Intentional Software

Charles Simonyi
Intentional Software Corporation
500 108th Ave NE, #1050
Bellevue, WA, 98004
+1 425 467 6600
charless@intentsoft.com

Magnus Christerson
Intentional Software Corporation
500 108th Ave NE, #1050
Bellevue, WA, 98004
+1 425 467 6600
magnus@intentsoft.com

Shane Clifford
Intentional Software Corporation
500 108th Ave NE, #1050
Bellevue, WA, 98004
+1 425 467 6600
shane@intentsoft.com

Abstract

Wysiwyg editors simplified document creation by separating the document contents from the looks and by automating the re-application of the looks to changing contents. In the same way Intentional Software simplifies software creation by separating the software contents in terms of their various domains from the implementation of the software and by enabling automatic re-generation of the software as the contents change. This way, domain experts can work in parallel with programmers in their respective areas of expertise; and the repeated intermingling can be automated. Intentional Software is supported by a Domain Workbench tool where multiple domains can be defined, created, edited, transformed and integrated during software creation. Key features include a uniform representation of multiple interrelated domains, the ability to project the domains in multiple editable notations, and simple access for a program generator.

Categories and Subject Descriptors D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.3.3 [**Programming Languages**]: Language Constructs and Features.

General Terms Management, Design, Human Factors, Languages, Theory.

Keywords Intentional Software, Generative Programming.

1. Introduction

For numerous practical reasons the creation of software has been historically approached from the point of view of the computer. As a result most software is expressed in a general purpose programming language. Consequently, the programs record what is required for the computer, the detailed instructions for execution, rather than the problem details. This would not be an issue if only computers looked at programs, but that is evidently not the case. Because programs are the only precise (i.e. machine processable) artifacts that record the programmers' work, any change (whether maintenance or extension) has to be done to the program, which is encoded in a way that does not clearly express the problem that was to be solved.

Fred Brooks expressed the frustration of programmers thusly [1]:

> *"Much of the complexity [the programmer] must master is arbitrary complexity, forced without rhyme or reason by the many human institutions and systems…'*

Of course this complexity is not at all arbitrary from the point of view of the domain experts such as the hospital administrators, the content creators for a website, or the aeronautical engineers who have software needs.

The goal of software is to implement a solution to a problem that is defined by human intent alone. Some people have expertise in the problem domain - they are domain experts. Some people have expertise in software creation – they are programmers. Together they create software, perhaps with some people in both roles.

This division of labor also leaves the domain experts frustrated. Why can they not contribute more directly to the software? Granted, their potential contributions in terms of their domain expertise would not be executable by computer, but they evidently comprise an ever growing part of the total effort. The domain experts' plea is: Programmers, if you are frustrated with complexity of the domain and with changes in the specifications, give us a system where *we,* domain experts, can record and maintain our contributions in a way that is convenient for you to process by software.

The paper will present the following argument:

1. First we review the well-known idea of generative programming that promises to involve the domain experts in the creation of software by factoring the program into three parts: *domain schema*, *domain code*, and *generator code*.
2. The main emphasis will be on the domain code. The novelty of this paper's approach is to introduce a tool, the *Domain Workbench* that allows the combination of multiple domains that can be viewed and edited in multiple projections.
3. The domain schema defines the terms of the domain code and allows the generator to recognize and process these terms.
4. The generator processes the domain code and produces executable target code. The generator can be created using typical software engineering techniques.
5. The need for multiple domains and multiple projections is introduced through a simple example familiar to many programmers: specification of a parser through production rules and tree building templates.
6. Finally, additional applications are discussed and related approaches are reviewed.

1.1 Direct Programming and Intentionality

For the creation of any software, two kinds of contributions need to be combined even though they are not at all similar: those of the domain providing the problem statement and those of software engineering providing the implementation. They need to be woven together to form the program.

The direct development workflow in use today (Figure 1) is the following: The domain expert communicates the problem

OOPSLA'06 October 22-26, 2006, Portland, Oregon, USA.

statement to the programmer. This is done in forms that can not be automatically transformed into code, such as specs, use cases, stories, and sketches. The programmer then weaves these intentions together with software engineering knowledge and implementation decisions and creates the source code that can run on a computer.

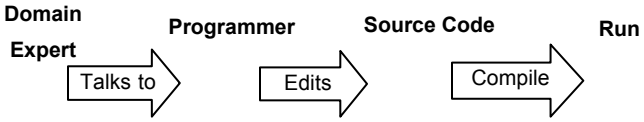


Figure 1. Direct programming workflow.

Persistent problems with this process stem from the fact that maintenance and correctness must be defined in the problem statement and in the implementation separately. However, maintenance has to be performed and correctness has to be evaluated on the source code mixture of the two.

Consider for example the following simple snippet from a banking application:

```
void Trnsfr(Acct payer, Acct payee, Dlrs amount)
{ AuditTrl(payer.Xact(amount),  payee.Xact(-amount)); }
```

This program is correct with respect to implementation types, yet it is incorrect in the domain: the payee should be credited the amount and vice versa.

When extending or otherwise maintaining a program, the programmer must mentally unweave the implementation (into Debits and Credits, for example), reason about the part in question, solve the problem and reweave the result. This repeated unweaving and reweaving is an added burden on the programmers that introduces many programming errors into the software and can increase costs disproportionate to the size of the problem.

The example could be refactored like this:

```
{ AuditTrail(payer.Debit(amount),  payee.Credit(amount)); }
```

By using the domain terms Credit and Debit, the efficiency of maintenance can be greatly improved. We may say that the second snippet is more *intentional* than the first – it is a more direct encoding of what was originally intended. Unfortunately, in complex systems a higher degree of intentionality is very difficult to achieve since a more complex domain vocabulary, domain relationships, and domain rules simply cannot be mapped into a programming language without a great deal of software engineering trickery which in turn makes maintenance exceedingly difficult.

1.2 Generative Programming

If we want to guarantee a high degree of intentionality, we have to express the problem in domain terms. Since by itself the expression won't necessarily constitute an executable program we have to rely on an additional transformational step: program generation.

Generative programming (Figure 2) is a technique where instead of focusing on the final program, the programmers focus on a generator program that takes *domain code* as its input and outputs the *target code* which is the final program. The domain code is expressed in a Domain Specific Language [4]. The target code can be a combination of any desired executable codes and other deliverables.

The generator is the place where programming still needs to get done; but the focus of the programmers' work has changed from writing the program directly, to writing the generator. Much of the programming effort is very much like direct programming – the program organization and the data structures *all need to be designed, defined, and written* except that now the target code can be parameterized by the domain code. The savings are achieved when the domain code is maintained and the generator is re-run. Now the computer does what ordinarily would be the programmer's task: distributing the domain code changes appropriately according to the complex correspondence between the domain and the chosen implementation.

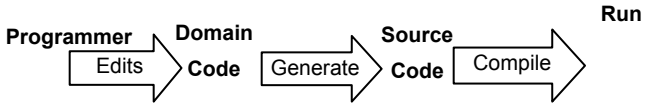


Figure 2 Generative programming workflow

The different applications of the generative techniques differ in how the domain specific language is defined, how it is edited, and how the generator itself gains access to the language. A classic overview of the range of possibilities is given in [3].

1.3 Intentional Software

By using a generator, the concerns of the domain experts and of the programmers can be separated. Nonetheless, many issues remain for practical application. The relative dearth of generative solutions today may be due to the lack of satisfactory resolutions for these issues: What is the language of the domain code? Who defines and who supports this language and its documentation? What if the problem spans more than one domain – for example banking and user interfaces? And what API does the generator use to access the domain code?

To create Intentional Software, we propose specific resolutions for the above issues and use a *Domain Workbench* for support.

The process starts by the domain experts–with support from the programmers—defining the *domain schema* (see below) that will serve as the interface between the Domain Workbench and the generator as well as their users the domain experts and programmers. The domain schema contains at least a distinct identity (also called an intentional definition or *def*) for each concept of the domain. [1]

Were the domain chess, for example, we would need a def for the pieces, the colors, the board and its squares, and the various states: initial, check, checkmate, draw. It is not required that we define the game in the schema – presumably that will be a part of the program that we create – but we need to have the vocabulary complete enough to define the game.

If the domain is banking, we need defs for kinds of customers, for accounts, for credit and debit, for loan, debt, and interest; and also for the different banking services, personnel titles, banking procedures and so on. Again, this is by no means a definition of

[1] Conceptions of William James [11], or acquaintance objects as explained by Peter Naur [17] are remarkably similar to the intentional definitions, in that they have an identity but no fixed definition other than through their participation in a stream of consciousness.

banking but something that is essential if we want to say something about banking.

The *domain code* is maintained by the domain experts. The need for a fixed language is eliminated by the Domain Workbench's ability to project contents from multiple interrelated domains in multiple views.

Finally, a *generator* is written by the programmers to define how the domain code input should be processed to get the intended implementation, see Figure 3. In other words, the generator represents implementation knowledge such as engineering design, algorithm choices, platforms and code patterns used (essentially all the work that programmers typically perform *except* for the domain details in the domain code). So the generator excludes some information that is in sources produced by direct programming, but also *includes* information that is not found in direct sources, namely in what precise ways the target code depends on the problem details.

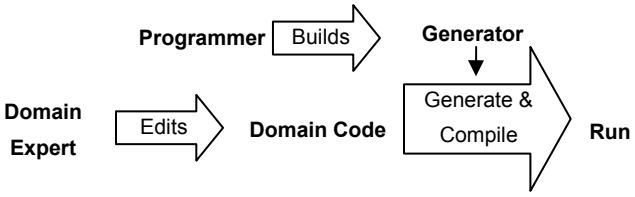


Figure 3. Intentional Software creation workflow

How do the domain schema and domain code differ from classes and objects? The main difference is the complete lack of *a priori* semantics. The domain expert need not make a decision whether the domain concept is an object, a quality, an action, a goal, or anything else. The need is just to establish an identity for it, whatever it may be, and that can be accomplished always in the same way—by a def in the schema.

On the other hand, how are the domain schema and domain code different from UML? The difference is the absence of *prior restrictions* of what can be represented in the domains that are described by the schemas. The generator is the ultimate guarantor of extensibility, but to do that the generator may need arbitrary forms in the domain code. This in turn is guaranteed by the extensibility of the schema language.

Through bootstrapping, circular arguments are permitted in software engineering. So we can say that the schema language is extendible with new properties (that might be considered meta-properties in this context.) It can also be projected and edited in a number of notations because every language in a Domain Workbench, partly with the guidance of schemas, can be so extended and so edited.

Given this ability to add properties to languages, we can enrich the schema language to contain the meta-information needed for the various domains. In effect, the schema is simply a database of meta-information to be used by the generator and by the editor.

2. The Domain Workbench

The Domain Workbench is used to define, create, edit, transform and integrate multiple domains. The Domain Workbench is an instance of the Language Workbench class of tools as described by Fowler [8].

Wysiwyg editors [14] are a useful historical metaphor for what can be done when the demand for the expression and manipulation of more complex data arises. Before Wysiwyg became popular, editors were called text-editors since they represented only the text contents of documents. As an intermediate stage of development, special character combinations (or "control codes") were used to encode what the formatting should be. For example, ^i might turn on italics. HTML and its successors still use this style of encoding. Today, a word-processing user does not have to make a distinction between the italic character *i* and its underlying representation. Wysiwyg editors separate the display of the document on the screen ("What you see") and what is printed ("what you get") from the underlying representation of the document. That separation makes it possible for the user to create and edit much more complex and effective documents.

The editor of the Domain Workbench applies the same Wysiwyg technique to software sources. It separates the display of the software and what is generated from the underlying representation of the software—whether schemas, domain code, or programming language source code. The representation is in a uniform format for all domains: the intentional tree. Display and editing is performed by projecting the tree on a display by a number of reversible transformations–some of which may be domain specific. All are sensitive to user options that select from the many available views to project.

2.1 A simple example

Consider the following program statement as we used to write it:

```
return a = b / (c + 1);
```

Naturally there is a projection of this statement that looks exactly like text:

```
return a = b / (c + 1);
```

But how does the intentional tree look? The editor has another projection that shows the underlying intentional tree like this:

```
Return
(
   Assign
   (
      a,
      Div
      (
         b,
         Plus
         (
            c,
            1
         )
      )
   )
)
```

There are differences in the information shown in the program text and tree representations. What happened to the semicolon (`;`) and the parentheses that are in the program text and not in the tree? These are artifacts of the programming language syntax. The semicolon is implied by the structure below the `Return`

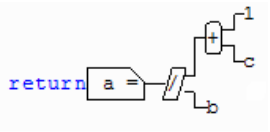
statement: it may or may not be needed depending on the syntax and depending whether the statement is the last in the statement list. At any rate the semicolon does not convey any intention.

Compare the above tree with this tree:

```
Plus
(
   Div
   (
      b,
      c
   ),
   1
)
```

which describes the different formula b / c + 1 instead of b / (c+1). So the parentheses are artifacts of the algebraic notation that takes the traditional precedences of the / and + operators into account. There is an intention expressed (namely which operator we want to apply to the result of the other) but we express this in terms of the intentional tree structure, not some additional participants like the left and right parentheses. By changing settings, we can project the original intentional tree into other different notations:

```
return a = b / c + 1;
```



Note that the intentions are so clear there is no need at all for parentheses. We realize that in the program text notation the unambiguousness of programming languages presupposes that we have perfect knowledge of possibly obscure language details, such as the relative precedence of "&&" to "||". When we lack this knowledge, the theoretically unambiguous notation is in fact ambiguous for us. By projecting using a different notation (that may also be ambiguous but along a different axis) we can easily resolve the ambiguity.

As we will see, mixing and integrating notations and domains with software programs make software creation more effective for the programmer. This also allows projections that non-programming domain experts can understand and edit.

2.2 Structured Source Code and Structured Editors

I mean, source code in files;
how quaint, how seventies!

Kent Beck

The Domain Workbench builds on two key ideas that are decades old: Structured Source Code and Structured Editors.

Consider two questions about a programming language. "How do you write an "if" in Ada?", and "Does Ada have coroutines?". The first question is about notation or syntax, the second is about semantics. These are the two main components of a text-based programming language. These two components are also reflected by the structure of a typical compiler: the front end embodies the knowledge of the syntax, while the middle part defines the semantics. (The back end of a compiler expresses the knowledge of the target machine, which is not important here.)

Structured source code means that we store, maintain and process the input to the compiler's middle part directly. In computer science this is called an Abstract Syntax Tree (AST). The intentional tree is similar to an AST, but there are some key differences that we will discuss in Section 2.3. The format of the tree structure could be XML [20], but we use the proprietary Intentional Tree [19] representation.

A *structured editor* is an editor that is cognizant of some higher organization in the edited data. Wysiwyg editors such as Bravo [14] or Microsoft Word can be called structured.

Especially when applied to program code, there are two major kinds of structured editors:

> A *syntax directed editor* is an editor that knows about a language syntax that is used ostensibly to help the user. Syntax directed editors [12] operate well within the constraints of the syntax with many of the advantages and disadvantages thereof that are outside of the purview of this paper.
>
> A *projecting editor* transforms, or projects, the underlying representation into one or more notations and may be able to invert the transformation to effect changes to the representation. CAD systems and modeling tools are typically projecting editors.

Early work on structured source code used in combination with structured editors had two goals. The now obsolete initial goal was to improve compiler performance. The other goal was to help implement syntax directed editors where it was thought that strict control of the input and context-specific typing prompts would help the user enter programs more rapidly and with fewer errors. By and large these efforts were disappointing in that limitations of the input tended to get in the way of the creative process where the program text often passes through intermediate stages of completeness and correctness. For example, one might start with the interior of a loop and work outward.

In contrast with the above goals, Intentional Software uses structured source code and a projecting editor not to help with enforcing a syntax but so that domains are not limited by syntax. Definition of new domains and arbitrary extension of existing domains, can be difficult to accomplish if a parsable and unambiguous syntax also has to be designed.

There are several reasons for separating the syntax from the domain. We gain notational flexibility in that many notations we observe both in programming domains and other domains can be equally accommodated. This also results in greater user satisfaction because individual preferences differ, not only out of caprice, but also due to individual backgrounds and experiences. Individuals prefer what is familiar from prior contexts.

But the most important reason for separation is that computer notation then no longer needs to be unambiguous. It can thus more closely resemble the domains' own notations. In fact very few if any of the everyday domain notations are unambiguous. A good example is a drawing of a building facade – it is just one view. We know that the complete definition of the building means a whole set of drawings, or more recently a CAD database from which any drawing can be printed as needed.

The difference between notations and syntax is subtle but important. Notations need not be parsable, so they can be quite freely chosen on the basis of qualities other than parsability. For example, we can pick notations that simply look good, or are traditional in the domain, or otherwise please the users. Even ambiguity need not be an obstacle. For example, the common mathematical notation A_j is ambiguous yet it can be what the user wants and likes. Note that the structure that is projected is never ambiguous, but the projection might be. In the proper context, an ambiguous projection need not create an ambiguous impression in the mind of the person using the editor. For example, mechanical drawings can be very ambiguous. A circle may well denote the void of a hole in a plate or the cross section of a steel shaft, yet the engineer familiar with the context would not be confused.

However, key advantages of text representation that have assured its preponderance are worth reviewing. Text representations are easy to implement, especially when using traditional input devices such as punched cards, keyboards, and glass teletypes. Text languages consequently have a very simple editing model. For example, to create the program fragment `a+b` one simply has to press the keys a, +, and b. So the editing model and the representation model coincide. But this advantage is not going to remain sufficient in the long run to compensate for the disadvantages enumerated in Section 2.5, especially when most other application programs have abandoned text based representation and introduced more effective editing models. For example, slide presentations and spreadsheets are not created just by using text syntax.

2.3 Intentional Tree

```
Plus
(
   Div
   (
      b,
      c
   ),
   1
)
```

The famous painting by Magritte ("*Ceci n'est pas une pipe*") is not a pipe but a painting of a pipe. Similarly, the illustration of the tree representation above is not the tree; it is a *projection* of the tree. The actual tree, the *domain code*, is a data structure with nodes that have an "isa" field pointing to the intentional definition (def) in the domain schema. This is indicated in the above projection by names such as "Plus" or "b".

A node may have children, and may also contain arbitrary binary data whose interpretation is based upon the node's definition. For example, a "`NumLit`" together with the appropriate bits (...001) is projected above as the constant 1. The definitions themselves, represented in a *domain schema*, are also nodes in trees and as such form a domain of their own. For example, we can define a *name* as a "`TextLit`" parameter underneath the definition:

```
def (name (TextLit "Plus"))
```

Figure 4 illustrates that there are no fundamental distinctions between nodes in the domain code and definitions in the domain schema.

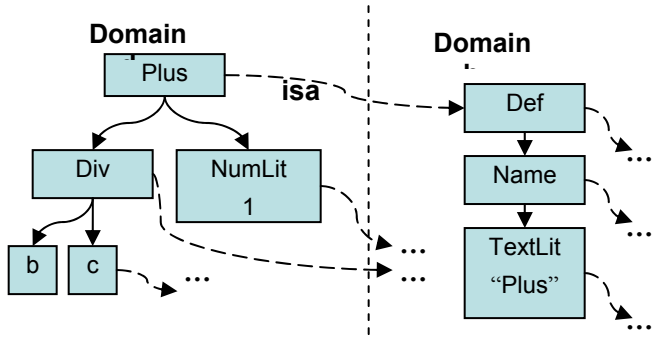


Figure 4. Domain Code and Domain

The `name` property of a definition is not fundamentally different from a string literal parameter in a print statement, or from a comment that looks like this in the tree:

```
Comment (TextLit 'this is a comment')
```

Note that there are no comment delimiters or string quotes stored in the string, as opposed to XML. The proper delimiters for any given notation will be supplied by the respective projection. Of course uses of names, string literals, or comments are completely different, but the underlying representation can be the same.

Treating names and other strings the same way eliminates the need for limiting rules on what can be a name or whether more than one name can decorate a node. Domains often use names that programming languages do not support, for instance spaces or characters from non-English alphabets. In the intentional tree names are like comments. Names are there to communicate with the users; they are not internal identifiers.

The representation of the intentional tree is implemented in a special purpose system [19], rather than using a general purpose encoding of, say, XML. In a well designed system, solutions to all the issues that come up in practical use can be implemented uniformly to meet the highest standards. Examples include how to maintain identity reliably and independently of names; renamings; what happens if a definition is deleted; how to handle undo; versioning and groupware; how to search efficiently; and so on.

In addition to maintaining identities, the most important requirement is the ability to refine any node at any time to an arbitrary extent by adding parameters without disturbing the existing parameterization. In domains this corresponds to the need for continuous refinement. Refinement is used to add operands, attributes, properties, formatting, subscripting, tags, or decorations. For example, we can make the word "is" in the comment italic, by adding a parameter like this:

```
comment(TextLit 'this '
        TextLit 'is' (Italics)
        TextLit ' a comment')
```

When strings are available it is always tempting to encode new information into the strings using special syntax. We could have written:

```
comment(TextLit 'this <i>is</i> a comment')
```

The disadvantage would be that none of the guarantees and benefits of the intentional tree representation would accrue to the text formatting.

2.4 Domain Schemas and Transformations

Domain definitions, such as TextLit or Plus, are defined as nodes in *Domain Schemas* that are also defined in an intentional tree using a *definition* domain as its meta-domain. The description of this meta-domain is outside the scope of this paper. It is worth noting, however, that even a minimal definition is usually sufficient to start using it in intentional domain code. Any new definition will inherit sufficient services to be useful. For example, it will show up in name lists within the domain, and it can be displayed in any number of the universal tree display projections such as the one in Section 2.3.

The semantic value of a definition is ultimately derived from intentional tree *transformations* that do something with it. For this the transformations need only a minimal definition – the identity of the intention. Meta-data associated with the definition –the Schema content – serves merely to parameterize the transformation so that more general transformations can be performed without requiring meta-data specific code.

For example, a name string in the definition allows the use of a general name lookup facility, possibly using multiple names of any specific node. Field descriptions under the definition enable the use of menus that can prompt the user for common parameters or auto completion. In addition, help text in the definition is used to generate a help facility.

For example, the definition of `Plus` can list the notations for `Plus`, including its possible names, possible symbols, methods for computing Plus, and information for generating code for Plus. The definition can also include help text and the precedence of addition operator in multiple standard languages. All of this information is there to support specific transformations, so that generic methods can be used to evaluate expressions, to project the operation in multiple ways, or to generate code or a help facility for Plus.

These benefits are there to support the schemas as well. Schema notation can evolve and improve as needs expand. Similarly, new kinds of information can be added to schemas.

Because each node is self-identifying through its schema, mixing trees from different domains becomes practical. See Sections 3.2, 4.1.1 and 4.1.7 for some examples.

2.5 Projections and Editing

Domain code and domain schemas are just data. They have no behavior on their own. Transformations add behavior, and a special type of transformation is the *projection* for editing in the Domain Workbench. But how can arbitrary editing work consistently across various notations? The well-established cut, copy, and paste model works well for most structured editing. The main difference is that in text editing, the user perceives two kinds of selections: a place and an extent. There may be many different ways of selecting extents: by drawing through them, by double clicking for words, or clicking on the left margin to select lines and paragraphs. When satisfied with a selection, the user can cut, copy, paste, or move the selected text.

In structured editing, a few more kinds of selections become available. This is not unusual. For instance, a spreadsheet application is a form of structured editor, where the selection has to distinguish between whole cells and the contents of a cell. Since the user is predisposed to making exactly that kind of distinction, the distinction between cell selections and contents selection in spreadsheets does not raise usability issues.

Similarly, in the Domain Workbench we use selection to distinguish the following editing situations:

1. Prepare for editing just one node in the tree ("crown selection")
2. Prepare for editing a subtree or forest ("tree/range selection")
3. Insert into a list ("place selection")
4. Parameterize ie. add new children ("under selection")
5. Operate on a subtree ie. splice above a node ("wrap selection")
6. Edit text contents ("text selection")

These selections provide an essentially complete set because any tree can be built, and editing operations to change a current tree into a desired tree are reasonably intuitive. For example, to negate an expression, the user wrap selects the expression in question and pastes the negation operator node.

Each selection is identified and highlighted in an image of the underlying node or subtree in a projection, so it is largely independent of the projections. For example, to change a division into a multiplication, the user crown selects the division sign (in whatever format is projected for example x/y or $\frac{x}{y}$) by clicking on its image and pasting a multiplication operator node (perhaps from a menu). This can be accomplished in any notation that has any image of the division, including if it is denoted by a fractional line. Often making a selection and seeing the highlight helps one to navigate complex formulas and visualize what operations apply to what operands.

When a selection is in text, editing can continue in a conventional manner. The keyboard is used to input intentional information in two stages. First, sufficient text is entered and projected at the place of editing to convey the user intention. This text is not yet a real edit on the intentional tree. Second, the editor must interpret the text and convert it into an intentional node or nodes at which point the edit–effectively a "paste"–takes place. Each interpretation at a minimum must recognize names by using name tables from which the intentional reference can be determined or by user interface techniques in case of name ambiguity or other complication. Numbers must be also recognized by the usual rules. At this point an arbitrary tree can already be built using the keyboard and the selections.

The simplicity of editing text-based languages comes at a high cost. In effect, the user is continually expected to manually project the underlying intention into the text representation, just so the system can claim simplicity for its editing model! Editing in the Domain Workbench editor can also be perceived as simple because the user can think of editing the underlying intention, not somehow trying to manipulate the projection. This may take a little getting used to just as Wysiwyg surprised its first users by making text flow from one line to another without typing carriage returns. If your expectation was that you are essentially using a typewriter (a "glass teletype") such behavior could have been surprising and disturbing. On the other hand, if your expectation is that you are interacting with the contents and you let the system

handle the text layout, the flowing of the text is now perceived as perfectly natural. In fact, when text does not flow that is the surprising exception that may indicate a problem.

Intentional interpretation of text can be extended to parsing of various simple syntaxes, such as the traditional syntax for expressions or code snippets from programming languages that are already defined to be parsable. Note that this is not the same as using syntax to encode a result. Parsing is just another input device to augment the mouse, menus, and control keys of the editor. Once the data has been intentionalized–turned into intentional tree form—all the guarantees will hold. To bring legacy definitions and code into the Intentional tree Parsing is a convenient way to bring legacy definitions and code into the intentional tree.

3. An Example: Grammars

Despite the fact that the Domain Workbench does not rely extensively on syntax information, grammars can still be used as an example of an intentional domain. A grammar is not executable by itself, but can be used as input to a generator that generates an executable program–in this example a parser. Grammar also has the advantage of being very familiar to computer scientists and comparable applications exist, most notably YACC [21].

An example is also valuable because a similar approach could be used to implement business software. Of course, business domains are very different from grammars. But for comparison purposes they have much in common. Most importantly, neither is executable. Furthermore, both are repositories of complex information that define a specific set of decisions and rules from some more general domain.

In the production rules of a grammar, the decisions described are the structure of the language to be defined, the nomenclature of the syntactic categories, and the like. The general domain is the theory of computer languages. For a business domain, the specific decisions described can include the nature of the process steps, their connection to each other, business rules and the nomenclature. A process domain can be a set of common process steps such as process initiation, communication, approval and the common agents and concepts such as customer, invoice, or payment. In both cases the theory comes with a large number of operations, combination rules for the operations, and whatever common notations the domain experts prefer.

The purpose of our comparison with YACC is not to belabor the obvious—that after more than 20 years of advancement in hardware and software technology we can produce better looking output. The purpose is to suggest that the Domain Workbench can simplify the creation of tools like YACC. The novelty is not the quality of the output, but the ease by which that quality can be achieved.

3.1 Intentional Production Rules

Figure 5 shows two production rules for a small part of the C# language shown as projections in the Domain Workbench editor. The underlying tree for the second of these is as follows:

```
ProductionRule
(
    Name( “using”),

    Concatenate
    (
        “using”,
        type-name,
        Optional
        (
            “=“,
            type-name
        ),
        “;”
    )
)
```

The production rule is expressed as the definition with a name so that it shows up in the name lists that the editor supports. Within the expression we have three kinds of operators:

terminals – coded as literals,

non-terminals – coded as nodes referencing the corresponding production rule,

meta-operators – such as `ProductionRule` or `Concatenate`.

Using the domain schema we can easily make changes such as the following without invalidating legacy code that has already been developed in this domain:

change or extend the nomenclature (the names of the meta-operators),

add new meta-operators,

extend the current meta-operators with new or different parameters,

introduce new notations,

and enjoy other benefits of the Domain Workbench such as groupware.

Legacy domain code only needs to be modified if the domain schema changes radically, *and* if it is desirable to eliminate references to the old schema design (for usability or maintenance reasons). Multiple schemas can easily coexist for an interim

```
namespace ::= "namespace"  type-name  "{" {using}* {namespace-member-declaration}* "}" [ ";"]

using ::= "using"  type-name [ "="  type-name] ";"
```

Figure 5. Projecting a subset of the C# grammar.

```
<namespace> ::= "namespace"  <type-name>  "{"  Repeat(<using>; 0.. ∞)
   Repeat(<namespace-member-declaration>; 0.. ∞)  "}"  Repeat(";"; 0.. 1);

<using> ::= "using"  <type-name>  Repeat("="  <type-name>; 0.. 1)  ";";
```

Figure 6. Projecting the subset using a different notation.

period—even using the same projections and the same nomenclature.

Domain-specific names can also be accommodated. For example, non-terminals in production rules often use hyphenated names as shown in Figures 5-8. The use of common terms, such as "Concatenate," will not limit use of these productions when mixed with other domains that also use the same term. Since names are used only to communicate with the user, name overloading affects only the user, not the system. In everyday life we are well equipped to handle occasional name ambiguity. For example, we would not otherwise use first names.

In the projection in Figure 5, the symbols * and [..] denote ZeroOrMore and Optional repetitions respectively. If the user forgets which one is which, the user can simply change to the projection shown in Figure 6 where the repetitions are shown explicitly. This projection is also editable. But that does not mean the lower and upper limits are arbitrary (for example `Repeat(...; 2..3)`) *unless* such an operator is also defined for the domain. To change the first Repeat clause, it (or the * in Figure 5) can be selected and "Optional" can be typed or chosen from a name list for pasting over. In response the system will echo `Repeat(...; 0..1)`.

This example shows the difference between what the intentional source is, how it is viewed, and how it can be edited. When these activities are separated, the domain designer can choose to invest more or less support into any or all of them depending on user feedback. The interesting thing is how much is possible with little or no special support.

Other projections can be created for the same domain with dramatically different effect. In Figure 6 we used the < > brackets instead of italics to distinguish non-terminals. Figure 7 uses capitals, as in COBOL manuals with a twist on the nomenclature. Figure 8 uses "railroad" diagrams where non-terminals are distinguished by both font and borders. Note that each of these projections uses exactly the same intentional tree representation. Only the projections are altered.

In Figure 8, in contrast to the "Repeat" case, it would be quite natural for users to change the first loop by picking "Optional" from a menu. This is because when the notation is graphical the user simply does not expect to type what the user wants to see.

3.2 Extending and mixing domains

Extending and mixing domains are useful capabilities. For example, we have already made a small extension the grammar domain: we have added a repetition construct to the basic BNF notation to shorten the syntax productions. (Section 3.4 shows in part the extra productions that would have been necessary had we not used repetition.)

The goal of these productions is not only to parse syntax but also to build trees. There are several requirements: we need to specify the *template* for the tree to build and we need to connect the parts of the productions to the variable parts of the template. The additional goal is easily met by wrapping the individual parts, giving them names when the non-terminal name would not be sufficient, and building the template by actually quoting the desired results. Compare the following native projection with those in Section 3.1:

```
ProductionRule
(
    ProductionOutput
    (
        Name("namespace"),

        Concatenate
        (
            "using",
            ProductionOutput
            (
                Name("Alias"),
                type-name,
            )
            Optional
            (
                "=",
                ProductionOutput
                (
                    Name("Reference"),
                    type-name,
                )
            ),
        ";"
        )
        Template
        (
            Using
                (
                    Reference,
                    Alias,
                )
        )
    )
)
```

These productions can be projected without the template-related information as in Section 3.1, or with it included (see Figure 9.)

The projection in Figure 9 neatly separates the production rules

```
NAMESPACE ::= "namespace"  TYPE-NAME  "{"  AnyNumberOf(USING)
   AnyNumberOf(NAMESPACE-MEMBER-DECLARATION)  "}"  Opt(";")

USING ::= "using"  TYPE-NAME  Opt("="  TYPE-NAME)  ";"
```

Figure 7. Projecting the subset of the C# grammar using CAPITALS.

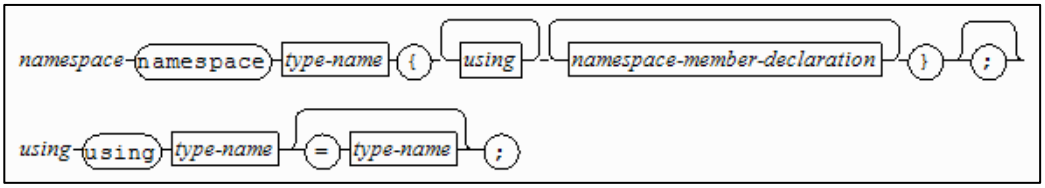


Figure 8. Projecting using railroad notation.

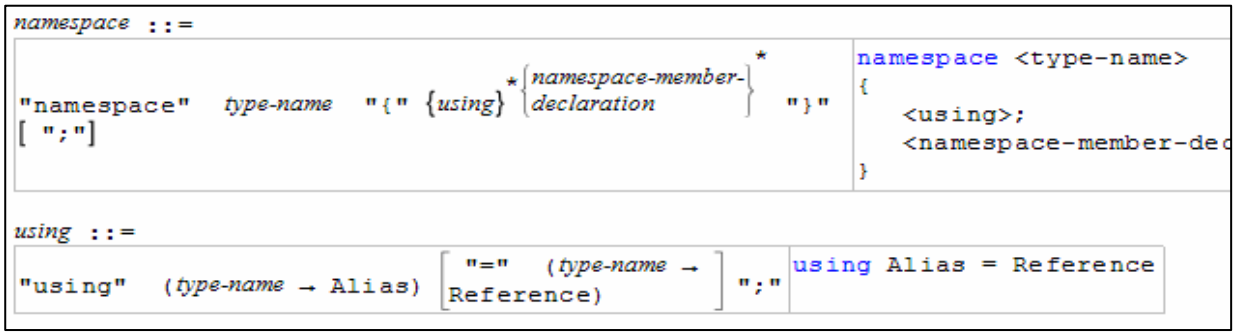


Figure 9. Mixing the grammar domain and the template domain.

from the templates that reference it. The projection of the templates (in the right column) uses the C# language domain.

Note that this domain code combines three domains: the domain of grammars, the domain of tree-building templates, and the domain of C# (where the actual definitions of the source intention *namespace* and *using* reside). The productions named *namespace* and *using* define the syntax only. Note also that the same name is used to denote two very different, yet related, things. If that is not tolerable, renaming and multiple names can always be used.

3.3 The Generator

The generator is a class written in C# that creates the parser (also in C#) as directed by the domain code. The Domain Workbench provides an API whereby the generator program can access the domain schemas and the domain code. The parser uses a simple name binder and textbook algorithms.

When the parser is given the above sources and the string

```
"namespace Foo { using Bar ; }"
```

the output will be a tree that is projected:

```
namespace Foo
{
   using Bar;
}
```

3.4 Comparison with YACC

These results can be directly compared with YACC files that perform essentially the same tasks. Following are the YACC statements equivalent to the domain code in Section 3.2:

```
namespace : 'namespace' type-name
    {
    nsCurrent = new Ns($2, nsCurrent);
    }
    '{' optional-usings
        optional-namespace-member-declarations
        namespace_close
        {
                nsCurrent = nsCurrent.parent;
        }

namespace_close : '}' ';' | '}'

optional-usings : | usings

usings : using | using usings

using : 'using' alias '=' type-name  {
         nsCurrent.AddChild(new Us($4, $2)); }
        |'using' type-name {
         nsCurrent.AddChild(new Us($2)); }

alias : type-name

optional-namespace-member-declarations :
 | namespace-member-declarations

namespace-member-declarations :
namespace-member-declaration
       | namespace-member-declaration
       namespace-member-declarations
```

First, note the extra productions above that were not necessary in the Intentional system. Those with names `"optional-*"` were implemented by the Optional construct. The production named `"namespace_close"` was necessary to allow the association of a template with both alternatives, with or without the semicolon.

Second, the parse tree is built using templates written in a general purpose language like C++. This assures flexibility, but it is complicated for normal cases. An API must be written for building the tree, in this example the classes `Ns` for `namespace` and `Us` for `using`. Irrespective of how the YACC template is expressed, how to connect it to parts of the production remains an issue. YACC uses the pragmatic approach of integer indices into the production. For example, $4 represents what was returned by the 4th item in the production:

```
'using' alias '=' type-name
```

namely `type-name`.

To summarize the differences:

i) Enhancing expressiveness of a domain can improve intentionality of the domain. The Domain Workbench expressed with 2 productions what took 8 productions in YACC. The "extra" 6 productions in YACC were unrelated to the domain, having more to do with the lack of expressiveness. Especially for larger, realistic problems, this expansion (a factor of 4 in this case) from the domain into a representation of the domain can obscure what is being described and is an obstacle to maintenance and enhancements.

ii) Domain enhancement can continue in the Domain Workbench. For example, the "OneOrMoreOf" construct could also specify the delimiter before and after the list and in between the list items. This further reduces the clutter and separates essential information (that there is a list) from the stylistic (that the list is separated by commas).

iii) When domains can be combined clearly (with reliable references across the boundaries,) we can create a template domain that is related to the target language thus avoiding or delaying use of a general purpose programming language (until truly exceptional situations arise).

iv) Replacement of fragile integer references (such as $4 in YACC) with intentional references also eliminates a large source of potential problems.

These advantages are individually small but generally applicable and therefore have great cumulative effect.

Just as it is simple to add capabilities to the domain schema, it is also possible to keep unnecessary constructs out of the domain; for example, eliminating the extra productions and omitting the templates from the projections. The purity of the domain helps with maintenance and with processing the domain for new purposes. For example, the grammar domain may be used to create a framework for documentation, or to generate a projection automatically from an intentional tree into the particular notation described by the domain.

4. Applications

The capabilities of the Domain Workbench can be applied in many domains including various Software Engineering domains. The ability to define, create, edit, transform and integrate multiple domains creates an opportunity to rethink some Software Engineering practices.

Depending on the focus of the Domain Schemas we can distinguish between *language oriented development* (where the notations, the schemas, and the generator closely mirror and extend existing general purpose languages) and *domain oriented development* (where a business domain is the starting point and the generator is built to create a domain specific solution.)

4.1 Language-oriented development

4.1.1 Mixing sources from multiple languages

Many software systems are implemented using multiple languages, for example Java, Javascript, and HTML. The biggest problems here are references from one language into another and sharing of definitions and other meta-data. In Section 2.3 we showed how the intentional tree representation expresses references. Similarly all meta-data can be shared and manual copying and translation of definitions can be avoided. The editor provides a common editing experience across all languages.

Figure 10 shows a projection where a SQL statement (using a SQL domain) has been mixed into a C# program (using a C# domain). Note the reference to the C# formal parameter `ADDRESSID` in the SQL code. This is not a SQL feature, but a simple generator can create the files for the fragments to run in standard C# and SQL environments.

4.1.2 Extending a language

Language features are currently associated with specific languages and a great deal of energy is spent explaining the coherence or cleanliness of the rather arbitrary feature bundles we call languages. Just as programmers are free to define APIs and data structures today (for better or for worse), using the Domain Workbench they will also be able to implement whatever language features will help them.

4.1.3 Names

Names in programming languages serve two purposes. One has to do with identification of references, and the other has to do with human factors (as a name is also a communication, mnemonic and thinking aid).

With these purposes separated, each purpose can be better served. A quantity can have multiple names. The old name after renaming can be retained to ensure continuity with legacy. Localized names can be used in national alphabets. Nicknames can be used for quick entry. Iconic names can be used for learning.

Names that follow a convention (for example using the type as a name) can be generated automatically.

4.1.4 Eliminating code copying and aspect fragmentation

Language extension should be considered when code is copied or when a particular concern (or aspect) is fragmented over many parts of the source. The former is frequently due to the difficulty of expressing performance concerns or for parameterization in a context where the language does not permit it. The latter gave rise to Aspect-Oriented Programming [13] with language features that centralize the expression of aspects.

Code copying and aspect fragmentation cause many of the most common maintenance problems [7].

4.1.5 Personal preferences

A frequent source of frustration in any environment arises from conflicts between personal preferences and external standards. Many of these conflicts go away in the Domain Workbench as

```
private SqlDataReader GetAddresses(int ADDRESSID)
{
  int SELECT;
  SqlConnection Connection = new SqlConnection("server=localhost;uid=sa;database=db_cust");
  SqlCommand Command = new SqlCommand(
    SELECT
        *

      FROM
        ADDRESS AS addr

      WHERE
        addr.ADDRESS_ID = ADDRESSID
  , Connection);
  Connection.Open( );
  return Command.ExecuteReader(CommandBehavior.CloseConnection);
}
```

Figure 10. Mixing source from a C# domain and a SQL domain.

```
procedure double Sum3(double j, double k, double m)
{
   double result;
   result = j + (m + k)
```

	A	B	C	D	E
1	j	m	k	expected	result
2	1	15	2	18	18
3	3	15	4	22	22
4	4	5	6	14	15
5	.	.	.	.	.

```
   ;
   return result;
}
```

Figure 11. Integrating dynamically evaluated tests into source.

each user can simply project the common information in whatever ways that user prefers.

4.1.6 Testing

Programmers' ability to improve testing suffers greatly from lack of support to express and process test intentions beyond Asserts. The main needs are:

a. The ability to associate test code (including asserts, pre and post conditions, white-box test and test data) with the code being tested without any permanent visual or other impact.

b. The ability to refer to parts of the algorithm being tested and affect their behaviors for testing without disturbing the source code.

c. Access to meta-data and other descriptions of the problem.

Figure 11 is an example of how test cases can be inserted directly in the source code. In this example, test case 4 signals an error, but the error is actually in the test data not in the code.

4.1.7 Eliminating the need for many tradeoffs

Languages tend to differentiate themselves by the position taken on two inherently opposing requirements, expressiveness vs. brevity. Some languages take one extreme, some the other. For example, the language APL [10] is famous for its extreme brevity. Others made pragmatic choices somewhere in the middle to minimize overall misery, while COBOL opted for garrulousness. The Domain Workbench can move from one choice to another whenever needed so the tradeoff disappears. We can focus on many new choices in the solution space, rather than continue to argue about yet another tradeoff.

4.1.8 Program transformation and refactoring

The intentional tree format is also convenient for program transformations. Both display generation and code generation employs tree transformations. These capabilities are easily harnessed for other code refactoring and code analysis tasks, with the additional benefit that results of each transformation can be inspected in many formats.

Note that program transformation is a domain of its own that invites support by DSLs.

4.1.9 Improving IDE interaction

Many functions of Integrated Development Environments are limited by the lack of intentional information in the source code. A trivial example is the lack of connection between a comment and the statements. When the source contains the connection at the outset, it is much easier to move the comment with the code, and to determine when the display of a comment should be suppressed in outlines.

When the editor can distinguish code that is used only for error cases, or only for testing, it becomes easier to include more testing code or more assertions. Their presence will not overwhelm viewing of the main program logic.

4.1.10 Reuse

Reusability is an important quality of code that is deeply connected to parameterization. The issues of reuse in general are complicated and outside the purview of this discussion. But the likelihood of reuse, can be greatly increased by parameterization. Fixed code will either match a situation or not. Parameterized code will match a larger number of situations.

Programmers are implored to make software more reusable. This is effectively a call for parameterization with respect to some expected range of differences. The parameterization guarantee of intentional trees always makes this possible. If notation becomes an issue, that can be resolved by extending the projections. Performance issues can be handled by specialization in code generation.

4.2 Domain-oriented development benefits

The domain schema, the domain code, and the generator together represent a factoring of the implementation. This factoring makes Intentional Software a plausible approach for dealing with the complexity of domains. The schema and generator can be improved incrementally as new knowledge is gained in the domain. The domain experts can update the domain code and re-run the generator for common changes in the problem that are still within the schema parameters. While not all desired changes can be made this way, the worst case outcome is just today's best practice. Namely the domain experts will have to talk to the programmers. This contrasts with direct programming (as discussed in Section 1.1) where every change requires human interaction that introduces delay, costs, and potential errors.

Programmers can also make substantial implementation changes in many instances by only changing the generator. The absence of domain detail helps reduce the scope of these programming tasks. The programmers' task of creating a generator, however, may be thought of as harder than just creating one implementation instance. On the other hand, it can also be argued that creating the generator is a purer, more explicit, and more transparent expression of what the programmers should be doing: verifying assumptions, incorporating testing into the code, making dependencies on domain details clear, and so on.

The concept of partial implementation is also worth noting. An intention recorded in the schema or in the domain code is valuable when the implementation is not yet started, is done partially, or is done "fully". Of course, there rarely is any "full" implementation. Just as there is no perfect implementation, a minimal implementation can be valuable during the development process.

Variants of generative programming have been tried in the past with generally positive results. One important bottleneck has been the need to specify the input to the generator in the form of a DSL typically including a syntax and parser. Even with automatic parser generators, just the *design* costs of a syntax can be

daunting. In addition, most DSLs retain a programming language flavor that hinders more direct involvement of non-programmers such as domain experts. For domain experts to be efficient, the domain descriptions need to mirror the domain intentions [6].

Furthermore, most problems involve multiple domains and legacy components written in other languages. However, DSLs have historically been difficult to mix with general purpose programming languages and other DSLs. Sections 3.3 and 4.1.1 discusses how this can be resolved in the Domain Workbench.

5. Other Generative Approaches

Several other approaches use generative techniques to tackle complexity in software development.

Code generation is commonly used for generative programming. For example, template libraries like STL [16] use code generation. The CASE products of the 80's were able to generate standard COBOL or C applications from specialized diagrams. These were fixed generators. If the users needed to do anything that was not in the generator's repertoire, the user had to maintain the generated COBOL or C code. For specific, well contained problems, code generation was and still is effective. A list of current code generators can be found at [2].

Wizards are another example of code generators popular in IDEs. Wizard dialogs often capture intentional information about the program. But once the programmer clicks Finish, the code gets generated and the succinct intentions are scattered across the generated code and lost.

Code generators work fine as long as the output of the generator does not have to edited. When the output of the generator is edited, problems can arise the next time the generator runs–the edits may be overwritten. Editing the output challenges the code generator to invert the encoding function. Most generators cannot perform the inversion, and even if they were helped they could not represent the inverse without an enhancement of the input.

Round trip engineering, two-way wizards, and synchronization were all developed to solve the problem of allowing modification of generated code and still allow reconciliation with the domain specific model. These remain standard features of many model driven tools.

In a round trip the same data persists in two places, in the model and in the code. That makes it difficult to maintain consistency. Typical approaches designate one copy as master and either pull on demand or push on change from the master to the redundant copies.

Another way around the problems is to separate the code that is generated into two parts: one that is editable by the programmer and one that is not. But this forces the programmers into whatever artificial separation is prescribed by the tool.

Domain Specific Languages shift customization into the language itself by trying to design the optimal language for each specific domain [4, 5, 9, 15, and 18]. One problem with DSLs is that, since by nature they are specific to one domain, they often become silos of their own. Furthermore, DSLs suffer from the same problems as General Purpose Languages: good languages are difficult to design, their syntax is limited both in the notation and expressiveness, they are difficult to evolve, and it has been difficult to combine multiple languages.

6. Summary

This paper presented the general ideas of Intentional Software. The Domain Workbench allows creation and editing of domain code. The intentional tree contains the domain code for Domain Specific Languages, General Purpose Languages, and meta-data. The use of Generative Programming turns the domain code into executable target code.

Past problems with structured editing have also been discussed. While past experience with structured editors has been generally negative, the need in this new context should cause us to re-examine the problems and implement pragmatic solutions. Most application software other than programming has already moved ahead toward recognizing more structure in documents.

The domain of grammar production rules was used as an example. Some snippets from the C# syntax were presented in a number of projections to show the interchangeability of popular notations and the corresponding tree-generating templates.

These examples were selected to show not only that syntax and semantics can be separated, but also that the users' intention—domain code–is a proper focus of investigation. The factoring described can yield benefits beyond direct benefits of notational flexibility and semantic extension.

7. Acknowledgements

The authors are grateful to all reviewers and colleagues at Intentional Software Corporation for their valuable contributions to the content of this paper.

8. References

[1] Brooks, Fredrick, No Silver Bullet – Essence and Accidents of Software Engineering, Computer Magazine, 1987.

[2] Code Generation Network www.codegeneration.net

[3] Czarnecki, K. and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA, USA, June 2000

[4] van Deursen, Arie, Paul Klint and Joost Visse**r** Domain-Specific Languages: An Annotated Bibliography, 2000 http://homepages.cwi.nl/~arie/papers/dslbib/ An updated list published here http://catamaran.labs.cs.uu.nl/twiki/pt/bin/view/Transform/DSLBibliographyAdditions

[5] Dmitriev, Sergey. *Language Oriented Programming: The Next Programming Paradigm*, 2004. http://www.onboard.jetbrains.com/articles/04/10/lop/

[6] Evans, Eric. *Domain-Driven Design*. Addison Wesley, 2004

[7] Foote, B. and Yoder, J., Big Ball of Mud, Fourth Conference on Pattern Language Programs, 1997.

[8] Fowler, Martin, Language Workbenches: The Killer-App for Domain Specific Languages? , 2005. www.martinfowler.com

[9] Greenfield, Jack, Keith Short, Steve Cook, Stuart Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*

[10] Iverson, K. *A Programming Language*, 1962. See also Programming Notation in Systems Design. IBM Systems Journal 2(2): 117-128 (1963)

[11] James, William, *The Principles of Psychology*, Henry Holt 1890

[12] Khwaja, Amir, Joseph Urban, *Syntax-directed editing environments: issues and features*, Proceedings of the 1993 ACM/SIGAPP symposium on Applied Computing: states of the art and practice.

[13] Kiczales G, et al, Aspect-Oriented Programming, *Proceedings of European Conference on Object-Oriented Programming*, 1997.

[14] Lampson, B.W. Personal Distributed Computing: The ALTO and Ethernet Software, ACM conference on the History of Personal Workstations, Palo Alto, 1986.

[15] Mernik, M, Heering, J, Sloane, A, When and How to Develop Domain Specific Languages, ACM Computing Surveys, vol 37, no 4, December 2005

[16] Musser, David R., Gillmer J. Derge, and Atul Saini *STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library,* Addison-Wesley, 2001

[17] Naur, Peter, A Synapse-State Theory of Mental Life*,* 2004, www.naur.com/synapse-state.pdf

[18] Simonyi, Charles. *Intentional Programming – An Ecology for Abstraction*. 1997, Invited Talk USENIX Conference on Domain-Specific Languages

[19] Simonyi, Charles. Intentional Program Tree Represented By High-Level Computational Constructs, US Patent Nos. 5790863, 5911072, 6070007, 6078746, 6097888 and 6189143

[20] Wilson, Gregory V., Extensible Programming for the 21st Century, ACM Queue, Vol 2, No 9, Dec/Jan 2004/2005.

[21] YACC. http://dinosaur.compilertools.net