**Part II**

# DSL Design

dsl engineering 55

This part of the book has been written together with Eelco

Visser of TU Delft. You can reach him at **e.visser@tudelft.nl**.

Throughout this part of the book we refer back to the five case studies introduced in Part I of the book (Section 1.11). We use a the following labels:

**Component Architecture:** This refers to the component architecture case study described in Section 1.11.1. J

**Refrigerators:** This refers to the refrigerator configuration case study described in Section 1.11.2. J

**mbeddr C:** This refers to the mbeddr.com extensible C case study described in Section 1.11.3. J

**Pension Plans:** This refers to the pension plans case study described in Section 1.11.4. J

**WebDSL:** This refers to the WebDSL case study described in Section 1.11.5. J

Note that in this part of the book the examples will only be used to illustrate DSL *design* and the driving design decisions. Part III of the book will then discuss the implementation aspects.

Some aspects of DSL design have been formalized with mathematical formulae. These are intended as an additional means of explaining some of the concepts. Formulae are able to state properties of programs and languages in an unambiguous way. However, I want to emphasize that reading or understanding the formulae is *not* essential for understanding the language design discussion. So if you’re not into mathematical formulae, just ignore them.

This part consists of three chapters. In Chapter 3 we introduce important terms and concepts including *domain*, model *purpose* and the structure of programs and languages. In Chapter 4 we discuss a set of seven dimensions that guide the design of DSLs: expressivity, coverage, semantics, separation of concerns, completeness, language modularization and syntax. Finally, in Chapter 5 we look at well-known structural and behavioral paradigms (such as inheritance or state based behaviour) and discuss their applicability to DSLs.

*3*

## Conceptual Foundations

*This chapter provides the conceptual foundations for the discussion of the design dimensions. It consists of three sections. The first one,* Program, Languages and Domain *defines some of the terminology around DSL design we will use in the rest of this chapter. The second section briefly address the* Purpose *of programs as a way of guiding their design. And the third section briefly introduces parser-based and projectional editing, since some design considerations depend on this rather fundamental difference in DSL implementation.*

### 3.1 Programs, Languages and Domains

|  |  |
| --- | --- |
| Domain-specific languages live in the realm of *programs*, *languages* and *domains*. So we should start by explaining what these things are. We will then use these concepts throughout this part of the book.  As part of this book’s treatment of DSLs, we are primarily interested in *computation*, i.e. we are aimed at creating executable |  |
| software1. So let’s first consider the relation between programs |  |
| and languages. Let’s define *P* to be the set of all conceivable programs. A *program p* in *P* is the *conceptual* representation of some *computation* that runs on a universal computer (Turing machine). A *language l* defines a structure and notation for *expressing* or *encoding* programs from *P*. Thus, a program *p* in *P* may have an expression in *L*, which we will denote as *pl*.  There can be several languages *l*1 and *l*2 that express the *same* conceptual program *p* in different way *pl*1 and *pl*2 (**fac-** |  |

**torial** can be expressed in Java and Lisp, for example). There may even be multiple ways to express the same program in a single language *l* (in Java, **factorial** can be expressed via recursion or with a loop). A transformation *T* between languages *l*1 and *l*2 maps programs from their *l*1 encoding to their *l*2 encoding, i.e. *T*(*pl*1) = *pl*2.

|  |  |
| --- | --- |
| It may not be possible to encode all programs from *P* in a given language *l*. We denote as *Pl* the subset of *P* that can |  |
| be expressed in *l*. More importantly, some languages may be |  |
| *better* at expressing certain programs from *P*: the program may be shorter, more readable or more analyzable.  **Pension Plans:** The pension plan language is very good at representing pension calculations, but cannot practically be used to express other software. For example, user defined data structures and loops are not supported. J  *Domains* What are domains? We have seen one way of defining domains in the previous paragraph. When we said that a language *l* covers a subset of *P*, we can simply call this subset the *domain* covered with *l*. However, this is not a very useful approach, since it equates the scope of a domain trivially with the scope of a language (the subset of *P* in that domain *PD* is equal to the subset of *P* we can express with a language *l Pl*). We cannot ask questions like: "Does the language adequately cover the domain?", since it always does, by definition.  There are two more useful approaches. In the *inductive* or *bottom-up* approach we define a domain in terms of existing software used to address a particular class of problems or products. That is, a domain *D* is identified as a set of programs with common characteristics or similar purpose. Notice how at this point we do *not* imply a special language to express them. They could be expressed in any Turing-complete language. Often such domains do not exist outside the realm of software.  An especially interesting case of the inductive approach is where we define a domain as a subset of programs written in a specific language *Pl* instead of the more general set *P*. In this case we can often clearly identify the commonalities among the |  |
| programs in the domain, in the form of their consistent use of a set of domain-specific patterns or idioms2. This makes build- |  |

ing a DSL for *D* relatively simple, because we know exactly what the DSL has to cover, and we know what code to generate from DSL programs.

**mbeddr C:** The domain of this DSL has been defined bottomup. Based on idioms commonly employed when using C for embedded software development, linguistic abstractions have been defined that provide a "shorthand" for those idioms. These linguistic abstractions form the basis of the language extensions. J

The above examples can be considered relatively general – the domain of embedded software development is relatively broad. In contrast, a domain may also be very specific, as is illustrated by the refridgerator case study.

**Refrigerators:** The cooling DSL is tailored specifically towards expressing refrigerator cooling programs for a very specific organization. No claim is made for broad applicability of the DSL. However, it perfectly fits into the way cooling algorithms are described and implemented in that particular organization. J

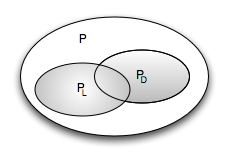
The second approach for defining a domain is *deductive* or *topdown*. In this approach, a domain is considered a body of knowledge about the real world, i.e. outside the realm of software. From this perspective, a domain *D* is a body of knowledge for which we want to provide some form of software support. *PD* is the subset of programs in *P* that implement interesting computations in *D*. This case is much harder to address using DSLs, because we first have to understand precisely the nature of the domain and identify the interesting programs in that domain.

**Pension Plans:** The pensions domain has been defined in this way. The customer had been working in the field of old-age pensions for decades and had a detailed understanding of that domain. That knowledge was mainly contained in the heads of pension experts, in pension plan requirements documents, and, to a limited extent, encoded in the source of existing software. J

In the context of DSLs, we can ultimately consider a domain

*D* by a set of programs *PD*, whether we take the deductive or inductive route. There can be multiple languages in which we can express *PD* programs. Possibly, *PD* can only be partially expressed in a language *l* (Figure 3.1).

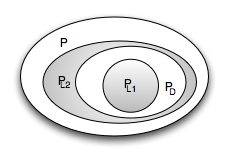
*Domain-Specific Languages* We can now understand the notion of a domain-specific language. A *domain-specific language*



*lD* for a domain *D* is a language that is *specialized* for encoding programs from *PD*. That is, *lD* is more efficient3 in represent-

|  |  |
| --- | --- |
| ing *PD* programs than other languages, and thus, is particularly well suited for *PD*. It achieves this by using *abstractions* suitable to the domain, and avoiding details that are irrelevant to programs in *D* (typically because they are similar in all programs and can be added automatically by the execution engine).  It is of course possible to express programs in *PD* with a general-purpose language. But this is less efficient – we may have to write much more code, because a GPL is not specialized to that particular domain. Depending on the expressivity of a DSL, we may also be able to use it to describe programs |  |
| outside of the *D* domain4. However, this is often not efficient |  |
| at all, because, by specializing a DSL for *D*, we also restrict its efficiency for expressing programs outside of *D*. This is not a problem as long as we have scoped *D* correctly. If the DSL actually just covers a subset of *PD*, and we have to express programs in *D* for which the DSL is *not* efficient, we have a problem.  This leads us to the crucial challenge in DSL design: finding regularity in a non-regular domain and capturing it in a lan- |  |

|  |  |
| --- | --- |
| understanding of the domain becomes more and more refined over time. In a DSL *l* that is adequate for the domain, the sets *Pl* and *PD* are the same. |  |
| *Domain Hierarchy* In the discussion of DSLs and progressively higher abstraction levels, it is useful to consider domains organized in a hierarchy5, in which higher domains are a sub- |  |

guage. Especially in the deductive approach, membership of programs in the domain is determined by a human and is, in some sense, arbitrary. A DSL for the domain hence typically represents an explanation or interpretation of the domain, and often requires trade-offs by under- or over-approximation (Figure 3.2). This is especially true while we develop the DSL: an iterative approach is necessary that evolves the language as our

set (in terms of scope) of the lower domains (Fig. 3.3).

At the bottom we find the most general domain *D*0. It is the domain of all possible programs *P*. Domains *Dn*, with *n >* 0, represent progressively more specialized domains, where the set of interesting programs is a subset of those in *Dn*−1 (abbreviated as *D*−1). We call *D*+1 a subdomain of D. For example, *D*1.1 could be the domain of embedded software, and

*D*1.2 could be the domain of enterprise software. The progressive specialization can be continued ad infinitum, in principle.

For example, *D*2.1.1 and *D*2.1.2 are further subdomains of *D*1.1:

*D*2.1.1 could be automotive embedded software and *D*2.1.2 could be avionics software6.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Languages are typically designed for a particular domain *D*. |  | | Languages for *D*0 are called general-purpose languages7. Lan- |  | |

guages for *Dn* with *n >* 0 become more domain-specific for growing *n*. Languages for a particular *Dn* can also be used to express programs in *Dn*+1. However, DSLs for *Dn*+1 may add additional abstractions or remove some of the abstractions found in languages for *Dn*. To get back to the embedded systems domain, a DSL for *D*1.1 could include components, state machines and data types with physical units. A language for *D*2.1.1, automotive software, will retain these extensions, but in addition provide direct support for the AUTOSAR standard and prohibit the use of **void\*** to conform to the MISRA-C standard.

**mbeddr C:** The C base language is defined for *D*0. Extensions for tasks, state machines or components can argued to be specific to embedded systems, making those sit in *D*1.1. Progressive specialization is possible; for example, a language for controlling small Lego robots sits on top of state machines and tasks. It could be allocated to *D*2.1.1. J

### 3.2 Model Purpose

We have said earlier that there can be several languages for the same domain. These languages differ regarding the abstractions they make use of. Deciding which abstractions should go into a particular language for *D* is not always obvious. The basis for the decision is to consider the *model purpose*. Mod-

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | els8, and hence the languages to express them, are intended |  | | for a specific purpose. Examples of model purpose include automatic derivation of a *D*−1 program, formal analysis and model checking, platform-independent specification of func- |  | | tionality or generation of documentation9. The same domain |  | | concepts can often be abstracted in different ways, for different purposes. When defining a DSL, we have to identify the different purposes required, and then decide whether we can create one DSL that fits all purposes, or create a DSL for each purpose10. |  |   **mbeddr C:** The model purpose is the generation of an ef- |

ficient low-level C implementation of the system, while at the same time providing software developers with meaningful abstractions. Since *efficient* C code has to be generated, certain abstractions, such as dynamically growing lists or runtime polymorphic dispatch, are not supported even though they would be convenient for the user. The state machines in the **statemachines** language have an additional model purpose: model checking, i.e. proving certain properties about the state machines (e.g., proving that a certain state is definitely going to be reached after some event occurs). To make this possible, the action code used in the state machines is limited: it is not possible, for example, to read and write the same variable in the same action. J

**Refrigerators:** The model purpose is the generation of efficient implementation code for various different target platforms (different types of refrigerators use different electronics). A secondary purpose is enabling domain experts to express the algorithms and experiment with them using simulations and tests. The DSL is not expected to be used to visualize the actual refrigerator device for sales or marketing purposes. J

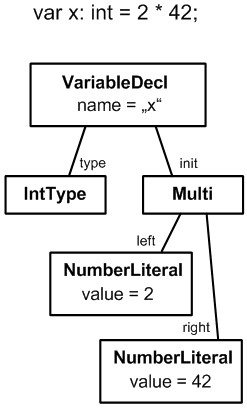
**Pension Plans:** The model purpose of the pension DSL is to enable insurance mathematicians and pension plan developers (who are not programmers) to define complete pension plans, and to allow them to check their own work for correctness using various forms of tests. A secondary purpose is the generation of the complete calculation engine for the computing center and the website. J

The purpose of a DSL may also change over time. Consequently, this may require changes to the abstractions or notations used in the language. From a technical perspective, this is just like any other case of language evolution (discussed in Chapter 6).

### 3.3 The Structure of Programs and Languages

The discussion above is relatively theoretical, trying to capture somewhat precisely the inherently imprecise notion of domains. Let us now move into the field of language engineering. Here we can describe the relevant concepts in a much more practical way.

*Concrete and Abstract Syntax* Programs can be represented in their abstract syntax and the concrete syntax forms. The *concrete syntax* is the notation with which the user interacts as he edits a program. It may be textual, symbolic, tabular, graphical or any combination thereof. The *abstract syntax* is a data structure that represents the semantically relevant data expressed by a program (Fig. 3.4 shows an example of both). It does not contain notational details such as keywords, symbols, white space or positions, sizes and coloring in graphical notations. The abstract syntax is used for analysis and downstream processing of programs. A language definition includes the concrete and the abstract syntax, as well as rules for mapping one to the other. *Parser-based* systems map the concrete syntax to the abstract syntax. Users interact with a stream of characters, and a parser derives the abstract syntax by using a grammar and mapping rules. *Projectional* editors go the other way round: user interactions, although performed through the concrete syntax, *directly* change the abstract syntax. The concrete syntax is a mere pro-



jection (that looks and feels like text when a textual projection is used). No parsing takes place. Spoofax and Xtext are parserbased tools, MPS is projectional.

While concrete syntax modularization and composition can be a challenge and requires a discussion of textual concrete syntax details, we will illustrate most language design concerns based on the abstract syntax. The abstract syntax of programs are primarily trees of program *elements*. Each element is an instance of a *language concept*, or *concept* for short. A language is essentially a set of concepts (we’ll come back to this below). Every element (except the root) is contained by exactly one parent

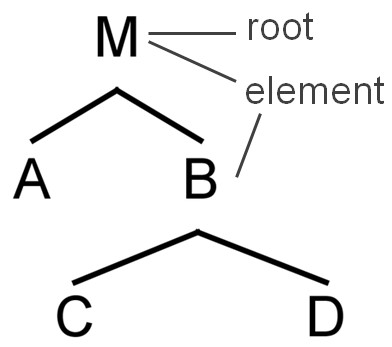
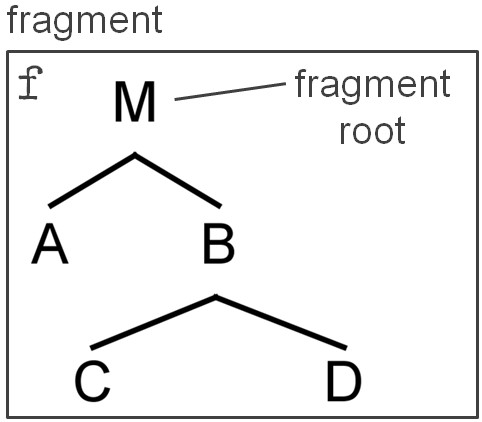


Figure 3.5: A program is a tree of program elements, with a single root element.

element. Syntactic nesting of the concrete syntax corresponds to a parent-child relationship in the abstract syntax. There may also be any number of non-containing cross-references between elements, established either directly during editing (in projectional systems) or by a name resolution (or *linking*) phase that follows parsing and tree construction.

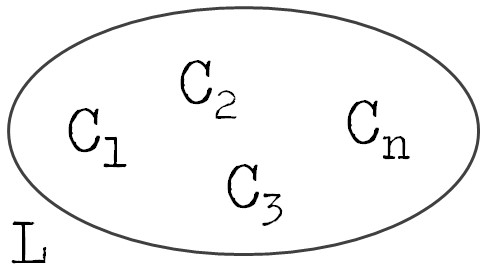
*Fragments* A program may be composed from several program *fragments*. A fragment is a standalone tree, a partial program. Conversely, a program is a set of fragments connected by references (discussed below). *Ef* is the set of program elements in a fragment *f*.

*Languages* A language *l* consists a set of language concepts *Cl* and their relationships11. We use the term *concept* to refer

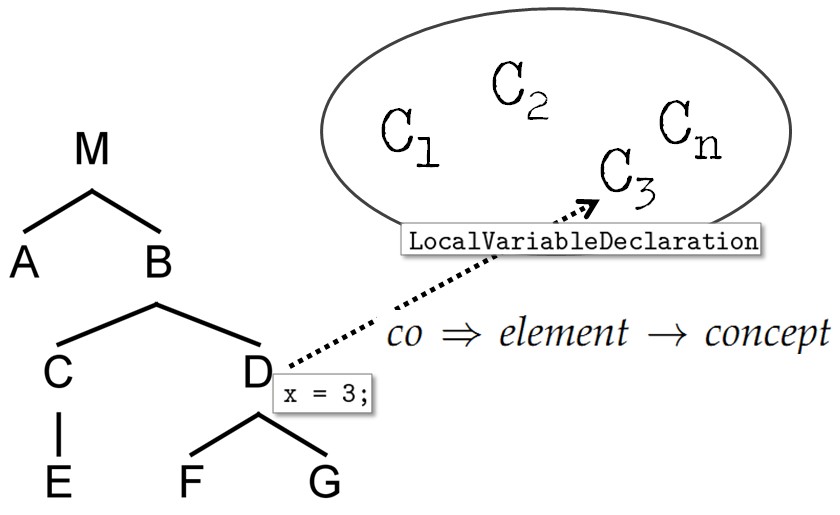


|  |  |
| --- | --- |
|  |  |
| syntax, abstract syntax, the associated type system rules and constraints as well as some definition of its semantics. In a fragment, each element *e* is an instance of a concept *c* defined |  |

to all aspects of an element of a language, including concrete

in some language *l*.

**mbeddr C:** In C, the statement **int x = 3;** is an instance of the **LocalVariableDeclaration** concept. **int** is an instance of **IntType**, and the **3** is an instance of **NumberLiteral**. J

F

*Functions* We define the *concept-of* function *co* to return the concept of which a program element is an instance: *co* ⇒ *element* → *concept* (see Fig. 3.8). Similarly we define the *languageof* function *lo* to return the language in which a given concept is defined: *lo* ⇒ *concept* → *language*. Finally, we define a *fragment-of* function *fo* that returns the fragment that contains a given program element: *fo* ⇒ *element* → *fragment* (Fig. 3.9).

*Relationships* We also define the following sets of relationships between program elements. *Cdnf* is the set of parentchild relationships in a fragment *f*. Each *c* ∈ *Cdn* has the properties *parent* and *child* (see figure Fig. 3.10; *Cdn* are all the

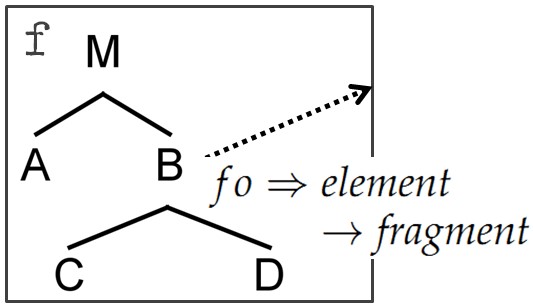
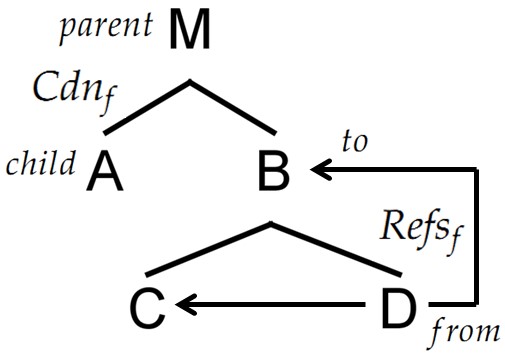


Figure 3.9: *fo* returns the fragment for a given element.



parent-child "lines" in the picture).

**mbeddr C:** In **int x = 3;** the local variable declaration is the *parent* of the **type** and the **init** expression **3**. The concept **Local- VariableDeclaration** defines the containment relationships **type** and **init**, respectively. J

*Refsf* is the set of non-containing cross-references between program elements in a fragment *f*. Each reference *r* in *Refsf* has the properties *from* and *to*, which refer to the two ends of the reference relationship (see figure Fig. 3.10).

**mbeddr C:** For example, in the **x = 10;** assignment, **x** is

a reference to a variable of that name, for example, the one declared in the previous example paragraph. The concept **LocalVariableRef** has a non-containing reference relationship **var** that points to the respective variable. J

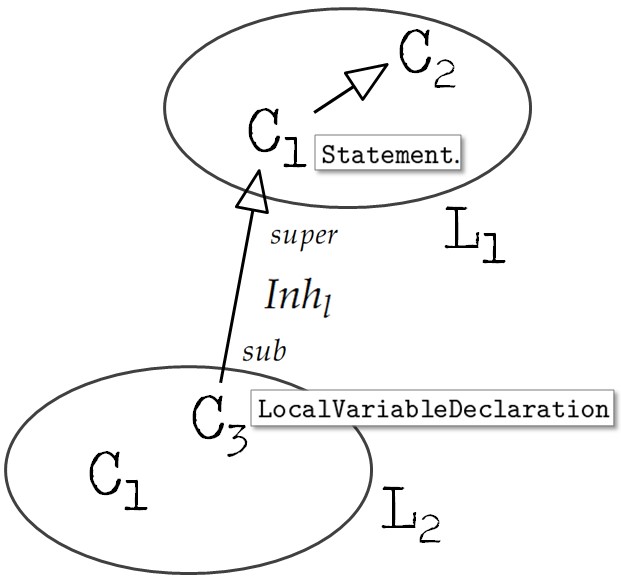
Finally, we define an inheritance relationship that applies the Liskov Substitution Principle (LSP) to language concepts. The

LSP states that,

In a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may be substitutes for objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)

The LSP is well known in the context of object-oriented programming. In the context of language design it implies that a concept *csub* that extends another concept *csuper* can be used in places where an instance of *csuper* is expected. *Inhl* is the set of inheritance relationships for a language *l*. Each *i* ∈ *Inhl* has the properties *super* and *sub*.

**mbeddr C:** The **LocalVariableDeclaration** introduced above extends the concept **Statement**. This way, a local variable declaration can be used wherever a **Statement** is expected, for example, in the body of a function, which is a **StatementList**. J



*Independence* An important concept is the notion of independence. An *independent language* does not depend on other languages. This means that for all parent/child, reference and inheritance relationships, both ends refer to concepts defined in the same language. Based on our definitions above we can define an independent language *l* as a language for which the following hold12:

∀*r* ∈ *Refsl* | *lo*(*r*.*to*) = *lo*(*r*.*from*) = *l* (3.1)

∀*s* ∈ *Inhl* | *lo*(*s*.*super*) = *lo*(*s*.*sub*) = *l* (3.2)

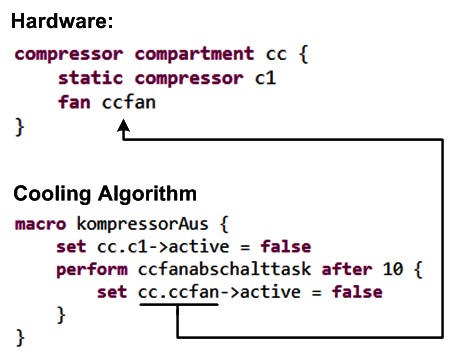
∀*c* ∈ *Cdnl* | *lo*(*c*.*parent*) = *lo*(*c*.*child*) = *l* (3.3)

Independence can also be applied to fragments. An *independent fragment* is one in which all non-containing cross-references *Refsf* point to elements within the same fragment:

∀*r* ∈ *Refsf* | *fo*(*r*.*to*) = *fo*(*r*.*from*) = *f* (3.4)

Notice that an independent language *l* can be used to construct dependent fragments, as long as the two fragments just contain elements from this single language *l*. Vice versa, a dependent language can be used to construct independent fragments. In this case we just have to make sure that the non-containing cross references are "empty" in the elements in fragment *f*.

**Refrigerators:** The hardware definition language is independent, as are fragments that use this language. In contrast, the cooling algorithm language is dependent. **BuildingBlockRef** declares a reference to the **BuildingBlock** concept defined in the hardware language (Fig. 3.12). Consequently, if a cooling program refers to a hardware setup using an instance of **BuildingBlockRef**, the fragment becomes dependent on the hardware definition fragment that contains the referenced building block. J



*Homogeneity* We distinguish *homogeneous* and *heterogeneous* fragments. A homogeneous fragment is one in which all elements are expressed with the same language (see formula 1.5). This means that for all parent/child relationships (*Cdnf* ), the elements at both ends of the relationship have to be instances of concepts defined in one language *l* (1.6):

|  |  |
| --- | --- |
| ∀*e* ∈ *Ef* | *lo*(*co*(*e*)) = *l* | (3.5) |
| ∀*c* ∈ *Cdnf* | *lo*(*co*(*c*.*parent*)) = *lo*(*co*(*c*.*child*)) = *l* | (3.6) |

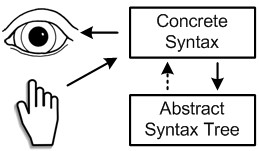
|  |
| --- |
| **mbeddr C:** A program written in plain C is homogeneous. All program elements are instances of the C language. Using the state machine language extension allows us to embed state machines in C programs. This makes the respective fragment heterogeneous (see Fig. 3.13). J |

### 3.4 Parsing versus Projection

This part of the book is not about implementation techniques. However, the decision of whether to build a DSL using a projectional editor instead of the more traditional parser-based approach can have some consequences for the design of the DSL.

So we have to provide *some* level of detail on the two at this point.

In the parser-based approach, a grammar specifies the sequence of tokens and words that make up a structurally valid program. A parser is generated from this grammar. A parser is a program that recognizes valid programs in their textual form and creates an abstract syntax tree or graph. Analysis tools or generators work with this abstract syntax tree. Users enter



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| programs using the concrete syntax (i.e. character sequences) and programs are also stored in this way. Example tools in this category include Spoofax and Xtext.  Projectional editors (also known as structured editors) work without grammars and parsers. A language is specified by defining the abstract syntax tree, then defining projection rules   |  |  | | --- | --- | | that render the concrete syntax of the language concepts defined by the abstract syntax. Editing actions *directly* modify the abstract syntax tree. Projection rules then render a textual (or other) representation of the program. Users read and write programs through this projected notation. Programs are stored as abstract syntax trees, usually as XML. As in parser-based |  | | systems, backend tools operate on the abstract syntax tree.  Projectional editing is well known from graphical editors; virtually all of them use this approach13. However, they can also be used for textual syntax14. Example tools in this cat- |  | |

egory include the Intentional Domain Workbench[[1]](#footnote-1) and JetBrains MPS.

In this section, we do not discuss the relative advantages and drawbacks of parser-based versus projectional editors in any detail (although we do discuss the trade-offs in the chapter on language implementation, Section 7). However, we will point out if and when there are different DSL design options depending on which of the two approaches is used.

1. **www.intentsoft.com** [↑](#footnote-ref-1)