*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 workbench1, and their focus has been on business

professionals and less on programmers as users for the DSLs[[1]](#footnote-1). 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.

*Generative Programming: Methods,*

*Techniques and Applications*. Addison-

Wesley, 1999

### 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 application3. The do-

|  |  |  |  |
| --- | --- | --- | --- |
| main 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 values4. 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: | | | . |
| 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:

|  |
| --- |
| **if** the patient  has a Weight Height ratio of less than 46 **and**  a cholesterol LDL level below 100 **and** does not take LDL medications **and**  the systolic blood pressure level is less than 125 **and** does not take blood pressure medication  **and**  the hemoglobin A1c test is equal or greater than 6.5 **and** does not take glucose medication  **then** <advice according to diabetes plan>. |
| 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 doubled5. 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

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 possible6. 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.

.

|  |  |
| --- | --- |
| *22.3.3 Implementation*  To implement this, we have used IDW to define a set of domain schemas7 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 key- |  |
| words that are useful for working with tree structures8. The |  |
| language also contains several dedicated DSLs for defining domain schemas, validators or projections9. The result of com- |  |
| piling 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 ques- |  |
| tionnaires and applies the business rules10. The web applica- |  |
| tion 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 pro- |  |
| duce a dynamic user interface11. 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. |  |

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.

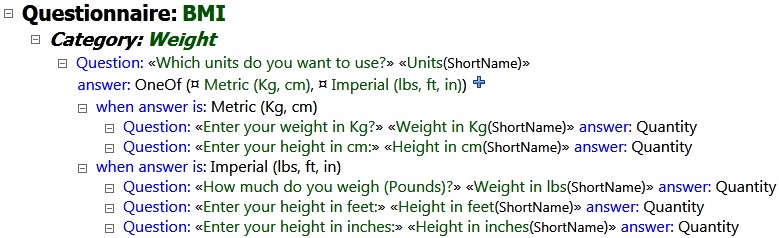
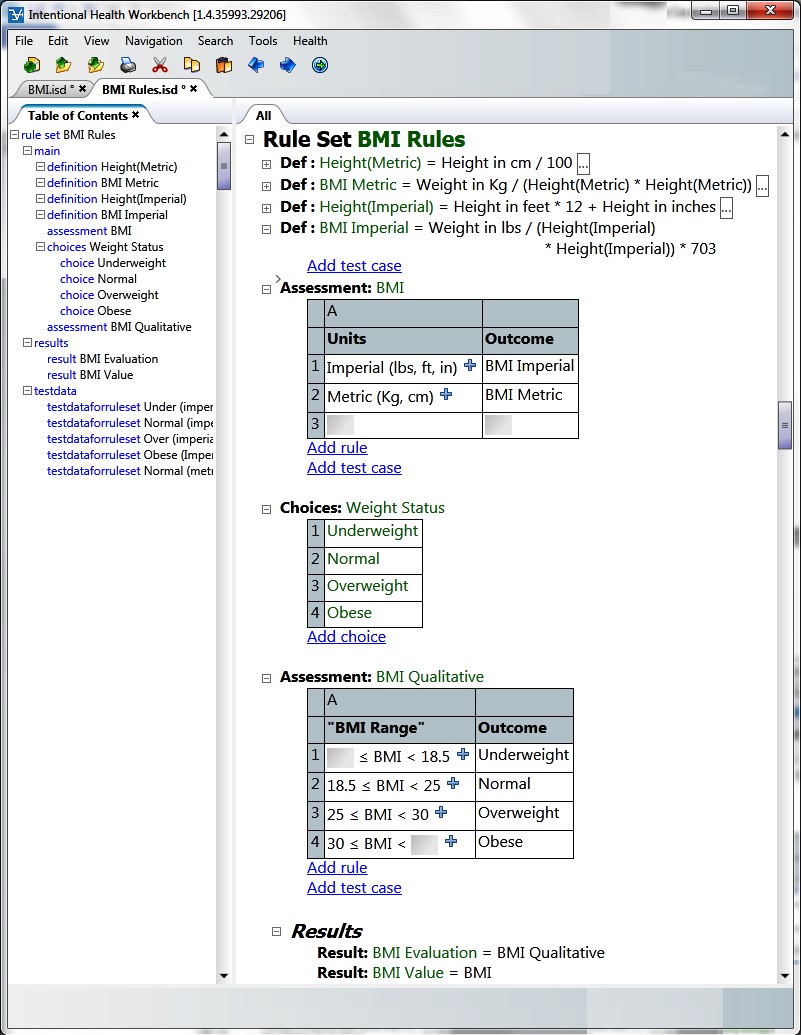


Figure 22.1: An example questionnaire, as seen and edited by the medical professional. Questionnaires are essentially trees of questions with the possible answers, as well as dependencies between questions (**when answer is...**).

Figure 22.2: This screenshot shows how the medical professional sees and edits business rules for questionnaire questions. In this example, the body mass index is calculated.

|  |  |
| --- | --- |
| *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 appli- |  |
| cation discussed here, others more general12. |  |
| 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 |  |

and has high potential for reuse. The business rules are also general enough to be reused on their own, independent of the questionnaires. This results in two main domains: the questionnaire 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 questionnaire 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 domain13. 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 business rules domain and intuitive editing.

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

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

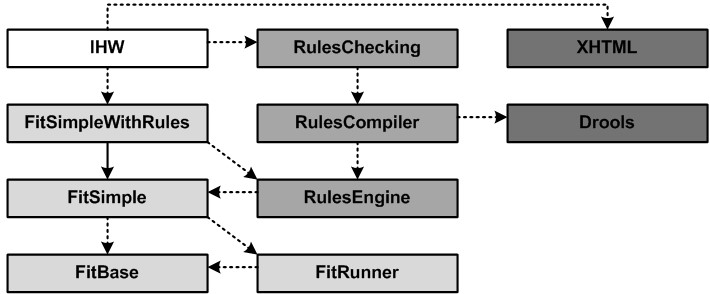
*RulesCompiler:* Generates the business rules from RulesEngine to files that the Drools business rule engine can use.

*FitSimpleWithRules:* Combines the simplified subset of the questionnaire domain FitSimple with the generic business rule domain RulesEngine.

*Drools:* Provides abstractions from the Drools business rules engine domain. Supports generation to the Drools file format.

*XHTML:* Provides abstractions from the XML and HTML domains. 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.

Figure 22.3: Dependencies between the ten domains that make up the system described in this chapter. The arrows represent an *includes* relationship.

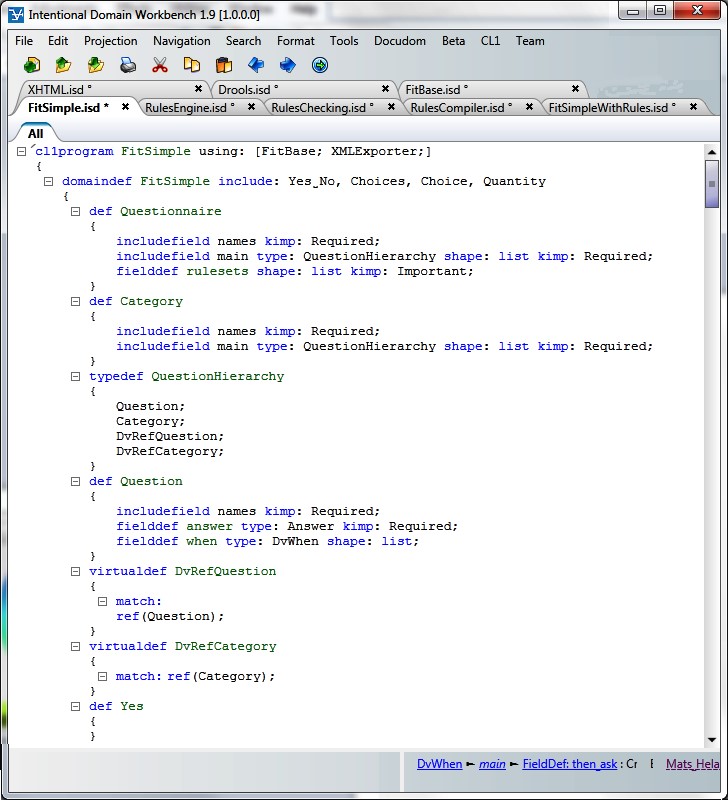
Like the *extends* relationship in MPS, *includes* is generic in the sense that it may be an actual include in terms of language concepts, or it represents a generic dependency. An example is the **RulesCompiler**. Its relationship with the **Drools** domain captures the fact that it generates Drools rules.

|  |  |
| --- | --- |
| *Defining a Domain* The schema for each language is 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 editor as soon as you have defined them14. |  |
| 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 domain, **def** defines a new type of node that can be used in the |  |
| domain15 and **fielddef** defines a field under a **def** where new |  |
| child nodes can be added.  While **def**s and **fielddef**s are similar to **EClass**es and **EFeature**s in EMF (and consequently also quite similar to MPS’ structure definition), there are a few differences. For example, 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 acceptable. If the same list of types is used in several places, we can package them in a reusable way using the **typedef** keyword. We can also reuse field definitions in multiple **def**s with the **includefield** keyword, potentially overriding (limiting, extending) their type16. |  |

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

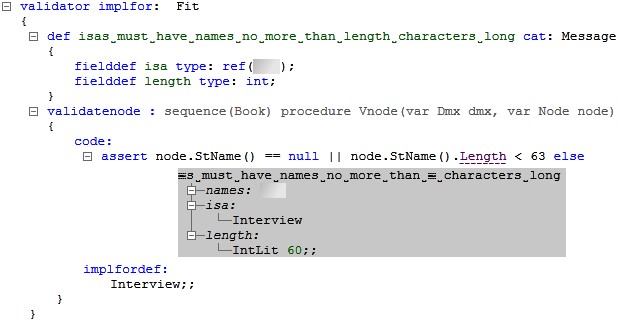
ment. The **Question** def, for example, has an **answer** fielddef with the **Answer** def as its type. Just using a **def** directly implies containment. Let us now look at references.

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-



ply 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.



|  |
| --- |
| 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[[2]](#footnote-2). A code snippet showing how such a validator could be |

implemented is shown below.

**validator implfor**: FitSimpleWithRulesets {

**def** Category too deeply nested **cat**: Message { }

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 implementation18.

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

|  |
| --- |
| validatenode : **sequence**(Book) **procedure** Vnode(**var** Dmx dmx, **var** Node node) { **code**:  **var** level = 0; **var** parent = node.Parent; **while** (parent != **null**) { **if** (parent.Isa == ‘(Category)) { level++;  } parent = parent.Parent;  } **assert** level < 5 **else** Category too deeply nested;  }  } |

The third constraint, about showing negative answer options in red, could be implemented in the presentation for the **AnswerOption** 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:

|  |
| --- |
| 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#)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 |  |

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 **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.

|  |  |
| --- | --- |
| 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 **def**s to concepts from the Abstract Projection Language (whose concepts all have names beginning with **A** to make them easily identifiable). These concepts are then transformed further, until, at least conceptually, | for **virtualdef**s. This allows nodes in a specific context to be projected differently. |
| we arrive at the level of pixels on the screen21. Some of the **A** |  |

Each **def**20 (**Category**, **Question**, or **Answer**) comes with its

constructs are quite primitive, such as **AVert**, which only specifies that its contents should be displayed as a vertical list, or **ASeq**, which specifies that the contents should be presented in



|  |  |
| --- | --- |
| 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 keyboard22. |  |

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

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 definition23.

|  |  |
| --- | --- |
| 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()**24, |  |
| 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 expres- |  |
| sions and values that the end users type in25. |  |

*Dynamic Schemas* Another case is when the DSLs that end

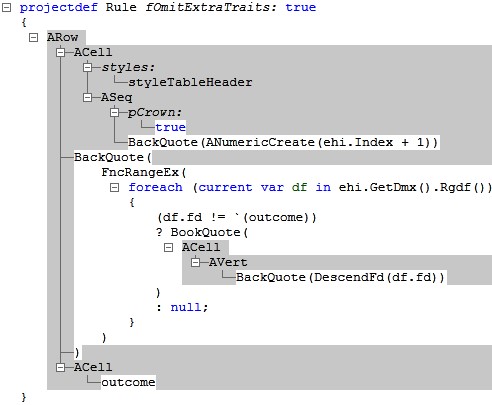
users edit influence each other dynamically, such as when one DSL is the schema language for a second DSL. Consider for example an Entities DSL in which users can define entity types with attributes. A second DSL allows users to define instances of the entities, specifying values for the attributes. 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[[3]](#footnote-3)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.

.



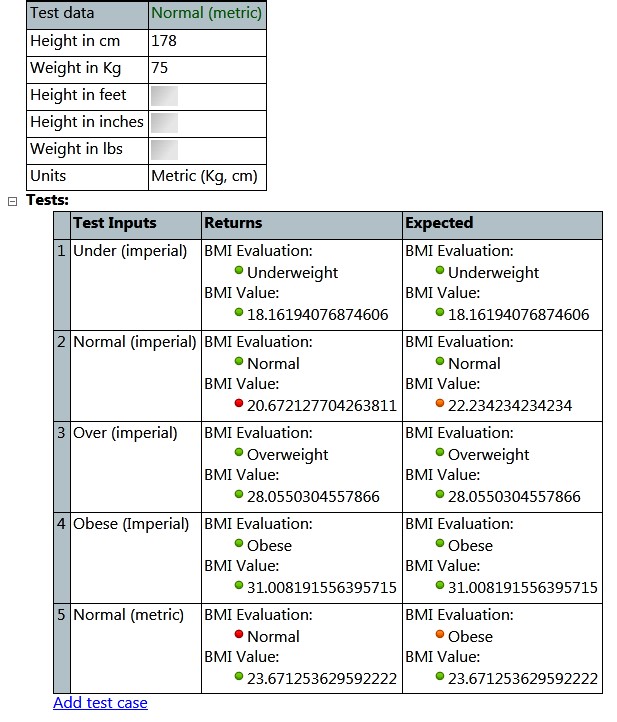
The schema of the projected node can be accessed with the expression **ehi.GetDmx().Rgdf()**, where **ehi** is the input node to the projection, **GetDmx()** retrieves domain context information about it and **Rgdf()** returns the fields that are expected under the node. Normally **Rgdf()** will only 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 one27. A simple C# expression (**ehi.Index + 1**)

|  |  |
| --- | --- |
| is also used to display the row index in a leading cell for each row. | projected in the declarative part of the projection rule. |
| 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 screen28. These transformations are twoway, so the projections built including such a transformation | 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 |

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-

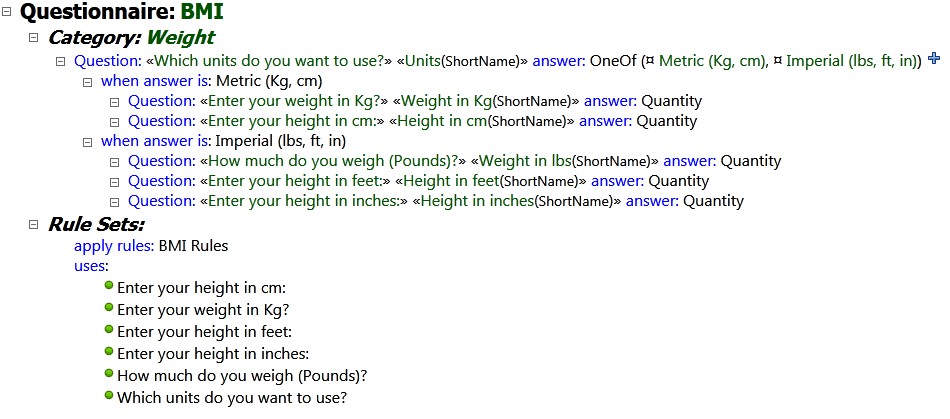


jecting dynamically derived information about the user input.

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[[4]](#footnote-4), the time has come to do something with the

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 edited30. 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.

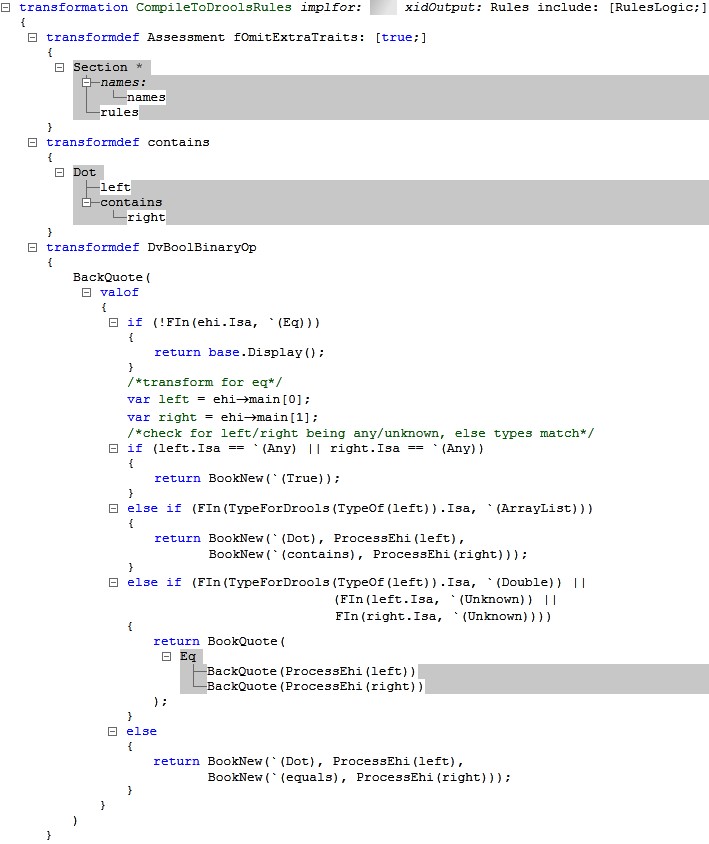
The workbench in this case study uses a transformation that takes the questionnaire domain as input and outputs a tree structure in the XHTML domain that is included with IDW. The resulting XHTML tree is then passed on to a second transformation that knows how to transform such trees to text. The result 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.

Figure 22.10: This is part of the transformation from the business rules domain into the Drools domain. The template-based approach is similar to projections.



### 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.

1. This has always been a focus of DSLs. However, as we will see in this chapter, focussing on non-programmers leads to different tradeoffs in the design of the languages and the tools. [↑](#footnote-ref-1)
2. h [↑](#footnote-ref-2)
3. [↑](#footnote-ref-3)
4. For this application this means that the structure is correct, all validators are happy, all tests are green and consistency analysis is satisfied. [↑](#footnote-ref-4)