# Case Study: Intentional Software in Healthcare

In this chapter we will investigate using Domain Specific Languages for business professionals. We will use technology from Intentional Software and use one of their projects in the healthcare domain as a case study. Intentional Software was one of the first companies to create a language workbench, but their focus has been on business professionals and less on programmers as users for the DSLs. 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.

## The project challenge

This case study describes an application where domain knowledge is captured and maintained directly by the domain experts using DSLs, validated at the domain level, and used for code generation to a final application. The domain is tele-health where patients with chronic diseases like diabetes, hypertension or obesity are staying at home, and are provided daily recommendations based on observed values on various daily measurements of the patient. A medical professional has defined what values to observe for what patient, and the rules for the daily individual recommendations based on those values. The input from the patient in the 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 scale to multiple health care providers that will have different set of criteria’s for different patients.

This application was first developed using a traditional approach where 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

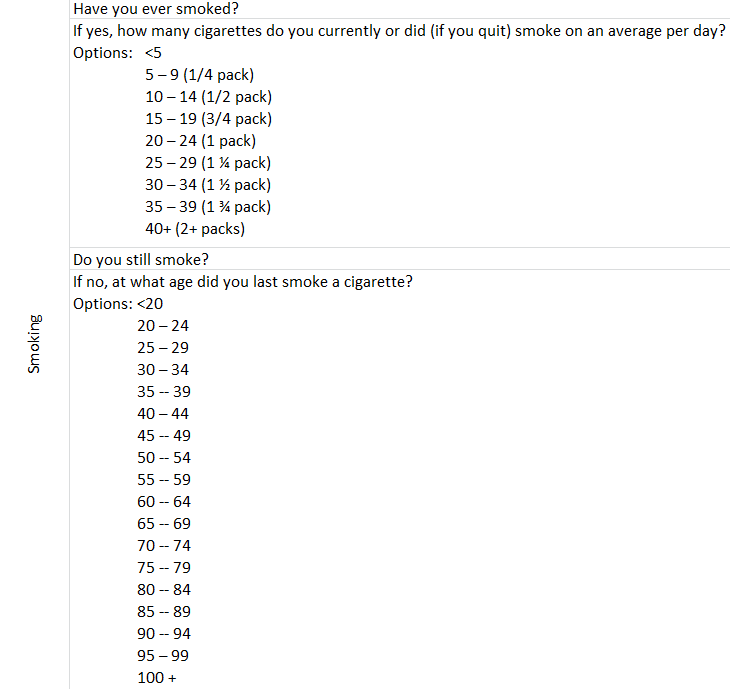
(a cholesterol LDL level below 100 and does not take LDL medications) and

(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. For each new observable attribute the number of rules doubled. Each rule was then encoded by a programmer into rules for a rules engine (Drools in this example). The patient data had a similar workflow where information for the patient recorded data was captured also in Excel sheets like the figure below. Once this information was confirmed with the medical doctor, XML schemas were created for this data to feed a custom patient application including the database that interpreted it.



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 it 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.

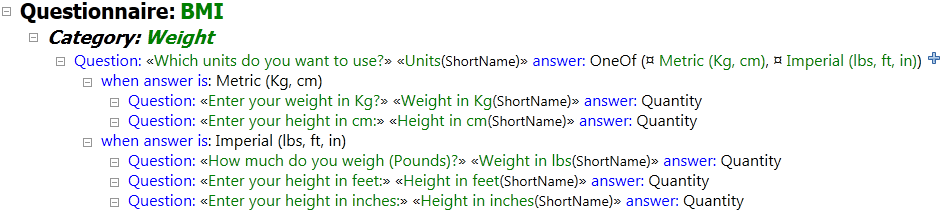
## Intentional Software

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

The Intentional platform provides a number of key technologies that make the DSLs especially suited for business users. As discussed before \todo{ref}, these include a projectional editor that allows multiple projections of languages to be edited in multiple syntactical formats, and with multiple semantic interpretations. 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 the combination and interaction across languages. A projection can integrate knowledge represented in multiple disparate languages.

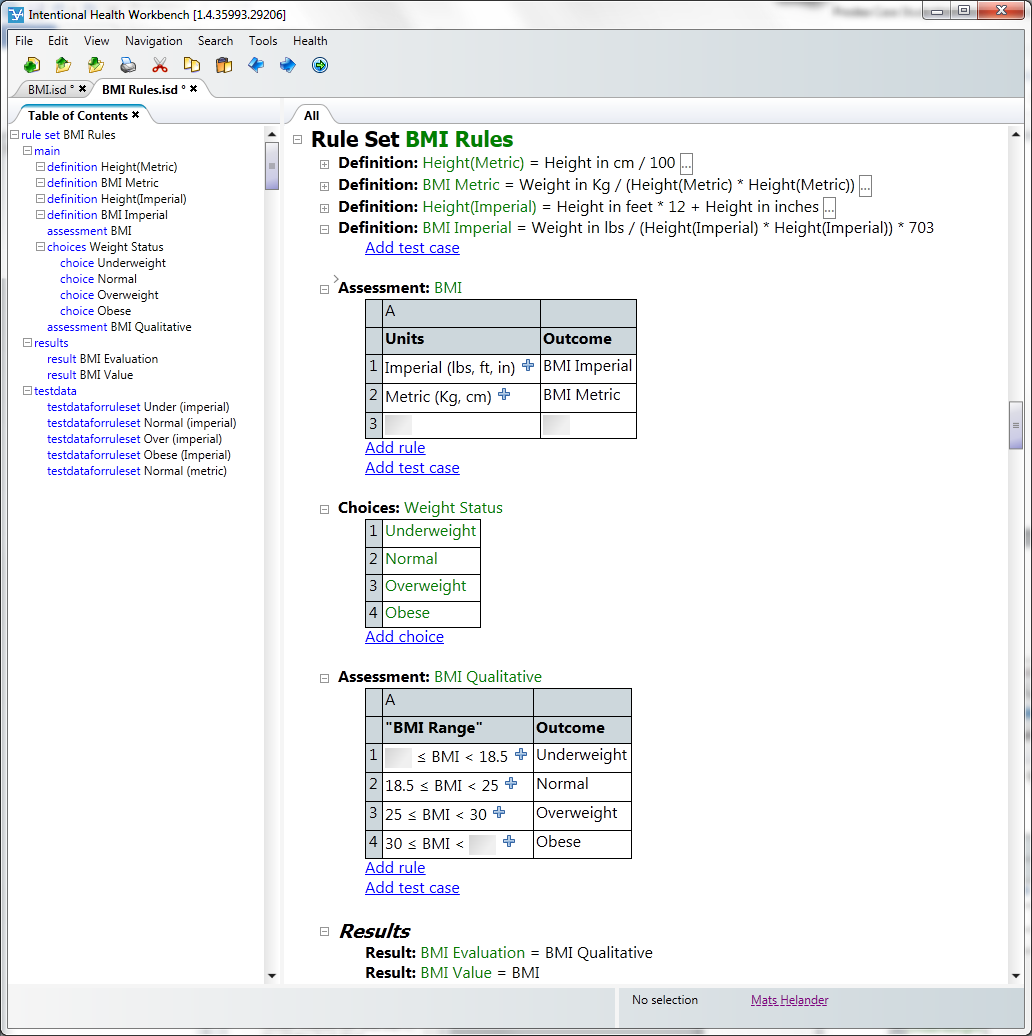
The Intentional Domain Workbench is a language workbench based on projectional editing that allows the developer to create projectional editors for custom languages with domain specific semantics and notation. Because of its structured editing approach where the user edits a tree-based data structure holding the semantic model rather than a text representation of the syntax, it is highly suitable for scenarios involving complex or composed languages (no parsing step) as well as situations where very flexible on screen representation of the language in the editor is required. The editable projections can contain a mix of text representation, tables, diagrams, graphics and symbols, all to make the notation as efficient and convenient as possible for the end user.

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 which in turn allows end users to fill out their answers.



In addition to defining the questions, business experts may also edit business rules that should be applied to the questionnaires as well as tests to ensure that the business rules are working correctly.

To implement this, we have used IDW to define a set of domain schemas along with logic for validation, transformations, evaluation, code generation and projectional editors. We do so using a custom language supported by IDW that extends C# with additional operators and keywords that are useful for working with tree structures. The language, called CL1, also contains several dedicated DSLs for defining domain schemas, validators, projections etc. The result when compiling the CL1 program 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.



The customer in this case study has 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, so we have used a modular language design with two main domains: the questionnaire domain and the business rule domain, that are in turn divided into subdomains to allow selection of what features to reuse. We then complement this with a “glue” domain that has an “include” relationship to the reusable questionnaire and business rule domains to 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 towards the particular customer requirements.

As its output the workbench in this case study generates files that result in a web application running the questionnaires and applying the business rules. The web application itself is developed separately and consists of web pages with JavaScript that consumes XML files describing the questionnaires and that are generated by the workbench. The JavaScript then uses these XML files to produce a dynamic user interface. 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 out to the DROOLS engine to access the running rules. To generate the XML we employ a reusable XHTML domain that comes with IDW and to generate the DROOLS rules we have created a new DROOLS domain.

In total we end up with ten domains:

FitBase: The generic questionnaire domain. Constains abstractions such as…

FitRunner: In-workbench execution of the generic questionnaire domain FirBase 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.

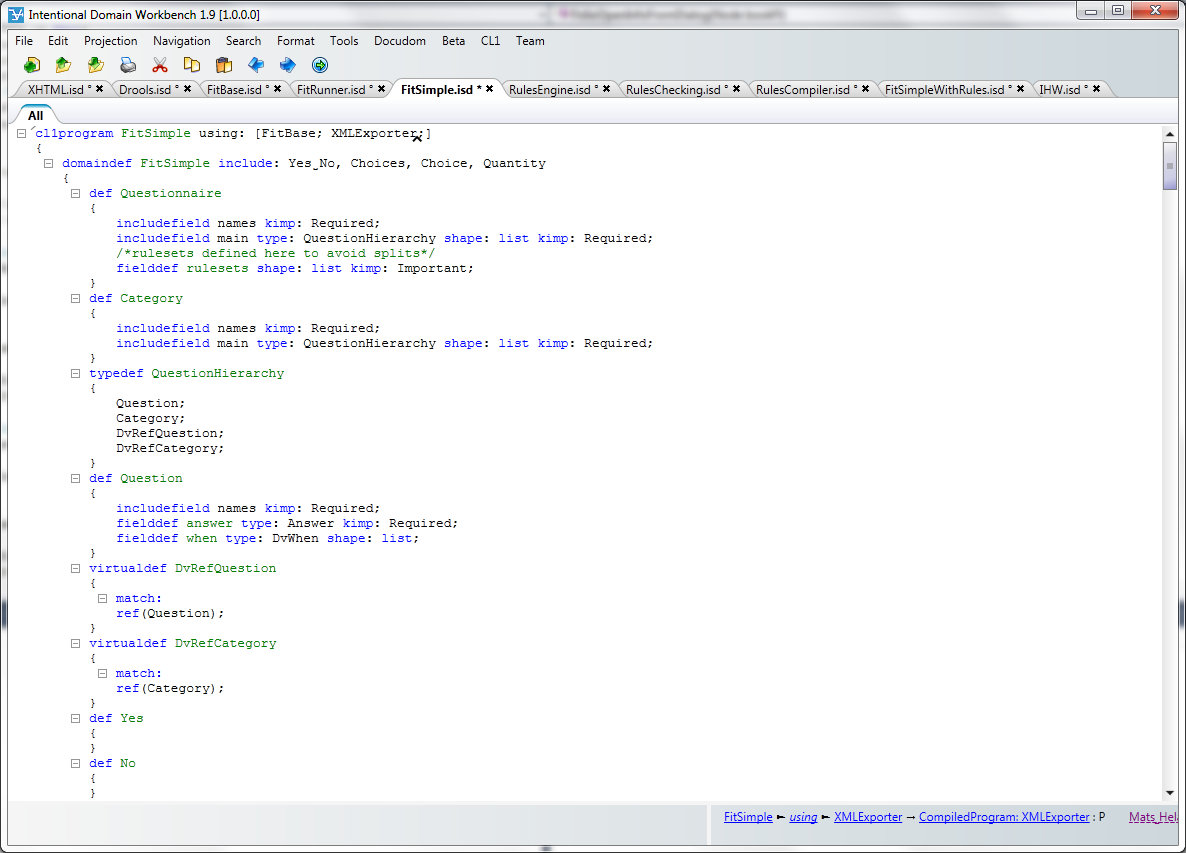
FitSimpleWithRules: Combines the simplified subset of the questionnaire domain FitSimple with the generic business rule domain RulesEngine.

DROOLS: Allows generation to the DRROLS business rules engine file format.

XHTML: Allows generation of XHTML and XML files.

IHW: The workbench that ties all the other domains together. Results when compiled 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.

The domains are all defined in the CL1 language. The schema for each language is defined using a DSL for schema definition that is included in CL1. 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 your trees until you create custom projections, so you can start editing and experimenting with your structures inside the 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 domain, 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.

In some ways def is similar to a class and fielddef is similar to a field in object oriented languages such as C# or Java, but there are also significant differences. For example, a fielddef can be assigned more than one type of node that it accepts. In Java or C# accepting a range of types in a field would require the creation of a super type that the field would use as its type. In CL1, a fielddef will simply accept 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 defs with the “includefield” keyword, potentially overriding what types they expect.

As we are working with tree structures, the default relationship between a node and its child node under a field is containment, such that if the parent node is deleted the child node is deleted as well. The “Question” def, for example, has an “answer” fielddef with the “Answer” def as its type. This means that when editing the tree and entering the “answer” field under a “Question” node the business user will be prompted to create a brand new “Answer” node. If the “Question” node is then deleted the “Answer” node below it is deleted as well. But IDW is also able to deal with references between any nodes in the tree structures.

The “Category” def reuses the “main” field (a commonly reused fielddef that comes with IDW) and overrides its type to expect those 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. This keyword allows us to create a special type for a reference to a particular other type (def or 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.

This means that when users enter the main field under a “Category” node they don’t have to type in a new “Question” node, they can just type in the name of an existing one and a reference to that node will be created under the field. If the “Category” node is deleted, the reference itself that is contained under its main field will be deleted, but the “Question” that was being referenced will be left unharmed.

Defining the basic shape of the tree that users should edit will usually be done in a declarative fashion using the schema DSL in CL1. However, sometimes more complex rules should apply, such as “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”.

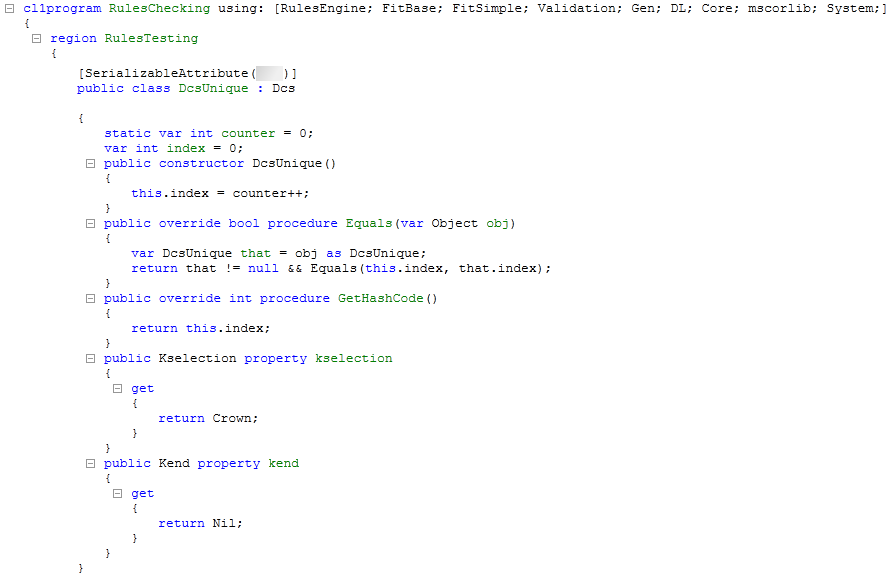
The first constraint about level nesting could be implemented using the DSL for writing “validators” in CL1. Validators run over the tree structure that the business users edit and assert conditions that go beyond what is reasonable to define in the declarative schema language, such as a recursive check to determine nesting levels. When a validator fails an error message is produced that 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” in CL1. CL1 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 the question nodes. The code would check if the question has been published and if so cancel the modification or delete operation from going through.

The third constraint about showing negative answer options in red could be implemented in the presentation for the answer option nodes using the DSL for defining projections in CL1. The code responsible for showing the answer option node on screen would simply use the C# if statement to check whether the option is negative (such as the “No” answer option on a “Yes/No” question) and if so present the node on screen using a red color.

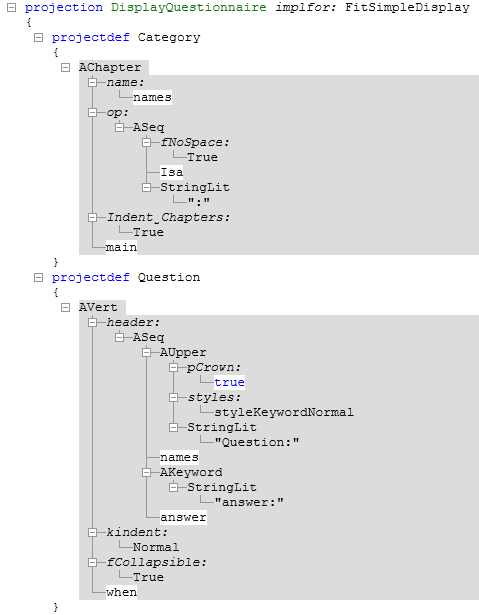
The three dynamic constraints could thus be implemented using the CL1 validation, behavior and projection DSLs, which work on top of the schema DSL. The basic shape of the tree would still be defined in a declarative way using the schema DSL. However, there are cases when it is useful to let the actual schema be influenced by dynamic behavior, such as when one DSL drives the schema of another DSL, an example of which will be shown later. For such scenarios the CL1 schema DSL does provide hooks for dynamically determining the schema for a domain.

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. CL1 is an extension to 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 in CL1, 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 inside a CL1 program uses a textual projection 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.



For example 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#). 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 the “procedure” keyword to select the whole method and hit the “Delete” button. 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.

The ability to write C# is not only useful when writing utility classes, several of the DSLs included with CL1 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 and 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.



Intentional Software’s projection engine is compositional, so each node type (“Category”, “Question”, “Answer” etc.) has its own instructions for how to be projected and then all the instructions are combined by the engine to show all the nodes of different types together on screen in an editable format. The projection for each type is defined in a declarative fashion where a template is specified for how to present nodes of that type to the user.

The projection DSL in CL1 comes with a set of constructs for how to display nodes in different ways. These constructs belong to the Abstract Projection Language DSL and all have names beginning with “A” to make them easily identifiable. Some of the “A” constructs are more primitive, such as “AVert” that only specifies that its contents should be displayed as a vertical list or “ASeq” that 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 keyboard.

Each “A” 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 enter a reference to the relevant fielddefs in the appropriate places in the projection definition. These references get a white background in the definition where the rest is gray.

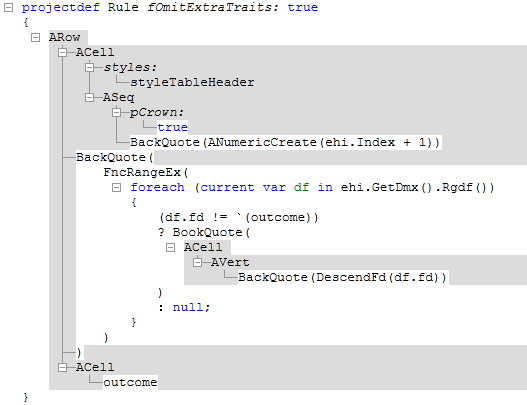
Templates are a good fit for a declarative DSL so projections can often be defined in an entirely declarative manner. When there is demand for dynamic behavior, the declarative context of the template can be broken out from using the “BackQuote()” function, where standard C# can be entered inside it. The C# code should end by returning a new piece of template that is inserted into the hosting template in the place of the BackQuote. A new piece of template can be created with the “BookQuote()” function, where declarative templates can be added inside it.

There are many cases where 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. 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 and a second “Instances” DSL where users can define instances of the entities with values for the attributes. When editing this second DSL, a user creates a new instance that they state should use the “Person” entity as its type. 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.

The CL1 schema language allows this by letting us hook in C# code to dynamically determine the fields that the schema should consider expected under a def or virtualdef. In the case described above, the code would return fielddefs for each attribute under the entity referenced in the type field of the instance and potentially from any super types of that entity. The IDW default projection would pick up on 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 all.

In the case of the customer workbench we have a “Rule” def that has one fielddef called “outcome” that is declaratively defined in the standard schema DSL and the rest of its fields are determined dynamically as described. 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.

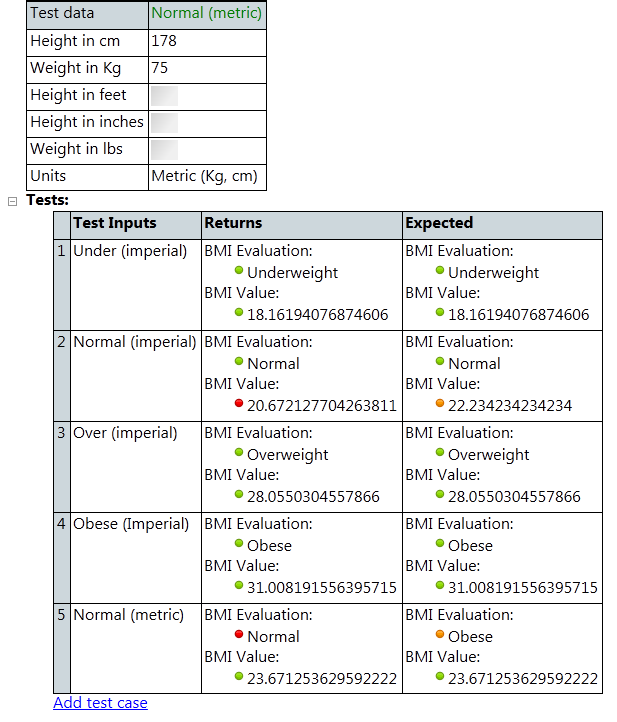


The schema of the projected node can be accessed with the expression “ehi.GetDmx().Rgdf()” where “ehi” is the input node to the projection, “GetDmx()” retrieves domain context information about it and “Rgdf()” returns the fields that are expected under the node. Normally Rgdf() will only return the fields 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 projectdef for the Rule def iterates over the fields that should go under the Rule def according to the schema and our overriding code and then uses the BookQuote() function to create a template piece with an ACell in each (but taking care to avoid doing so for the “outcome” fielddef which is already represented in the projection). A small C# expression is also used to display the row index in a leading cell for each row.

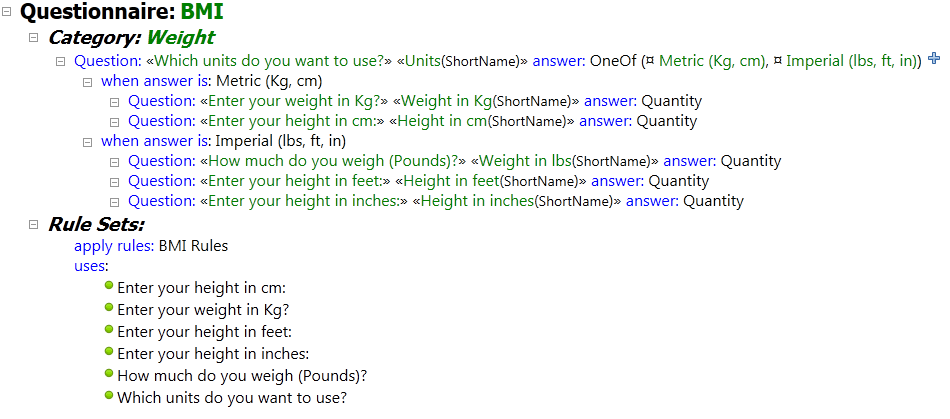
The ability to mix C# into projections opens up the possibility to create very powerful dynamic projections including DSL evaluation and even running of test cases for a DSL directly in the editor for that DSL. Projections can also be combined with transformations, such that the tree structure edited by the user undergoes a series of transformations before being projected on screen. Such transformations are two-way so 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 to break 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.

For the customer discussed here a testing framework was created so that business rules are evaluated with test data and the results are verified against expected values. The tests are ran continuously so that whenever the user modifies the business rules the tests go red or green as the rules break or correspond to expectations.



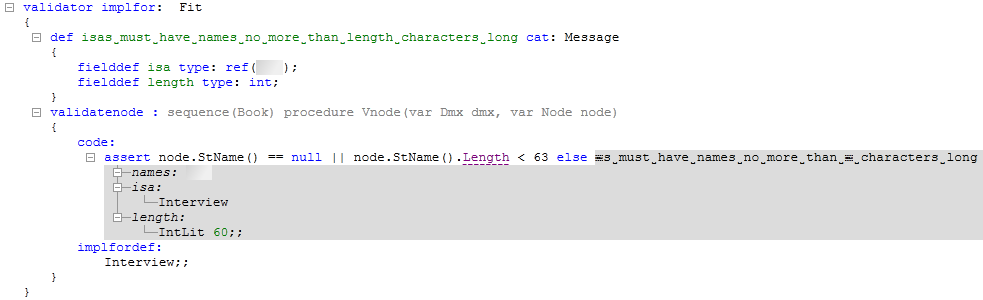
The evaluation of the business rules is implemented in C# that works by evaluating each node in the tree structure according to its type in a recursive fashion. This C# is packaged in a helper class called by a transformation that passes the test inputs to the evaluation method and decorates the transformation’s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

In this case study we see another interesting example of projecting dynamically derived information about the user input. The projection for a questionnaire calls out to C# code performing coverage analysis on the combined business rules and questionnaire domains. The validators ensure that when business rules are applied to questionnaires, the rules do not refer to questions absent from the questionnaire.



The tests and coverage 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, which is a central place for collecting custom error messages from validators and system error messages from built-in generic validators alike.

Validators are a good place for extending the semantic definition of your language beyond the pure schema definitions. 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 is adhered to by users when they edit their DSL code.



One common 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. CL1 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 out to the recursive IDW type calculator and if a problem is discovered an appropriate error shows up in the error pane. In this customer’s case the workbench has a lot of expressions in the business rules and they are all being validated for type consistency.

Once the input is known to be consistent – all validators are happy, all tests show green lights, coverage analysis is satisfied – the time has come to do something with the information the business user has provided. For this customer that 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.

CL1 includes a DSL for defining how to create output folders and files and together with the DSL for transformations constitute 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 that the information stays 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.

Some kind of summary? Use of the DSL? Customers happy? Any closing words?