internal milestones; however, due to the lack of extended documentation, sufficiently structured literature, and a large and thriving community, so far the development of the NetBeans plug-in has been harder and its appeal less than expected.

Among other major Prolog implementations [2], the most widely spreaded user interface is a command-line prompt in a text-based console. Some systems just provide that kind of interface to Prolog developers: Ciao Prolog, GNU Prolog, YAP Prolog on the open source side, and SICStus Prolog on the commercial side, do not offer any other means of interacting with their Prolog engine, nor do they provide any facility for editing Prolog code. Other implementations supply graphical interfaces for Prolog programming, and even some innovative and useful tools for inspection, profiling, and debugging. For example, the open source SWI-Prolog provides a graphical tracer with embedded stack view, an execution profiler, a cross-reference dependencies analyser, and a navigator to outline (directories of) Prolog programs; Strawberry Prolog, a commercial implementation with a light version downloadable for free, offers debugging features and the display of the proof tree at runtime. However, all such tools suffer from the disadvantages of in-house applications: they hardly have a uniform, standard interface to present to the user; they are often not sufficiently integrated with each other; and they are not cross-platform, being instead programmed with niche graphical toolkits (SWI-Prolog tools are based on XPCE) or available only for specific operating systems (Strawberry Prolog just runs under Windows and Linux).

The only Prolog implementation to build its graphical environment upon an established platform such as Eclipse is Amzi! Prolog, a commercial distribution developed by Amzi! Incorporated.[4] The Amzi! Prolog IDE provides wizards for

[4] `http://www.amzi.com`

the creation and import of projects and files, an editor with syntax highlighting and syntax error markers, a full-fledged integrated debugger, outline and cross-reference views, and facilities for compilation of libraries and applications. Despite the supply of many more tools with higher quality than our initial prototype, the interaction with the Prolog engine always occurs through a text-based console that represents a strict porting of the command-line interface distributed with any other Prolog implementation (see Fig. 7). Moreover, launch configurations are not really exploited, and developers are always limited to have only one Prolog engine active at a time within the same project.

5 Conclusions and Future Work

We are currently testing the prototype environment in some Prolog development scenarios derived from teaching and research settings. In the immediate future, both a deeper integration of some of the existing tuProlog plug-in tools (e.g. better mechanisms for configuring engines and browsing their knowledge base, platform-wide access to the Prolog AST) and extension towards the multi-paradigm side of the project will be pursued.

References

1. McAffer, J., Lemieux, J.M.: Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications. Addison-Wesley (2005)
2. Wikipedia: Prolog. Available online at `http://en.wikipedia.org/wiki/Prolog`. Retrieved on August 2nd, 2008
3. Denti, E., Omicini, A., Ricci, A.: tuProlog: A light-weight Prolog for Internet applications and infrastructures. In Ramakrishnan, I., ed.: Practical Aspects of Declarative Languages. Volume 1990 of LNCS. Springer (2001) 184–198
4. Denti, E., Omicini, A., Ricci, A.: Multi-paradigm Java-Prolog integration in tuProlog. Science of Computer Programming **57**(2) (August 2005) 217–250
5. Doets, K.: From Logic to Logic Programming. The MIT Press (1994)
6. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1997)
7. Müller, M., Müller, T., Van Roy, P.: Multiparadigm programming in Oz. In Smith, D., Ridoux, O., Van Roy, P., eds.: Visions for the Future of Logic Programming: Laying the Foundations for a Modern successor of Prolog. (December 1995) Workshop in association with International Logic Programming Symposium (ILPS'95).
8. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
9. Schrijvers, T., Serebrenik, A.: Improving Prolog programs: Refactoring for Prolog. In Demoen, B., Lifschitz, V., eds.: Logic Programming, 20th International Conference, ICLP 2004, Proceedings. Volume 3132 of LNCS., Springer (2004) 58–72
10. Gamma, E., Beck, K.: Contributing to Eclipse: Principles, Patterns, and Plug-Ins. Addison-Wesley (2004)
11. ISO/IEC: Information technology — Programming languages — Prolog — Part 1: General core. International Standard ISO/IEC 13211-1 (June 1995) Joint Technical Committee ISO/IEC JTC1 SC22.
12. Boudreau, T., Tulach, J., Wielenga, G.: Rich Client Programming: Plugging into the NetBeans Platform. Prentice Hall (2007)