Language Workbenches: The Killer-App for Domain Specific Languages?

Most new ideas in software developments are really new variations on old ideas. This article describes one of these, the growing idea of a class of tools that I call Language Workbenches - examples of which include Intentional Software, JetBrains's Meta Programming System, and Microsoft's Software Factories. These tools take an old style of development - which I call language oriented programming and use IDE tooling in a bid to make language oriented programming a viable approach. Although I'm not enough of a prognosticator to say whether they will succeed in their ambition, I do think that these tools are some of the most interesting things on the horizon of software development. Interesting enough to write this essay to try to explain, at least in outline, how they work and the main issues around their future usefulness.

12 June 2005

Martin Fowler

Translations: Russian Korean

Contents

For a long time there's been a style of software development that seeks to describe software systems using a collection of domain specific languages. You see this in the Unix tradition of 'little languages' which generate code via lex and yacc; you see it in the Lisp community with languages developed inside Lisp, often with the help of Lisp's macros. Such approaches are much liked by their advocates, but this style of thinking hasn't caught on as much as many of these people would like.

In the last few years there's been an attempt to support this style of development through a new class of software tool. The earliest and best known of these is Intentional Programming - originally developed by Charles Simonyi while at Microsoft. However there are other people doing similar things too, generating enough momentum to create some interest in this approach.

At this point I'm going to coin some terminology that I'll use in the rest of this essay. As usual there's no standard terminology in this field, so don't expect the terms I use to be used in this style elsewhere. I'm going to give a brief definition here, but will explain much more about them as the essay goes on - so don't worry if you don't follow the definitions immediately.

The two main terms I'm specifically coining for this article are 'Language Oriented Programming' and 'Language Workbench'. I use **Language Oriented Programming** to mean the general style of development which operates about the idea of building software around a set of domain specific languages. I use **Language Workbench** as a generic term for this new breed of tools. So a language workbench is one way to do language oriented programming. You may also be unfamiliar with the term **Domain Specific Language** (usually abbreviated to **DSL**). It is a limited form of computer language designed for a specific class of problems. Some communities like to use DSL only for problem domain languages, but I'm following the usage that uses DSL for any limited domain.

I'm going to start by briefly describing the current world of language oriented programming with an example, an overview of the different flavors, and various arguments about the pros and cons of the approach. If you're familiar with language oriented programming you may want to skip through this stuff, but I've found that many, indeed most, developers aren't that familiar with these ideas. Once these are explained I'll then build on them to explain what language workbenches are and how they alter the trade-offs.

As I wrote this article, it turned out to be too much for a single article, so I've separated some parts of the discussion into other articles. I'll mention as I go in the text where it makes sense to go off an read those, they're also linked just below the contents. In particular take a look at the example using MPS - this shows an example DSL built using one of the current language workbenches and is probably the best way of getting a feel for what they will be like. You'll need to get through the general description of language workbenches here before it'll make much sense.

A simple example of language oriented programming

I'm going to begin by running through a very simple example of language oriented programming and the kind of situation that leads to it. Imagine we have a system that reads files and needs to create objects based on these files. The file format is one object per line. Each line can map to a different class, the class is indicated by a four character code at the beginning of the line. The rest of the line contains the data for the fields of the class, these vary depending on what class we are talking about. The fields are indicated by position rather than delimiter. So a customer ID number might run from characters 4-8.

Here's some sample data

```
#123456789012345678901234567890123456789012345678901234567890
SVCLFOWLER         10101MS0120050313.........................
SVCLHOHPE          10201DX0320050315........................
SVCLTWO           x10301MRP220050329..............................
USGE10301TWO          x50214..7050329...............................
```

The dots indicate some mumbly uninteresting data. The comment line at the top is to help you see the character positions. The first four characters indicate the kind of data - SVCL indicates a service call, USGE a record of usage. The characters after that represent the data for the object. So the characters from position 5 to 18 on a service call indicate the name of the customer.

To turn these into objects you might be tempted to write specific code for each case, I hope that after a few you'd want to simplify the task by writing a single reader class that you can parameterize with the details of the fields for each class.

Here I have a simple class to do this. A reader class reads the file. A reader can be parameterized with a collection of reader strategy classes - one for each target class. So for our example we'd have one strategy for service calls, another for usages. I hold the strategies in a map keyed by the code.

Here's the code to process a file

```
class Reader...
    public IList Process(StreamReader input) {
      IList result = new ArrayList();
      string line;
      while ((line = input.ReadLine()) != null)
        ProcessLine(line, result);
      return result;
    }

    private void ProcessLine(string line, IList result) {
      if (isBlank(line)) return;
      if (isComment(line)) return;
      string typeCode = GetTypeCode(line);
      IReaderStrategy strategy = (IReaderStrategy)_strategies[typeCode];
      if (null == strategy)
        throw new Exception("Unable to find strategy");
      result.Add(strategy.Process(line));
    }
    private static bool isComment(string line) {
      return line[0] == '#';
    }
    private static bool isBlank(string line) {
      return line == "";
    }
    private string GetTypeCode(string line) {
      return line.Substring(0,4);
    }
    IDictionary _strategies = new Hashtable();
    public void AddStrategy(IReaderStrategy arg) {
      _strategies[arg.Code] = arg;
    }
```

It just loops through the lines, reads enough to figure out what strategy to call, and then hands over to the strategy to do the work. To get the reader to do the job you create a new reader, load it up with strategies and let it loose on the files you want to process.

The strategies can also be parameterizable. We only need one strategy class, when we instantiate it we can parameterize it with the code, target class, and details of what character positions on the input map to which fields on the target class. I hold the latter in a list of field extractor classes.

```
class ReaderStrategy...
    private string _code;
    private Type _target;
    private IList extractors = new ArrayList();
    public ReaderStrategy(string code, Type target) {
      _code = code;
      this._target = target;
    }
    public string Code {
      get { return _code; }
    }
```

I can add field extractors to the strategy once I've instantiated it.

```
class ReaderStrategy...
    public void AddFieldExtractor(int begin, int end, string target) {
      if (!targetPropertyNames().Contains(target))
        throw new NoFieldInTargetException(target, _target.FullName);
      extractors.Add(new FieldExtractor(begin, end, target));
    }
    private IList targetPropertyNames() {
      IList result = new ArrayList();
      foreach (PropertyInfo p in _target.GetProperties())
        result.Add(p.Name);
      return result;
    }
```

To process the line the strategy creates the target class and uses the extractors to get the field data

```
class ReaderStrategy...
    public object Process(string line) {
      object result = Activator.CreateInstance(_target);
      foreach (FieldExtractor ex in extractors)
        ex.extractField(line, result);
      return result;
    }
```

The extractors simply pull the data out of the right bit of the line, and use reflection to put the value into the target object.

```
class FieldExtractor...
      private int _begin, _end;
      private string _targetPropertyName;
      public FieldExtractor(int begin, int end, string target) {
        _begin = begin;
        _end = end;
        _targetPropertyName = target;
      }
      public void extractField(string line, object targetObject) {
        string value = line.Substring(_begin, _end - _begin + 1);
        setValue(targetObject, value);
      }
      private void setValue(object targetObject, string value) {
        PropertyInfo prop = targetObject.GetType().GetProperty(_targetPropertyName);
        prop.SetValue(targetObject, value, null);
      }
```

So far what I've described is a very simple library for doing this kind of thing. Essentially I've built an abstraction which I can then use to specify the concrete work. To use the abstraction I need to configure the strategies and load them into the reader. Here's examples of this for the two example cases.

```
    public void Configure(Reader target) {
      target.AddStrategy(ConfigureServiceCall());
      target.AddStrategy(ConfigureUsage());
    }
    private ReaderStrategy ConfigureServiceCall() {
      ReaderStrategy result = new ReaderStrategy("SVCL", typeof (ServiceCall));
      result.AddFieldExtractor(4, 18, "CustomerName");
      result.AddFieldExtractor(19, 23, "CustomerID");
      result.AddFieldExtractor(24, 27, "CallTypeCode");
      result.AddFieldExtractor(28, 35, "DateOfCallString");
      return result;
    }
    private ReaderStrategy ConfigureUsage() {
      ReaderStrategy result = new ReaderStrategy("USGE", typeof (Usage));
      result.AddFieldExtractor(4, 8, "CustomerID");
      result.AddFieldExtractor(9, 22, "CustomerName");
      result.AddFieldExtractor(30, 30, "Cycle");
      result.AddFieldExtractor(31, 36, "ReadDate");
      return result;
    }
```

I look at this as two different styles of code. The Reader and Strategy classes are an abstraction, this last bit of code is configuration. When you're building these kinds of library classes it often helps to think of these two pieces: abstraction and configuration. The abstraction may be a class library, a framework, or just a set of function calls. The abstraction may be reusable in many projects, but it doesn't have to be. The configuration code tends to be specific; rather simple, straight-ahead code.

Since the configuration is pretty simple and likely to change more often than the abstraction, a common approach is to separate it further and take the configuration out of C# altogether. The current fashion is to put it in an XML file.

```
<ReaderConfiguration>
  <Mapping Code = "SVCL" TargetClass = "dsl.ServiceCall">
```

```
    <Field name = "CustomerName" start = "4" end = "18"/>
    <Field name = "CustomerID" start = "19" end = "23"/>
    <Field name = "CallTypeCode" start = "24" end = "27"/>
    <Field name = "DateOfCallString" start = "28" end = "35"/>
  </Mapping>
  <Mapping Code = "USGE" TargetClass = "dsl.Usage">
    <Field name = "CustomerID" start = "4" end = "8"/>
    <Field name = "CustomerName" start = "9" end = "22"/>
    <Field name = "Cycle" start = "30" end = "30"/>
    <Field name = "ReadDate" start = "31" end = "36"/>
  </Mapping>
</ReaderConfiguration>
```

XML has its uses, but isn't exactly easy to read. We could make it easier to see what's going on by using a custom syntax. Perhaps like this:

```
mapping SVCL dsl.ServiceCall
  4-18: CustomerName
  19-23: CustomerID
  24-27 : CallTypeCode
  28-35 : DateOfCallString

mapping  USGE dsl.Usage
  4-8 : CustomerID
  9-22: CustomerName
  30-30: Cycle
  31-36: ReadDate
```

Since you're now familiar with the problem, you should be able to read the syntax with no help from me.

As you look at this last example, you can see that what we have here is a very small programming language - one that's suitable (only) for the purpose of mapping fixed length fields into classes. It's a classic example of the Unix tradition of 'little languages'. It is a Domain Specific Language for the task.

This language is a Domain Specific Language, and shares many of the characteristics of DSLs. Firstly it's suitable only for a very narrow purpose - it can't do anything other than map these particular fixed length records to classes. As a result the DSL is very simple - there's no facility for control structures or anything else. It's not even Turing complete. You couldn't write a whole application in this language - all you can do is describe one small aspect of an application. As a result the DSL has to be combined with other languages to get anything done. But the simplicity of the DSL means it's easy to edit and translate. (I'll expand on the pros and cons of DSLs shortly.)

Now look again at the XML representation. Is this a DSL? I would argue that it is. It's in an XML syntax - but it's still a DSL - indeed in many ways it's the same DSL as the previous example.

This is a good moment to introduce a common distinction that you run into in programming language circles - the distinction between abstract and concrete syntax. The **concrete syntax** of a language is its syntax in its representation that we see. The XML and custom language files have different concrete syntaxes. However both share the same basic structure: you have multiple mappings, each with a code, a target class name, and a set of fields. This basic structure is the **abstract syntax**. When most developers think about programming language syntax they don't make this separation, but it's an important one when you use DSLs. You can think of this in two ways. You can either say we have one language with two concrete syntaxes, or two languages that share the same abstract syntax.

This example thus raises a design issue - is it better to have custom concrete syntax for a DSL or an XML concrete syntax. The XML syntax can be easier to parse since there are so many XML tools available; although in this case the custom syntax was actually easier. I'd contend that the custom syntax is much easier to read, at least in this case. But however you view this choice the core trade-offs around DSLs are the same. Indeed you can argue that any XML configuration file is essentially a DSL.

Let's go back a step further, back to the configuration code in C# - is this a DSL?

While you're thinking of that look at this code. Does this look like a DSL for this problem?

```
mapping('SVCL', ServiceCall) do
        extract 4..18, 'customer_name'
        extract 19..23, 'customer_ID'
        extract 24..27, 'call_type_code'
        extract 28..35, 'date_of_call_string'
end
mapping('USGE', Usage) do
        extract 9..22, 'customer_name'
        extract 4..8, 'customer_ID'
        extract 30..30, 'cycle'
        extract 31..36, 'read_date'
end
```

This second piece of code relates to the C# one. Those of you who know my language likings will have guessed that this last example is in fact ruby code. In fact it's the exact moral equivalent of the C# example. It looks much more like a custom DSL due to various ruby features: minimally intrusive syntax, literals for ranges, and flexible runtime evaluation. This is the full configuration file which can be read in and evaluated in a object instance's scope at runtime. But it's still pure ruby and interacts with the framework code through the method call `mapping` and `extract` which correspond to `AddStrategy` and `AddFieldExtractor` in the C# example.

I would argue that both the C# and ruby examples are DSLs. In both cases we use a subset of the capabilities of the host language and capture same ideas as our XML and custom syntax. Essentially we are embedding the DSL into our host language, using a subset of our host language as the custom syntax for our abstract language. To an extent this is more a matter of attitude than of anything else. I'm choosing to look at the C# and Ruby code through language oriented programming glasses. But it's a point of view with a long tradition - Lisp programmers often think of creating DSLs inside Lisp. The trade offs for these internal DSLs are obviously different than for external DSLs, but a number of similarities remain. (I'll expand on these trade-offs later too.)

So now I've shown an example of a DSL I can better define language oriented programming. Language oriented programming is about describing a system through multiple DSLs. It's a graduated thing, you can use a little language oriented programming in a system where just some of its functionality is represented in DSLs; or you can represent most functionality in DSLs and use a lot of language oriented programming. How much language oriented programming you use is hard to measure, especially if you use in-language DSLs. Typically, like with any reusable code, you write some DSLs yourself and use other DSLs from elsewhere.

Traditions of language oriented programming

As my example shows, language oriented programming isn't something new - people have been doing language oriented programming for quite some time. So it's worth taking a look at language oriented programming in the state as it currently stands before we look at what language workbenches bring to the picture.

There are number of styles of language oriented programming that are out there. This is a good moment to summarize a few of them.

Unix Little Languages

One of the most obviously DSLy parts of the world is the Unix tradition of writing little languages. These are external DSL systems, that typically use Unix's built in tools to help with translation. While at university I played a little with lex and yacc - similar tools are a regular part of the Unix tool-chain. These tools make it easy to write parsers and generate code (often in C) for little languages. Awk is a good example of this kind of mini-language.

Lisp

Lisp is probably the strongest example of expressing DSLs directly in the language itself.. Symbolic processing is embedded into the name as well as practice of lispers. Doing this is helped by the facilities of lisp - minimalist syntax, closures, and macros present a heady cocktail of DSL tooling. Paul Graham writes a lot about this style of development. Smalltalk also has a strong tradition of this style of development.

Active Data Models

If you run into some more sophisticated data modeler types, they'll show you how highly variable parts of a system can be encoded by data in database tables (often referred to as meta-data tables, or table-driven programs). Code can then interpret the data in the tables to carry out the behavior.

This is essentially a DSL whose concrete syntax is the database tables. Often these tables are managed through some form of GUI interface to edit this active data. Usually people doing this don't think about creating a language and usually the difficulty of working with the relational concrete syntax helps to keep the languages small and well focused.

Adaptive Object Models

Talk enough to hardcore object programmers, and they'll tell you about systems they've built that rely on composition of objects into flexible and powerful environments. Such systems are built of sophisticated domain models where most of the behavior comes from wiring up objects into configurations to handle a range of complex cases. OO people treat adaptive object models as active data models on steroids.

Such adaptive models are an in-language DSL. Experience so far indicates that they allow people familiar with the adaptive model to be extremely productive once the model is developed and shaken down. The dark side is that such models are often very difficult for new people to understand.

XML Configuration Files

Visit a modern Java project, and you'd be forgiven for thinking that there's more XML in the system than Java. Enterprise Java system use a range of frameworks, most of which boast complex XML configuration files. These files are essentially DSLs. XML makes it easy to parse, although not as easily readable as a custom format might be. People do write plug-ins for IDEs to help manipulate the XML files for those who find that angle brackets hurt the eyes.

GUI Builders

Ever since people started to build GUIs, systems have been around that allow you lay out the GUI through drag and drop of controls. Visual Basic is probably the most famous example, but I've used similar screen builders for character screens long before GUIs became common. These tools either store layout in a closed format, generating suitable code for execution; or they try to put all the necessary information in the generated code. Although they are visually nice, we increasingly see that although it makes for attractive demos, there are limitations with this style of interaction. So much so that many experienced GUI developers discourage using GUI builders for reasonably complex applications.

GUI builders are a form of DSL, but one where editing experience is quite different from the textual programming languages that we are used to. Hence they are often not thought of as languages by people building them - which some see as part of their problem.

Pros and Cons of language oriented programming

Reflecting on these styles we can see that various forms of language oriented programming are pretty popular. Generalizing grossly, I find it useful to divide them into two broader styles. **External DSLs** are written in a different language than the main (host) language of the application and are transformed into it using some form of compiler or interpreter. The Unix little languages, active data models, and XML configuration files all fall into this category. **Internal DSLs** morph the host language into a DSL itself - the Lisp tradition is the best example of this.

I've coined the external/internal terms for this article since there's not a clear pair of terms for what I feel is a useful distinction. Internal DSLs are often called 'embedded DSLs' but I've avoided the 'embedded' term because it gets confused with embedded languages in applications (such as VBA embedded into Word which if anything is an external DSL.) However you'll probably come across the embedded term if you look around at more writing on DSLs.

The trade-offs for external DSLs and internal DSLs are fairly different, so it's best to examine them separately.

External DSL

I define an external DSL as one that's written in a separate language to the main language of an application, such as the last two forms in our simple example. Unix little languages and XML configuration files are good examples of this style.

The key strength of an external DSL is that you are free to use any form that you fancy. As a result you get a lot of ability to express the domain in the easiest form possible to read and modify. The format is limited only by your ability to build a translator that can parse the configuration file and produce something executable - usually in your base language..

An obvious disadvantage then follows from this - you have to build this translator. For a simple language, like I've shown above, this is not difficult. Although more complex languages make it harder - it's still not that bad. Parser generator and compiler compiler tools exist that can help you manipulate quite complex languages, and of course the whole point of DSLs is that they are usually quite simple. XML restricts the form of the DSL, but makes it very easy to parse.

The big disadvantage of external DSLs is that they lack what I call **symbolic integration** - that is the DSL isn't really linked into our base language. The base language environment isn't aware of what we're doing. Now that programming environments are getting more sophisticated, this becomes an increasing problem.

For a simple example, consider if we want to rename the properties on the target class in my simple example. With a first class modern IDE, automatic refactorings for renaming are habitual. But such a rename won't propagate into the DSL. There is what I'll call a **symbolic barrier** between the world of C# and the file mapping DSL. We can translate our mapping into C#, but the barrier limits our ability to manipulate the overall program.

This lack of integration hits us in lots of ways with tooling. Firstly - how do we edit our DSL? A text editor will do the job - but modern IDEs increasingly make text editors look primitive. I should get a pop-up list and completion on the field names, red squiggles if the character ranges overlap. But to do this I need an editor that understands the semantics of my DSL.

Maybe I can live without a semantic editor. But then think about debugging. My debugger can step into the C# translations, but can't get into the true source itself. What I really would like is a full-blown IDE for my DSL. In the days of text editors and simple debuggers this wasn't a big issue - but we now live in a post-IntelliJ world.

A particularly common objection to external DSLs is the **language cacophony** problem. This concern is that languages are hard to learn, so using many languages will be much more complicated than using a single language. To some extent this concern is based on a misconception about DSLs. Those having the concern often imagine multiple general purpose languages, which indeed could easily result in cacophony. But DSLs tend to be limited and simple which makes them easier to learn. This is reinforced by their closeness to the domain. DSLs don't look like regular programming languages.

Fundamentally in any reasonably sized program you are dealing with a bunch of abstractions that you need to manipulate, such as the file reading example in the introductory example. Commonly we manipulate these abstractions using objects and methods. This works, but provides a limited grammar to express what we want to say (although how limited depends on our base language). Using external DSLs gives us an opportunity to have a grammar that is easier to manipulate. The question is whether the added ease of manipulating through the external DSL is greater than the cost of understanding the new DSL in the first place.

Related to this issue are concerns over the difficulty of designing DSLs - language design is hard thus designing multiple DSLs will be too hard for most projects. Again this objection often rests upon thinking about general purpose languages rather than DSLs. Here I think the fundamental issue is getting a good abstraction - that's the hard part of the task. The difference between API design and DSL design is then rather small - so I don't think designing DSLs is going to be significantly harder than designing good APIs.

For many people, one of the big strengths of an external DSL is that the DSL can be evaluated at runtime. This allows commonly changed parameters to be altered without recompiling the program. This is a major reason why XML configuration files have become so popular in the Java world. While this is an important issue with statically compiled languages, it's important to remember that many languages can easily evaluate expressions at runtime, so for them it's not a problem. There's also growing interest in mixing compile-time and runtime languages, such as IronPython in .NET. This would allow you to evaluate an IronPython internal DSL in the context of a mostly C# system. This is a common technique in the Unix world mixing C/C++ with scripting languages.

Internal DSL

Internal DSLs flip the pros and cons of ex-language DSLs. We eliminate the symbolic barrier with our base language. We also have the full power of our base language available to us at all times, together with all the tooling that exists in our base language. Lisp and adaptive object models are examples of internal DSLs.

One of the problems in discussing this is that there is a big difference between mainstream curly brace programming languages (C, C++, Java, C#) and those languages like Lisp that are particularly suited to internal DSLs. The internal DSL style is much more achievable in Lisp or Smalltalk than in Java or C# - indeed advocates of dynamic languages point this out as one of their major strengths. We're seeing some of this be rediscovered with scripting languages - consider the meta-programming capabilities of Ruby and how they are used by the Rails framework. This problem is compounded by the fact that many programmers have never used a dynamic language seriously, and thus don't have an appreciation of their capabilities (and true limitations.)

Internal DSLs are limited by the syntax and structure of your base language. More dynamic languages suffer less of a limitation. They have a minimally intrusive syntax (such as lisp, smalltalk, and scripting languages) which tends to work better than mainstream curly brace languages, something that's very visible when you compare the C# and ruby examples. Language features such as closures and macros are also valuable. While much of this machinery is missing from C based languages, we're seeing features that can support some of this thinking. Annotations (attributes in C#) are a good example of this kind of language feature which could be quite useful for this kind of purpose.

While you have the tooling of your base language, this base language doesn't actually know what you are up to with your DSL - so the tools don't fully support the DSL. You're still better off than with a text editor, but there's much room for improvement.

Having the full power of the language available to you in the DSL is a mixed blessing. If you're familiar with the base language, all is well. However one of the strengths of a DSL is that it allows people to program in it without knowing the full base language - which makes it simpler for lay programmers to enter domain specific information directly into the system. An internal DSL can make it hard to do this because there are many places where a user can get confused if they aren't familiar with the full base language.

One way of thinking about this is that a general purpose programming language gives you lots of tools - but your DSL uses only a few of these tools. Having more tools than you need often makes things harder - because you have to learn what all these tools are before you can figure out the few you use. Ideally you want only the actual tools you need for your job - certainly no less, but only a few more. (Charles Simonyi discussed this idea with the notion of degrees of freedom.)

There's an analogy here with office tools. Many people complain that modern word processors are so difficult to use because they have hundreds of features, far more than any single person needs. But since all these features are needed by somebody, an office program ends up satisfying everyone by building a large system. An alternative would be to have multiple office tools, each focused on a single task. Each of these tools would then be much easier to learn and use. The problem, of course, is that it's expensive to build all these special purpose office tools. It's a very similar trade-off to that between general purpose programming languages (with internal DSLs) and external DSLs.

Since internal DSLs are close to the programming language, this can present a difficulty when you want to express something that doesn't map well to the programming language itself. For example, in Enterprise Applications it's common to have a notion of layers. These layers can be defined to a large extent by using the package construct of the programming language, but it's hard to define the dependency rules between the layers. So you might put all your UI code in MyApp.Presentation and your domain logic in MyApp.Domain but there's no mechanism with an internal DSL to indicate that classes in MyApp.Domain should not reference classes in MyApp.Presentation. To some extent this again reflects the limited dynamism of common languages - this kind of thing was possible in Smalltalk since you have deeper access to the meta-levels.

(As a comparison, it would be interesting to see my more complex example developed in one of these dynamic languages. I probably won't get around to it, but I suspect someone else might, in which case I'll update the further reading.)

Involving Non-Programmers

One of the themes that winds constantly across both forms of language oriented programming is the involvement of **lay programmers**: domain experts who are not professional programmers but program in DSLs as part of the development effort. The goal of lay programming has been a constant goal of the software world - indeed many believed the early high level languages (COBOL and FORTRAN) heralded the end of programmers as users would use them. This reminds us of what I call the **COBOL inference** - that most technologies that are supposed to eliminate professional programmers do nothing of the sort.

Despite the COBOL inference, people do succeed in getting direct user input into programs from time to time. One way of doing this is to carve out a part of the problem that is sufficiently easy and limited that users can safely and comfortably program in this space. You then turn each of these user programmable areas into a DSL. These DSLs can be quite sophisticated - MatLab is a good example of a quite complex DSL that works because it is focused on a domain.

The advantage of an external DSL for a user programmable DSL is that you can drop all the baggage of your host language and present something that's very clear for the user. This matters particularly for languages with a more restrictive syntax. But even with simple languages you have an issue with internal DSLs in that a user can easily do things that make sense in the language but are out of scope of the DSL. This gets the user confused with what looks to them as odd behavior and cryptic error messages.

Many proponents of language oriented programming have a vision of the future where all the domain logic of a system is done by users. Programmers then write the necessary support tools to allow them to edit and compile these programs. While this would not mean the end of professional programmers - it would greatly reduce how many you need (since much of these tools would be reusable) and it would remove much of the communication issues that slow down software development today. This lay programmer vision is an attractive one - but the COBOL inference hangs mockingly over it.

In the end I see lay programming as a valuable thing to obtain, but not the whole point of language oriented programming. A good DSL makes professional programmers more productive even if it isn't embraced by user programmers. A good DSL may end up having professional programmer to write it - but be usefully reviewable by domain experts.

The lay programmer argument is a high stakes bet. If someone justifies some technology based primarily on enabling large scale user programming I overflow with skepticism. Yet if such an approach could succeed it would provide an enormous benefit. This wouldn't come from eliminating professional programmers, but on improving the often dire state of communication between domain experts and programmers. This lack of communication is often the biggest roadblock in software development projects.

Summarizing the Trade-Offs in language oriented programming

For me the fundamental issue in language oriented programming is the benefit of using DSLs versus the cost of building the necessary tools to support them effectively. Using internal DSLs reduces the tool cost - but the resulting constraints on the DSL itself can also greatly reduce the benefits, particularly if you are limited to C-based languages. An external DSL gives you the most potential to realize benefits, but comes at a greater cost to design the language, build the translator, and consider tools to support programming.

This is why language oriented programming hasn't caught on so much. Both in-language and ex-language techniques have significant disadvantages. As a result there is a gnawing gap - a sense that we should be able to do more with DSLs than we currently have.

This leads nicely into the justification for language workbenches. Essentially the promise of language workbenches is that they provide the flexibility of external DSLs without a semantic barrier. Furthermore they make it easy to build tools that match the best of modern IDEs. The result makes language oriented programming much easier to build and support, lowering the barriers that have made language oriented programming so awkward for so many.

Today's Language Workbenches

I'll start by briefly mentioning some of the tools that I've come across that fit this category of language workbench. Remember that all of these are in the early stages of development. We're still some years from seeing language workbenches that can be used for large scale software development.

Intentional Software

The godfather of these tools is Intentional Programming. Intentional Programming was originally developed by Charles Simonyi at Microsoft Research. A few years ago Simonyi left Microsoft and created his own company to develop Intentional Software independently. As is common with such startups, he's not been very open about developments. As a result there is a dearth of information about what is in Intentional Software and how it might be used.

I've had the opportunity to spend a little time with Intentional Software and several of my colleagues at ThoughtWorks have worked closely with Intentional over the last year or so. As a result I've had the opportunity to peek behind the Intentional curtain - although I'm restricted in how much I can say about what I saw there. Fortunately they intend to start opening up about their work over the next year or so.

(As a terminological note the Intentional folks use the term "Intentional Programming" to refer to the older work they did at Microsoft and "Intentional Software" to refer to what they've been doing since.)

Meta-Programming System

A newer initiative is the Meta Programming System developed by JetBrains. JetBrains have quite a reputation amongst software developers due to their superb IDE tools.

JetBrains's experience with IDEs is relevant to language workbenches in a couple of ways. Firstly their success with IntelliJ gives them a lot of credibility in the tools world - both for their technical ability and for their pragmatism. Secondly much of the capabilities of a language workbench are tied very closely to the features that make post-IntelliJ IDEs so capable.

JetBrains have spent a couple of years building a sophisticated environment for developing web applications called Fabrique. The experience of building Fabrique convinced them that they needed a platform to build these kinds of tools more effectively in the future - this desire is what led them to develop MPS.

MPS is strongly influenced by what has been made public about Intentional Software. It's had much less time in development than Intentional's work, but JetBrains believe in a very open development cycle. They have made MPS available under an Early Access Program as soon as they have something that's usable. Currently they hope to do this in the first half of 2005.

I've been fortunate in working quite closely recently with Sergey Dmitriev - the lead behind MPS. It helps that the MPS activity comes out of JetBrains's Massachusetts office, which makes it easy for me to visit them. As a result of this geographic similarity and their openness, I've used MPS to help describe some detailed examples (although they won't make much sense until I've got a bit further with this article. Don't worry I'll give you the link again when its time.)

Software Factories

Software Factories is an initiative headed by Jack Greenfield and Keith Short at Microsoft. There are several elements to software factories which I won't go into here (other than saying don't let the terrible name put you off.) The element that is relevant to this article is the DSL effort - language oriented programming plays a major role in Software Factories.

The software factories team has a background in Model Driven Development. They include people who have been active in CASE tool development and also many leading lights of the OO community in the UK. So it's no surprise that their DSLs tend to a more graphical approach. Unlike most CASE tool people, however, they take a serious interest in semantics and control over code generation.

Much of my discussion here refers to the traditional programming of an application. The Software Factories team in particular is also very interested in using DSLs for other areas of software development that often don't get automated such as deployment, testing, and documentation. They are also exploring simulators for situations where you don't want to execute the DSL directly in development - such as deployment DSLs.

The DSL team at Microsoft have been making downloads available for several months as part of Visual Studio 2005 Team System.

Model Driven Architecture (MDA)

If you've been tracking the OMG's MDA, you'll notice many similarities between what I've been saying about language workbenches and the MDA vision. It's a contentious issue, but for now I'll say that some visions of MDA are forms of language workbench - but not all of them. I'll also say that I believe that building a language workbench on top of MDA is seriously flawed. I've written a connected article to discuss this in more detail, but it won't make much sense until you've finished with this one.

Elements of a Language Workbench

Although these tools are all different, they do share some common characteristics and similar parts.

One of the strongest qualities of language workbenches is that they alter the relationship between editing and compiling the program. Essentially they shift from editing text files to editing the abstract representation of the program. Let me spend some paragraphs to explain that last sentence.

In conventional programming we edit the text of the program by using a text editor on text files. We then make that file executable by running a translator that turns those text files into something the computer can understand and execute. That translation may occur at execution time, as for scripting languages like Python or Ruby, or as a separate step for compiled languages such as Java, C# and C.

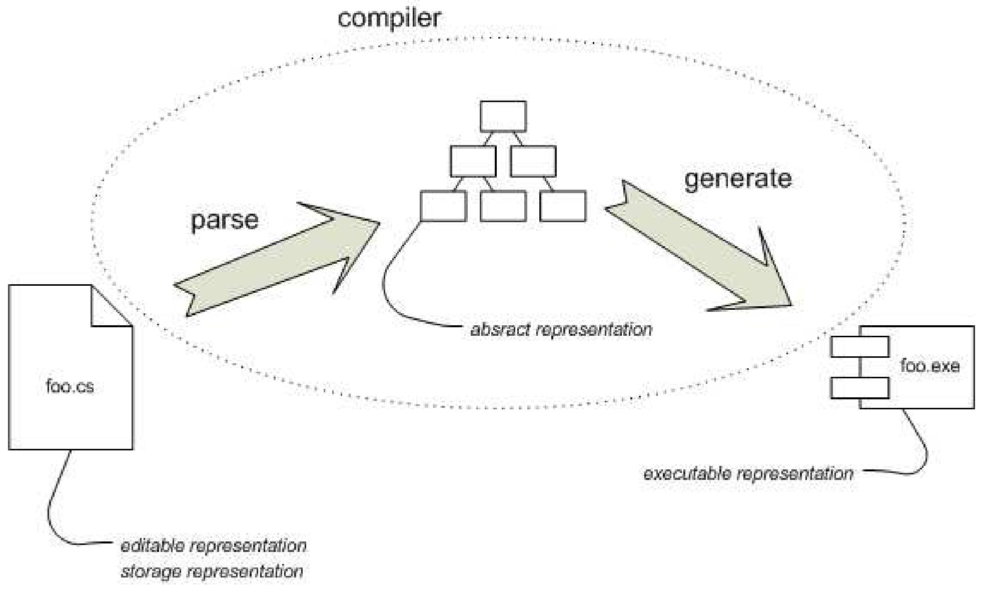

Figure 1: A outline of traditional compilation.

Let me break this process down a bit. Figure 1 shows a simplified view of the compilation process. To turn foo.cs into something executable, we run the compiler on it. For the purposes of this discussion we can break the compilation process into two steps. The first step takes the text from the file foo.cs and parses it into an abstract syntax tree (AST). The second step walks this tree generating CLR byte codes that it puts into an assembly (an exe file).

More detail on how you do code generation from an external DSL.

We can think of the program having a number of representations where the compiler translates between the representations. The source file is the editable representation - that is this is the representation that we manipulate when we want to change the program. It's also the storage representation - the one that's kept in source code control and used should we want to get at the program again. When we run the compiler the first phase maps the editable representation to the abstract representation (the abstract syntax tree), and then the code generator turns that into the executable representation (the CLR byte code).

(There are more translations on the executable code before it's really the final executable. But once we have the byte code the compiler's work is done and all that's left remains with later stages outside its scope.)

The abstract representation is very transient - it only exists while the compiler is running and serves only to separate the compilation into two logical steps. This transience is, of course, a large part of why it's so hard to get symbolic integration between external DSLs. Each language runs through a separate compilation, so there's no linking between the abstract representation. Things only come together with the generated code, at which point key abstractions are lost.

The more sophisticated post-IntelliJ IDEs bring a significant change to this model. When the IDE loads the file it creates an abstract representation in-memory, which it uses to help you edit the file. (Smalltalk did a limited version of this too.) This abstract representation helps with simple things like method name completion and sophisticated things like refactoring (automated refactoring is a transform on the abstract representation).

My colleague Matt Foemmel described how this struck him one time while working in IntelliJ. He made a change that was strongly assisted by these features and suddenly realized that he wasn't typing text - instead he was running commands against the abstract representation. Although the IDE translated these changes in abstract representation back into the text - it was really the abstract representation he was manipulating. If you've had a similar feeling while working with a modern IDE you're getting a sense of what a language workbench does.

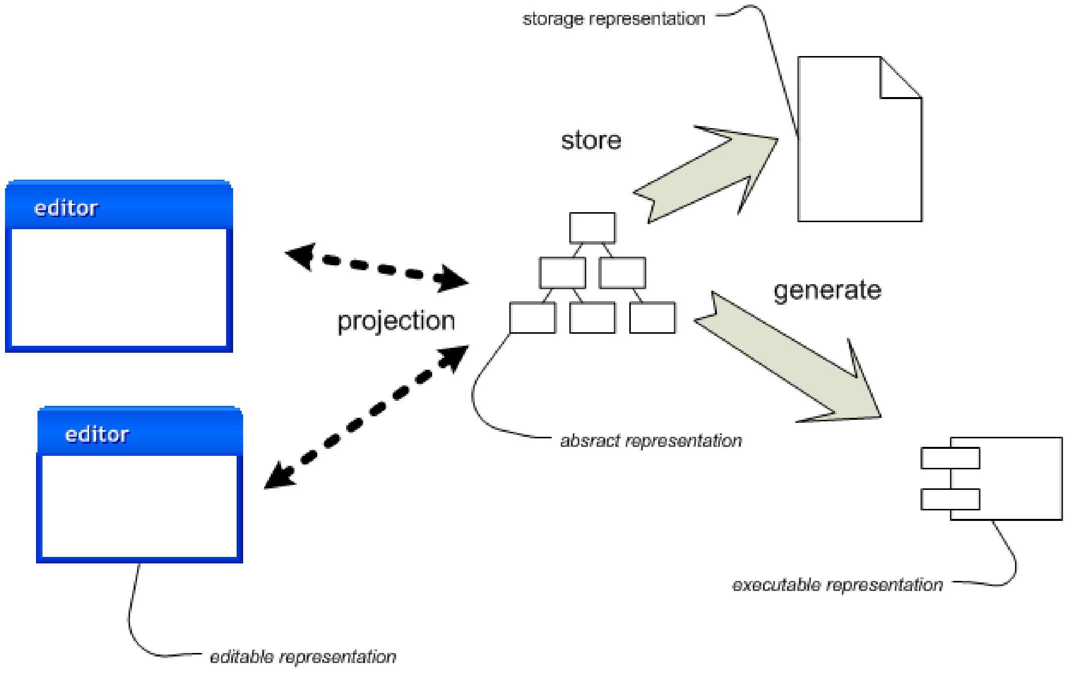

Figure 2: Manipulating representations with a language workbench.

Figure 2 shows how this process works with a language workbench. The key difference here is that the 'source' is no longer the editable textual files. The key source that you manipulate is the abstract representation itself. In order to edit it, the language workbench projects the abstract representation into some form of editable representation. But this editable representation is purely transient - it's only there to help the human. The true source is the persistent abstract representation.

The fact that the editable representation is merely a projection of the abstract representation leads to a few points. Perhaps the most important is that there is no need for the editable representation to be complete - some aspects of the abstract representation can be missing if they aren't important to the task at hand. Furthermore you can have multiple projections - each showing different aspects of the abstract representation. Since the projection is inside the language workbench the editable representation is much more active than a text file. This **projecting editor** is tightly bound up with the language itself. As a result in thinking about your editable representations you actively think about how an editor works with them. This leads to different ideas than you would get from a purely passive editable representation such as text.

A language workbench separates the storage representation from the editable representation. The storage representation is now a serialization of the abstract representation. A common way to do this is XML - but this XML isn't designed for human editing. Having XML as the storage representation is helpful for tool interoperability - although such interoperability is likely to be very hard.

The code generation is pretty much the same, although such tools are likely to treat traditional source as the executable representation. If they do generate regular language source files, these files aren't really source and like other generated code should not be edited directly. As language workbenches mature we should see more reliance on generating non-editable structures such as byte-code.

One non-obvious, yet important feature, for a language workbench is that the abstract representation has to be comfortable with errors and ambiguities. Traditionally people have felt that if you are to have an abstract representation it needs to be kept correct - you shouldn't be able to put incorrect information into it. This assumption, however, led to lousy usability. Post-IntelliJ IDEs realized this and react gracefully to erroneous states. For example you can perform refactorings on programs that have compilation errors (very necessary for good usability.)

This becomes even more important if you want to capture complex information from multiple sources. You can't keep everything consistent and correct all the time. So you have to deal with ambiguous and erroneous states - highlighting errors rather than refusing input. You should also allow people to easily enter non-computable information (such as documentation) into the model. This way scanned napkins can be linked directly to the resulting DSL code.

Defining a new DSL

With this kind of setup in place, there are three main parts to defining a new DSL:

- Define the abstract syntax, that is the **schema** of the abstract representation.
- Define an **editor** to let people manipulate the abstract representation through a projection.
- Define a **generator**. This describes how to translate the abstract representation into an executable representation. In practice the generator defines the semantics of the DSL.

This is the main trio, but there will be variations. As I indicated earlier, there is no reason why you can't have multiple editors or generators for a DSL. Multiple editors could be common. Different people may like different editing experiences. For example, Intentional's editor allows you to switch between different projections of the same model easily so you can view a hierarchic data structure as lispy lists, nested boxes, or a tree.

Multiple generators might appear for several reasons. You may want them to bind against different frameworks that do similar things. A good example of this would be the irritatingly multiple dialects of SQL. Another reason is for different implementation trade-offs with different performance characteristics or library dependencies. A third reason would be to generate different languages: allowing a single DSL to generate either Java or C#, for example.

Another optional extra might be to define translators for the storage representation. We can assume that language workbenches will come with a default storage schema that handles serialization of the abstract representation automatically. However you may want to generate alternative storage representations for interoperability or transfer between tools. Unlike the generator this would have to be a two way representation.

A different kind of generator would define human readable documentation - the language workbench equivalent of javadoc. Although most interaction with the language workbench would come through the editors, there'll still be a need to generate web or paper documentation.

Defining a Language Workbench

A concrete example of how to define a DSL using JetBrains's Meta-Programming System (MPS), which shows you how a language workbench works.

There is no generally accepted definition of what makes a language workbench. This isn't surprising as I've just made up the term for this article! But it strikes me that to avoid the rampant ambiguity that surrounds so many topics in the software business (eg components, Service Oriented Architecture), I should try to make a first stab of the essential characteristics of a language workbench, which I now can do briefly as I've provided the necessary background.

- Users can freely define new languages which are fully integrated with each other.
- The primary source of information is a persistent abstract representation.
- Language designers define a DSL in three main parts: schema, editor(s), and generator(s).
- Language users manipulate a DSL through a projectional editor.
- A language workbench can persist incomplete or contradictory information in its abstract representation.

How language workbenches alter the trade-offs for language oriented programming.

A while ago I discussed the trade-offs for language oriented programming. Language workbenches clearly affect that trade-off with a number of new things to consider.

The most obvious change that a language workbench makes to the equation is the ease of creating external DSLs. You no longer have to write a parser. You do have to define abstract syntax - but that's actually a pretty straightforward data modeling step. In addition your DSL gets a powerful IDE - although you do have to spend some time defining that editor. The generator is still something you have to do, and my sense is that it isn't much easier than it ever was. But then building a generator for a good and simple DSL is one of the easiest parts of the exercise.

The second big plus of a language workbench is that you get symbolic integration. The ability to take an excel-like formula language, and just plug it into your own specialized language is pretty nifty. As is the ability to change symbols in one language and have those changes ripple through the whole system, which is a plausible thing to consider with a language workbench (I'm not sure if any of them can do that yet.)

This issue of refactoring is one of the big issues in language workbenches. When I explain using a language workbench it's easy to fall into the trap of describing it as "first define a DSL, then build stuff using it." If you've read much of what I've written in the past that notion should set off many alarm bells. I'm a big advocate of evolutionary design - which in this context means that you need to be able to evolve a DSL and any code built in the DSL together. That's a hard problem, but one that was acknowledged since early on in Intentional's development. It's too early to tell how well evolving a DSL concurrently with its use will work out in mature language workbenches - but a lack of this capability would be a big negative against them.

The biggest medium term problem that I see for language workbenches is the risk of vendor lock-in. There are no standards for defining the trio of schema, editor and generator. Once you define a language in a language workbench you are tied to that language workbench. There's no standard for interchange between the different language workbenches - this would leave you with reimplementing the trio if you wanted to change your language workbench. It may be that over time we would see some kind of special storage representation designed to interchange DSLs - an interchange representation. But unless a robust story appears here, vendor lock-in remains a big risk. (MDA claims to offer an answer to this, but it's partial at best.)

One mitigation to this is if you see the language workbench as a tool to help you generate sources. An example of this would be to use a language workbench to get all your Java XML configuration files under control. If the worst happened and you had to ditch the language workbench then you still have the generated configuration files. Providing you pay attention to clean looking generated files, you might not even be worse off than writing them yourself. Even for deeper capabilities you could still generate well-structured Java code. This does mitigate the risk to some extent, at least you won't be completely high and dry. But vendor lock-in is still something to think about.

This question about tooling is one of the consequences of moving away from text files as sources. Other issues come up - issues that we've managed to solve with text but now have to rethink with the central role of an abstract representation. High on my list is that of version control. We know how to do effective version control for textual sources with good diff and merge capabilities. To be effective language workbenches will need to be able to provide diff and merge of the abstract representation itself. In theory this should be solvable and unlocks the opportunity of real semantic diffs (where renaming a symbol is understood as that act not just something you have to infer from its result as you do with text.) Intentional seem to have a good solution here, but we're yet to try it out in practice.

Back on the positive note, the combination of custom language and editor may finally open the way to make DSLs editable by non-programmers. In addition the symbolic integration removes the problem of user code and the core program getting out of sync. The use of editors may be the single biggest tool to help break the COBOL inference - providing environments where the tool is customized for user interaction.

This promise of bringing domain experts more directly into the development effort is perhaps the most tantalizing part of the language workbench promise. Time and time again we see that whatever tools we programmers use to boost our productivity there's a sense that we are optimizing the idle loop. On most projects I visit the biggest issue is the communication between the developers and the business. If that's working well, then you can make progress even with second rate technology. If that relationship is broken, than even Smalltalk won't save you.

Most proponents of language oriented programming talk about involving domain experts more. Indeed I've even heard claims of secretaries happily programming in Lisp's internal DSLs. Most of the time, however, these efforts have failed to really take off. By combining the advantages of a focused external DSL with a sophisticated editor and development environment, maybe we can finally begin to chip away at this problem. If so the upside will be enormous. Indeed its striking how much this user-involvement is seems to be the primary driving force behind Charles Simonyi's work, underpinning most of the decisions in Intentional Software.

The biggest short term limitation of these tools is maturity. It will take a while before these tools will hit even the leading edge of developers. But as we know that can change quickly - just reflect on tool and language choices a decade ago compared to now.

Changing our conception of DSLs

In this article I discus how the OMG Model Driven Architecture relates to language workbenches.

The examples I've used in this article are actually pretty uninteresting examples of DSLs. I used them because they were easy to talk about and build. But even the more complex agreement DSL is pretty conventional - it's easy to see how it could be done as a traditional textual DSL. Many people look to producing graphical DSLs, but even these don't capture the full potential. The biggest danger in using the term 'language' is that it can lead people to miss the point of what can really be done with language workbenches.

When I was talking with my colleagues about OOPSLA 2004, the biggest buzz was some demonstrations by Jonathon Edwards on Example Centric Programming. The key idea was an editor that showed not just program code, but also the results of example executions in that code. The idea was that although we manipulate an abstraction, we often find it easier to think in terms of concrete cases. This leaning towards examples is a large part of the appeal of Test Driven Development - I think of it as Specification by Example.

Edwards has developed his ideas further into a tool called Subtext. Subtext shares some principles of language workbenches - in particular the idea of moving away from textual source code. While subtext is less interesting in supporting easy definition of new languages, it provides an interesting glimpse of the kind of thinking that could develop as language workbenches make us think about language and tool as deeply intertwined.

Indeed this may be the strongest reason why language workbenches may be able to avoid the baleful influence of the COBOL inference. As I argued earlier, we constantly come up with technologies to empower users as lay programmers, but regularly fail. Let's consider one technology that has really succeeded in making lay programmers effective - spreadsheets.

Most programmers don't think of spreadsheets as a programming environment. Yet many lay programmers create sophisticated systems using them. Spreadsheets are a fascinating programming environment that suggest characteristics for a lay programming tool might need:

- Immediate feedback - including showing the results of example calculations right away.
- Deep integration of tool and language
- No textual source
- No need to show all information all the time - formulae are only visible when you edit the cell containing them, otherwise the value is shown.

Spreadsheets are also very frustrating. Their lack of structure encourages experimentation, but often I feel a touch more structure could make certain problems much easier to deal with.

So when we think of the DSLs in a language workbench, we should be thinking less of the kinds of languages I've shown here - or of the graphical languages beloved by modelers. Instead we should be thinking of things like the next generation of spreadsheets.

Conclusions

My main purpose in writing this was to give you an introduction to language workbenches. At the least I hope you now understand enough to hold your end up if your manager asks you to replace your entire programming environment with them.

As I see it, language workbenches offer two principal advantages. One is improved programmer productivity by giving them better tools for the job. The other is improving the productivity of development by forging a closer relationship with domain experts by allowing domain experts more opportunity to contribute directly to the development base. Only time will tell if these advantages will actually be realized. Looking at the two I would say that improving productivity is more likely to happen but carries less impact. If language workbenches made a serious impression on the relationship between development and domain experts it could have a tremendous effect - but it has to overcome the COBOL inference to succeed.

Perhaps the most interesting thing that I've come to realize is that we probably have little idea what DSLs will look like once we've had experiences with language workbenches. So far my thinking is still very much constrained by thinking about what textual and graphical languages are like. Yet the interplay of editors and schema opens up possibilities that are quite different to most people's idea of an external DSL. If language workbenches live up to their hopes, in ten years time we'll look back and laugh at what we now think DSLs should look like.

As I've indicated, language workbenches are still in a very early stage of development. It will be several years before we are able to seriously kick their tires. I'm not going to make predictions about whether they will, as their advocates hope, change the face of software development. I'm not much of a technology futurist. What I do believe is that language workbenches are one of the most interesting ideas that's out there on the edge of our vision. If they do realize their potential, they'll certainly have a huge effect on our profession. Even if not, I suspect they'll lead to plenty of interesting ideas.

So I suggest you keep an eye on this space. It's an interesting field and one with enough life to stay interesting for many years. I've been fortunate to have a good view of it in recent months, and I intend to continue my interest for a while yet.

Further Reading

I decided to put references to further reading here on my bliki. This way it's easier to keep track of updates.

Acknowledgments

My deepest thanks go to my fellow-ThoughtWorker Matt Foemmel. Matt has long been a lynch-pin toolsmith at ThoughtWorks and has constantly been looking for ways to jump forwards in our development work. He started getting interested in Intentional Programming in early 2004 and I've benefited greatly from his investigations. His active involvement with development at Intentional Software this last year has been very helpful to me in understanding this environment.

When I heard that one of the few software tools companies I admire were working in this field I was immediately interested. The fact that Sergey Dmitriev was based a few miles away from me in Boston made it even better. Sergey gave me incredible access to MPS as he's been developing it. He and his team has taken this agreement example and implementing it in MPS so that I could describe something that wasn't entirely vaporware. Igor Alshannikov helped me out when I had the inevitable problems you get with software that's still in mid development.

Intentional Software has been very quiet in the last few years as they've been developing their ideas. Charles Simonyi has granted me remarkable access to their tools and plans. I've also been able renew my collaborations with Magnus Christerson, who is also now at Intentional.

Like many people in the UK in the 80's and 90's, I benefited greatly from the leadership of the OO community there from

Steve Cook. Since then he's helped me through thickets of the UML specification and for this article he's been very helpful with information on Microsoft's Software Factories initiative. It's helped to see many long term friends of mine on this project: Keith Short, Jack Greenfield, Alan Wills, and Stuart Kent have all been great sources of information.

I've had several entertaining visits to MIT, thanks to professor Daniel Jackson. In particular he introduced me to Jonathon Edwards. Not for the first time I didn't really understand dramatic ideas when I first saw them, but I do learn eventually.

One of the greatest things about being at ThoughtWorks is ready access to very talented people doing interesting things. In this case it's been mighty useful to have access to people working closely with the Intentional tools: Matt Foemmel, Jeremy Stell-Smith, and Jason Wadsworth.

And speaking of fellow ThoughtWorkers, Rebecca Parsons and Dave Rice have been fine intellectual sounding boards - essential to keeping my thinking on track.

As well as providing this kind of background information to write these articles, I've also received helpful reviews on an early draft from Rebecca Parsons, Dave 'Bedarra' Thomas, Steve Cook, Jack Greenfield, Bill Caputo, Obie Fernandez, Magnus Christerson and Igor Alshannikov

Thanks for Reuven Yagel, Dave Hoover and Ravi Mohan for spotting and sending me typos.

Significant Revisions

12 June 2005: First publication.

Guides

Intro
Design
Agile
DSL
Delivery
About
Me

Popular Articles

New Methodology
Dependency Injection
Mocks aren't Stubs
Is Design Dead?
Continuous Integration

Books

Domain-Specific Languages
Refactoring
Patterns of Enterprise Application Architecture
UML Distilled
Analysis Patterns
Planning Extreme Programming
Signature Series

Site Sections

FAQ
Articles
Bliki
Books
EAA Catalog
EAA Dev

ThoughtWorks

Blogs
Careers
Mingle
Twist
Go