CoreCard Software White Paper Series

# Introducing the concept of DSL to a non-technical person

**Jan 2012**

CoreCARD SOFTWARE

# Contents

# Introduction

This white paper is aimed at introducing the concept of DSL to the business/non-technical person and explaining how the use of DSLs can greatly increase the productivity of developers and make programming changes faster with less risk to the business. Let's start with the acronym DSL; it stands for Domain Specific Language. In very simple terms DSL is a programming language that is meant to solve problems and design solutions in a given problem domain.

A DSL, in contrast to a General Programing Language (GPL), can't be used outside the domain it was designed for. For instance, the DSL used by CoreCard Software was created specifically to reduce the time required to develop and make changes to its highly sophisticated financial transaction software products. DSL provides a meaningful level of abstraction to the programmer such that he/she does not have to deal with unnecessary details in executing the tasks for the domain. Some such abstractions provided by CoreCard's DSL are Multi-currency support, interface to DBMS, PCI security compliance, Account hierarchy and Aging chains.

If the term "abstraction" seems too abstract, the following example may help. In a financial account management system, one can transfer an account from one financial product (such as "90 days same-as-cash") to another (such as "revolving credit limit"). This can be done by creating a new account under the destination product, personalizing it with account holder information, and transferring all balances, transactions, payments, status flags and other details to the new account. Alternatively, you can create an imaginary entity/object that has all the information needed to complete the account transfer and can invoke all individual processes needed to do so. Let us call this entity "Transfer account to new product". What we have done here is abstraction. Transferring an account to a new product now merely requires invoking the abstract entity "Transfer account to new product" without worrying about any individual processes, entities or other details.

DSLs are not a new concept; they have always existed in the form of special purpose languages. But they are more practical now due to the availability of tools for modeling and creating DSLs. In fact, you might have used a DSL without being aware of it. Excel spreadsheets and SQL are two very powerful and widely used Domain Specific Languages. Here is how: Excel spreadsheets have built in functions for all manipulations that you might want to do on tabular data. The user need not go into details of how these functions work or write step by step instructions, but only needs to know which ones to use for what purpose (Microsoft has done a good job naming the functions in user friendly fashion). Another key trait that makes Excel instructions domain specific is that they can't be used elsewhere. Of course, similar functions can be written in any GPL but they won't be single line instructions or might not be as easy to understand as in Excel. Similarly, SQL is a handy tool/language for accessing, managing and updating relational databases only. It can't be used elsewhere. Once again, any GPL can do the job, but it might require several lines of code to replace a single SQL command. Other widely used examples of horizontal DSLs are HTML, WS* standards and Business Process Execution language. Some other DSLs that programmers might be familiar with are Ant, Rake, Make, CSS, RSpec.

Excel and SQL are examples of DSLs for a horizontal domain which are applicable across a large range of applications. Vertical DSLs, by their nature, are much narrower. They apply to a specific industry vertical

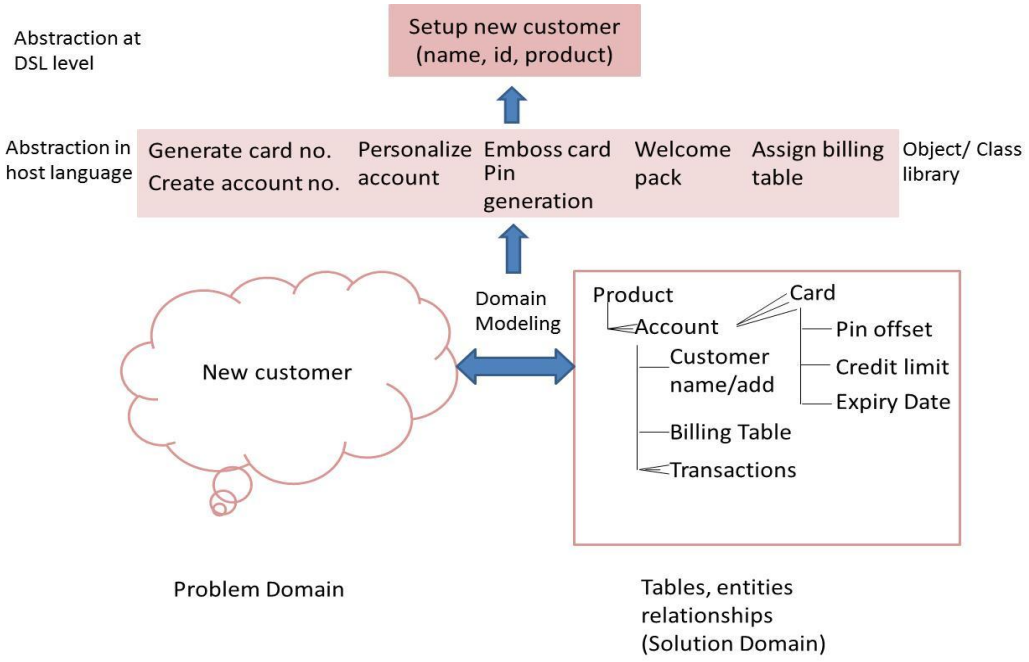
or to a small problem within an industry. Since most vertical DSLs are developed within companies as their intellectual property, you might not have come across them. Some vertical DSLs used by big corporations are Python (Nokia), East-ADL (Volvo), Amphion (NASA), SLL (Lucent) and Vampire (Philips, for medical applications).

## Analogy from the tangible world

A robot made for automotive assembly lines can only assemble cars and does that pretty well. It has been fitted with tools with which it can lift stuff, move and fix doors, seats, lights, engine, and music player in the right positions. We can call this robot a domain specific tool (for cars only). In contrast, a wrench, fork lift or a screwdriver can be considered general purpose tools. ie. they can also be used in house construction or in a nuclear power plant. Though general purpose tools (wrench, screwdriver) can be used in a wide range of applications, they won't be nearly as efficient when it comes to assembling a car. It takes a lot of effort, time and expertise to build a domain specific tool (a robot), but is worth every penny when it comes to its specific application. The same concepts apply to software DSLs.

## Domain Modeling

A critical step in creation of a DSL is **Domain Modeling**. Domain modeling in software engineering can be considered as creating a conceptual model that relates to the problem domain and defines artifacts and tools for the solution domain. In a domain model, the entities, processes, their collaborations and restrictions are named using a terminology that business users or domain experts use. After creating a domain model, meaningful abstractions can be defined in the host general purpose programming language to solve the problem at hand. These GPL abstractions can be in the form of objects/classes/methods contained in libraries. A DSL, which is another level of abstraction, can be written on top of these GPL abstractions. The following diagram will help understand this concept better.




This diagram illustrates modeling and creating DSL for a "New Customer" entity in the credit card problem domain. This process can be extended to all entities and processes/business rules in the problem domain to form a complete domain specific language. A DSL user would now only need to program "Setup new customer (name, id, product)" in order to create a new customer account in the system. Using a DSL abstraction isolates the user from low level processes, artifacts, rules and restrictions and makes programming much easier, faster and freer of bugs.

## Features of a DSL

- *Limited expressiveness* – A DSL programming language is said to be of limited expressiveness because unlike a GPL it can't support varied data structures, controls and abstraction structures. DSLs support only a bare minimum of features required for its domain. Ex. SQL is a powerful language for relational databases, but it can't be used to display web pages. Though limited in expressiveness across domains, DSLs are very expressive within a domain.
- *Syntax* – DSLs have their own syntax which can be much different from the host language.
- *Fluency* – Because the domain model and the DSL are designed in collaboration with domain experts/business users, a DSL's syntax offers a very high degree of fluency and readability to the users in the given domain.
- *Aggregation/Abstraction* – DSLs allow programmers to perform a logical business function with minimal instructions. This has been illustrated with the earlier examples of "Transfer account to new product" and "Setup new account". This aggregation/abstraction also allows reusing the code wherever required to do the same function. This feature makes DSLs seem like a close cousin of APIs, but the difference lies in the fact that APIs are still difficult for business users to read and understand. Martin Fowler, in his book "Domain-Specific Languages", draws a nice

analogy from English language. As he explains, APIs can form the vocabulary of a language but lack the grammar whereas DSLs provide both vocabulary and grammar.

## Benefits

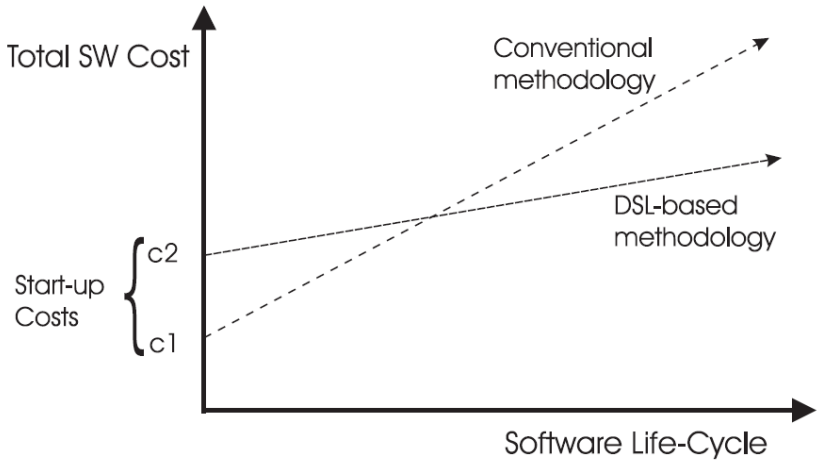
- *Less steep learning curve* – DSLs focus on a small problem area, are very fluent to read and are close to the business rules. A DSL, with the help of its abstractions, provides a clear view of the domain. These attributes allow programmers to learn DSLs faster than GPLs and also to start thinking in terms of business problems.
- *Fewer errors, easier to find mistakes and modify* – DSLs have limited expressiveness making it hard to say the wrong thing. Also because of English-like syntax, it's easier to spot mistakes and modify. Programmers are not exposed to the details of programming; in our example of transferring an account to a new product, the programmer need not know how to transfer each transaction over to the new product. This makes it significantly more difficult to go wrong.
- *Avoids duplication of work* – DSLs help avoid duplication of work by gathering common code and reusing it wherever needed. Expertise and skills of domain experts are captured in the DSL and reused by other users, allowing faster development cycles and response to new business requirements.
- *Significantly reduced coding effort* – BASIC language provided significant productivity improvement over assembler by allowing programmers to write one line of code instead of five or more instructions in assembler language. Similarly a single line of DSL code might represent hundreds or even thousands of lines of general purpose programming language code providing great productivity gains.
- *Improved communication with Domain Experts* - The business users or domain experts can read the DSL code easily to understand what is going on, spot mistakes and suggest improvements. DSLs allow domain experts to talk more effectively with the programmers who write the rules, thus enforcing better code quality, reduced time and defects. Business users can also write pseudo code which can be easily refined into proper DSL code by programmers. However, this argument does not apply to all DSLs. Most horizontal domain DSLs do not allow for better communication between programmers and domain experts/business users because they provide abstraction of technical solutions, not of business rules. For ex. SQL will not help communication in a financial services company, whereas a DSL written for financial services vertical would.

## Potential Drawbacks

- *Upfront cost, time and effort* – Considerable resources have to be deployed to develop a model/library and a DSL encompassing them. From CoreCard's experience in the very rapidly changing financial transaction markets, the significant upfront investment in creating its DSL is justified by shorter development time for new features and improved responsiveness to new market requirements. And building a model and library for your organization or domain helps even if you choose to develop applications in GPL rather than DSL.

- *Expertise required for building DSL* – You cannot hire just any developers and expect them to begin to immediately write DSLs. They have to know your domain and understand the abstractions in your model and libraries.
- *App developers have to be trained* – GPL app developers have to be trained on the specific DSL and model. But DSL, being high level abstraction of entities, objects and processes, is easier to learn and, once trained and fluent in the DSL, app development proceeds faster and with fewer errors.
- *Blinkered abstraction* – Programmers might try to fit a new problem into existing abstractions rather than create a new one to solve the problem. DSL architects and owners in any organization have to be alert to spot and eliminate such forced and inefficient solutions.

Finally, what matters for an organization is the aggregate cost of software development and maintenance in the long run. The following graph published by Martin Karlsch in his thesis for domain specific languages represents how use of DSLs impacts total software costs.




Though initial costs and effort might discourage some organizations, a well thought out and developed DSL will more than pay for the initial investment over time. DSL is most suitable for domains which have complex and very specific process flows and business rules. Decision to use a DSL Vs GPL should be made after careful deliberations.

**References**

**The knowledge and concepts about DSLs for this white paper have been drawn from the following sources:**

(http://www.dsmforum.org/cases.html).

http://www.ibm.com/developerworks/library/ar-mdd4/

http://blogs.msdn.com/b/publicsector/archive/2005/10/03/476666.aspx

http://www.theenterprisearchitect.eu/archive/2009/05/06/dsl-development-7-recommendations-for-domain-specific-language-design-based-on-domain-driven-design

http://queue.acm.org/detail.cfm?id=1989750

http://karlsch.com/frodo.pdf

Book: Domain Specific Languages By Martin Fowler

Book: DSLs In Action by Debashish Ghosh