# Domain-Specific Languages for Business Applications - Modelling User Interfaces

*Daniel Stieger, Wolfgang Messner, Oscar Rodriguez, Vaclav Pech*

Business application development covers various problem domains, like data persistence, business logic handling, and user interface design. A single DSL is not sufficient to achieve productivity and quality gains, nor are multiple DSLs that are not neatly integrated with each other. Only a holistic approach, where several DSLs can be used seamlessly to model solutions for the different problem domains, can bring significant improvements in development.

In this article, we will showcase a DSL we are using at modellwerkstatt.org to model standardized user interfaces for our business applications. The inherent logic of the DSL is capable of visually displaying whole entity-graphs in a very simple, declarative manner. We will point out how easy it is to interact and weave in plain Java code, an interaction that leads to more flexibility and safety, for example, type-safety. After distinguishing between internal and external DSLs, we will introduce JetBrains MPS and dive straight into our UI DSL. Finally, we will summarize our thoughts on DSL interaction and extension.

## Internal vs. External DSLs

Domain-Specific Languages (DSLs) are widespread in software development nowadays. They can be defined as “programming languages that raise the level of abstraction beyond programming by specifying the solution that directly uses concepts and rules from a specific problem domain.” (Kelly and Tolvanen, 2008). While GPLs (General Purpose Languages), like Java or Kotlin, are widely used across various problem domains, DSLs are tailored to be very productive in a specific domain. They provide suitable abstractions aligned with the problem domain to express solutions in a neat and succinct way, often usable by non-programmers. Typically, SQL is referred to as an example of a DSL. SQL allows users to handle data, tables, and databases without any programming knowledge. SQL provides a higher level of abstraction with specific concepts like “select” or “update” in the domain of “data in rows and columns.” SQL is expressive and focused, but at the same time, it pins down the user to a particular problem domain. Other well-known DSLs include CSS, Regular Expressions, and XPath. In addition to these commonly-used DSLs, numerous highly-specific DSLs have been developed in-house by many organizations to help their users capture their business knowledge and automate the generation of repetitive GPL code.

DSLs implemented as so-called internal DSLs, building on a hosting GPL, are widespread. According to Fowler's *Domain-Specific Languages,* “Internal DSLs are a particular form of API in a host general-purpose language, often referred to as a fluent interface” (Fowler, 2010). An internal DSL defines concepts and operations that exist only in a specific domain, providing additional assistance in that domain. JMock and JooQ are well-known examples for internal DSLs built on Java. However, internal DSLs have big limitations. The syntax of an internal DSL can only be what the syntax of the hosting GPL allows. Without that, the resulting code could not be parsed and compiled by the host-language compiler. Furthermore, IDE support for syntax highlighting, code completion and type checking is also restricted and depends on the hosting GPL. This might not only limit the usage of internal DSL to GPL-savvy developers, but it also imposes heavy constraints on syntax and visual representation.

In contrast, SQL could be characterized as an external DSL, since SQL is not embedded or hosted by another GPL. SQL comes with its own keywords, operations, and syntax. It was conceived without any (syntactical) relationship to any other GPL; it is completely independent. External DSLs can be optimized to capture all the necessary information regarding the problem domain, maximizing expressiveness without any restrictions on readability and comprehensibility. The visual representation, the look and feel of an external DSL, could be built to suit language requirements as best as possible, not only for developers, but also for the domain experts themselves. Only then can formulating, communicating, and understanding solutions for a particular problem domain be improved. This was the very reason for coming up with DSLs in the first place.

Nevertheless, external DSLs come with some drawbacks of their own. First, since the language is not hosted by a GPL, out of the box, there is no IDE available. Especially so if the DSL targets a very specific narrow domain, where the low number of potential DSL users do not justify the effort required to create the language-specific tooling or a complete IDE. Yet, DSL users demand intelligent tools to provide assistance and make them productive. Second and more seriously, external DSLs cannot easily reference external artifacts, e.g. code written in a GPL or declarations persisted as XML. An external DSL is somehow closed in itself, as long as there are no importers and exporters (generators to produce executable GPL code) developed additionally. For practical purposes, interacting with the whole development environment is crucial, because not all aspects of a software solution can be provided with a single DSL - which is where language workbenches come in helpful.

## JetBrains MPS

JetBrains MPS is a language workbench designed to define, reuse, and compose DSLs, providing additional tooling for these languages. Unlike other environments, MPS does not rely on the parser technology to provide extensive tool support. Usually, tools and compilers parse the textual representation of code with well-defined grammar to instantiate the abstract syntax tree (AST). This structured tree is then used to look up and transform given information into executable code.

MPS employs a projectional editor to manipulate the AST directly. While the projectional editor might look and feel like a text editor, the user is editing the tree structure in real-time. The code is not persisted in parsable text – the AST itself is persisted, thus avoiding any need for parsing. So, no syntactic ambiguities can arise, no semicolons or brackets are needed to delimit concepts. The projectional editor allows for concise and appropriate domain-specific visual representations. Code completion and error checking can be easily provided by investigating and browsing through the AST, all this is supported by the MPS IDE in a straightforward manner. To sum things up, MPS provides general tool support for a DSL, so that it has not to be built specifically.

Building on the AST as a core foundation of MPS comes with a major advantage. Since grammar ambiguities are eliminated, DSLs can be combined in any possible way. Existing DSLs can be extended and enriched with new concepts. DSLs can be built upon each other, inheriting features and capabilities. This allows for simple reuse of existing language concepts, speeding up development but also fostering a common base.

Let's turn to our showcase to experience this notion from a practical perspective.

## DataUx – a UI Language Built Upon Java

The idea of language combination is nicely demonstrated with DataUx, a language we've built at modellwerkstatt.org to specify user interfaces for data-rich applications written in Java – typically business apps like invoice management or stock tracking. DataUx provides various language concepts to specify forms for data entry, tables for the visualization of collections, and layouts. DataUx draws on the idea of advanced property binding to provide actual data-values for the UI elements. Java Objects with bean properties (getter and setter for specific identifiers) are arranged in an object graph. Then – and this is very special from our point of view – the whole graph is bound to the UI. In our simple example, an Order object with various properties contains also a list of Order-Positions. The following screenshot shows a UI for an Order with its properties and positions. The second screenshot is the specification with DataUx of this very UI.

*Screenshot 1: Demo-Application UI with one Order containing multiple Order-Positions.*

*Screenshot 2: Order UI specification for our Demo Application with the DataUx language in MPS.*

Screenshot 2 above shows the ‘Order main page’ which is bound to a list of Orders. In a Grid Layout consisting of only one column (1* = maximized with weight 1), we placed two UI elements in two rows, the first one minimized as much as possible (-1), the second one maximized again (1*). The delegate-form shows values from six properties of the Order divided up into 2 columns; the table shows the list of OrderPositions of our Order. The DISABLED option sets the whole form into read-only mode; the table is read-only by default anyway.

A key idea of DataUx was inspired by beans binding and the master-detail pattern. Each PagePane is capable of visually representing a graph consisting of objects, where a “selected object” exists for each object type. In our example, two object types are involved: the Order and the OrderPosition. Thus two selections are relevant: the currently selected Order and the currently selected OrderPosition. Now, all UI elements have to be bound to an object type to get loaded with the currently selected object of that type. Within single UI elements, properties of objects can be accessed and bound.

The Delegate Form is bound to the currently selected Order, the Table is bound to the list of positions of the currently selected Order (written as Order.pos). Therefore, the delegate form can visually display properties of the Order object, whereas the table can show a visual of the OrderPositions properties as columns. Five properties of OrderPosition and their column-width are declared.

In our “selected object” logic, the table is not only used to display the lists of objects, it also determines the selected object. The selected row in the table corresponds directly with the “selected object” of that particular type. In our example, changing the table selection will change and define the selected Object of the type OrderPosition. Thus, we could enhance our UI by presenting a visual of the details of an OrderPosition (its properties) in another form, bound straight to OrderPosition.

Our “Order main page” from screenshot 2 is typed as a list of Orders, but when passing in only a single Order, that Order automatically becomes the selected object of that type. That is a special feature of our PagePane concept. If we passed multiple Orders to that page, an additional table would be necessary to define a selected Order. Only then a selected Order could be provided, which is necessary for the detail form and the OrderPosition table. A more complex example is shown on screenshot 3, where multiple Orders are expected. As suggested in the last paragraph, we added a detail form to show the properties of the selected OrderPosition.

*Screenshot 3: More complex UI with master-detail arrangements for our Demo-Application.*

Drawing on this object-binding logic, one can easily specify UIs for complex object graphs with DataUx. The user binds UI elements to an object type to access particular properties of that object. When executed, the UI element gets loaded with the currently selected object of that type. That's probably the most complex matter that has to be explained when using the DataUx language. However, this object-binding logic is checked by the typesystem of DataUx, together with many other consistency checks. Context-sensitive completion and even an "autocomplete UI element" is implemented to provide intelligent editing.

What is remarkable in the context of our DSL discussion is the relationship between DataUx and Java. Obviously, even though DataUx is not compatible with Java syntax, it references Java concepts like properties and classes. Even type-checking rules based on Java are present. Furthermore, in between the UI elements, external Java artifacts are referenced. How is this achieved?

JetBrains MPS ships with a full-featured Java implementation. This GPL is implemented in MPS, exactly in the same way as a custom DSL would be. A nice projectional editor conveying the look and feel of a smart Java text editor is in place, which supports the definition of classes and interfaces, their members, and of course expressions. Since MPS also comes with a jar importer, it can be used as a stand-alone, full-featured Java IDE. We took advantage of this Java implementation and extended it with DataUx. This allows us to reuse powerful concepts, especially expressions and types. By doing so, DataUx gets basic programming support without any effort.

When specifying UI elements with the DataUx language, users can reference declared Java objects and their properties, as long as they are loaded and available in the IDE. Smart scopes for class references, properties or selections are calculated while editing. As seen on screenshot 2, the DataUx option "Table Summary Line" provides a closure-similar concept with a variable labeled as "allObjects" (the type would be list<OrderPosition>). To the right, the option is expecting a Java expression returning a string (type Java.lang.String). In our example here, the static method sum() from the MyUtil class is called. Though, in the context of this option, any valid Java expression is possible, starting from method calls over string concatenations to Java variable references. Smart scopes for all references are calculated specifically in the option's context. All expressions are type checked.

The DataUx language is taking full advantage of the existing MPS Java language concepts, specifically expressions and typesystem handling, which are typically very complex elements of a language. Reusing Java expressions in the DataUx language, while quite straightforward, delivers additional value to the language user. Expressions deliver a great deal of variety to adapt to specific use-case requirements, without having to fall back to some externally defined, unintegrated, plain programming code. It is also the capability of extending another language that delivers the key advantage here: while still not being dependent on the syntax of the MPS Java language, we are able to easily reference concepts declared in that language, for example, a static method called in the table option "Table Summary Line".

DataUx was developed for one of our major customers, an Austrian retailer. Eight developers on the customer's team are using DataUx to implement all the necessary UIs, without resorting to any additional aspects of the UIs in plain Java code. The development of the language could be achieved in a couple of days, once the idea and the structure of the language are set. The UI runtime with its several implementations in different technologies took us longer.

DataUx is currently available via GitHub.

## Conclusion

DataUx is a very straightforward DSL that makes UI design accessible to non-programmers. With its flat learning curve, almost no explanation is necessary. It is also very easy to understand and maintain existing UIs. Furthermore, a user interface specified with DataUx is technology-independent when it comes to its implementation. A very important advantage, since UI technologies are changing swiftly these days. In fact, we provide DataUx runtime environments written for Apache Pivot, JavaFX, Vaadin, and HTML5. The user can just pick the desired platform and MPS will generate the necessary code from the DataUx UI descriptions.

DataUx demonstrates how JetBrains MPS is capable of mixing external DSLs, by letting DataUx extend the internal Java DSL implementation. While maintaining an attractive visual representation, important Java concepts like expressions are reused in DataUx. It is also this language extension that allows DataUx to interact with declarations made in Java. Generally, with MPS, languages can reference concepts from other languages and, in this way, reference specifications modeled with other languages. The projectional editor works with the AST and lets language creators build, reuse, and modify languages, but most importantly, lets different DSLs interact with each other. MPS is fostering interaction between different problem domains.

mps_modellwerkstatt.png

*Figure 1: Multiple DSLs for business application modeling, extending each other.*

Indeed, defining the user interface for a business application with DataUx is only one part of the story. Typical business applications are crossing multiple domains like data persistence (e.g. SQL querying and mapping), business logic and user interfaces. As JetBrains MPS allows interaction between DSL, we devised a language for each specialized part of a business application (see figure 1). Three highly targeted DSLs are available, forming a complete development stack. This way we can provide developers with the most expressive language for the task at hand while still building a complete business application within one development environment. Without integration barriers, high-level abstractions for each domain are given when building business software, improving ease of formulation, understanding, and thus maintainability and quality in the long term.

## References

https://github.com/danielstieger/moware35/tree/master/dataux

http://www.jetbrains.com/MPS

Kelly and Tolvanen, 2008: Domain-Specific Modeling: Enabling Full Code Generation, Wiley-IEEE Computer Society.

Fowler 2010: Domain-Specific Languages, Addison-Wesley Educational Publishers Inc.

application Database Interface (computing) Java (programming language) Concept (generic programming) Domain-Specific Language sql MPS (format) Object (computer science) Property (programming)