

# Patterns for the definition of Programming Languages

Arno Haase, [Arno.Haase@Haase-Consulting.com](mailto:Arno.Haase@Haase-Consulting.com)

## Introduction

Creating programming languages has been a pastime of computer people since even before there were computers. Recently however it has become increasingly common for programming languages to be defined, be it as part of frameworks and libraries, be it within the scope of a single project.

This trend is partly due to the excellent tool support available today, both for creating parsers and for writing editors and even interactive debuggers for a newly created language.

Another reason is the wide availability of powerful computers, relaxing the need for programs that are extremely optimized for performance. This affects both runtime performance and build time overhead, the latter being the bigger issue since interpreters frequently have far less actual impact at runtime than their critics claim.

This paper reflects on this “roll your own programming language” trend and attempts to identify recurring patterns. It intends to help language designers with little experience to avoid common pitfalls and find better solutions for coming up with domain specific languages to supplement a 3GL. It does not however attempt to provide an introduction to domain analysis or language design.

It focuses on the programming languages themselves and their elements rather than the tooling around the language. Compilers, interpreters, editors, refactoring support and debuggers are extremely important parts of the ecosystem of a language, and their quality is an important factor for their acceptance. They are however large topics of their own that have been dealt with elsewhere, and I choose to ignore them for the purpose of this paper.

I would ask the Eurolop workshop to concentrate their feedback on the granularity and completeness of the patterns. Does the way they are cut make sense? Are core aspects missing?

And finally, I would like to thank Markus Völter for the feedback he provided while shepherding this paper.

## EVOLVING LANGUAGE

When creating a programming language, start with a minimal feature set, growing it from this foundation.

\* \* \*

Creating a programming language is a daunting task. Part of the challenge lies in technical issues – writing an efficient compiler or interpreter, providing editor support etc. – but there has been significant progress in these areas, making them increasingly simple. What remains is the challenge of tackling the intrinsic complexity of defining a language that is a good fit for the problem at hand.

There is no truly “general purpose” language. Hardware drivers and demos are programmed in assembly language even today, and languages such as C++ which support a wide range of paradigms are perceived as unwieldy and brittle.

There are proven features in existing languages, but it is not desirable to create a language that contains as many of them as possible since such a bloated language would be hard to learn. More importantly, it would provide so much freedom in extending and redefining itself that conceptually it would take on the nature of a toolbox for creating programming languages (internal DSLs in the terminology introduced by Martin Fowler). Common Lisp with its macro mechanism falls into this category. This is not a bad thing, but it moves the role of language designer to the user of the language, giving her the responsibility for choosing and defining the language features to be used in the actual code.

**Therefore**, start a new programming language with a minimal feature set and let it evolve from there based on experience with actual usage, be it in a lab setting or in real-world scenarios. Making a language self-hosting, i.e. writing the compiler and/or runtime environment for a language in that same language, is a common means towards that end for full-blown languages.

If a language is to be a good fit for a particular kind of problem, it needs to evolve based on experience in an agile way. It must do so without breaking backward compatibility. So having a minimal number of powerful language features with few and well defined points of interaction is the most desirable scenario.

Starting with a clearly scoped `DOMAIN SPECIFIC LANGUAGE` is a good foundation because it helps to keep a language focused during its evolution. Having an established `EXTENSION POINT` mechanism removes the pressure to add rarely needed features that endanger the focus of the language.

Putting new features into a `STANDARD LIBRARY` instead of the language core helps to avoid unnecessary bloat of the language.

\* \* \*

Widely used programming languages like C++, C# or Java are going through a constant evolution, having new features added to them and – more rarely – existing features redefined or “fixed”.

On a smaller scale, DSLs developed for a concrete program should evolve as feature requests become apparent.

## DOMAIN SPECIFIC LANGUAGE

Keep a language focused in the face of evolution by splitting off explicit domain specific languages that are EVOLVING LANGUAGES with scopes of their own.

\* \* \*

The strength of a programming language lies in the abstractions it provides. They can make it a good fit for a set of problems, or cause programmers to avoid it because they perceive it to cause more problems than it solves.

The beauty and expressiveness of domain specific languages is that they are a good fit for their domain, i.e. their abstractions closely reflect the domain abstractions. For non-technical domains, this means that the language concepts are largely non-technical, and for specialized technical domains, the abstractions should be from that specialized technical domain.

While it is tempting to add concepts from a more general domain or a different specialized domain, that quickly undermines the integrity and focus of a language because it is no longer grounded in a single domain.

**Therefore**, start a new language with a clear scope. Make it explicit which domain the language describes, and refactor it so that it keeps doing so.

If concepts from a different domain start to creep in, split of another DOMAIN SPECIFIC LANGUAGE for this new domain, and describe the different aspects of the system with these different languages.

Frequently, the backbone of a system is written in a general purpose language (GPL) which combines the aspects that are described by the different DSLs.

\* \* \*

It is common to use one or more frameworks that come with their specific configuration languages to describe particular aspects of a system, e.g. O/R persistence mapping or a system's component structure.

In model-driven development, it is common to describe a system using DSLs. A description of data types is a common starting point. As the system architecture evolves, persistence information can creep into the data type definitions, necessitating enhancements to the data type DSL. That can arguably be seen as stretching the DSL's scope without breaking it.

If however code for several programming languages is to be generated from these definitions, adding such descriptions to the DSL would really bloat it – especially since different people would work on these different definitions – and splitting off the target language specifics into a different DSL is the natural way to go.

## DECLARATIVE CONFIGURATION LANGUAGE

If the domain can be reduced to making choices from a set of alternatives, extract the logic to the interpreter / compiler, giving the language the declarative structure of a configuration file.

\* \* \*

Based on the XP principle of „doing the simplest thing that could possibly work“, a programming language – especially a DOMAIN SPECIFIC LANGUAGE – should be as simple as possible.

It should be both easy to use and easy to read. Ideally, it should provide just the amount of flexibility that is necessary to describe a solution, guiding the programmer by not offering superfluous degrees of freedom.

This is especially important for simple domains that can be described at a high level of abstraction. Any program can be written in a general purpose language. The abstractions of GPLs are however technical in nature, and they are designed not to pose any restriction on their expressiveness, giving them inherent complexity.

This mismatch is particularly pronounced in contexts where end users, i.e. non-programmers, are the desired users of a language.

**Therefore**, reduce the abstractions of a domain to making choices from pre-defined value sets when possible. This makes writing a program in the newly defined language as simple as writing a configuration file, which is in fact the same thing. Providing a “default” or “template” configuration file is a convenient way of documenting language features and guiding programmers.

Such declarative configuration languages can be extremely expressive. For example, the configuration of a web server could contain a flag *concurrency-model* with the possible values of *single*, *threads* and *processes*. While this choice is very simple to express in the configuration, it has profound impact on the behavior of the system.

This places the burden of implementing the programming logic on the runtime environment of the configuration language. Over time, such languages have a tendency of growing significantly as more and more options are added, especially if the language is successful.

Such a language can incorporate an `EXTENSION POINT` mechanism to alleviate the necessity of extending the language for every rarely needed special case. This places very little burden on the user of the language, making it a good choice to add to the language when first the need arises.

\* \* \*

The configuration of the Apache web server is an example of a powerful and comprehensive declarative configuration language. It provides a large number of choices to configure the behavior of a web server installation. It ties in with the modular plugin mechanism of Apache, allowing different plug-ins to define their own configurations that only they will know how to evaluate.

Frameworks often come with a configuration language. Examples of this include the configuration and mapping files of the Hibernate O/R mapper. This split illustrates how two related domains were split to be described by two different languages.

The configuration file proper, *hibernate.properties*, defines global settings such as the database connection parameters, the configuration of the connection pool and the second level cache, and default settings to tune performance. The mappings files use XML to describe mappings between Java classes and database tables, linking attributes to columns.

Both make use of `EXTENSION POINTS`. The definition of the type for a mapped property can e.g. be either one of a fixed set of values like *string*, *date* or *clob*, or it can be the fully qualified name of a Java class that defines a special mapping.

Another example of a declaration-style DSL is the well-known *make* utility that performs build operations based on configuration files, the so-called *make files*.

Make files basically consist of a declarative description of dependencies between artifacts, and the description of operations to create dependent artifacts. These commands can be any valid shell command, providing an `EXTENSION POINT` and relieving the *make* language of the burden of describing arbitrary commands.

## EXPRESSION LANGUAGE

If the side effects of a language – I/O in particular – follow a recurring pattern, move them from the language to its runtime environment so that the language itself only needs to deal with providing values from input.

\* \* \*

In many domains, the interesting point of variability is in the transformation (or “processing”, as it is often called) of data, while side effects such as I/O follow a more or less recurring pattern.

Such a distinction between transformation code and side-effects is not always obvious. Especially in non-functional programming paradigms, transformations of data are often expressed as modifications of existing data structures, making their nature more difficult to spot.

Still, the complexity inherent to true side effects like I/O is quite different from that of data transformations. Output channels have an unpleasant habit of being unreliable, necessitating special error handling code. In addition to that, I/O code frequently needs to take concurrency into account to some degree.

So giving a newly developed language the features necessary to describe arbitrary I/O operations means turning it into a fully grown general purpose language. Transforming data on the other hand boils down to evaluating expressions.

**Therefore**, restrict a language to the evaluation of expressions and move all code that deals with the results to its supporting runtime environment whenever possible.

If dealing with collections is an important part of the domain, provide the language with closures and a powerful collection types based on higher order functions so that it becomes possible to transform collections in a side-effect-free way.

If the language grows to a point where code duplication becomes an issue or loops are necessary, introduce functions. If performance in the execution of loops is an issue, implement tail recursion.

Provide an `EXTENSION POINT` mechanism for special kinds of expressions that go beyond the scope of the language, such as providing the current time stamp or incrementing a persistent counter to ensure uniqueness across several invocations.

\* \* \*

The validation of data is a simple and common example of `EXPRESSION LANGUAGES`, and many systems and frameworks come with DSLs for this. It basically consists of a list of predicates (i.e. boolean expressions) that define the validity of a given data element. These predicates are typically annotated with declarative data such as the data type to which they apply, the severity (error or warning) they denote or an error message describing their failure to an end user.

DSLs for describing input formats are another typical example. Such DSLs describe e.g. how the columns of a comma separated value file should be parsed into an internal data structure.

Template languages are a third group of DSLs in this category. Web languages like JSP or ASP are basically a means of transforming the data that is passed to them into a string representation that is then sent to the calling browser by their

runtime environment. Other template languages like XSLT, Xtend or Velocity basically do the same thing.

These template languages very frequently come with some kind of `EXTENSION POINT` mechanism. And while it is usually considered to bad practice to use that for side effects, that is an inherent possibility, making it possible to abuse them beyond their originally intended scope.

All these different DSLs share the trait that they take input provided by their runtime environment and provide output to the runtime environment based on this data. While it is their responsibility to transform the data, it is up to the runtime environment to handle I/O.

## STAND-ALONE LANGUAGE

If the domain requires a language to control the way a program interacts with the rest of the world, create a language for stand-alone programs.

\* \* \*

Every program needs to interact with its environment – if it does not provide any kind of output, it might as well not have run in the first place.

I/O support introduces significant complexity into a language, but since some part of every program must perform I/O, at least one of the used programming languages must support it.

**Therefore**, explicitly add I/O support to a programming language if its programs are intended to be runnable on their own.

Make the range of supported I/O channels and their associated programming models explicit. While it is possible to create a full-blown GPL, that is by no means necessary to incorporate I/O into a language. In particular, it is not necessary to make the whole range of I/O of the underlying operating system available to every programming language. Defining only means for file I/O for example avoids the concurrency issues inherent in server side socket communication.

The difference between a `STAND-ALONE LANGUAGE` and an `EXPRESSION LANGUAGE` is similar to the difference between using a library and using a framework. A program in a `STAND-ALONE LANGUAGE` builds on features provided by the language, but it maintains control over the flow of things. A program written in an `EXPRESSION LANGUAGE` on the other hand is “called” by the language runtime that also takes care of the actual flow of the program.

On a general note, obviously, the term “stand-alone” is relative. The vast majority of programs today rely on an operating system to run, and the pipes-and-filters architecture of Unix shell programs is an example of a runtime

environment that takes care of the vast majority of the I/O issues for programs, even if they are written in GPLs.

The distinguishing feature of a Stand-Alone programming language is that it *enables* programs control their I/O.

\* \* \*

Obviously, general purpose language like Java, C# or C are STAND-ALONE LANGUAGES. Haskell, Common Lisp, Visual Basic or Perl also fall into this category.

Stand-alone languages by their very nature need to address technical issues entailed by their support for I/O, making them rather wider in scope and more elaborate than is often warranted for DOMAIN SPECIFIC LANGUAGES.

The Xpand template language however is an example of a DSL that controls its output. It has language elements to control which part of the generated output should go to which file. This specific I/O model requires only a minimum of overhead, yet it enables the language to stand on its own.

## SYNTACTIC ICING

Design a language for conceptual cleanness first, adding convenience syntax on top rather than permitting syntactic concerns to dominate the language design.

\* \* \*

Syntax is an important part of the definition of a programming language, since syntax is what programmers will be writing and, even more importantly, reading.

Introducing special syntax for common cases however has a tendency of introducing ambiguities and subtle issues. It can also get in the way of language evolution because all subsequent changes and extensions must maintain this special syntax.

**Therefore**, first and foremost design a language for conceptual cleanness. If common idioms are difficult to express, first check if an extension to the STANDARD LIBRARY could resolve this, possibly using an EXTENSION POINT mechanism in the library code. Evolving a library is simpler than evolving the language itself.

If that proves insufficient, consider ways of adding features to the language itself that provide ways of concisely expressing the idiom. Adding such features often solves a group of more general problems and makes the language more powerful rather than just simplifying a specific idiom.

Only if neither of these approaches prove satisfactory, add special syntax to the language. If you do this, be thorough in giving the new syntax a very precise meaning. Align it with existing language features as far as possible (rather than deliberately cutting across several features) to simplify language evolution.

The term “syntactic sugar” is commonly used to describe convenience syntax. However, it is misleading in several ways. Firstly, it suggests the syntactic enhancement is (and should be) an integral part of the language, when it actually should be as far removed from the language core as possible. Secondly, sugar can be added to a recipe early on, whereas syntactic enhancements should be postponed as far as reasonably possible.

For these reasons, the term `SYNTACTIC ICING` (as in “icing on a cake”) more accurately describes the concept.

\* \* \*

Programming languages that focus on string processing often have several kinds of string literals, enclosed in double quotes, single quotes and sometimes slash characters. This is done to simplify having string literals that contain quote characters. It is a good example of syntactic icing – neither an extension to the `STANDARD LIBRARY` nor adding language features could easily provide a substitute, and it is very well aligned with the language feature “string literal”.

Support for regular expressions in Groovy is another good example. There is an operator `==~` that combines two strings, indicating if the first of them is matched by the regular expression in the second. This convenience syntax builds on the regex support in Groovy's `STANDARD LIBRARY`, and its semantic is defined in terms of the standard library classes. One caveat however is that it introduces a dependency of the language on its `STANDARD LIBRARY`.

Defining a special syntax for operations like *select*, *collect* or *exists* on collections the way OCL does however is bad style. These operations basically require passing an expression as a parameter, and introducing higher order functions and closures provide a more general and more powerful solution without introducing special cases to the language.

## EXTENSION POINT

Give a language features to include functionality written in another more widely scoped language, making it possible to include unforeseen special cases without compromising the language's focus.

\* \* \*

Every programming language is limited in scope. It is desirable and even necessary for a language to maintain its focus, the alternative being for it to grow wildly and become unwieldy to use and a nightmare to maintain. This is particularly true for DSLs with their clearly defined domain, but even GPLs like `C#` or `Java` are not a good fit for e.g. writing low-level driver code.

And in the majority of cases, a program in a language needs to do only things expressible in that language, at least for well defined and well chosen languages. Nonetheless, every now and then a situation arises where it would be nice to go slightly beyond the scope of the language.

**Therefore**, give a programming language `EXTENSION POINTS` that allow code written in a more general language to be integrated. For a DSL, that can be code written in the underlying GPL, whereas GPLs like C or Java profit from the ability to call arbitrary assembly code.

This maintains the language's integrity and scope, avoiding bloat and marking exceptions as such. At the same time, it allows users of the language to add their own enhancements to the language, addressing cases that are situation specific.

\* \* \*

`EXTENSION POINTS` are a very common feature in GPLs. Java has JNI that allows calls to arbitrary other languages, and many C dialects have syntax to include assembler code.

Integrating code at the linker level is another way to make calls that go to a deeper level of abstraction, at least for linked languages. And script languages in the Unix culture commonly have means to call other executables, providing a highly natural interface for calls to code written in a lower-level language.

DSLs written in a GPL often have syntax to reference arbitrary code written in that language, even `DECLARATIVE CONFIGURATION LANGUAGES` or `EXPRESSION LANGUAGES`. Mapping files for the Hibernate O/R mapping framework written in Java e.g. accept the fully qualified name of a Java class implementing a certain interface in several places, a mechanism commonly used in Java.

## STANDARD LIBRARY

Provide a well-defined library of frequently useful functionality, giving users convenience without bloating the language core itself.

\* \* \*

As a programming language is adopted more widely, an increasing number of feature requests tends to arise, and idiomatic usage patterns appear.

While it is possible in principle to address all of these in the language itself, that approach very quickly bloats the language, makes it difficult to learn and maintain, and causes subtle interactions between language features that are difficult to keep track of.

Ignoring them however is not an option either. We want our languages to be convenient to use, and code duplication due to missing functionality that needs

to be incorporated in most new programs can quickly turn into a maintenance nightmare.

Documentation and sample code can alleviate this some, but what we really want is to provide our users with ready-made solutions.

**Therefore**, incorporate frequently needed functionality into a `STANDARD LIBRARY` written in the language itself. Such a library is more flexible than the language core itself, yet it still provides reliably supported functionality.

Make it explicit which code is part of the standard library and which is not, and be particularly thorough in QA and maintaining backward compatibility for standard library code.

Evolving the standard library is simpler than growing the language core in several regards. Firstly, the library is simpler to develop than new language features, because it does not require extensions to the compiler / interpreter and can leverage existing IDE support.

Secondly, dependencies on other features are easier to track in a library than the language core. Thirdly, it is possible to release code as part of a non-standard library and gather experience with it before submitting it to the rigor of standardization. And fourthly, the standard library can serve as sample code of high quality, illustrating idiomatic usage of the language.

Obviously, support for `MODULES` is a prerequisite for defining a `STANDARD LIBRARY`.

\* \* \*

Practically all GPLs come with standard libraries.

## **MODULE**

For programs of non-trivial size, give a language a feature to split a program into several independent building blocks that can be spread over several source files.

\* \* \*

Initially, programs written in a newly created programming language may be small enough to fit into a single file. That is especially true for `DECLARATIVE CONFIGURATION LANGUAGES`. But as a language is more widely adopted, programs have a tendency to grow.

That can lead to several issues. One of them is that can be somewhat difficult to find your way around a single file of several thousand lines of code – it is just too long, and comments can only do so much to help.

In addition to that, the dependencies between different parts of a program tend to become hard to keep track of as it grows.

And finally, it is difficult for several people to work on different parts of a file at the same time. Diff and merge operations can deal with this to some degree, but it quickly becomes painful, and the ability to split a program into several files makes tracking of changes in version control a lot simpler.

**Therefore**, give a language support for splitting a program into several files if several people need to work on a program at the same time or program size becomes too big to manage in a single file.

If the language goes beyond simple declarative configuration, use this module mechanism as a means of encapsulation. Introduce a separation between program parts that are visible from outside the module, and those visible only inside it. Such support for encapsulation adds very little overhead to the language, while at the same time providing a powerful means for structuring programs.

As an aside, modules are a different concept from abstract data types, but classes in object-oriented languages frequently combine the two.

\* \* \*

DECLARATIVE CONFIGURATION LANGUAGES frequently allow configurations to be split into several files, especially if they provide an extension mechanism. The configuration of the Apache web server is an example of this.

Java incorporates the splitting into files at the level of the language core, forcing every public classes to be in a separate file of the same name as the class.

C provides modularity at two levels. The first of these is the include directive which has no semantic meaning but allows a compilation unit to be split into several files, enabling header files to be shared between C files. The second mechanism is at the linker level: Compilation units can call functions defined in other compilation units, and compilation units can have data private to them.