This PDF is an excerpt of the full DSL book that contains only the chapter on Business DSLs, contributed by Intentional Software. You can get the full book (PDF or printed version) via http://dslbook.org
22

DSLs for Business Users

This chapter has been written by Intentional’s Mats Helander. You can reach him via mats@intentsoft.nl.

In this chapter we will examine using DSLs for business professionals. The example is a system in the healthcare domain – essentially a system for defining questionnaires and the business rules to process them. A secondary purpose of this chapter is to provide an impression of Intentional Software’s technology for defining DSL: the example system is built with the Intentional Domain Workbench.

22.1 Intentional Software

Intentional Software was one of the first companies to create a language workbench\(^1\), and their focus has been on business professionals and less on programmers as users for the DSLs\(^2\). Business professionals are often the source of domain knowledge. Today this knowledge has to be captured and explained to software engineers for it to be actionable. Agile principles help bridge this gap, but this communication gap remains the biggest obstacle in software development today. DSLs for business professionals have the potential to bridge this gap.

\(^1\) Intentional Software was started by Charles Simonyi after he left Microsoft. There he had been one of the early employees and served as the chief architect for Excel and Word, introducing the principle of WYSIWYG. During his later years he ran the Intentional Programming research project in Microsoft Research (which is described very well in Czarnecki and Eisenecker’s Generative Programming book\(^3\)). His company, Intentional Software, continues to work on the ideas pioneered by Intentional Programming.

\(^2\) This has always been a focus of DSLs. However, as we will see in this chapter, focusing on non-programmers leads to different tradeoffs in the design of the languages and the tools.
22.2 The Project Challenge

This case study describes an application in which domain knowledge is captured and maintained directly by the domain experts using DSLs, validated at the domain level, and used for code generation to create an executable application\(^3\). The domain is tele-health, where patients with chronic conditions or diseases like diabetes, hypertension or obesity stay at home, and are provided with daily recommendations based on observed values of various daily measurements of the patient. A medical professional has defined which values to observe for each particular patient, and the rules for the daily individual recommendations based on those values\(^4\). The input from the patient at home is provided through sensors, medical devices and patient interactions with the system through mobile devices, set-top boxes or web interfaces. The system needs to be flexible enough to address the requirements of multiple health care providers that will have different sets of criteria for different patients.

The system described in this chapter replaces a legacy system developed using a traditional approach in which domain knowledge was captured in big Excel documents that encoded the physician's rules. A typical rule looked like this:

\[
\text{if WHTR < 46 and (LDL < 100 and No LDL Meds) and (SBP < 125 and No BP Meds) and (HgbA1c >= 6.5 and No Glucose Meds)}
\]

This Excel text should be interpreted as:

\[
\begin{align*}
\text{if the patient} \\
\text{has a Weight Height ratio of less than 46} \\
\text{and} \\
\text{a cholesterol LDL level below 100} \text{ and does not take LDL medications} \\
\text{and} \\
\text{the systolic blood pressure level is less than 125} \text{ and does not take blood pressure medication} \\
\text{and} \\
\text{the hemoglobin A1c test is equal or greater than 6.5} \text{ and does not take glucose medication} \\
\text{then <advice according to diabetes plan}>.
\end{align*}
\]

The Excel spreadsheet had hundreds of rules like this. The repetition resulting from lack of abstractions available to the rules programmer meant that for each new observable attribute the number of rules doubled\(^5\). Each rule was then transformed by a programmer into rules for a Drools rules engine. The patient data had a similar workflow, in which information for the patient-recorded data was captured also in Excel sheets. Once this information was confirmed with the doctor, XML documents were created for this data to feed a custom web

---

\(^3\) The DSL is complete: no manual coding of "business logic" is required. If that were necessary, the premise of a DSL for business professionals would be infeasible.

\(^4\) This is not an expert system. All decisions are made originally by medical doctors.

\(^5\) The lack type checks, testing, refactorings, and all the other amenities we are used from an real IDE also hampered productivity and maintainability.
application application to be used by the patient to fill in the data.

The medical professional was overwhelmed with the complexity. It was clear that the doctors knew exactly what intentions they wanted to express, but the complexity to express them became a big bottleneck. Furthermore, when the doctor wanted to add or make any changes to the application, it had to go through a convoluted process, with limited traceability, to update XML documents, Drools rules, database schemas and other application-dependent logic.

22.3 The DSL-Based Solution

22.3.1 Intentional Domain Workbench

Intentional Software provides a knowledge processing platform to allow business professionals to turn their specialized expertise into software. The development environment, the Intentional Domain Workbench (IDW), is a language workbench for building DSL-oriented applications for business users. These applications can be run stand-alone, and can optionally also generate applications using various languages and runtimes (such as XML and Drools in this example).

The Intentional platform provides a number of key technologies that make the DSLs especially suited for business users. In particular, this includes a projectional editor that allows languages to be edited in multiple syntactical forms, and with multiple semantic interpretations. It can use and mix textual, tabular and graphical notations to approximate the needs of a business domain as closely as possible. The projections of a language can potentially be ambiguous, but that does not cause a problem, because they are just projections of an underlying consistent representation, and a user can always switch to another projection to resolve any ambiguity. The platform also allows for combination and interaction across languages. A single projection can integrate knowledge represented in multiple disparate languages.

22.3.2 Overview of the Solution

The purpose of the custom language workbench application examined in this case study is to let business experts edit questionnaire definitions that are used as input to a web application that in turn allows end users to fill out their answers. Fig. 22.1 shows an example definition of a questionnaire.

---

As we will see, it also supports Word document-like headings, and more generally, looks a bit like Microsoft Office. This also helps acceptance with business users.
In addition to defining the questions, the medical professional can also define business rules that should be applied to the questionnaires, as well as tests to ensure that the business rules are working correctly. Fig. 22.2 shows an example of such rules; we will get back to testing later.

### 22.3.3 Implementation

To implement this, we have used IDW to define a set of domain schemas\(^7\) along with logic for validation, transformations, evaluation, code generation and projectional editors. All of these concerns are implemented with a custom language supported by IDW that extends C# with additional operators and keywords that are useful for working with tree structures\(^8\). The language also contains several dedicated DSLs for defining domain schemas, validators or projections\(^9\). The result of compiling the language definition is a custom workbench: a standalone Windows application that lets the business experts edit the defined domains in a projectional editor where all the rules for validation, projection layout and such are applied. Fig. 22.2 shows the editor for business rules with definition expressions, assessment tables, choice lists and results.

As its output the workbench in this case study generates files that are fed into a web application that executes the questionnaires and applies the business rules\(^10\). The web application itself is developed separately and consists of web pages with JavaScript that consumes the XML files generated by the workbench. The JavaScript then uses these XML files to produce a dynamic user interface\(^11\). The workbench also generates business rule files in a format that the Drools business rule engine can consume, and the web application can in turn call the Drools engine to access the running rules.

---

\(^7\) A schema is the structure definition of a domain. It roughly corresponds to the abstract syntax or meta model, even though the meta meta model of the IDW is substantially different from EMF or MPS’ structure definition.

\(^8\) This approach is similar to MPS: MPS’ BaseLanguage extends Java with constructs useful for working on the meta level of languages.

\(^9\) This is similar to MPS and Spoofax, which also come with a set of DSLs specific to various aspects of language definition.

\(^10\) The patient interacts with this web application in addition to the sensors mentioned earlier.

\(^11\) The web application acts as an interpreter for an intermediate language whose serialization format is XML.
Domain Structure  The IDW is very suitable for modularizing, reusing and composing languages ("domains" in the terminology of Intentional Software). Consequently, the application consists of several domains, some of them specific to the application discussed here, others more general\(^\text{12}\).

We use two domains that are motivated by the underlying technology: to generate the XML, we employ a reusable XHTML domain that comes with IDW. To generate the Drools rules, we have created a Drools domain (which may be reused for other applications in the future).

Similarly, the domains that are closer to the business domain are also modularized. The medical professionals in this case study have a particular subject they want to create questionnaires about, but the questionnaire domain itself is general

\(^{12}\text{Some are generalized because future reuse is anticipated, others are existing languages reused in this application.}\)
and has high potential for reuse. The business rules are also
gen¬eral enough to be reused on their own, independent of the
questionnaires. This results in two main domains: the ques¬
tionnaire domain and the business rule domain. These are in
turn divided into subdomains to allow selection of features to
reuse. We then complement this with an adapter domain that
includes the reusable questionnaire and business rule domains,
and define how they should work together. Finally, we have an
overarching domain for the application that we call Intentional
Health Workbench (IHW), which adapts the combined ques¬
tionnaire and business rule domains to the particular customer
requirements. In total we end up with ten domains (Fig. 22.3
shows an overview of the relationships between them):

FitBase: The generic questionnaire domain\(^\text{13}\). Contains abstrac¬
tions such as interviews, questions and answers.

FitRunner: In-workbench execution of the generic questionnaire
domain FitBase, allowing the business expert editing the
questionnaires to experiment with filling out answers inside
the workbench.

FitSimple: A simplification of the generic questionnaire domain
FitBase to a subset suitable for combination with the busi¬
ness rules domain and intuitive editing.

RulesEngine: The generic business rule domain, with table-style
ing and in-workbench evaluation of business rules.

RulesChecking: Consistency validation of the rules in the generic
business rule domain RulesEngine.

RulesCompiler: Generates the business rules from RulesEngine
to files that the Drools business rule engine can use.

FitSimpleWithRules: Combines the simplified subset of the ques¬
tionnaire domain FitSimple with the generic business rule
domain RulesEngine.

Drools: Provides abstractions from the Drools business rules
ing domain. Supports generation to the Drools file for¬
mat.

XHTML: Provides abstractions from the XML and HTML do¬
mains. Supports generation of XHTML and XML files.

IHW: The workbench that ties all the other domains together.
When compiled, this results in the workbench application
that lets business users edit questionnaires and business rules,
test them and generate output for the web application and
the Drools business rule engine.

![Diagram](image)

**Figure 22.3** Dependencies between the
ten domains that make up the system
described in this chapter. The arrows
represent an includes relationship.
Like the extends relationship in MPS,
includes is generic in the sense that it
may be an actual include in terms of
language concepts, or it represents a
generic dependency. An example is the
RulesCompiler. Its relationship with
the Drools domain captures the fact
that it generates Drools rules.

### Defining a Domain

The schema for each language is de-
defined using a DSL for schema definition. Because no parser
is involved, we only have to define the data structure of the
tree that the user will edit. IDW provides a default projection
for all domains until you create custom projections, so you can
start editing and experimenting with your structures inside the
text editor as soon as you have defined them.

Defining a schema for a domain is all about deciding what
types of nodes there may be in the tree structure and what
types of child nodes to expect under them. To define the tree
structure schema for a domain, we use the keywords `domaindef`,
`def` and `fielddef`. A `domaindef` is used for defining a new do-
main, `def` defines a new type of node that can be used in the
domain and `fielddef` defines a field under a `def` where new
child nodes can be added.

While `defs` and `fielddefs` are similar to `EClasses` and `EFea-
tures` in EMF (and consequently also quite similar to MPS’
structure definition), there are a few differences. For ex-
ample, a `fielddef` can be assigned more than one type. In EMF,
accepting a range of types in a field would require the cre-
ation of a supertype that the field would use as its type. A
`fielddef` will take a list of types that are all considered ac-
ceptable. If the same list of types is used in several places, we
can package them in a reusable way using the `typedef` key-
word. We can also reuse field definitions in multiple `defs`
with the `includedef` keyword, potentially overriding (lim-
itig, extending) their type.

As we are working with tree structures, the default relation-
ship between a node and its child node under a field is contain-

---

14 This is a very useful feature, because it allows the incremental definition
of concrete syntax. Also, if a concrete syntax definition is broken (in the sense
that it has a bug) or is ambiguous, the program tree can always be unambigu-
ously rendered in this tree view like default notation (even if that notation is
not necessarily as elegant as a custom projection).

15 It is essentially what’s called a language concept in this book.

16 In all the other tools described in
this book, fields are always owned by
a language concepts. In IDW they can
stand on their own and can be included
in `defs`. This provides an additional
level of modularization and reuse.
This design also allows associating
additional specifications with a field,
such as constraints or projections. These
can be reused along with the field.
Figure 22.4: A domaindef defines a new domain/language. It can include other domains, making their contents available to the domain. A def defines a new language concept. defs contain fields. New fields are defined with fielddef, existing fields are included via includefield. Each field defines a shape (list, set, single element) and a type (one or more other defs). When including a field, these can be overridden. The main field is reused by many defs and has special editing support. virtualdefs use a match pattern to select existing nodes. Based on this virtualdef, projection rules or other aspects can be defined.

ment. The Question def, for example, has an answer fielddef with the Answer def as its type. Just using a def directly implies containment. Let us now look at references.

The Category def reuses the main field (a commonly reused fielddef that comes with IDW) and overrides its type; it expects those types listed in the QuestionHierarchy typedef. When we look for the definitions of the types in that list we discover that two of them are not defs, but use the virtualdef keyword. A virtualdef can use an arbitrary match pattern (not just ref). This allows a new virtual def to be assigned to any node that matches the match clause. You can then define projections or constraints for this new virtualdef. They will ap-
apply to all nodes that match the match clause in the virtualdef. In this case DvRefQuestion defines a type for references to Question nodes, and DvRefCategory defines a type for references to Category nodes, allowing questions and categories to be reused in multiple places.

**Constraints and Behavior** Defining the basic shape of the tree that users should edit will usually be done in a declarative fashion using the schema DSL. However, in many cases, additional constraints are required. These are typically implemented in validators. Validators can enforce scoping rules (that the referenced variable is available in the scope), scan for illegal names or naming collisions, and ensure that any number of domain-specific business rules for the DSL are adhered to by users when they edit their DSL code. Fig. 22.5 shows a simple validator that ensures that the length of a name does not exceed a specified maximum.

![Validator Example](image)

Here are some more examples of constraints: "categories should not be nested more than five levels deep", "questions may not be modified once they are published" or "negative answer options should be displayed in red".

The first constraint, about level nesting, could be implemented using the DSL for writing validators that comes with IDW. A code snippet showing how such a validator could be implemented is shown below.

```validator implfor: Fit
{
  def isas_must_have_names_no_more_than_length_characters_long cat: Message
  {
    fielddef isa type: ref();
    fielddef length type: int;
  }
  validatesnode : sequence(Book) procedure Vnode(var Dmx dmx, var Node node)
  {
    code:
      assert node.StName() == null || node.StName().length < 63 else
    }
}
```

![Validator Code Snippet](image)

Figure 22.5: This constraint checks for the maximum length of all kinds of names. Notice how the error message itself is represented as a def: it is of category Message.cat represents structural subtyping, in the sense that all the fields of Message are also added to Category.

Validator run over the tree structure as it is edited and assert conditions that go beyond what is reasonable to define in the declarative schema language, such as a recursive check to determine nesting levels. When a validator fails, an error message is produced, which is shown together with any error messages that the system generates if the user breaks the schema constraints defined in the schema DSL.
The second constraint, about preventing modification to published questions, could be implemented using the DSL for behaviors. Behaviors are a bit like database triggers, in that they contain code that is triggered to run on events that signal changes to the tree structure, such as when nodes are inserted, modified or deleted. In this case we could use behaviors to associate code that should be run on modify and delete events for Question instances. The code would check if the question has been published and if so, prevent the modification or delete operation from executing. The following code snippet shows a possible implementation:

```csharp
def behavior implfor: Question {
    overproc: Execres procedure CanEdit() {
        if (rh->published) {
            return Error("May not modify published question!");
        }
        return Success();
    }
}
```

The third constraint, about showing negative answer options in red, could be implemented in the presentation for the Answer-Option nodes using IDW’s DSL for defining. The code responsible for showing the AnswerOption node on screen would simply use the C# `if` statement to check whether the option is negative (such as the No answer option to a Yes/No question) and if so, present the node on screen using a red color. We will see examples of code using conditionals in a projection later on.

While we would use three different DSLs to define the three constraints described above, we would also mix that DSL code with standard C# code. The DSLs that come with IDW extend C#, so in addition to the DSL keywords for defining schemas, validators, behaviors and projections, it is also possible to write standard C# classes, and even to mix C# code into declarative DSLs, such as the projection DSL. Some DSLs, such as the validator and behavior DSLs, expect to be implemented using C#
and have no declarative way to be implemented. The projection for C# code uses a textual notation that looks basically like standard C#, but because it is a tree projection, albeit one that looks like text, there are a few differences from what the same code would look like in a text editor. Consider the following example code:

```csharp
program RulesChecking using: [RulesEngine, FitBase, FitSimple, Validation, Gen, DL, Core, mscorlib, System] {
    region RulesTesting {
        [SerializableAttribute()]
        public class DcsUnique : Dcs {
            static var int counter = 0;
            var int index = 0;

            public constructor DcsUnique() {
                this.index = counter++;
            }

            public override bool procedure Equals( var Object obj ) {
                var DcsUnique that = obj as DcsUnique;
                return that != null && Equals(this.index, that.index);
            }

            public override int procedure GetHashCode() {
                return this.index;
            }

            public Kselection property kselection {
                get {
                    return Crown;
                }
            }

            public Kend property kend {
                get {
                    return Nil;
                }
            }
        }
    }
}
```

In contrast to C#, there is the `procedure` keyword. This is shown in the projection simply to give the user something to click on if they want to select the whole procedure (or `method` as they are more commonly referred to in C#)\(^\text{19}\). Clicking on the `public` keyword lets the user change that keyword to for example `private`, and lets the user enter additional modifiers such as `static`. Clicking on the name lets the user change the name. But if the user wants to delete the whole method, they just click on the `procedure` keyword to select the whole method and hit the `Delete` key. In the tree structure, the name, the modifiers and the whole method body are child nodes contained by the `procedure` node, so deleting that node will delete all the contained child nodes as well. The `constructor` keyword is there for the same reason – something to click on to select the whole thing – as is the `var` keyword in the field definitions. When

\(^\text{19}\) In IDW, the `procedure` keyword would be known as the `crown` of a subtree.
generated to C# source code for compilation, these additional keywords are not included in the output.

Another use case for validators is to verify the types in expressions edited by users. Depending on the DSL, the expression \texttt{1 + True} may or may not be illegal, but many languages would prevent the addition of a Boolean value to an integer. IDW includes a DSL for defining the rules for the type calculus in a mix of declarative and C# code, and uses recursive evaluation to determine the resulting type from an expression. The validator will then call the recursive IDW type calculator, and if a problem is discovered an appropriate error shows up in the error pane. In this customer case the workbench has a lot of expressions in the business rules and they are all validated for type consistency.

Projection The ability to write C# is not only useful when writing utility classes; several of the DSLs included with IDW support the ability to mix C# code into the DSL code. The projections are one example, where some projections are written in an entirely declarative manner using just the keywords from the projection DSL, while others make use of mixed in C# to produce dynamic behaviors. Before looking at examples of such mixed code we will examine a couple of purely declarative projections first.

Each \texttt{def}\textsuperscript{20} (\texttt{Category}, \texttt{Question}, or \texttt{Answer}) comes with its own projection rules. The projection of the overall tree is then a composition of the projections of all involved nodes. The projection for each type is defined in a declarative fashion, where a template is specified that defines how nodes of that type should be presented to the user (Fig. 22.6). The parts with gray background in Fig. 22.6 constitute the template, whereas the parts with white background are either references to fields that should be projected in place or full blocks of imperative code.

Projection works by mapping domain \texttt{defs} to concepts from the Abstract Projection Language (whose concepts all have names beginning with \texttt{A} to make them easily identifiable). These concepts are then transformed further, until, at least conceptually, we arrive at the level of pixels on the screen\textsuperscript{21}. Some of the \texttt{A} constructs are quite primitive, such as \texttt{AVert}, which only specifies that its contents should be displayed as a vertical list, or \texttt{ASeq}, which specifies that the contents should be presented in

\textsuperscript{20}Projections can also be defined for \texttt{virtualdefs}. This allows nodes in a specific context to be projected differently.

\textsuperscript{21}As a language developer, you don’t have to care about anything below the \texttt{A} level. You just map your domain concepts to concepts from the \texttt{A} language.
a sequence – horizontal or vertical is up to the presentation engine and depends on available screen estate. Others are more high-level, such as `AChapter`, which presents its contents in the form of a word processor-style chapter (thick text, optional chapter numbering and indentation, etc). To project something as a graph, we just have to use the `AGraph`, `AGraphNode` and `AGraphEdge` constructs. To project something as a table, we use `ATable`, `ARow` and `ACell`. `AImage` displays a bitmap image. `AButton` and `AHyperLink` make it possible to add buttons and links to the projections that execute C# code or move focus to a different place in the projection when clicked, providing an alternative to having the user type everything in with the keyboard\(^{22}\).

Each A Language construct has a number of fields where values can be entered in the template. Sometimes this will be literal information that should be displayed, such as the

---

\(^{22}\) Of course developers can define new, higher level projection concepts that are in turn transformed to A concepts. It is also possible to define new concepts on the abstraction level of A and then project them manually to the next finer level.
string literals "Question:" and "answer:" in the projection for the `Question` def. Other literals control the behavior of the projection, such as the `True` value under the `Indent_Chapters` field in the `AChapter` projection for the `Category` def. To make the child nodes of the projected type show up in the projection, we just put a reference to the relevant fielddefs in the appropriate places in the projection definition.

Templates are a good fit for a declarative DSL, because projections can often be defined in an entirely declarative way. When there is demand for dynamic behavior, the declarative context of the template can be broken out from using the `BackQuote()` function: standard C# can be entered inside it. The C# code should end by returning a new "piece of tree" that is inserted into the hosting template in the place of the `BackQuote`. A new piece of tree can be created with the `BookQuote()` function inside of which declarative structures can be created.

There are many cases in which dynamic behaviors in projections are useful. Common examples include changing the color depending on the displayed values, showing or hiding some of the values depending on editing modes or access rights, and even displaying the results from dynamic evaluation of expressions and values that the end users type in.

### Dynamic Schemas

Another case is when the DSLs that end users edit influence each other dynamically, such as when one DSL is the schema language for a second DSL. Consider for example an Entities DSL in which users can define entity types with attributes. A second DSL allows users to define instances of the entities, specifying values for the attributes. The schema language allows this by letting us hook in C# code to dynamically determine the fields that the schema should consider under a `def` or a `virtualdef`.

Let us look at an example. When creating a program expressed in the second DSL, a user may want to create an instance of the `Person` entity defined with the first DSL. The `Person` entity in turn contains `firstName` and `lastName` attributes. The editor should then read the definition for the `Person` entity and go on to present two fields under the new instance, label them `firstName` and `lastName`, and let the user enter the names for their new `Person` instance. This works by hooking in code into the Instances DSL that returns `fielddef`s for each attribute under the entity referenced in the `type` field.

---

23 These references get a white background in the definition where the rest is gray.

24 A `Book` is a basically a subtree literal that can be inserted into another tree.

25 This can be used nicely for hooking interpreters that execute tests based on the data entered by the user.

26 Here we assume that each `Instance` in the instances DSL references the `Entity` it instantiates in a field called `type`. 
of the instance, and potentially from any supertypes of that entity. The IDW default projection would detect this and present **firstName** and **lastName** fields ready to be edited under a **Person** entity. In a custom projection dynamic code would be used to iterate over the appropriate fields and create projections for them.

In the case of the workbench in this case study we have a **Rule** def, which has one fielddef called **outcome** that is declaratively defined in the standard schema DSL; the rest of its fields are determined dynamically, as described above. In the projection we want to display each rule as a row in a table and each dynamic field under a rule as its own cell. The **outcome** field should also get its own cell, which is defined in the declarative way in the template, but for the dynamic fields we have to break out from the declarative template context and write some C# code. Fig. 22.7 shows the respective code.

![Diagram of code](image)

The schema of the projected node can be accessed with the expression `ehi.GetDmx().Rgdf()`, where `ehi` is the input node to the projection, `GetDmx()` retrieves domain context information about it and `Rgdf()` returns the fields that are expected under the node. Normally `Rgdf()` will only return the fields we have declared in the schema DSL, but in this case it has been overridden for the **Rule** def to return a set of fields that...
are determined dynamically by other input to the Rule DSL. The C# code in the projection definition for the Rule (shown in Fig. 22.7) iterates over the fields that should go under the Rule def according to the schema and our overriding code, then uses the BookQuote() function to create a piece of tree with an ACell in each one. A simple C# expression (ehi.Index + 1) is also used to display the row index in a leading cell for each row.

The ability to mix C# into projections opens up the possibility of creating very powerful dynamic projections including DSL evaluation, and even running of test cases for a DSL directly in the editor for that DSL. Projections can also be combined with transformations, such that the tree structure edited by the user undergoes a series of transformations before being projected onto the screen. These transformations are two-way, so the projections built including such a transformation continue to be fully editable. They work in a similar way to projections, in that they let the developer create templates in their target language (rather than the A language) declaratively, but with the option of breaking out into C# code. By moving calls to things like test evaluation into a transformation that precedes the projection, code with different types of responsibilities is separated by concern and kept simple and to the point.

A testing framework was created for the case study discussed in this chapter, so that business rules can be evaluated with test data and the results verified against expected values, all directly in the workbench. The tests are run continuously, so that whenever the user modifies the business rules, the tests go red or green as the rules break or correspond to expectations. Fig. 22.8 shows an example of such a test case.

The evaluation of the business rules is implemented as an interpreter that works by evaluating each node in the tree structure according to its type in a recursive fashion. The code for this is packaged in a helper class called by a transformation that passes the test inputs to the evaluation method and decorates the transformation’s output tree with the results. The projection then takes the decorated tree, presents the editable input values, expected values and calculated test results (not editable), and compares the test results with the expected values to show green, red or orange bullets as appropriate.

In this case study we see another interesting example of pro-

27 But taking care to avoid doing so for the outcome fielddef, which is already projected in the declarative part of the projection rule.

28 In fact, projections, transformations and code generation work essentially the same way in IDW. As we will see later, code generators in IDW are implemented as transformations between a source domain and a target domain. Projections are in turn just transformations that have the Abstract Projection Language as their target domain. There are two differences. First, projections are evaluated lazily, such that only the parts of the projection required for showing what is on screen will actually be executed, whereas for a code generator the whole transformation will always be executed at once. The second difference is that projections are automatically triggered as the program tree changes, whereas code generators are executed only on demand (e.g., by the user pressing a button in the Workbench.)
The testing aspect of the Health workbench lets users create test data and evaluate the business rules directly in the IDE. As mentioned earlier in this book, in-IDE-testing is an important ingredient to making a DSL accessible to non-programmers: in some sense, the tests make the DSL “come alive” and help build understanding about the semantics of the language.

The projection for a questionnaire calls out to C# code that performs consistency analysis on the combined business rules and questionnaire domains. The analysis ensures that when business rules are applied to a particular questionnaire, the rules do not refer to questions absent from that questionnaire.

The tests and consistency analysis are implemented and presented in a way specific to the application, but it is also possible to use the IDW validation framework to ensure validity of user inputs. The developer then writes validators that run in the background against the tree structure as it is being edited by the user. When a rule in a validator is broken, it yields an error message, which is shown in the IDW error pane, a central place for collecting custom error messages from validators and system error messages from built-in generic validators alike.

Transformation and Generation

Once the input is known to be consistent\(^\text{29}\), the time has come to do something with the

\(^{29}\) For this application this means that the structure is correct, all validators are happy, all tests are green and consistency analysis is satisfied.
Figure 22.9: The consistency analysis ensures that when a set of business rules is used together with a particular questionnaire, all questions evaluated by the rules are actually present in the selected questionnaire.

As we discussed in Part II, this allows us to reuse parts of the overall transformation. For example, the Drools and XML domains are likely to be reusable in other applications.

Table 22.2.10 shows the transformation to the Drools domain. Again, parts with a gray background are declarative, whereas

information the medical professional has provided. In this case study this means invoking code generation to produce the XML files that the JavaScript in the web application will consume, and the files with business rule definitions for the Drools engine.

IDW includes a DSL for defining how to create output folders and files, and, together with the DSL for transformations, it constitutes how code generators are defined. While it is possible to take the tree that the user has edited and generate raw text files in the target format directly, it is often a better approach to use a transformation to create a tree structure in the domain of the target format from the tree structure that the user edited. Such transformations that result in information being generated to files rather than being presented to the user on screen do not have to be two-way, as there is no requirement for the information to stay editable.

The workbench in this case study uses a transformation that takes the questionnaire domain as input and outputs a tree structure in the XHTML domain that is included with IDW. The resulting XHTML tree is then passed on to a second transformation that knows how to transform such trees to text. The result of this transformation is finally passed to a file generator defined with the DSL for creating files and folders, with the result that the text is saved to files on disk. To generate the Drools files a similar transformation chain is executed, but with the difference that both the Drools domain and its transformation to text had to be developed for the project.

Fig. 22.10 shows the transformation to the Drools domain. Again, parts with a gray background are declarative, whereas
the white background signifies imperative code. We can see
the use of the `FIN()` function which determines if a given item
is in a list of items. We also see the use of `BookNew()`, which
creates a single node that can be inserted into a larger tree.

```java
transformation CompileToDroolsRules implfor: xidOutput: Rules include: [RulesLogic:]
{
  transformdef Assessment fOmitExtraTraits: [true;]
  {
    Section
      - names:
        - names:
        - rules
  }
  transformdef contains
  {
    Dot
      - left
      - contains
        - right
  }
  transformdef DvBoolBinaryOp
  {
    BackQuote{
      valOf
        { if (!FIN(enh.Isa, "Eq"))
          {
            return base.Display();
          }
          /*transform for eq*/
          var left = enh.main[0];
          var right = enh.main[1];
          /*check for left/right being any/unknown, else types match*/
          if (left.Isa == "(Any)" || right.Isa == "(Any)"
            {
              return BookNew("(Truc)");
            }
          else if (FIN(TypeForDrools(TypeOf(left)).Isa, "(ArrayList)"
            {
              return BookNew("(Dot), ProcessEnh(left),
                          BookNew("(contains), ProcessEnh(right))");
            }
          else if (FIN(TypeForDrools(TypeOf(left)).Isa, "(Double)" ||
                   (FIN(left.Isa, "Unknown") ||
                    FIN(right.Isa, "Unknown")))\n            {
              return BookQuote{
                Eq
              BackQuote(ProcessEnh(left))
              BookQuote(ProcessEnh(right))
            },
          else
            {
              return BookNew("(Dot), ProcessEnh(left),
                          BookNew("(equals), ProcessEnh(right));
            }
        }
    }
  }
```
22.4 Wrapping Up

With the code generators in place the whole workbench is ready for use by the medical professionals. They can now define interviews and rules that they can run and validate against tests directly in the workbench. When they are satisfied with their work, they hit the button to generate the XML and Drools rules for the web application, which can then be used immediately by end users. All the time the workbench guides them in their work with helpful validation messages, auto-completion and running tests, allowing for consistently high quality in the generated applications.

To implement the workbench we used several DSLs for schema definitions, projections, transformations and more in concert. The final product also combines several domains, with the two most prominent domains for interviews and for business rules split up into individually reusable subdomains. The projectional approach is well suited for such complex language composition and provides flexible notation, which makes it a powerful technology in scenarios that target business professionals. The ability to mix DSLs with GPLs such as C# ensures that each DSL can remain short and to the point without taking on all the burdens of a GPL as requirements grow in complexity.
Domain-Specific Languages are programming languages that are tailored to a particular application domain. By incorporating knowledge about that domain, DSLs can lead to more concise and more analyzable programs, improved code quality, tighter stakeholder integration and faster development speed. This book provides a thorough introduction to DSLs, relying on today’s state-of-the-art language workbenches, lots of examples and years of experience. The book has four parts, each focussing on a different aspect of DSL development.

1 Introduction: This part introduces DSLs in general and discusses their advantages and drawbacks. It defines important terms and concepts and introduces the case studies and tools used in most of the remainder of the book.

2 DSL Design: This part focusses on the design of DSLs – regardless of implementation technologies. It discusses seven design dimensions, explains a number of reusable language paradigms and points out a number of process-related issues.

3 DSL Implementation: This part provides details about the implementation of DSLs, using lots of examples. It uses three state-of-the-art but quite different language workbenches: JetBrains MPS, Eclipse Xtext and TU Delft’s Spoofax

4 DSLs in Software Engineering: This part discusses the use of DSLs for requirements engineering, architecture, implementation and product line engineering, as well as their roles as a developer utility and for implementing business logic.

The author, Markus Voelter works as a researcher, consultant and coach for software architecture, model-driven software development, domain-specific languages and product line engineering. He has worked in enterprise, web and embedded systems. He lives in Stuttgart, Germany and spends his free time flying gliders and publishing podcasts.

http://www.voelter.de
voelter@acm.org
@markusvoelter

dslbook.org