Split along Technology not Language

By Michael Scharf, October 2005

Abstract

In the first part of this paper, we will describe the problems of having the same technology duplicated in different language toolkits. The focus of each language toolkit is to completely supporting everything needed to develop in that language. This leads to duplication of subsystems. Extending or replacing subsystems becomes difficult, because each language has its own set of extension points. We propose to split along technology lines for components that are programming language independent.

In the second part of this paper, we will focus on two examples of technologies that are relative language independent: code editors and build systems. We propose an editor helper plugin that provides some extension points (for editor extensions) and helper classes to be used by editor implementers to plug the extensions into their editors. For the build system we propose a build support plugin that provides an API to be used by plugins that need information from the build system (AST and DOM parsers). Build system implementers would have to provide those APIs.

Part I

The problem

A multi language solution faces two different problem spaces:

1. How to efficiently (quickly) write a new DT by reusing toolkits and infrastructure?
2. How to split existing language DTs into language specific plugins and less language dependent plugins?

Both problems are interrelated: if many parts of a DT can be moved into language independent plugins, the burden of supporting a new language might be reduced to only provide the parts that are language specific and some glue code for the language agnostic components.

If there is a set of related programming languages, one could imagine providing common (configurable) symbol representation (see proposal by Markus Schorn). And tools based on the symbol representation.

In short:

- Language implementers want a simple jumpstart
- Tool vendors want to replace/enhance a specific technology
- Integrators want to mix and match components from different providers for one or multiple language DT's

Goals

- Make is easier to write a new language specific DT by providing language agnostic components
- Define extension points that can be used by multiple languages
- Allow vendors to provide extensions to components
- Allow parts of a DT to be replaced with alternative implementations (editor, build, debugger)

Proposal

- Split projects along technology boundaries instead of programming language:
  - Editor
  - Build
  - Debugger
  - Profilers
  - Indexing
  - Static analysis (language specific)
- Extract common implementation to be used in multiple plugins (refactoring ltk, indexer)
- Separate language specific features from language neutral features
- Provide a common plugin structure and common extension points for the different technologies:
  - Allow to replace technology lines in a DT
  - Allow to extend an DT (even if parts have been replaced)
  - Allow to write extensions for multiple DTs (languages)

Personas: Different Perspectives

Depending on which side you look at the problem of multiple languages, you have a different perspective.

The Users Perspective

- Want complete and consistent language toolkit
- Configure eclipse for specific needs
  - Use of specific tools
  - Follow company policy

    - Mix and match tools
- Use 3rd party plugins

The DT Developers Perspective

- Minimize effort
- "monkey see monkey do" is easier than defining common infrastructure
- Creating abstractions/extension points is "expensive" (it has to be maintained and be stable over time)
- Focus is on the specific language and not on a language abstraction (else the problem is to "hard")
- Different levels of language support (presented by John Duimovich at eclipsecon 2004)

The Extension Writer Perspective

- Want consistent set of extension points (plug into multiple DTs)
- Extend functionality in a language independent consistent way (like adding gutter or highlighting to editors)

The Tools Provider Perspective

- Be able to plug into any environment
- Be able to replace parts of a DT with specific tools (provide alternative implementations):
    - Team support is great example (CVS, ClearCase, PVS....)
    - Editor (either replace or enhance with plugins) ADT replaces the JDT editor but cannot coexist with other ADT-like extensions (against the eclipse promises)
    - Specific debuggers (GDB, DBX, ...)
    - Build systems (there is not THE build solution)

The Tool Integrator Perspective

Put together tools from different vendors

- specific editor
- specific build system
- specific debugger
- specific extensions
- ...

How to split technology?

There are two ways to split technology: along product lines and along technology lines.

1. Split along Product Lines

- JDT

- CDT
- Cobol DT
- ADA DT
- Fortran DT
- ...

2. Split along Technology Lines

- Editor
- Build (make)
- Debugger
- Profiler
- Static analysis (language specific part)
- Refactoring
- …

A fundamental problem with eclipse as language development environment is the way technology is split. The focus at the moment is around languages instead of functionality (imagine a world, where CVS would be integral part of JDT....).

The split along "product lines" facilitates fast and complete support of a particular language at the price of duplication of technologies. A technological split might not fully satisfy the specific needs of a language toolkit. But it enables vendors specialized on a particular tool type (editor, debugger, build system etc) to provide those tools for multiple language DTs. Just like it is possible today to use different team support tools.

Instead of creating one LTK, it would make sense to focus on different technology lines and making each technology line a more or less independent project. There are some connections between the technologies and it will be necessary to create an infrastructure to put things together. But each technology has its own independent life and its own community. However, it would be necessary to synchronize the releases.

The following we will focus on two technologies: code editors and a build plugin.

Part II

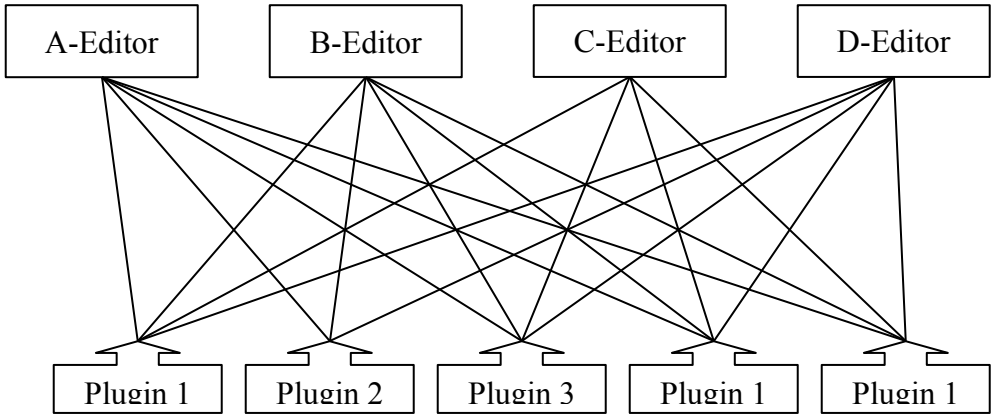
We will pick put two examples, the source code editors and the build system and outline how they could live as independent technologies. Migration of existing DT is also considered.

The Source Code Editors

Currently, eclipse has a very flexible editor toolkit. Every language DT uses the toolkit and builds its own editor. Unfortunately, the editor toolkit provides little help when it

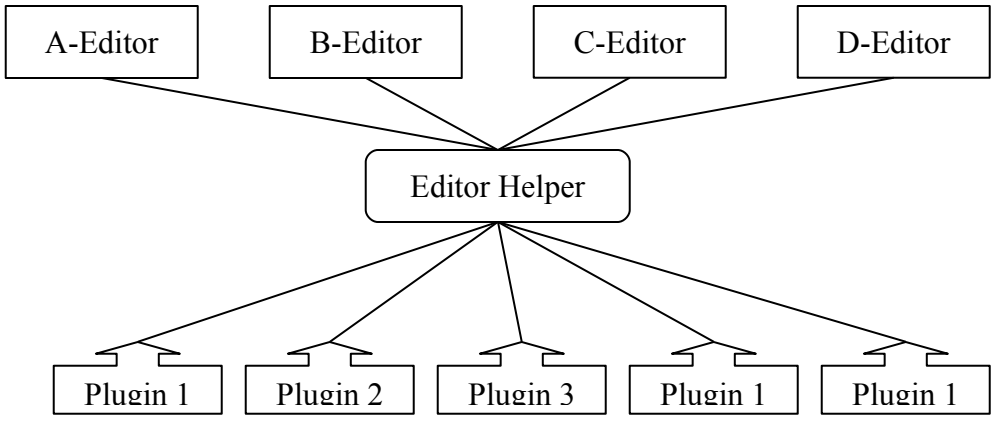
comes to provide extension points. Each editor invents its own extension points. But most editors for code editors are very similar. Therefore, it is relatively easy to write a generic editor that can be extended to add new languages. Although it might be a nice goal to have such an extensible editor, it would be very hard to migrate all existing editors to a new generic editor. Therefore we propose an editor helper plugin that provides extension points for common behavior. The extension points can be used by plugin writers if they want to contribute or extend existing editors. There will also be a set of helper classes that can be used by editor writers to add extensibility to their editors.

Current Situation of the Editor



- There are very few common extension points for editors
- Editors invent their own extension points
- Plugin writers have to know the ID of the editor (they cannot contribute to a content type or a programming language)
- Contributions to a language embedded in another language is hard

Proposed Editor Helper Plugin



Editors can use contributions from the Editor Helper for

- Formatting
- Folding
- Outline contributions
- Navigation
- Syntax/Semantic highlight
- Rulers
- Hovers
- Content Assist
- …

The Editor Helper would provide some helper classes that existing editors can use to allow contributions to their already existing implementation.

Scanners

Plugin writers can extend editors services from the list above. One fundamental concept would be that plugins can contribute Scanners. Scanners can be used to provide the context for some of the above extensions. Scanners can be used to for:

- Providing the scope of Hovers or Code Completion
- Syntax highlighting
- Folding
- Outline
- Comment recognition
- Bracket matching
-

Providing a scanner would yield in a simple integration of a new programming language.

Contribution for Content Type or Programming Language

At the moment context menus and other extensions are done for Editor Ids. Editor Ids are very specific and make it hard to use another editor. Many contributions could be done based on programming language. Since there is no real concept of programming language in eclipse, the content type can be used instead. But it should be considered to add the concept of a programming language. Minimally, a Programming Language has an id, a name, a variant and a version. E.g. C: KR-C, ANSI-C or Java: Java 1.3, Java 1.4 etc. There should be a kind of registry, where Programming Language can be registered with a kind of URI. Contributions can be made on specific Programming Language or language dialects and there will be no need to abuse editor ids.

Examples of Plugins for Code Editors

- Profilers could add gutters and background highlighting to visualize data.
- Profilers could contribute to hovers to explain the performance problems
- Comment parsers could add content assist, navigation support, etc
- Code generators parsers could provide content assist for the code that will be generated
- External documentation could be used to populate hovers
- External code formatters could easily be added
- Special language extensions could contribute to code coloring
- Special comment parsers and preprocessors could contribute content assist, formatting, folding, hovers etc.
- Special language bindings (most languages provide ways to call C code) could provide navigation to methods implemented in other languages.
- …

Build System Support

There are various build systems: make based, ant based, script based etc. Even make based build systems can look very different. Many companies have their own custom build system and don't want to change it. On the other hand, many compiled languages cannot be parsed and understood unless the compiler flags of every file are known. Therefore it would be very helpful if there would be a Build Support plugin that provides the information needed to successfully analyze the source code. The choice of a build system is similar to the choice of a configuration management system. For configuration management the eclipse platform provides the team support. Similarly there should be build support. There is no reason at all to bind a build system to a particular language. Most build systems are neutral to the programming language. Therefore this is a perfect example of a technology that could live outside and specific language toolkit.

Basic Build Support

The Build Support plugin should provide the following information

- Is a file compiled (e.g. not all .c files in a project might be compiled)

- Get the build tool (compiler) for a file. It might be necessary to provide abstract descriptions of compilers and compiler versions. For example, for C/C++ there are many versions of compilers and each compiler has build in preprocessor defines.
- Language id, variant and version (see above, in the editor section)
- Get the build flags (the defines) for a file
- Allow registration of listeners for changes to the build settings

Implementations of build systems can plug into the Build Support plugin and provide the information. The build

Advanced Build Support

Optionally, the Build Service should provide information about assembly of build targets. Many compiled languages have no clear concept of modules/packages. Without an understanding of the link structure, the code analysis can be misleading. Some symbols might be multiple defined within a project, but when objects are linked there is no conflict, because they either end up in different executables or there is a clear hierarchy of shadowing (e.g. the link order for .a files).

- Build Sets would be something like working set for the build products (which files go into an executable or into a library)
- Build sets can be assembled hierarchically (like executable consist of libraries and object files…)

If there would be a Build Support plugin, different build plugin could emerge (like different team plugins) and allow users to choose the build system that fits best their needs, or even write their own, without loosing integration with the language toolkits.

Conclusions

We have seen that there is a lot of technology that exists currently within language toolkits that should exist independently. The tight coupling of editors, build systems, debuggers, indexing etc to languages cause a lot of duplication and is bad for the eclipse eco system. There should be a set of technology driven plugins that allow better interoperability and exchangeability between different implementations. For the editor we propos an Editor Helper plugin, that provides basic extension points, and for the build system, we propos a Build Support plugin, that provides the basic build information to language toolkits.