Design principles for domain-specific languages

Technical Report

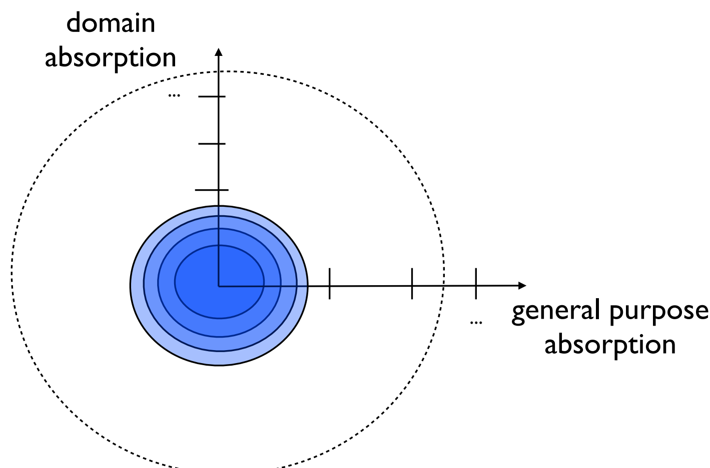
Thomas Cleenewerck
tcleenew@vub.ac.be

Motivation

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [4]. Ever since its conception, domain-specific languages have been attributed with a considerable amount of software engineering qualities. DSLs can reduce the complexity; improve development qualities (correctness, productivity, reliability, conciseness, readability); and facilitate requirements checking and rapid prototyping. Despite their popularity in research and engineering fields like model-driven engineering and language design, DSLs are bit waning.

Over the years the term “domain-specific language” has been somewhat a victim of its own popularity. The term is used in a broad range of engineering and research fields. Everybody nowadays seems to be doing something that is labelled as domain-specific language design and engineering. DSLs are becoming so overloaded and ambiguous to the point that they become hollow, nearly useless and having no distinguishable properties of their own. In this document, you will not find yet another attempt to ‘claim’ the term or yet another purpose for using it. Instead we will be looking at how to make DSLs distinguishable and ultimately more meaningful.

DSLs bear in their name two distinguishing characteristics: domain-specificness and languages. It is only in the combination of both characteristics that we are able to distinguish it from what DSLs are not. Approaches that deviate from either characteristic are considered to suffer from DSL degradation. Based on these two characteristics we identify two threats for DSL degradation (see figure below): domain absorption and general-purpose absorption.



- DSLs should be kept small and focused towards a well-scoped problem domain. When growing DSLs beyond the borders of its initial domain towards new domains, mediocre support creeps into the DSLs, rapidly resulting in inconsistent syntax and semantics.
- DSLs should be kept simple. Growing DSLs towards general-purpose language concepts impregnates the DSL with concepts alien to its users aka the domain experts. Moreover from a technical point of view, this generalization blurs the borders with more conventional domain-specific abstractions that can be offered by programming languages such as libraries, frameworks, macro's.

Both degradation threats ultimately endanger the life-time expectancy of a DSL. In case of domain degradation, a DSL does a bit of everything but nothing especially good. The limitations that would otherwise be considered an improvement because of the gained simplicity and effectiveness, turn into annoying shortcomings because of the increasing restrictiveness and complicatedness. In case of general-purpose degradation, the introduction and maintenance of a DSL no longer improves the development process but is merely perceived as overhead.

In order to address the DSL degradation we propose 7 design principles. The principles present a practical checklist to ensure qualitative distinguishable DSL designs that can compete against existing engineering solutions.

7 Design Principles

Representation

"A language with radically different syntax, […], has no hope of finding broad adoption, regardless of the brilliance of its design. This tends to constrain the rate at which languages evolve." [8]

Definition

Representation addresses the syntactical way of expressing the concepts of DSLs and should not be confused with analysing the concepts of a domain into a domain model.

Distinguishable property

Unlike classical software engineering approaches DSLs have the freedom to use a syntactical formalism that best fits with the problem domain at hand, whereas classical software engineering approaches are all embedded in the formalism of GPLs. This entails both the ability to offer new syntactic forms but also the ability to discard existing syntactic forms. Although it is quite obvious for libraries and frameworks that they cannot define new syntactic forms, it is also true for the case of preprocessing facilities such as marcro's. Totally impossible is the ability to discard existing syntactic forms.

Upon designing a representation one should consider:

- Concrete syntax: includes amongst others tokens, keywords, delimitors that enable us to write unambiguous sentences
- Code layout: arrangement of the different grammatical sentences

Effectuate

The syntactical representation of a DSL should aim to adopt the most optimal syntax for the DSL users. Optimal syntax can be achieved by opting for:

- Domain Syntax: Syntactic representations common to the domain of the language.

- Distinguishable [19]: The different parts of a DSL program should be easily distinguishable from one another.

- Familiar: It is better not invent new syntax when there are well established and satisfying existing alternatives. Syntaxes of main-stream programming languages are after all based on experiences in decades of language development.

Contribution to design qualities:

Readability: Distinguishable syntax exploits our synesthetic memory and facilitates reading DSL programs because with a blink of an eye one can recognize those parts of the DSL program that are relevant at the time of reading.

Conciseness: A choice must be made on whether to emphasize on making programs more readable, or making them easy to write. Generally, readable programs are longer and require more typing. Most general-purpose languages tend to favour writability to achieve conciseness. However, typing the program is only a part of the programming process. As DSL programs can address a brother audience than trained developers, after the program is written, more time will be spent looking at the program than actually writing it. Moreover DSL programs tend to be inherently more concise as it uses semantically rich domain-concepts.

Consistency: Consistency is either "uniform" or "harmonious," depending on whether a set of constructs is indivisible or divisible into different smaller constructs [15]. From the perspective of text characteristics, one should take care of syntactic, stylistic and spatial aspects.

Examples

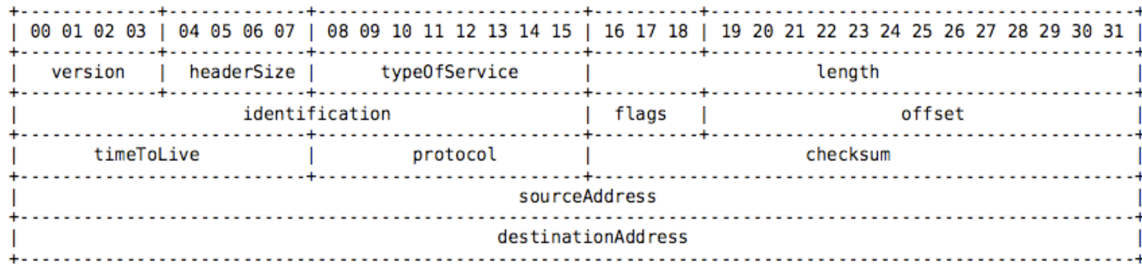
- Example of domain syntax: Boardgame design and setup in BOB

```
Board 8 x 8 {
    W B W B W B W B
    B W B W B W B W
    W B W B W B W B
    B W B W B W B W
    W B W B W B W B
    B W B W B W B W
    W B W B W B W B
    B W B W B W B W

    . B . B . B . B
    B . B . B . B .
    . B . B . B . B
    . . . . . . . .
    . . . . . . . .
    W . W . W . W .
    . W . W . W . W
    W . W . W . W .
}
```

Bulk of the boardgames are rectangular. They are composed by aligning squares on which pawns can be positioned. In game manuals, gamers are instructed to construct and set up the board using schematic diagrams. These diagrams have been adopted as such in the language BOB.

- Example of distinguishable syntax: ETF Requests For Comments (RFCs) in the form of "ASCII-art diagrams". [2]



Network protocols are a perfect example of the clumsiness of traditional programming languages obfuscating the simplicity of the protocols and the internal structure of the packets they exchange. The above suggested syntax shows how transparent a simple TCP/IP implementation can be; an unusually expressive alternative to complex, repetitive, opaque and/or error-prone code.

- Example of familiar syntax: nesting

Hierarchical decomposition dates back from the 70s promoted by the rather outdated Structured Design (SD) methodology [3]. Nevertheless it remains a very powerful idea that works fine together with the contemporary paradigms such as object orientation. The fundamental idea is that a large system must be divided into manageable sub-systems or parts in several layers deep. This principle is reflected in language design by means of nesting. In DSLs, nesting plays an important role to better structure the solution for a problem (see later). The syntactic forms in DSLs used to indicate nesting adopted the syntaxes of general purpose programming languages. Among the familiar forms are begin…end, { … }, (…).

When a nested expression has too many nesting levels, humans tend to become confused about the meaning of the code. Experimentation has shown in case of novice programmers one should not rely heavily on parentheses for grouping [12].

Other references

Parnas Tables [13]
Tabular query forms [12]

Absorption : the implicit use of assumed structure & behaviour

Definition

Being specific to a domain, DSLs should absorb the common practices present within that domain. Being a language, DSLs should absorb those commonalities implicitly. As such the users of the DSL can safely and reliably assume that the DSL will behave, as users would expect it to do. Users are thus relieved of explicitly keeping track of ‘the obvious’ and allowed to concentrate on the essence of the problem. As people are notoriously known for their lack of discipline in such matters, DSLs are a valuable contribution.

Distinguishable property

Absorption is an important guard against the threat of genericity. General-purpose solutions such as libraries can absorb common domain practices but the users have to explicitly and consistently use the interface of a library or framework. These reusable code artefacts can even be so complex that they are equipped with guidelines, recipes and example code to ensure their proper usage.

Effectuate

With absorption a DSL is able to *deduce* additional information concerning the required behaviour and structure of a program so that it can be omitted by the user. Absorption can be either achieved by:

- Exploiting context: based on the structure of the surrounding code of a particular construct

- Automation: the DSL program contains sufficiently rich information such that a smart processor can automate certain tasks for the user

Contribution to design qualities

Read- and writability: Absorption improves the writeability and readability. Stating the obvious over and over rapidly becomes very tedious. Not having to state it improves the writeability. Reading the obvious over and over confiscates the essential meaning of the program and distracts the reader. Not having to read the obvious improves the readability.

Reliability: Absorption improves reliability as it can shield the users from unexpectant interacting features drawn from the combination of libraries and host language features. Absorption also renders programs less error prone, as people are notoriously known for their lack of discipline in consistently and repetitively exercising the same task.

Examples:

- Example of automation: Implicit access to context values

Context values are values which actual value depends on the context in which it is being used. A context value library provides a couple of new routines like `cv-set` and `cv-get` to modify or read the actual value of a context value. A DSL however can absorb context values as a new primitive value along side other common primitive values such as integers and strings. A program in this language can then use the usual `set` and `get` routines for modifying or reading context values. The language implementation will dispatch to the right routine based on the given value. Having the `cv-set` and `cv-get` implicitly present in the language improves writability, reduces errors and improves reliability. As the first may be obvious, the later two are rather tricky. Firstly, primitive operators like / cannot deal with contextual values. The contextual values must first be reduced to their actual value before they can be used primitive computations. If not, an error occurs. Secondly, contextual values are passed by reference in functions and not, as you would expect, by value. Hence, a developer has to copy this value before calling a function. If not, reliability degrades significantly as may cause nasty side effects bugs that are hard to debug.

- Example of implicit context: implicit references to temporary variables

The creation of hierarchical structures is a common undertaking e.g. HTML for a website, Java swing components for desktop user-interfaces. General-purpose languages are rather ill equipped for constructing these structures as it involves the treading of many temporary variables. In the Java code excerpt below where a simple user interface is created, this is clearly visible when looking at the enumerated variables.

```
JFrame jFrame_0;
jFrame_0 = new JFrame();
jFrame_0.setTitle("Welcome!");
      JPanel jPanel_0;
      jPanel_0 = new JPanel();
      BorderLayout borderLayout_0 = new BorderLayout();
      jPanel_0.setLayout(borderLayout_0);
            JLabel jLabel_0;
            jLabel_0 = new JLabel();
            jLabel_0.setText("Hello World");
      jPanel_0.add(jLabel_0, BorderLayout.CENTER);
```

```
jFrame_0.setContentPane(jPanel_0);
```

DSLs can absorb this treading because a construct an exploit the surrounding code in which it is used. The above Java code can be written in SWUL as follows:

```
JFrame frame = frame {
      title = "Welcome!"
      content = panel of border layout {
        center = label { text = "Hello World" }
      }
};
```

There is no need for threading variables, for example when a label is created in the context of a panel, it gets added to that panel.

Standardization: offer a structured way of solving a problem

"*Limitation of the universe of discourse is so important for the design of special-purpose languages that it should be the first and most carefully considered step"* [7]

"*One might suspect that the language would not improve by having to conform to a restrictive defining tool. But experience shows that it does. In some sense there is no art unless there is a restriction of the medium. In some perverse way, the human mind, in coping with restriction, produces its best results"* [14]

Definition

Standardization restricts a DSL to better assist users in how to write a program that will solve their problem.

Distinguisable property

In GPLs, the main concerns compositionality, semantic simplicity and regularity give in the extreme rise to involution[14]. In languages such as scheme, basically everything in the language is an expression. However in many GPLs, involution is confined to certain parts of their grammatical and semantical specification. Object-oriented languages for example offer language constructs for the definition of classes, instance variables, visibility modifiers, etc. DSLs confine involution even further. A significant part of their grammatical and semantical specification is there to assist users in how to write a program that will solve their problem. Instead of letting the programmer figure out on how to solve a problem, DSLs restrict the possible alternatives to the minimal set a domain-expert is willing to consider at given stage in writing his program. For this reason, classical software engineering approaches like frameworks and libraries compensate by offering literature describing guidelines, cookbooks, idioms on how to be used.

Effectuate

Offering a standardized way of solving a problem can be done by:

- Restricting the grammar: the grammar contains an explicit recipe guiding programmers step by step towards a solution for their problem using among others hyponyms, conventions, cookbooks, idioms, patterns, best practices, routines, scenarios.

- Enforcing restrictions via semantics: the language implementation performs semantic checks on the program entered by the user. These checks are typically performed at compile time along with a type check.

Contribution to design qualities

Reliability: Restriction is only acceptable for most users when there is gain, so the restrictions must improve qualities regarding the overall user experience. Users expect to be supported in their endeavours in every step of the way. Standardization does so by guiding them to express solutions for their problems. In [6], the designers of the SpecTRM-RL language for process control systems of airplanes chose to limit the amount of technical details as much as possible. After evaluating their DSLs with the targeted users, they had to redesign their language to explicitly adopt *operation modes* of machine components. Users also do not want to get stuck with a restricted language if something goes wrong. They favour a reliable solution above total freedom, if a DSL can check or spot semantic errors.

Readability and writability are also improved by standardization. Restricted grammars are easily memorized, allowing users to quickly focus their attention on the part of the program of interest and facilitating the writing of new programs.

Locality: A well-standardized way of solving a problem improves locality. The further a statement is influenced by another statement, the harder it is to remember the relevant details.

Lexical Coherence/Disentanglement: code that logically belongs together should be physically adjacent in the program. code that is not related should not be interleaved. It should be easy to tell where one logical part of the program ends and another one starts. Lexical coherence is strongly related to disentanglement. Disentanglement is a response against the phenomenon of crosscutting concerns in order to attain the design quality of modularity. A concern corresponds to code that logically belongs together. When concerns are mixed, the interactions are encoded implicitly through the dependencies and interactions between fragments of code that implement the different concerns. Hence, these concerns become more difficult to understand.

Examples:

- Example of grammar restriction: SQL

After the advent of SQL, querying databases has dramatically been simplified. There is no need for programming anymore. People without those skills can easily be thought how to read and write simple queries. The query language enforces a rigid structure and thus standardized the way to formulate a query. Users no longer have to devise a way for writing queries. The grammar of the language carefully dictates the sequence of decisions that have to be made to formulate a query: once starts with a selection of all the columns you want to retrieve, proceed with an indication of the tables to consult, followed by a condition on the rows to be returned.

- Example of semantical restrictions: Devil [5]

The most common semantic check found in programming languages is the declaration/usage check. Programmers must declare an entity before they can start using it. The other way around, checking that every declaration is used at least once, is less common and only presented as a warning. For DSLs, this check makes more sense. Devil is a language for writing device drivers. In Devil, unnecessary variable declarations consume up valuable memory space in CPU's and GPU's. Logically, Devil enforces that each variable declaration is used at least once.

In general purpose programming languages the amount of semantic checking is bounded by the information stated in the program. Therefore program code is augmented with additional information. Semantic checks often rely on the use of types at declaration sites to prohibit the programmer from using a wrong type at usage sites. This is the point where DSLs distinguish themselves from plain languages, DSLs can exploit information beyond what is written in the program. This is due to the fact that DSLs operate within a particular domain. The DSL Devil nicely illustrates this. For reasons of efficiency, the device programming language prohibits the construction of *overlapping entities* i.e. entities (e.g. variables) that share parts (e.g. register bits).

References

- Performing analyses to detect common errors in the specification by providing explicit information that is difficult or impossible to extract from general-purpose languages. An example of this is checking that the bits of each register belong at most to one field [17]

Abstraction: consider independently or separately from something

"*The point of programming languages is to prevent our poor frail human brains from being overwhelmed by a mass of detail*" [18]

Definition

Abstraction is by far the most overloaded and thus possibly the most confusing principle. The term 'abstraction' has been attributed to any action during the design of software in which at some point knowledge is formalized and structured. As a consequence, the term as such can no longer act as a first guiding principle. We therefore reintroduce the term anew, by emphasizing its distinguishable property. When *abstracting*, one reduces the amount of information by explicitly and intentionally layering new concepts on top of the existing ones. The choice what to remove depends on the *abstraction level*. If the abstraction level is platforms for example, then one should not consider native or native inspired concepts.
Abstracting should not be confused with ignoring. Ignoring information means you are excluding or at best scoping the setting in which you are working. While abstraction creates concepts on which some information no longer needs to be considered.
Views are in a similar way also different from abstraction. In views, information is not shown but has to be taken into account in order to attain a complete understanding of the software. Abstractions are sufficiently semantically rich concepts that do not require the information that has been left out in order to understand the solution being described in the language.

Distinguisable property

One of the strengths of GPLs is the ability to define new abstractions. We distinguish between two levels: base and meta-level abstractions. All base abstractions behave in a similar fashion, they share the same language semantics, as they are instantiated using the same set of language constructs. Meta-level abstractions can alter the language semantics and can thus define new semantics. However, most meta-levels are reflective, and are thus constructed using the base set of language constructs. Hence new abstractions can never fully outgrow the GPLs. Hence abstractions remain polluted with jargon of the GPL. DSLs have a meta-level that defines the semantics by interpretation of by translation, and are thus unpolluted.

Effectuate

DSLs make abstractions from:

- Technical complexity required by skills that transcend the boundaries of the domain one is working in.
- Irrelevant details: Information that is not considered an essential part of the problem domain.
- Redundant or possibly confusing information that would clutter up programs such as concrete instances, examples, synonyms, etc.

Contribution to design qualities

Non-ambiguity: Abstraction allows us to focus on what is essential in a solution. There is no superfluous information that could otherwise cause misunderstandings, confusion, cluttering or the need to sift through and hook together fragmented information. Reducing redundancy implies non-ambiguity: a concept has only a single meaning in the real world.

Uniqueness/parsimony: Reducing redundancy also implies the condition that there must not be two concepts with the same meaning.

Examples

- Example of technical complexity: MPS (File format reader)

The file format reader DSL of Fowler is a small DSL used for processing text files. The objective of the language is to parse the file's content into objects. The DSL doesn't produce a parser such as typical parser generators (e.g. Yacc or Javacc) would do. Instead one has first abstracted away form these low level details and created a library for parsing the files and creating the objects. The DSL operates on top of this library. For parsing a particular kind of file, glue code has to be written that uses and composes various library functionalities such as reader strategies and field extractors.

```
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;
}
```

The abstraction has been pushed to its limits, because not a single expression in the glue code could be absorbed into the library. In other words, every single expression is specifically written for parsing a particular kind of file. At this point, from the solution perspective; glue code is considered as a pure configuration to parse a particular kind of file. However, from a problem perspective, the glue code is not a pure configuration. As there are numerous expressions, which are not related to the problem domain of configurations but are technical complexities due to the limitations of the language in which this solution is expressed. We observe the following purely programmatic inspired program text:

- distinction between variables and strings (when using reflection, strings can be variables): *CustomerName* must be a string, while it actually refers to an instance variable *CustomerName* of the class Usage.
- distinction between objects & values (abstractions are new structures encapsulating the primitive values): when using the parsed value one must access it through an object.

- creation of new objects, method calls, references to objects, etc.
- explicit control flow like looping
- variable passing: threading of the intermediate variable *result*
- meta programming: types such as *Usage* are used as parameters
- boilerplate code: setup of the library with c*onfigure* and structure of the solution using the auxiliary functions *ConfigureServiceCall* and *ConfigureUsage*.

Stripping the above program of programmatic text results in the following concise specification that can be processed by a DSL:

```
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
```

- Example of irrelevant details: Spec-TRM-ML[6]

Spec-TRM-ML is a DSL for process control systems of airplanes. Its specifications are solely expressed in terms of the system's components and the state variables of the controlled system. Specifically, "private" variables and procedures relating to the implementation are not a part of the expert's view of the controlled system, and thus are considered irrelevant. The specification therefore adopted a blackbox model of the system's components.

- Example of redundancy: BOB[1]

Too often, technically oriented "modelers" jump straight into excruciating detail, dense jargon, and complex graphics, incomprehensible to domain-experts and other participants. The process of eliminating redundancy requires one to balance disambiguation against the need for distinguisability. When designing a language for boardgames a lot of terminology with subtle differences had to be processed. Consider for example the concepts: capture, move, jump move. Disambiguation exposes the subtle differences among the three: removing an enemy pawn, being hindered by an enemy pawn, and ignoring an enemy pawn. However upon evaluating the need to be able to distinguish it, these dealings with enemy pawns where considered concrete instances of a move's behaviour and all of those concepts were abstracted into the single concept of a move.

Compression: reduce the code footprint while retaining the same amount of semantic details

Definition

Compression retains the amount semantic details; it merely requires a less elaborate programming style. Compression, brevity[18] or compactness [7] is often measured by counting the number of lines of code. Although crude, this makes compression by far the easiest to

measure but is at the same time also one of the most controversial principles. There are couple of rules of thumb that we can derive from design qualities to guide compression:

- Locality: Users need to be able to display a single indivisible unit of functionality on their programming editors at once without the need to scroll, zoom or navigate.
- Proportionality: Small problems should require small DSL programs, and any increase in the size of the problem statement should require a proportional increase of the DSL program.
- Frequency: The more frequent a language feature will be used, the more convenient its syntax should be. An infrequently used feature can be given a long name and less convenient syntax. In [7], a study of the English language revealed that the most frequently used words are the shortest ones.

Distinguisable property

The ability offered by GPLs to compress code is predefined and fixed. It all boils down the expertise and creativity of a developer to formulate a solution for a given problem. Developers therefore rely on idioms and patterns to elegantly shape the architecture and design of their solutions. As DSLs are not bound to the formalism of existing GPL, compression can be fine tuned down to the last character.

Effectuate

Compression can be attained by:

- Factorization: avoiding reoccurring patterns in the code
- Increasing conciseness by:
  - removing elaborate statements,
  - changing the syntactic sugar,
  - omitting default statements,
- Intelligent program text analysis such as overloading

Contribution to design qualities

Understandability: A short program will have a tendency to be more easily comprehended than a long one. Even the addition of comments with the intention of rendering the program more readable had the adverse effect [7].

“The fewer the lines of code the better” is a crude measure as it may negatively impact other engineering qualities like readability, maintainability. Perl is one of the languages that pushes this idea to the extreme. A whole chess program can be written in a single line of code, however few are able read it let alone thoroughly understand the program. The dual nature of compression is due to the fact that the semantic density increases. And as such one must keep this density within the capabilities of DSL users.

Readability and writability: Usually readability and writability are mentioned in one breath together. What improves readability generally improves writability. However, compression is a principle that clearly separates the two. From a writability perspective, a compressed construct has many advantages. It is easier and faster to write down. The reader of a compressed program is burdened with remembering the more self explanatory program [7].

Locality: A compressed specification localizes the necessary information to better comprehend a code fragment.

Examples

- Example of reoccurring patterns: WebDSL [6]

The WebDSL facilitates the creation of J2EE web applications by abstracting from many technical details and concerns of J2EE. When the authors arrived at the point where business objects had to be presented, a new challenge was inadvertently raised. As websites contain a lot of similar page layouts, a template system was added to the language to be able to declare new templates and reuse them afterwards. WebDSL code was compressed 10 fold by this mechanism.

- Example of increasing syntactic sugar: WebDSL [6]

The WebDSL facilitates the creation of J2EE web applications by abstracting from many technical details and concerns of J2EE. However, the authors found that those abstractions weren't sufficiently concise. Quite some boilerplate code had to be written and rewritten each time a link had to be created to a page displaying some business object. The authors added more syntactic sugar to their language to address this issue. A new construct was created that simply replaced the elaborate statement.

- Example of overloading:

Overloading is a commonly used practice in many general-purpose languages for operators and methods. In C++, the same operator name can be used for a wide range of operand types. It increases compression as it relieves the programmer of having to explicitly state the specific operator for adding integers, floats or other types. In essence, overloading a concept in a language means that it has different meanings depending on the context or way the concept is being used. Overloading is one of the major host language features used by embedded DSLs to use native host language conveniences for DSL constructs. Computing the discriminant using the BigDecimals is done through the rather verbose statement:

```
discriminant = (b.multiply(b)).subtract((BigDecimal(4)).mult(a).mult(c));
```

Whereares if operator can be overloaden the following can be achieved:

```
discriminant = b*b - 4*a*c;
```

Generalization: Inferring common case from specific cases

Definition

In a first attempt to bridge the gap between the solution perspective and the domain perspective, domain knowledge is particularized, exemplified and fragmented by the expert so that the domain language designer can ultimately understand the domain. It is up to the latter to prune, combine and reorganize the knowledge to a smaller set of general concepts while retaining the same semantic details. As such the effort of building languages is reduced and the necessary coverage of the problem domain can be more easily ensured.

Distinguisable property

Generalization is common activity during analysis and design of a software system. Yet, the goals are not entirely the same. During analysis, generalization is used as a means to structure and relate information concerning the domain or problem at hand. During design, generalizations are intended to minimize code duplication, and exploit polymorphic behaviour (maintainability,

separation of concerns, code compression). The goal of generalization for DSLs is to reduce the amount of concepts by replacing a group of more specific cases with a common case.

Effectuate

Generalization can be attained by:

- Inducing: form a new concept to surplant a series of concrete concepts by a theory that covers subtle variations
- Collapsing: search for concepts which are more specific than another one and collapse them into a new enriched concept e.g. hypernyms/hyponyms

Contribution to design qualities

Longivety: Generalization is an important principle that contributes to the overall completeness of the DSL. In a generalized concept, the domain coverage is explicitly formalized and turned into a computable form as the combination of variation points covered by a generalized concept is made explicit. In turn, the longevity of the language is easier to control. Generalization can unlock possibilities that were not explicitly mentioned but may become interesting in future evolutions.

Simplicity: Generalization also has a distinct impact on simplicity. It is sometimes easier to solve a general problem than to solve a specific one [7].

Readability and writability: Usually readability and writability are mentioned in one breath together. What improves readability generally improves writability. Generalization is a principle that clearly separates the two. From a writability perspective, a generalized construct has many advantages. The writer only needs to recall a single concept and can apply it in many different ways. However, the reader of a generalized program has to work his way through the code each time interpreting the general case to the specific one.

Examples

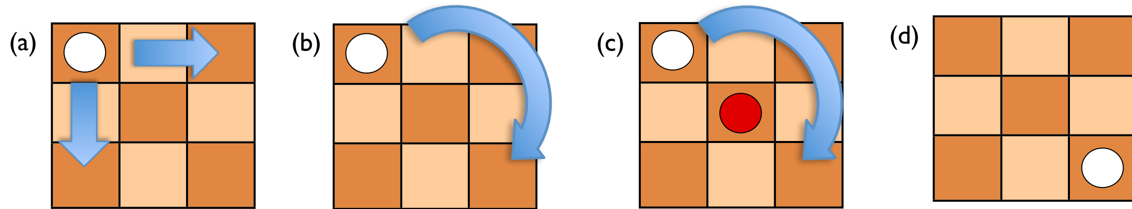
- Example of induction: BOB [1]

During the design of BOB a large number of moves have been catalogued from a broad range of boardgames. As an example consider all the moves of checkers: one step diagonal forward left and right, on step diagonal backward left and right, jumping over a pawn, multiple steps diagonal forward and backwards. Overwhelmed by the sheer amount of enumerating all the moves found in board game documentations two coordinate systems were introduced to cover them: a single dimensional system enumerating the squares form top to bottom, left to right, and the another a two dimensional cartesian system consisting of rows and columns.
BOB also nicely illustrates that generalization has its price. Consider the following code fragment describing a checker move for white pawns:

```
Moves W {
      move (x,y) to (x+2,y+2)
      if { isEnemy(x+1, y+1)
           isFree(x+2, y+2) }
      then { delete(x+1, y+1)
             moveTo(x+2, y+2) }
 }
}
```

First of all, (a) the reader must interpret the coordinates differently depending whether white pawns start off at the top of board or at the bottom of the board. Secondly, (b) the move needs to be pictured on a mental representation of board to understand that this move advances two steps diagonally to the right. Thirdly, (c) the conditions in the if-branch describing the setup of board

before the moves takes place have to be added to the mental image. And lastly, (d) the branch describes the setup of the board after the move is performed.



- Example of induction: Hypernyms

Hypernyms and hyponyms are often used as valuable sources of information in domain modelling for creating generalization and specialization hierarchies. The difference hower with the creation of language is the need to collapse those into a single concept. In BOB the attack move as a specialization of a normal move has been collapsed with in a single moves concept, where the behaviour attached to a move reveals when a move is normal or an attack move.

Optimization: Programming performance improvements

"*Preparing an engineering or scientific problem so that it could be placed on a computer was an arduous and arcane task that could take weeks and required special skills.*" [9]

Definition

DSLs distinguish themselves from general-purpose languages by outperforming their developers, similar to the transition from assembly to general-purpose languages. Back then performance was very important. It took a great deal of knowledge and time to complete an engineering task. But advances in the programming field were only accepted when programs run as efficient as a hand-code program would do. It is striking though how little the situation has changed in essence, the key difference being that our programs have gotten bigger and more diverse. Programs nowadays still take a great deal of knowledge to construct, consume a lot of time. Advances in programming languages are greeted with considerable amount scepticism. I therefore dare to conclude that DSLs likewise will have to demonstrate a leap in programming performance involving execution efficiency, developer time and effort, etc.

Distinguisable property

Programming performance of DSLs in terms of productivity has been compared against general-purpose solutions. In [6],[12] speedups have been produced ranging from a factor of 10 to 100 compared to GPL.

Fine-tuning performance of a software application is a complex undertaking. One of the rules of thumb is to optimize those parts your program that are very frequently executed and consume a considerable proportion of the total execution time. Furthermore, when those pieces of code are scattered within your program this is not an easy task. As DSLs abstract from these pieces of code by its declarative nature (stipulating the what rather than the how), adjusting them to improve performance is a lot easier. DSLs also capture the knowledge of how to turn an application into a more performant one, involving the peculiarities of amongst others third party components, external services, platform details, hardware specifications. In a DSL, this knowledge can be shared in many applications instead of describing them in the form of guidelines and best practices and hoping that developers apply them.

Contribution to design qualities

Portability: Being abstract and small, DSL code is can more easily be retargeted to support other platforms.

Effectuate

Optimization can be achieved by

- Enhancing the language runtime: adding new algorithms and functionalities
- Finetuning the semantics of the language: changing the execution order (permutations, caches, lazyness) or exploiting performance peculiarities of the language runtime
- Special-purpose constructs: give the user the ability to use optimized constructs when appropriate

Examples

- Example of adding new algorithms: BOB[1]

Boardgames can rapidly contain hundreds of rules. Stratego for example contains 25 different pawns, and require nearly a thousand rules to be described formally. In order for the game to be able to run on a mobile device, decision-pruning algorithms are used to significantly reduce the number of rules to be checked for each move.

- Example changing the execution order: Mori Algebra[11]

The algebra (called MORI algebra) is a domain-specific version of relational algebra and serves as intermediate language in the compiler from MORI/SQL to PL/SQL. The point of defining this domain-specific algebra was that terms in the algebra can be optimized using automatic transformations. Bind- and bind-not-restrictions are moved "downwards" in a query term. The earlier these restrictions are computed, the fewer rows have to be dealt with in the intermediate result tables. It is of special importance to try to move these restrictions below product operators, as these are the ones causing blowups in result table sizes.

- Example of the introduction of special-purpose constructs: QUIS[10]

In the QUIS (eQuation based UI in Smalltalk) DSL language, user interfaces are described by using bidirectional equations between a user interface and a data source. If the former changes then the later is updated and visa versa. Some of those equations can be quite time consuming to compute e.g. reading a high-resolution picture from a disc and displaying its thumbnail. In the event of bulk updates, like initializing the application's data model with a file, the user interface may become irresponsive and may in addition slow down the application's execution. In order to prevent this from happening a new operator is introduced which is non-blocking and builds up a queue of changes to be able to skip to the last entry instead of processing obsolete ones.

References

Higher-level optimizations that depend on domain knowledge that would be unavailable to a C compiler: remove redundant writes to idempotent registers and aggregate multiple writes to adjacent fields [16]

Finding the sweet spot: balancing principles

Standardization vs generalization

Domain knowledge is often particularized, exemplified and fragmented. Standardization and generalization are two principles that act upon this form of knowledge in opposite directions. Adopting this form of domain knowledge explicitly in the language strengthens the standardized form of DSL programs, while this decreases the generalization of those programs. Finding the sweet spot depends on the stability of the domain.

If the problem domain is unstable, then having explicit and specialized constructs in the DSL endangers the domain coverage. In unstable domains, the concepts used to program may no longer be sufficient in the near future. Hence, requiring new explicit and specialized constructs. The DSL risks to become bloated with a wide range of constructs. As a result, DSLs may get complex to maintain and complex to use. Generalization can reduce this complexity significantly as it reduces the number of language constructs, provided that the increased complexity of a more general concept is acceptable (see section about generalization).
For stable problem domains, whether DSLs should or should not contain explicit and specialized constructs depends more on the expertise of the DSLs' users. If expertise and the opportunities of training are rather low, than one should generalize with caution. Generalization is a scientific and formal process, and therefore a generalized concept may not be as familiar as the more specialized ones.

Standardization and Absorption vs compression

Standardization and absorption can be beneficial to compression. By using more explicit constructs one can make DSL programs smaller and thus more compressed. Semantics that is absorbed into the DSL is made implicit and thus not necessary to write down.

Generalization vs Optimization

A generalized solution is easier to optimize than a series of specific ones. [todo look into partial evaluation]

Compression vs Representation

When pushed to the extreme, compression makes writing code more difficult and leads to poorly readable code. Caution is advised to ensure that the resulting quality of well-represented code is not reduced by an over emphasis on compression.

Absorption vs Compression

Pushing frequency to the extreme case implies that a frequent programming pattern is *not an explicit* feature but an *implicit* feature of the language.

Abstraction vs Compression

Abstractions in the form of patterns increase code size, they do not compress. Because compressions are often computed using rather crude measurements, it is better to balance abstraction and compression depending on the added complexity of an abstraction. If the added complexity is considerable but not essential than compression is to be favoured.

Standardization vs Optimization

Assisting the user when solving problems in a DSL may come at a cost. Not all languages employ static mechanisms to increase the reliability. Some languages increase runtime checking to facilitate debugging or to improve error handling. The relationship between standardization and optimization is not purely of a conflicting nature. There is rather an opportunity here: optimization can undo some of the performance penalties incurred by standardization.

[1] Denis Coppens, Thomas Cleenewerck, Kris Deschutter, Build Your Own Boardgames (BOB), http://soft.vub.ac.be/~tcleenew/BOB/website/index.html

[2] Ian Piumarta , Extended Example: A Tiny TCP/IP Done as a Parser. Technical Report STEPS Toward The Reinvention of Programming., 2007-2008

[3] Edward Yourdon and Larry L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and System Design. Prentice-Hall, 1979 (Facsimile edition 1986). ISBN 0-13-854471-9.

[4] Arie van Deursen, Paul Klint, Joost Visser, Domain-Specific Languages: An Annotated Bibliography, CWI, Amsterdam, The Netherlands, http://homepages.cwi.nl/~arie/papers/dslbib/.

[5] aurent Réveillère and Fabrice Mérillon Charles Consel Renaud Marlet Gilles Muller and Fabrice Merillon and Charles Consel and Renaud Marlet and Gilles Muller, A DSL Approach to Improve Productivity and Safety in Device Drivers Development, In Proceedings of the 15 th IEEE International Conference on Automated Software Engineering (ASE), IEEE Computer Society Press, 101-109, 2000,

[6]Eelco Visser, WebDSL

[7] Weinberg 1971, The Psychology of Computer Programming

[8] Douglas Crockford, The World's Most Popular Programming Language Has Fashion and Luck to Thank, 2008, http://www.insideria.com/2008/03/the-worlds-most-misunderstood.html,

[9] Steve Lohr , Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists, & Iconoclasts

[10] Thomas Cleenwerck, eQuation based UI in Smalltalk (QUIS), http://soft.vub.ac.be/~tcleenew/ConfigurationLanguagePortal/portal.html.

[11] Niels H. Christensen, Domain-specific languages in software development and the relation to partial evaluation, PhD thesis, DIKU, Dept. of Computer Science, University of ?

[12] John F. Pane, A Programming System for Children that is Designed for Usability, School of Computer Science Computer Science Department Carnegie Mellon University Pittsburgh, PA , May 3, 2002

[13] Joanne M. Atlee, Parnas Tables: A Practical Formalism, David L. Parnas Symposium

[14] W M McKeeman: Programming Language Design; In: Compiler Construction (LNCS 21); 1974

[15] David K. Farkas, The Concept of Consistency in Writing and Editing , Journal of Technical Writing and Communication, Volume 15, Number 4 / 1985, 353 – 364.

[16] Christopher L. Conway, Stephen A. Edwards, NDL: A Domain-Specific Language for Device Drivers, In Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES), 30—36, 2004, ACM Press.

[17] Scott Thibault, Renaud Marlet, Charles Consel, Domain-Specific Languages: from Design to Implementation Application to Video Device Drivers Generation, EEE Transactions on Software Engineering, 25, 363—377,1999

[18] Paul Graham: Five Questions about Language Design; private notes 2001.

[19] Daniel L. Moody, The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering, IEEE Transactions on Software Engineering, pp. 756-779, November/December 2009 (vol. 35 no. 6), IEEE Computer Society