FAQ

DSLs, LOP and Programming

Here you can find answers to the most frequent questions regarding MPS.

What are DSLs? How are they different from "real" programming languages?

A DSL is a language optimized for a specific class of problems. It is usually less complex than a general-purpose language, such as Java, C or Ruby. A DSL may not even be Turing-complete and only state facts about the domain of interest. Usually DSLs are developed in close coordination with the people, who are experts in the field for which the DSL is designed. In many cases, DSLs are intended to be used not by software people, but instead by non-programmers, who are fluent in the domain the DSL addresses. This requires the language's notation and tool support to be optimized for non-programmers, for example, by using mathematical symbols, a mix of textual and graphical notations or a simplified IDE that does not expose DSL users to the full complexity of an IDEA or Eclipse.

What are the benefits of DSLs? Why should I care?

Using DSLs can reap a multitude of benefits. The most obvious benefit of using DSLs is that - once you've got a language and an transformation engine - your work in the particular aspect of software development covered by the DSL becomes much more *efficient*, simply because you don't have to do the grunt work manually. This is most obvious if you generate a whole truck load of code from a relatively small DSL program. If you are generating source code from your DSL program (as opposed to interpreting it) you can use nice, domain-specific abstractions *without paying any runtime overhead*, because the generator, just like a compiler, can remove the abstractions and generate efficient code. If you have a way of expressing domain concerns in a language that is closely aligned with the domain, your *thinking becomes clearer* because the code you write is not cluttered by implementation details. In other words: using DSLs allows you to separate essential from incidental complexity. DSLs, whose domain, abstractions and notations are closely aligned with how domain experts (i.e., non-programmers) express themselves, allow for very good *integration between the techies and the domain people*. Using DSLs and an execution engine makes the application logic expressed in the DSL code *independent of the target platform*. Using DSLs can increase the *quality* of the created product: fewer bugs, better architectural conformance, increased maintainability. This is the result of the removal of (unnecessary) degrees of freedom, the avoidance of duplication in code and the automation of repetitive work. In contrast to libraries and frameworks, DSLs can come with *tools*, i.e. IDEs, that are aware of the language. This can result in a much improved user experience. Code completion, visualizations, debuggers, simulators and all kinds of other niceties can be provided.

How do DSLs and regular Code fit together?

There are two fundamentally different ways of how traditional code and DSL code can be integrated. The first one keeps DSL code and regular code in separate files. The DSL code is then transformed into programming language code by a code generator, or alternatively the program loads the domain-specific code and executes it. This first approach, with seperated General Purpose Language (GPL) and DSL code is termed external DSLs. Think of SQL as an example of an external DSL. An alternative approach mixes DSL code and general-purpose code in the same program file, leading to a much tighter integration between DSL code and programming language code. The DSL reused the grammar and the parser of the GPL and exploits available extension options of the host language. It is worth to mention that some GPL are more suitable for extension than others.
Both approaches can make sense, depending on the circumstances, and MPS supports both.

Traditionally, DSLs have been embedded in programming language code by using the meta programming facilities of the host language. The DSL's sturcture and syntax was defined by writing code in the language into which the DSL code were to be embedded. Usually the IDE didn't know about the DSL and hence did not provide support (code completion, custom error checking, etc.). With MPS, you use the MPS framework with its specialized DSLs for language development to define language extensions. The IDE knows about them, so the system can provide full IDE support for the domain-specific embedded languages.

What is Language Oriented Programming?

The term language oriented programming has been coined by Sergey Dmitriev, the CEO of JetBrains and "father" of MPS in a 2004 article called Language Oriented Programming: The Next Programming Paradigm. Other people have come up with related approaches, usually under different names; a primary example is Charles Simonyi and his Intentional Programming approach, and Martin Fowler has described the approach in his 2005 article Language Workbenches: The Killer-App for Domain Specific Languages?.

The core idea is that we don't just use one language when developing software, but rather use those languages that fit each of the tasks best. In contrast to polyglot programming, which on the surface advocates a similar approach, language oriented programming explicitly encourages developers to build their own DSLs, or to extend existing languages with domain-specific concepts as part of the approach. Developing a new language should become an integral part of software development and not left to the Übergeeks. To make this feasible, languages workbenches such as MPS are an important ingredient of the language oriented approach.

Why do I want to extend a language? Aren't libraries good enough?

There are a couple of differences between a library and a language extension. A language extension can come with its own syntax. With MPS' projectional editor, this syntax can be arbitrary and is not at all limited by the syntax of the extended language. A language extension also comes with its own constraints and type system, so your IDE can report errors statically. More generally, language extensions, in the way MPS supports them, are fully integrated into the IDE: you get code completion, syntax highlighting and refactoring support for your new language constructs. Finally, a language extension is executed by compile-time transformation and translated into the target programming language code, so there is no runtime overhead that reflection or stacks of indirections would impose. This may be important in particular when targetting resource-constrained systems.

Projectional Editing

What is Projectional Editing?

In parser-based approaches, users use text editors to enter character sequences that represent programs. A parser then checks the program for syntactic correctness and constructs an abstract syntax tree (AST) from the character sequence. The AST contains all the semantic information expressed by the program i.e. keywords and purely syntactic aspects are ommitted.

In projectional editors, the process happens the other way round: as a user edits the program, the AST is modified directly. This is similar to the MVC pattern where every editing action triggers a change in the AST. A projection engine then creates some representation of the AST for the user to interact with. This approach is well-known from various graphical editors. When editing a UML diagram, for example, users don't draw pixels onto a canvas for an "image parser" to read the drawing, parse it and then create the AST. That would be way too limiting on what you can draw so as the engine would understand. Rather, the editor creates and instance of Class as you drag a class from the palette to the canvas. A projection engine renders the diagram, in this case drawing a rectangle for the class. You can then re-arrange visual elements on the screen without changing the meaning of your diagram.

This approach can be generalized to also work with text editors.Every program element is stored as a node with a unique ID (UID) in the AST. References are based on actual pointers (references to UIDs). The AST is actually an ASG, an abstract syntax graph, from the start because cross-references are first-class rather than being resolved after parsing. The program is then persisted to disk as XML, but this process is transparent to the user.

Why use Projectional Editing? Aren't text editors good enough?

Since no parsing is used in projectional editors, and the mechanism works basically like a graphical editor, notations other than text can be used in the editor. For example, MPS supports tables as well as simple diagrams. Since these non-textual notations are handled the same way as the textual ones (possibly with other input gestures), they can be mixed easily: tables can be embedded into textual source, and textual languages can be used within table cells. Textual notations can also be used inside boxes or as connection labels in diagrams.

After composing separately developed languages, the resulting language may become ambiguous in parser-based systems since most grammar formalisms are not closed under composition. In projectional systems, this cannot happen. Any combination of languages will be syntactically valid (semantics is a different issue). If a composed language would be ambiguous, the user has to make a disambiguating decision as the program is built. For example, in MPS, if in a given location two language concepts are available under the same alias, just typing the alias won't bind, and the user has to manually decide by picking one alternative from the code completion menu.

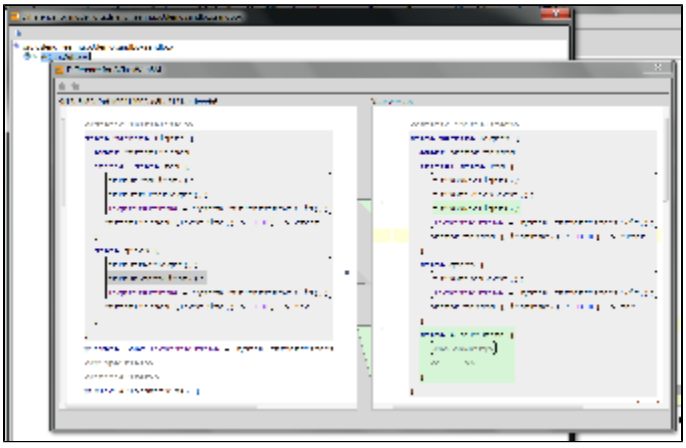
What are the key benefits of using projectional editors instead of parsed languages

Projectional editor gives the Abstract Syntax Tree (AST) the prominent role in code manipulation. Code is persisted, versioned, refactored and edited in the AST form, which eliminates the need for repetitive text-to-model-and-back transformations. AST is a much richer representation of code, which avoids ambiguities. The AST can be projected on the screen for editing in any way the language designer desires. With projectional editors your languages can:

- allow for non-parseable notations - tables, images, GUI components may become part of the language syntax. The AST can easily accommodate for such language elements and the projectional editor can draw any graphical shapes on the screen, if needed.
- combine multiple languages, potentially developed by different vendors, in a single piece of code. Ambiguities that parsers would have problems to recover from are not an issue for projectional editors.
- handle context-sensitive and positional grammars
- provide form-like notations - typically when targeting non-professional developers and domain experts the preference could be to use languages that offer strict notation with clear indication of the "moving" parts where the user is supposed to provide a value
- offer multiple notations for a single language - users can view and edit the same piece of code using several different editors, each suitable for a different task. Some editors may use graphical or tabular notations, some may show extra information from other sources, some may reduce the amount of visual clutter yet all are tied to the same piece of code (AST). The developer can thus choose the best visualization of the cdd for the given task.
- use diagramming notations in combination with text - think of electrical circuits, for example, modeled as graphical schemes with embedded pieces of code attributed to individual elements of the scheme

If it isn't text, how does the integration with version control work? Diff, Merge?

Special support is needed for infrastructure integration. Since the concrete syntax is not pure text, a generic persistence format needs to be used. Therefore, special tools need to be provided for diff and merge. MPS provides integration with the usual VCS systems, such as SVN or Git, and handles diff and merge in the tool, using the concrete, projected syntax. Note that since every program element has a UUID, a move can be distinguished from delete/create providing more useable semantics for diff and merge. The picture below shows an example.



How long does it take to get used to the editor?

In early 2010 we performed an experiment as part of a training. Ten people, who had never used MPS before, were taught the principals of MPS development. After two days all of them agreed that, while the editing experience is a bit unusual at the

beginning, the editor is in fact not worse at all than text editors, it is just different. After two days, you get used to it and feel no discomfort. In fact, and not surprisingly to us, some of the people claimed they prefer the MPS way of editing things, since in many context you're selecting/changing/removing elements based on the syntax tree anyway and projectional editors have an edge over text-base IDEs in this regard.

MPS Specifics

How is MPS licensed?

MPS is licensed under the Apache 2.0 License. Apache 2.0 is very liberal and allows you to use the system in basically any context you like. Companies like e.g. Realaxy have used MPS to build commercial applications.

Which languages are already in MPS so I can extend them?

MPS comes with a language called BaseLanguages, which is essentially Java 6 (with optional extensions for syntax of Java 7 and 8) plus a pile of custom extensions such as closures, collection API, regular expressions, extension methods, tuples and builder support. This language is also used for many aspects of language development in MPS, in which case it uses a couple of additional extensions for working with program trees. MPS also comes with an implementation of XML that can be used and extended. There is also an implementation of C available as an additional base language, see mbeddr.com. Other languages are being gradually developed by third-parties and as open-source projects (e.g. JavaScript). Check out the MPS Languages Repository page for more details.

Is there a language repository somewhere?

Since languages can be packaged and distributed as jar files, nothing prevents languages from being shared the same way libraries are. A handful of language plugins (both by JetBrains and third parties) for MPS as well as for IntelliJ IDEA have been made publicly available at the JetBrains plugin repository:

- MPS Languages Repository page in the documentation
- MPS language plugins site
- IntelliJ IDEA language plugins developed in MPS site

Are there some books on MPS?

Yes, there are:

- Fabien Campagne wrote a reference guide - The MPS Language Workbench Volume I
- Markus Voelter wrote a book on DSL engineering featuring MPS - DSL Engineering
- Numerous academic papers have been published over the past couple of years - check out the MPS publications page

How much effort does it take to add new base languages to MPS?

The time it takes to add new languages obviously depends a lot on the complexity of the language. Small DSLs can be added in a matter of minutes or hours, if you are familiar with MPS. Any language added to MPS can be used as a base language and can be easily extended by you or others. But that's probably not what you wanted to know :) Adding *real* programming languages as new base languages is a little bit more work, because general-purpose languages are typically rather large (with the exception of languages like Lisp, which would be trivial to add :-)). A good benchmark is the C base language recently implemented by the mbeddr.com folks. They reported 2-3 person month of work to implement all of C in MPS.

If projection is so cool, when can we do notations other than text?

MPS already supports notations that are not possible with parser-based systems. For example, MPS ships with an extension to BaseLanguage that can do fraction bars and big math symbols (such as the big symbol sign). In addition, MPS supports tables, as demonstrated by the decision tables example in the mbeddr project. It is also possible to embed any swing widget into an editor. Support for graphical notations out of the box has been added in the MPS 3.1 release.

Are there any successful projects developed with MPS?

MPS has originally been developed as an in-house project at JetBrains, so there are several use cases within JetBrains. A well-known example is the mbeddr.com project, which provides an extensible C-based IDE for embedded software development.

MPS is gradually being adopted by different industries - electrical engineering, insurance, government organizations, to

implement powerful DSLs for their respective domains.

How does MPS scale?

The mbeddr folks have done a little load test. They have found that single root elements (the equivalent of files in a classical IDE) can easily go up to 4.000 lines without serious performance issues. They have also measures that a C system with up to 100.000 lines of C code should work fine.

MPS is Open Source, but still "owned" by JetBrains. Huh?

Yes, JetBrains is the main contributor to the project providing more than 10 full-time developers. External contributions are welcomed, though, either as ideas shred through the forum, requests and bug reports in the tracker or patches to the code. Regular contributors may become project commiters and participate on the project planning an implementation process.

Can I import Java code into MPS?

MPS works hard on making Java interoperability easy. For example, you can use any Java libraries in MPS without problems. You can mix-and-match Java and MPS code on the same project so that your MPS code can see and use your Java code and vice versa - in Java you can use the code generated from MPS. Also, MPS can parse snippets of Java code that you decide to paste into MPS and transform them into valid BaseLanguage code..

Which Java versions are supported?

MPS can run on JDK 1.8 and later. Check out the MPS Java compatibility page for more details.

How about Java IDE Integration?

Currently, MPS is a standalone application, implemented in Java so it runs on all major platforms. In the 3.0 version MPS added support for deploying languages into IntelliJ IDEA. Language *development* is still done with the MPS standalone application, but language *use* is now possible in IntelliJ IDEA. This gives users much better options when integrating MPS-based DSLs into your Java and other applications.

How about Eclipse Integration?

While technically feasible, integration with Eclipse is currently not actively pursued. Depending on the interest among MPS users and their ability to contribute to or sponsor such activity, JetBrains is ready to consider implementing Eclipse integration.

How can I get support?

There is a MPS community forum, where you can get support. An issue tracker is available for submitting and tracking bugs. Last but not the least, you can also get professional support from people, who have real hands-on experience with MPS.

How does MPS compare to other language workbenches?

This is of course hard to say in an objective way from where we stand, but let's try :-) Unlike most other language workbenches, MPS uses a projectional editor. We've already discussed the benefits of this approach in the FAQ above. Also, MPS is the most comprehensive system in that it provides support for language structure, syntax, type system, constraints, refactoring, transformation and code generation all in a single integrated package. As of now, other alternatives such as MetaEdit+ provide better support for grapical syntax, and Xtext provides better Eclipse integration and has a bigger community surrounding it. For more detailed comparison you have to make up your own mind, and consider all facts in the context of your own project's needs. You may also consider checking out the MPS case studies, reading some of the academic papers published about MPS and getting in touch with experienced consultants.

How can I learn about using MPS?

You should take a look at the freshly renovated documentation page. It holds logically organized references to overview articles, tutorials on MPS basic and advanced concepts as well as screencasts and pointers to interesting case studies.