LWC 2013 – MetaEdit+

Version 1.1, April 5, 2013
Based on LWC Task Version 2013

Risto Pohjonen
rise @metacase.com

Abstract

MetaEdit+ is a mature language workbench that supports graphical diagram, matrix and table representations. We demonstrate it with an implementation of the 2013 Language Workbench Competition Questionnaire Language (QL) assignment as presented in http://languageworkbenches.net.

The MetaEdit+ implementation consists of two integrated graphical modeling languages – one for defining the sequence of questions and their interdependency and other for additional layout and styling – and a set of JavaScript generators to create a questionnaire form for a web browser to run.

Getting started

You can obtain an evaluation version of MetaEdit+ from www.metacase.com/download. Follow the instructions in the evaluation email / below to install MetaEdit+. Do not log in yet.

Windows

Run the installer and accept defaults. After installation you will have:

- MetaEdit+ app → C:\Program Files\MetaEdit+ 5.0 Evaluation
- User files & repositories → ...\My Documents\MetaEdit+ 5.0

MetaEdit+ starts automatically after installation. You can also start it from the Start Menu.

Mac OS X

Open the distribution .dmg and drag the MetaEdit+ application into your Applications folder.

Execute MetaEdit+ by clicking its icon in Applications.

- First execution of MetaEdit+ will create required folders and files under your Documents folder, which takes about 20 seconds.

Linux

Execute the following set of commands from the shell (requires root privileges or sudo):

```
cd /usr/local
sudo mkdir mep50eval
acd mep50eval
sudo tar -xzvf ~/Desktop/Linux/MetaEdit.tgz
export PATH=$PATH:/usr/local/mep50eval
cd ~
mesetup
```

Executing mesetup will create ~/metaedit and place user files and repositories there.

Start MetaEdit+ by executing:

```
cd ~/metaedit
metaedit
```

Do not log in to the demo repository: we want to get the LWC2013 repository.

The repository for the LWC tasks can be downloaded from:

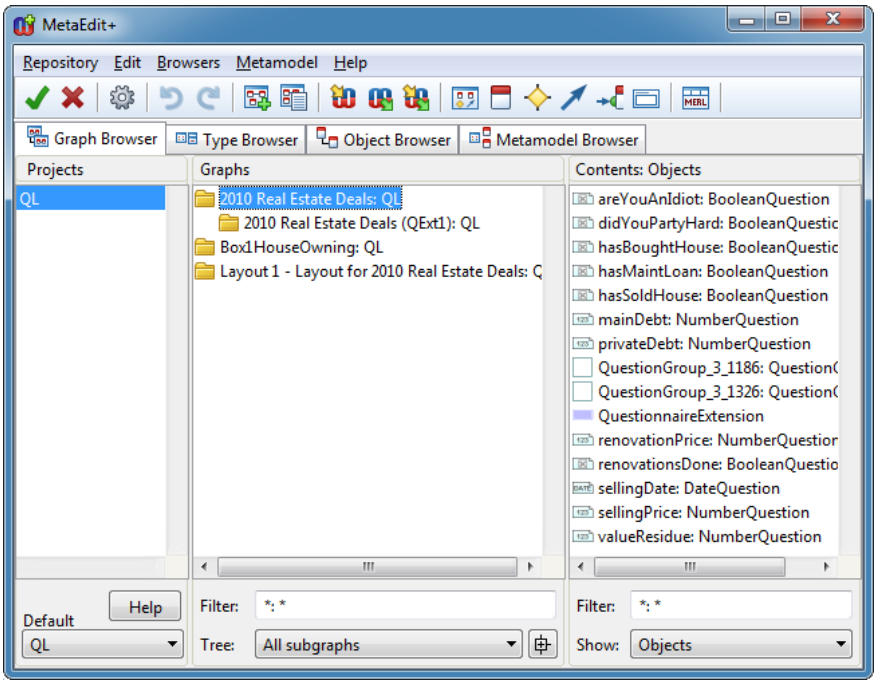
- http://www.metacase.com/support/50/repository/LWC2013.zip

Unzip the LWC2013 repository directory structure into your MetaEdit+ working directory:

- Windows: My Documents\MetaEdit+ 5.0
- Mac OS X: Documents/MetaEdit+ 5.0
- Linux: ~/metaedit

In the MetaEdit+ Startup Launcher, press F5 to update the repository list, which should now show LWC2013 as well as the standard demo repository. (If you can't see it, make sure you unzipped correctly, giving paths like My Documents\MetaEdit+ 5.0\LWC2013\manager.ab.)

Select the LWC2013 repository, and you will see the 'QL' project it contains in the list on the right. Make sure you have chosen 'sysadmin' as the repository user and have 'QL' project selected in the project list, and press Login. Once the login procedure has completed, you will see the following kind of MetaEdit+ launcher:

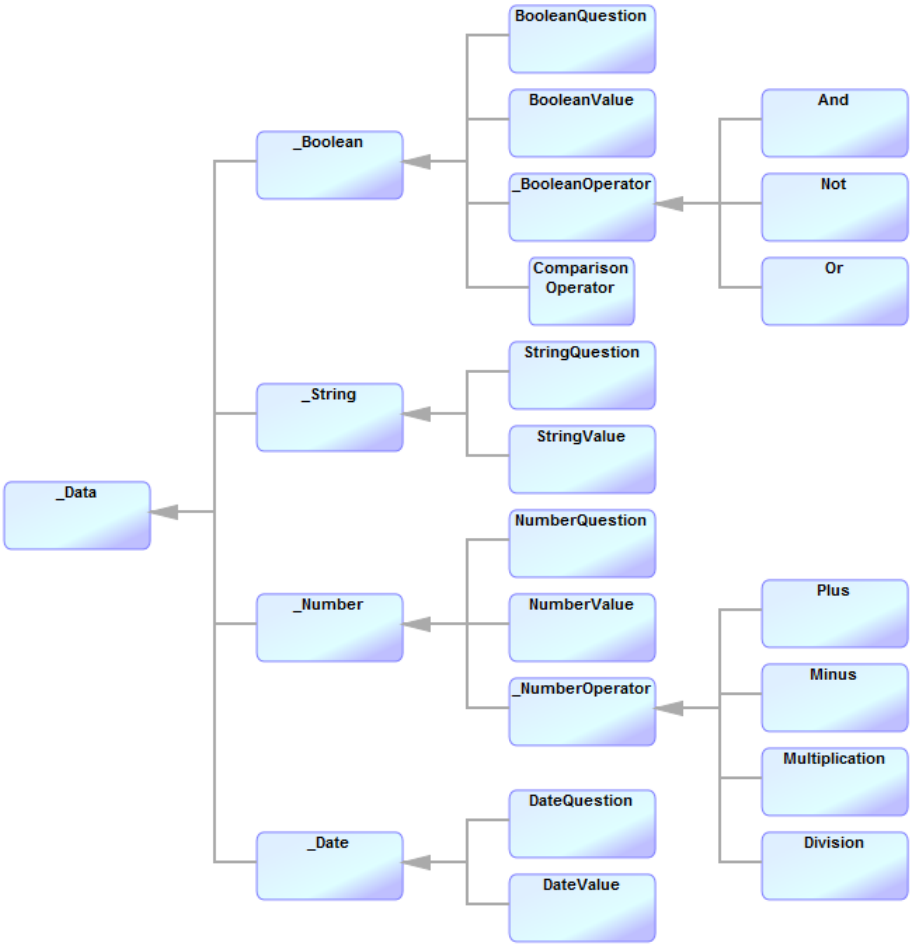


Questionnaire Language

The basic task of the LWC 2013 assignment called for an implementation of a language with abilities to define a questionnaire as a set of sequential and conditional questions and their dependencies.

Metamodel

The actual metamodeling was carried out employing MetaEdit+'s form based metamodeling tools. However, for a better understanding of the language definition, we illustrate the metamodel with two diagrams. The first presenting the type hierarchy of our QL implementation is shown below:



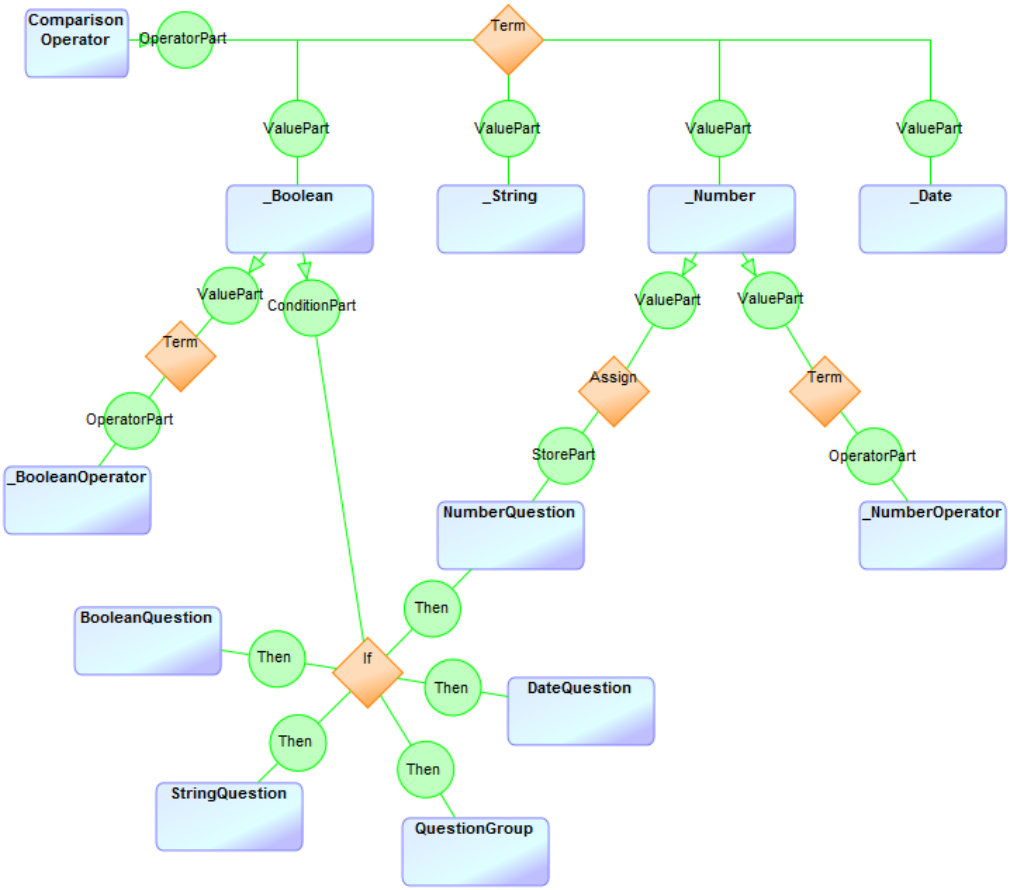
The fundamental idea of the language definition was to emphasize strong data typing within the modeling language itself. Our motivation here was to ensure that all type conflicts could be checked and reported not only during the code generation or run-time but already while modeling. Hence, all object types in Questionnaire Language are derived from the direct subtypes of abstract root type '_Data'. The original assignment required the following types to be implemented: boolean, string, date, integer, decimal and money/currency. First three were implemented as matching types called '_Boolean', '_String' and '_Date'. However, during the domain analysis we concluded that the assignment didn't make otherwise distinction of behavioral aspects of numeric data types (integer, decimal and currency), so we combined them as a single '_Number' data type. The single numeric type enables us to keep the metamodel simple (e.g. one comparison operator definition can handle all numeric cases) and to rely on JavaScript's default behavior when generating code for various data type operations. The original assignment was vague about the enforcement of data type, leaving open questions like what to do when user is about to assign a decimal value into an integer element, so we

used our best judgment and decided to allow it to happen on the modeling language level and have the value rounded in the JavaScript.

All further object types were then sub-typed from these four elementary data types. The '_Boolean' type was specialized further as boolean question, boolean question value, boolean operators as well as various comparison operators. Similarly, '_String', '_Date' and '_Number' types were specialized as their respective constant and question subtypes. '_Number' acted also as a base-type for the arithmetic operations of addition, subtraction, multiplication and division as required by the assignment.

In addition to the data type hierarchy, the QL metamodel includes two additional object types: QuestionGroup and QuestionnaireExtension. QuestionGroup is a simple grouping element that visually nests one or more question types so that they can be referred to as a one body. A QuestionnaireExtension object is an element that is used to split the questionnaire definition among the multiple diagrams if needed - basically it just provides a forward reference to the next QL diagram that continues the definition.

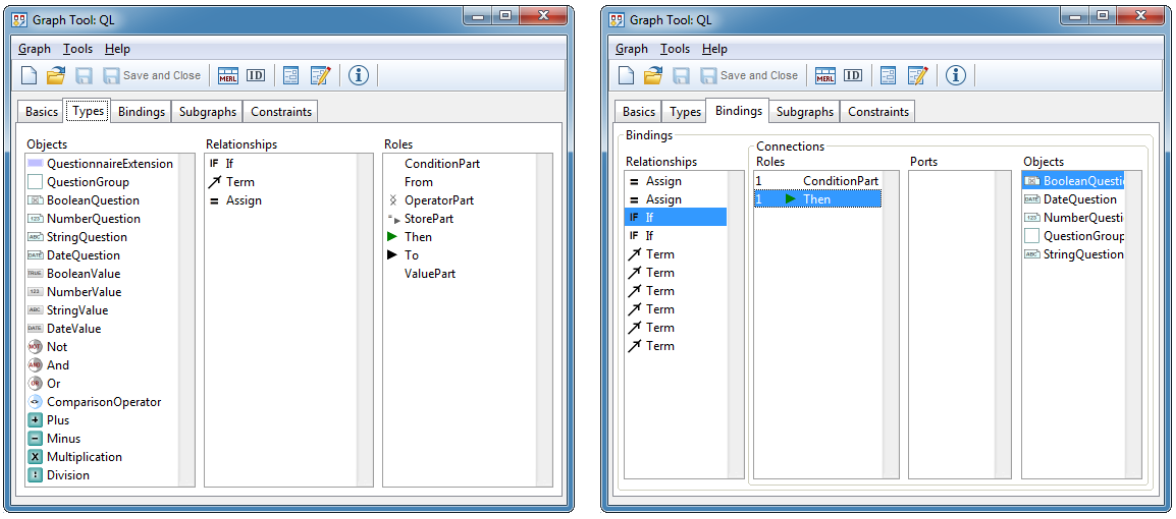
Having the data type hierarchy defined, the next step was to extent our metamodel with relationships that define the interplay between the aforementioned elements. The QL metamodel defines three relationship types: If, Assign and Term. An illustration of these relationships with their binding rules is provided below:



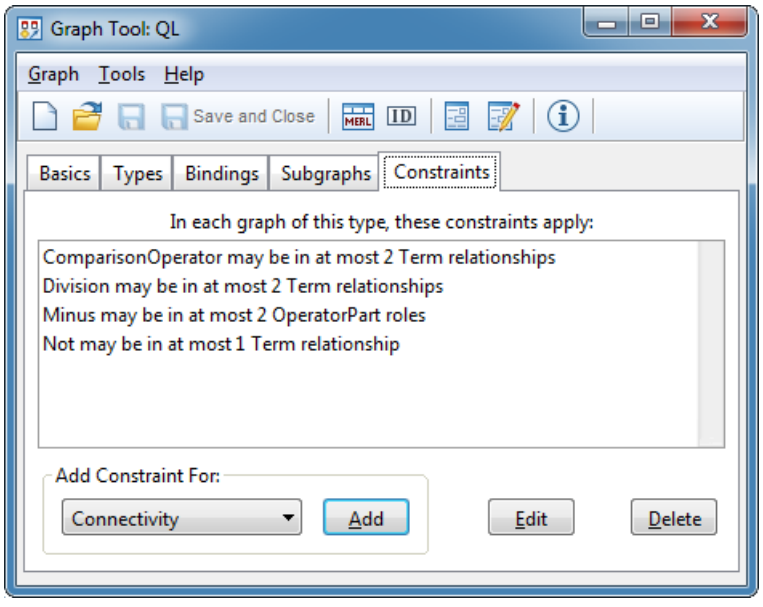
The binding rules for our relationships reads as follows:

- An 'If' relationship connects any boolean type with a question or question group.
- An 'Assign' relationship connects any number type with a number question.
- A 'Term' relationships connects various operators with their input values: boolean operator with boolean values, number operators with numeric values and comparison operator with either boolean, string, number or date values.

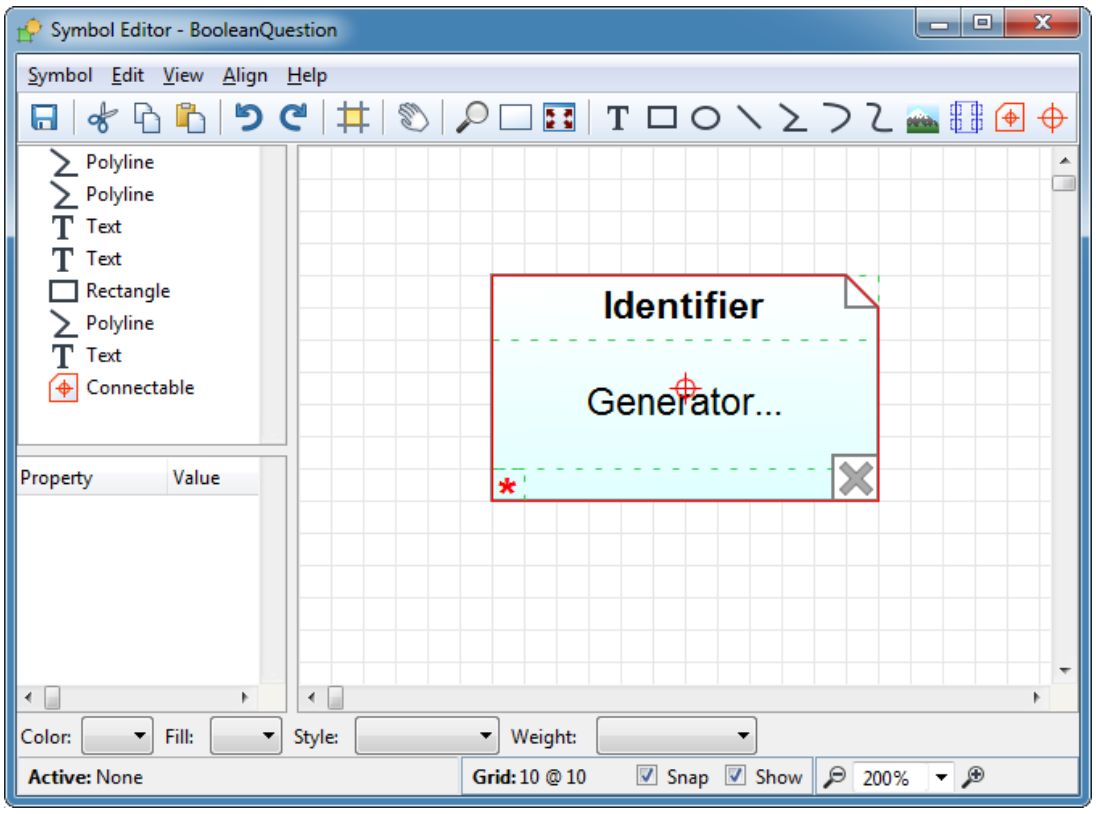
As mentioned earlier, the metamodel was entered into MetaEdit+ using its form-based metamodeling tools. The screenshots below show the metamodel definitions in the native toolset:



The conceptual metamodel definition was completed with a few further constraints that were needed to enforce rules like "comparison operator may be connected only with two values", "boolean operator Not is an unary operation", etc. These rules were defined using the Constraint definer tool in MetaEdit+:



The symbols for the QL concrete syntax were defined using the Symbol Editor in MetaEdit+. Figure below shows a notational symbol for 'BooleanQuestion':



Wrapping up the summarization of the QL metamodel here, we will next take a look at the code generator.

Code Generator

By default, the MetaEdit+ solution for the QL provides two code generators to output the JavaScript implementation for the questionnaire defined in a model: 'Autobuild' and 'HTML'. The 'HTML' generator is the one that actually does most of the work by creating the actual HTML and JavaScript code while 'Autobuild' is a mere wrapper that can be used to automatically load and execute the .html file created by 'HTML' in a browser (please note that when appearing on the Diagram Editor toolbar the 'Autobuild' generator is referred as 'Build').

The 'HTML' generator is divided in three subsections called '_HTML_Header', '_HTML_Body' and '_HMLT_Footer'. As the names imply, each of them is responsible for generating the code of the corresponding section of a HTML document. In addition to their usual HTML tags the header and body sections of the generated HTML document will also include chunks of JavaScript code. The code in header section implements the base domain framework for the QL, providing the services that are elementary for executing the questionnaire in a browser. These include the variables for storing the answer to the question and their utility methods, the data structures for internal storage of the question definitions, basic HTML element creation facilities, etc. The further JavaScript code within the body section has the sole function of defining the set(s) of question(s) that constitute the questionnaire to be displayed in the browser.

The generator transforms the questionnaire definition found in the models into an intermediate JavaScript data structure that loosely resembles the structure of the dynamically generated target output HTML. This structure builds upon two concepts: Page and Question. Question is the low-level atomic concept that encapsulates both the text node carrying the question label text as well as the information about related input type. Page is the natural top-level concept mapping individual questionnaire pages to HTML pages by incorporating the related Questions.

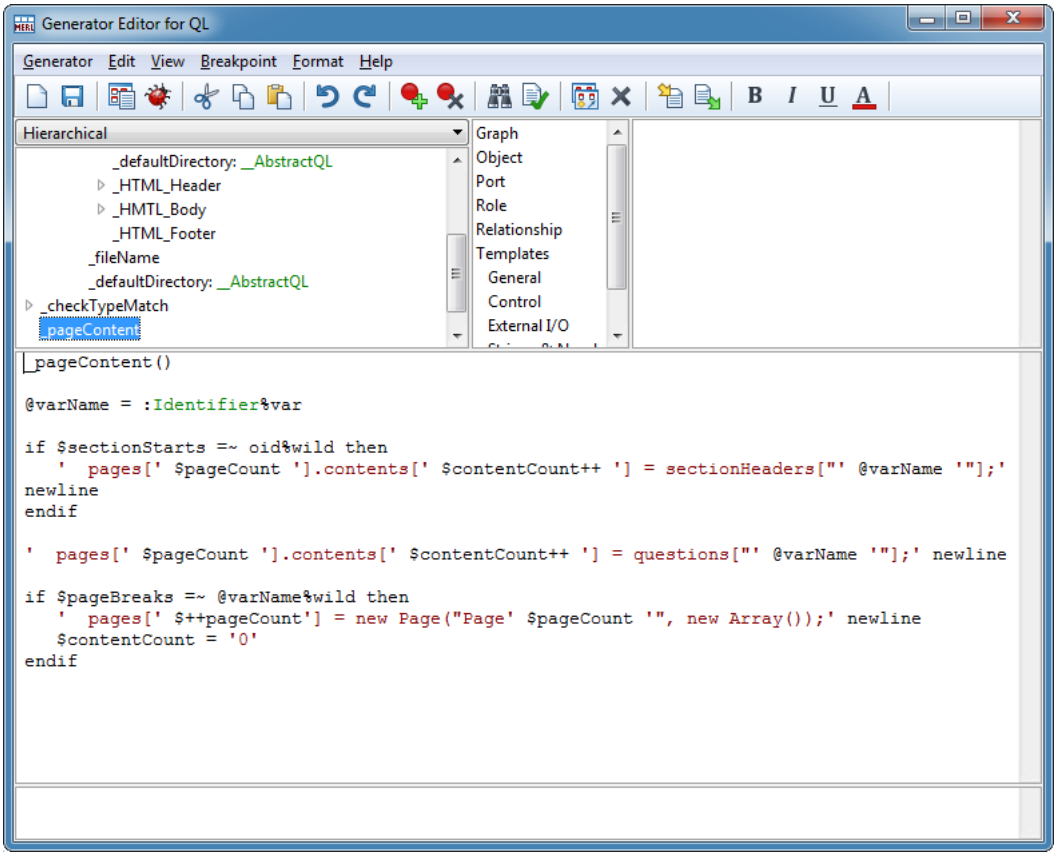
Within this structure the generation of the rest is rather straightforward procedure. The JavaScript framework function buildPage() creates the required HTML document elements and nodes while iterating through the page content elements. Each question is encapsulated within a that is then appended into a page-level element. In the basic case the resulting structure looks like below:

```
<div>
 <div>Question label text node<input></div>
 <div>Question label text node<input></div>
 .
 .
 .
 <div>Question label text node<input></div>
</div>
```

When the user toggles a checkbox or enters a value into a text/number/date field, the validity of input is checked and if accepted, the resulting changes (e.g. changing question visibility or storing a calculated value) are dynamically propagated into the page. These checks and changes have been culled from the relationships found in the QL model and implemented by the JavaScript framework's checkValues() and propagateChanges() functions, respectively.

The required serialization of the questionnaire data proved to be a bit difficult at first. As the good JavaScript practices (and browser implementations) forbid the browser for directly accessing underlying file system, there was no straight way to store the data on the disk. As we wanted to rely on the basic browser support, we also ruled out any server side solutions and additional browser extensions. Therefore we use HTML5 Web Storage feature to store the data internally. Unfortunately there are some compatibility issues with the Internet Explorer's implementation of Web Storage which renders it unusable for our purposes. Hence, when running the generated code in the Internet Explorer, we use a fall-back solution that will provide the same outcome as HTML5 Web Storage but only within the same session.

The code generator was written within MetaEdit+ using its MERL generator definition language, Generator Editor and Generator Debugger. The generator definition environment is shown below:

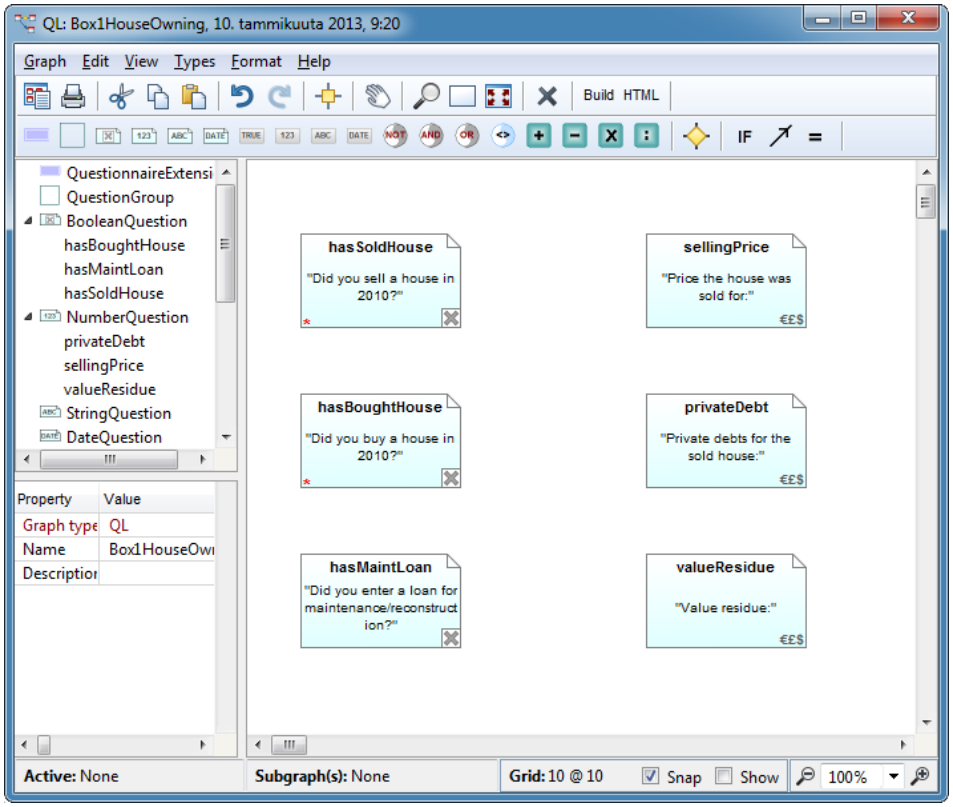


Having covered the metamodel and the code generator, it is now time to learn how to define questionnaires with our DSM environment.

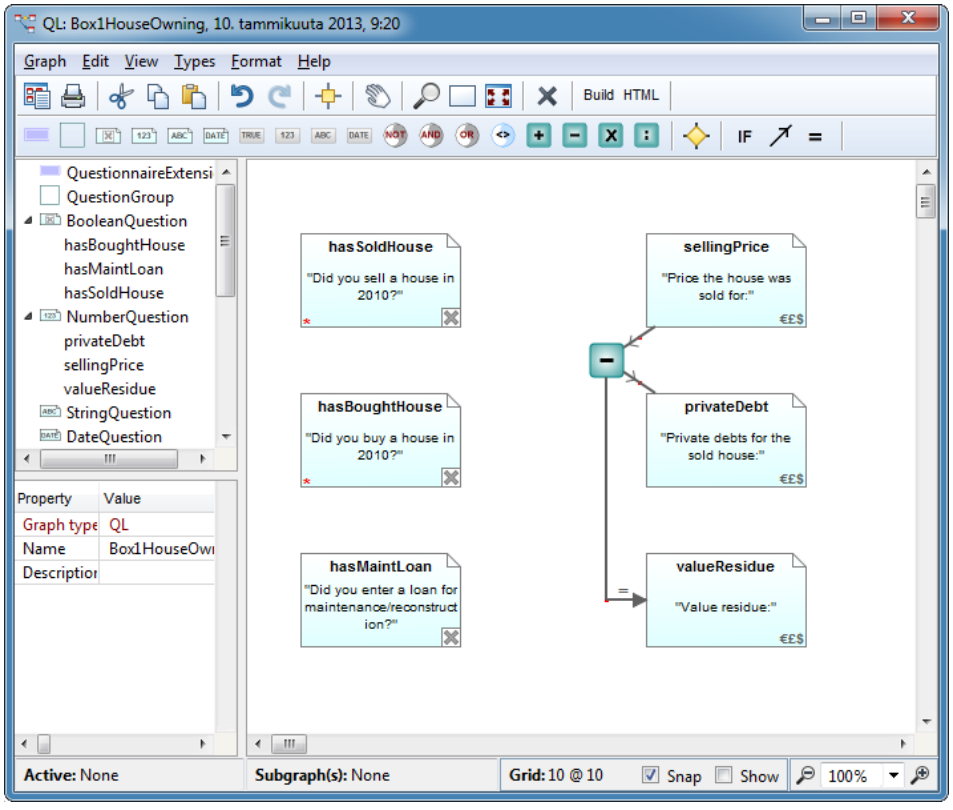
Working with the Questionnaire Language

The basic workflow for creating a simple questionnaire as the one presented as an example in the original assignment follows these steps:

1) Create a new QL diagram by selecting **Create Graph...** from the popup menu in the MetaEdit+ main window's 'Graphs' list. In the Create Graph dialog, select 'QL' as graph type and 'Diagram' for 'Open as'. Press **OK**.
2) Enter name and description for the new QL model and press **OK** to open it in the Diagram Editor.
3) Add Question objects into this new diagram. Select the Question type you want from Diagram Editor's toolbar, place mouse cursor on the drawing area and press left mouse button. Enter identifier name and label test for each Question and set whether it is optional or not. The sequence of the questions is defined by the X/Y position of a Question object – the order is always primarily from top down and secondarily from left to right. The question sequence of our example would thus look like this:

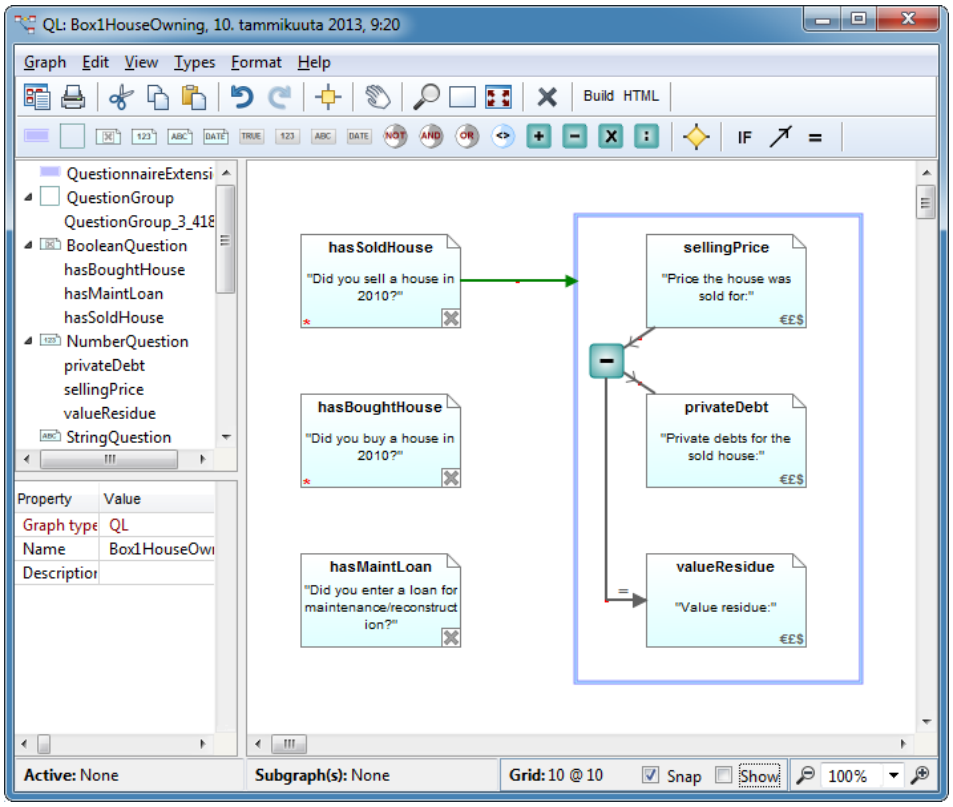


4) Add possible arithmetic, boolean or comparison operators and connect them to Questions with their respective relationships. A relationships is drawn by first selecting its type from the toolbar, placing cursor on top of the first object, pressing the left mouse button down, dragging the cursor on top of the other object (by keeping the button down) and releasing the button on top of that object.



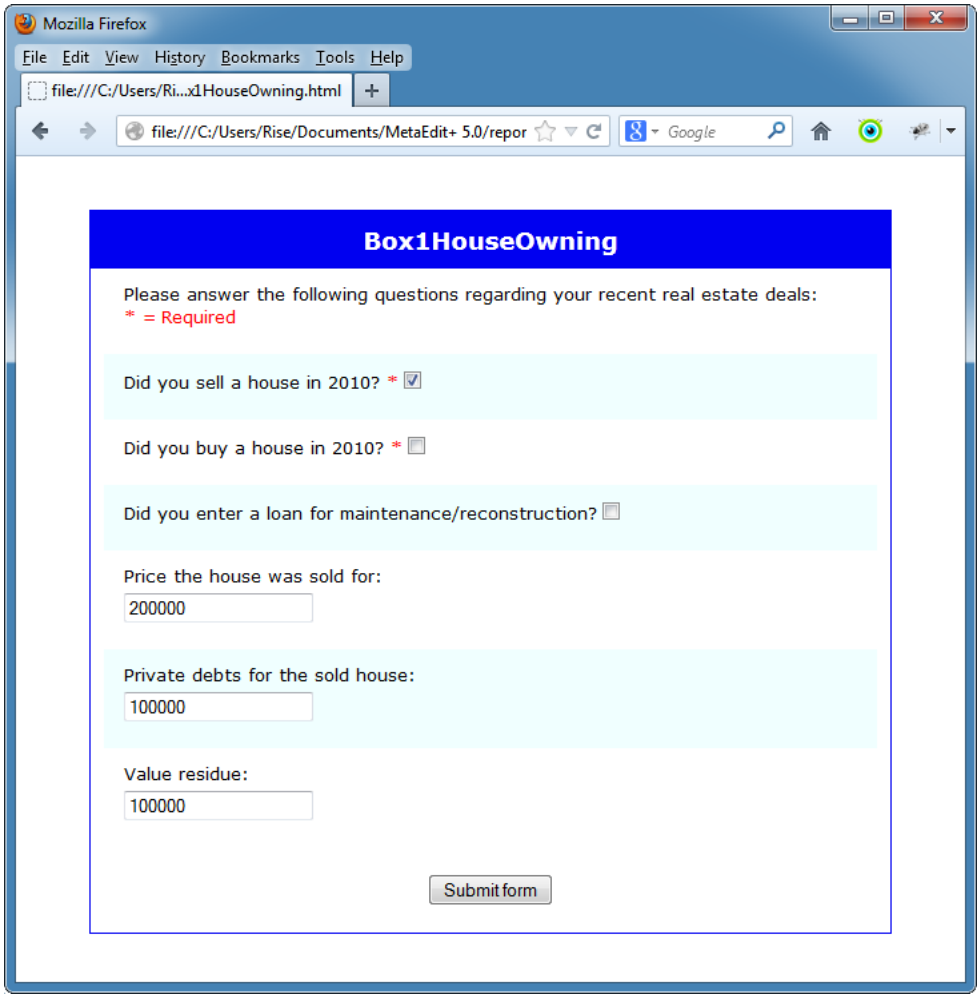
At this point it is worth noting how to use the non-asymmetric arithmetic operators. For example, the subtraction operation gives different results depending on the order of its terms (i.e. 10 - 5 <> 5 - 10). In our QL implementation this order is defined by the positional dependency of terms: the leftmost element is considered as left-hand side of the operator (if the horizontal position of both elements is equal, the topmost element is considered as left side). This interpretational order is visualized with small arrow heads near the operator.

5) Define optional question groups by drawing QuestionGroup perimeter object around them and add visibility toggle rules using the 'If' relationships. The If relationships can connect any element carrying a boolean value or result (e.g. BooleanQuestion, BooleanValue, boolean operator or comparison operator) into a Question or QuestionGroup element. In our example we can hence simply define the visibility toggle by drawing a True relationship between 'hasSoldHouse' Question and the group containing 'sellingPrice', 'privateDept' and 'valueResidue':



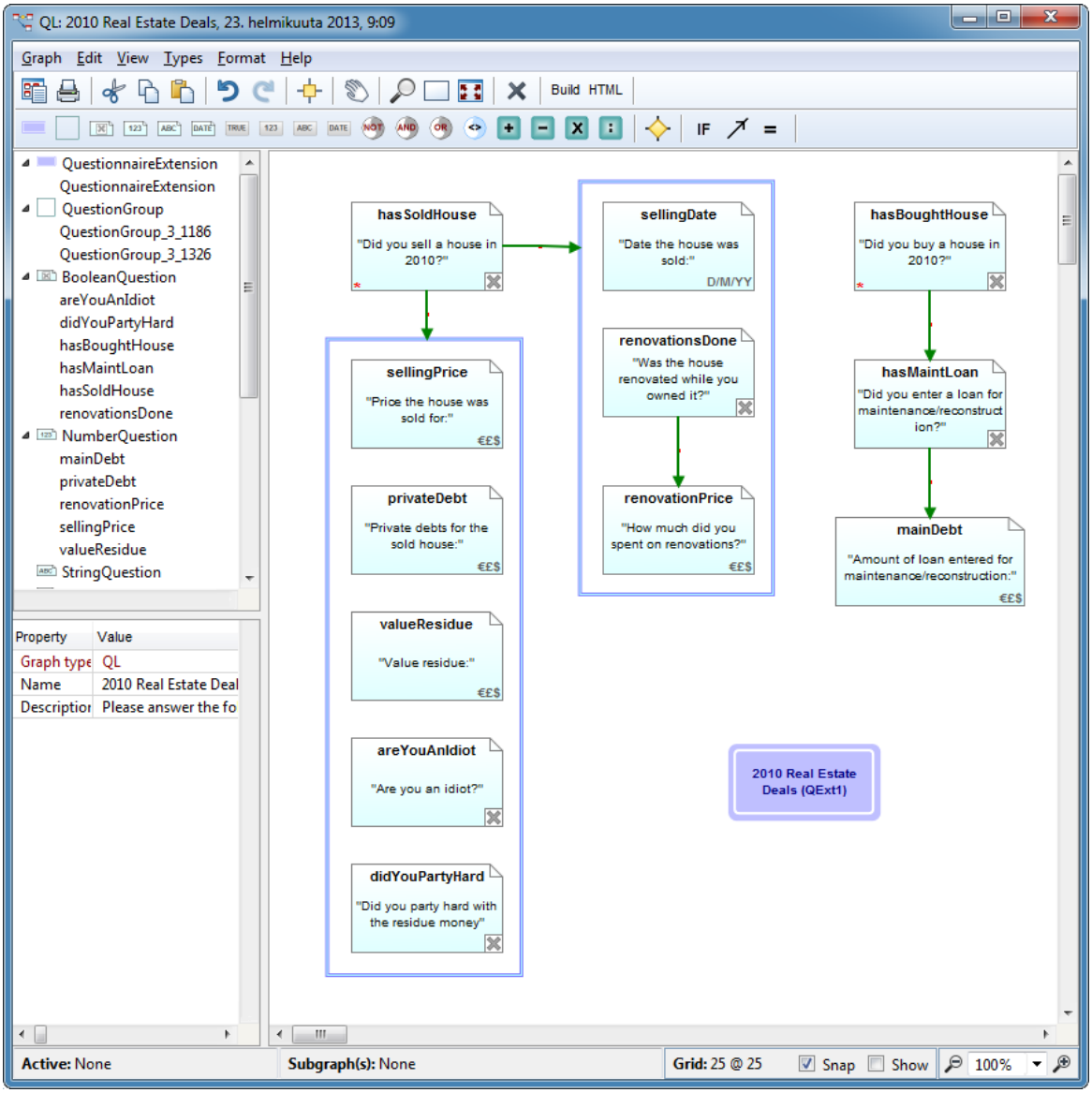
6) Having completed the definition of our example questionnaire, we can now proceed to try it out by generating and executing the JavaScript code. To generate the code and automatically load it into a browser, press 'Build' generator button in the Diagram Editor toolbar.

7) When the code has been generated and loaded into the browser, you can try out the questionnaire:

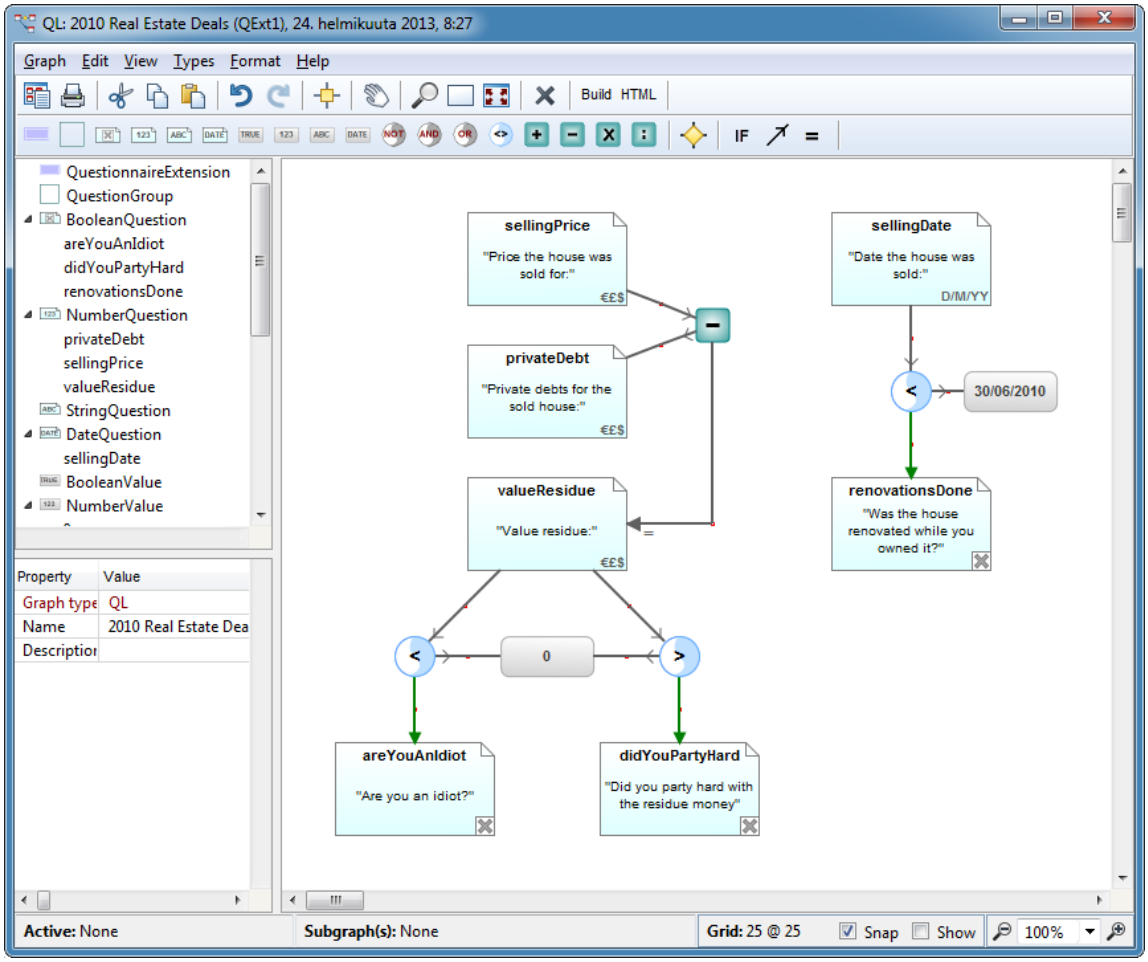


8) Once you have answered the relevant questions, press 'Submit form' button to save the questionnaire data. You will now see a page summarizing your answers

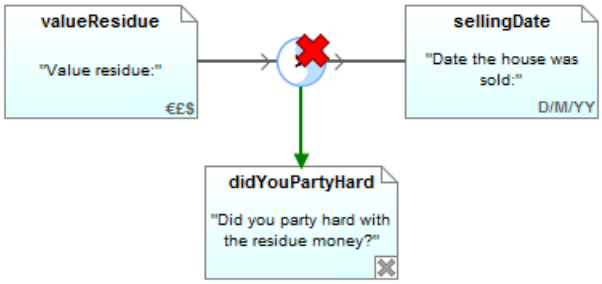
To have a better idea of what the QL is capable of, let us consider a bit extended version of our example questionnaire:



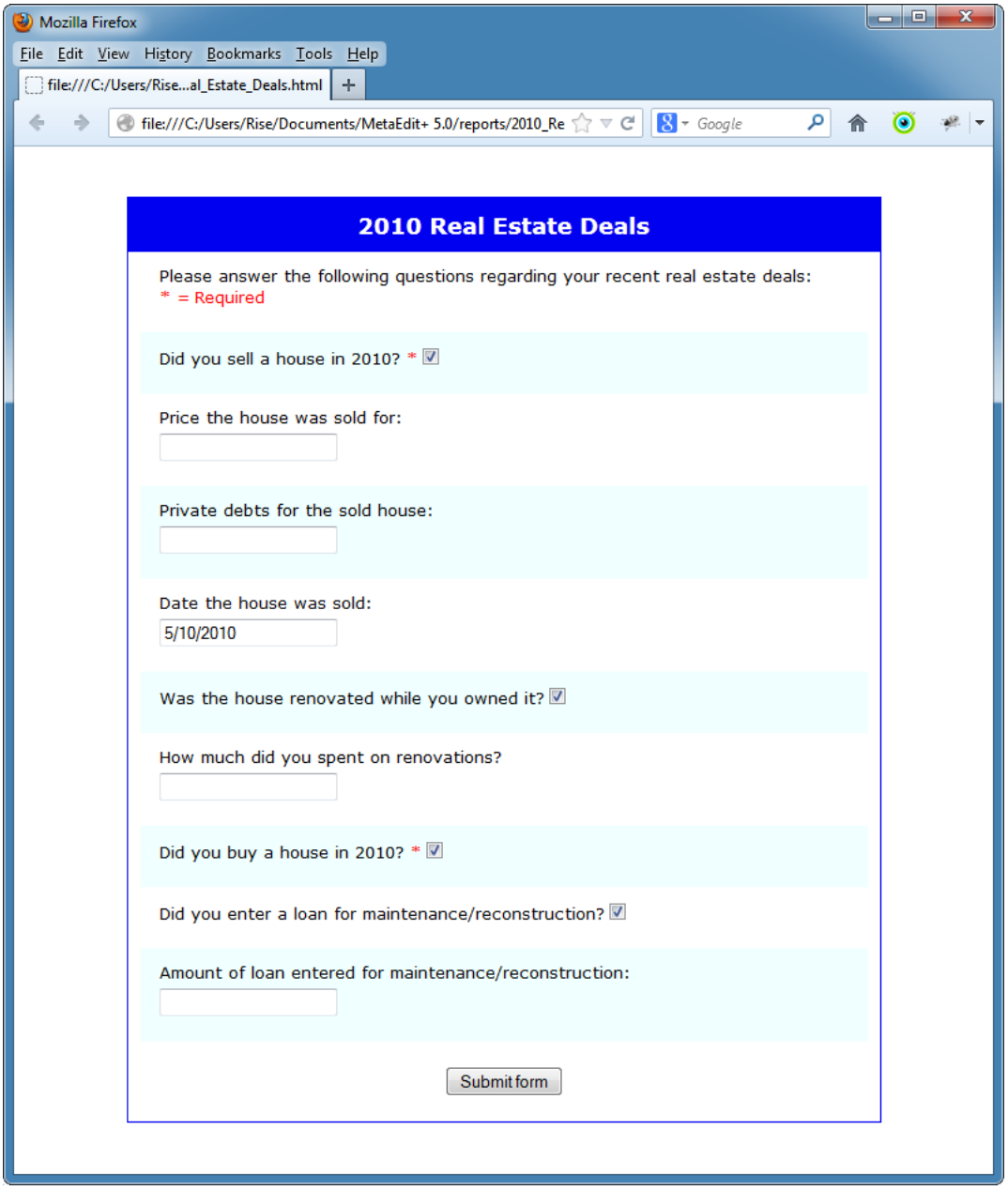
As can be seen, we now want to know a bit more details regarding the possible real estate transactions. This diagram defines the complete question sequence for our extended questionnaire and the basic visibility toggles that are driven directly by the answers to the questions. However, adding more complex toggling rules may hinder the readability of single diagram and therefore we are now using a QuestionnaireExtension element to extend the questionnaire definition into another QL diagram:



The definition of the calculation and assignment of valueResidue's value we already saw in the basic example is now placed in this diagram. Furthermore, depending on the financial situation of the house owner after the transaction, further questions are revealed. Similarly, if a house was sold during the first half of the year, more questions are to be asked. As a side note, the left-right ordering of comparison operator's terms follows the same logic as ordering of terms in arithmetic operations (i.e. left/topmost element is considered as the left-hand side term, visualized by small arrow heads). In the case there is a type mismatch between the compared elements, a warning sign will appear on top of the comparison operator symbol:



Generating code for this example will result with a questionnaire page with more dynamic behavior:



And so concludes the summarization of our solution for the basic task of LWC 2013 assignment. The assignment also included two more optional tasks, which we are discussing the following chapters.

Questionnaire Layout Language

The LWC 2013 assignment also suggests two further optional extensions to the QL. First one of these is an additional layout and styling language (QLS). The elementary requirement for QLS is the independence from QL while still being integrated with it – the models made with base QL should remain untouched while the layout and styling aspects are defined. Another important aspect is the referential integrity between QL and QLS – only defined questions shall be used within the layout and type compatibility must be matched.

Metamodel

The metamodel for QLS turned out to be surprisingly simple, consisting only the following four object types:

- Question types: these were reused from the base QL definition, enabling the user to bring a question from an existing QL model as a reference to the QLS model.
- PageBreak: a simple object that defines a point where the question set is split, both subsets being rendered on their own pages.
- SectionHeader: an object storing the text for section header.
- Style: an element that stores style definitions for colors, fonts and input widgets.

In addition to these, there is only one relationship type, SectionStart, that defines the bindings between SectionHeaders and Questions. No additional constraints were required for QLS.

Code generation

The code generator for QLS differs very little from the base QL code generator – actually it reuses most of its code base with the original. The only notable difference can be found from the JavaScript code generated within the HTML <body> tag. Where the original QL generator creates a single page containing all questions, its QLS counterpart must provide more complex structure with the questionnaire distributed on several pages and subsections with individual style attributes. This is done in two steps: first we create all style definitions and assign them into the questions (this is done in cascading fashion so that we always refer to nesting style if needed) and after that we create the set of pages according to the PageBreaks found from the model and infiltrate the possible section breaks into the question sequences on each page. The base JavaScript framework then takes it further from here.

The only other minor difference between the QL and QLS generators is the QLS reimplementation of 'Autobuild' and 'HTML' generators to enable them to work with slightly different top-level construct (i.e. generation start from the QLS diagram, following the reference to the original QL model and providing the QLS specific generation of HTML <body>).

It is, however, worth to note that we heavily refactored the framework generator after analyzing the features we needed to support the QLS. As a result, the framework now includes a few features that are not needed for the basic QL models (like handling of styles, sections and navigation buttons). Still, most changes were justified also from QL's point of view, as they corrected some bugs in the original and made the whole architecture more comprehensible (e.g pages didn't originally have a code level concept of their own but were hacked on top of the Div and Question hierarchy just before the HTML elements were to be appended into the HTML document). The refactored framework code is also more robust and more flexible for possible future requirements.

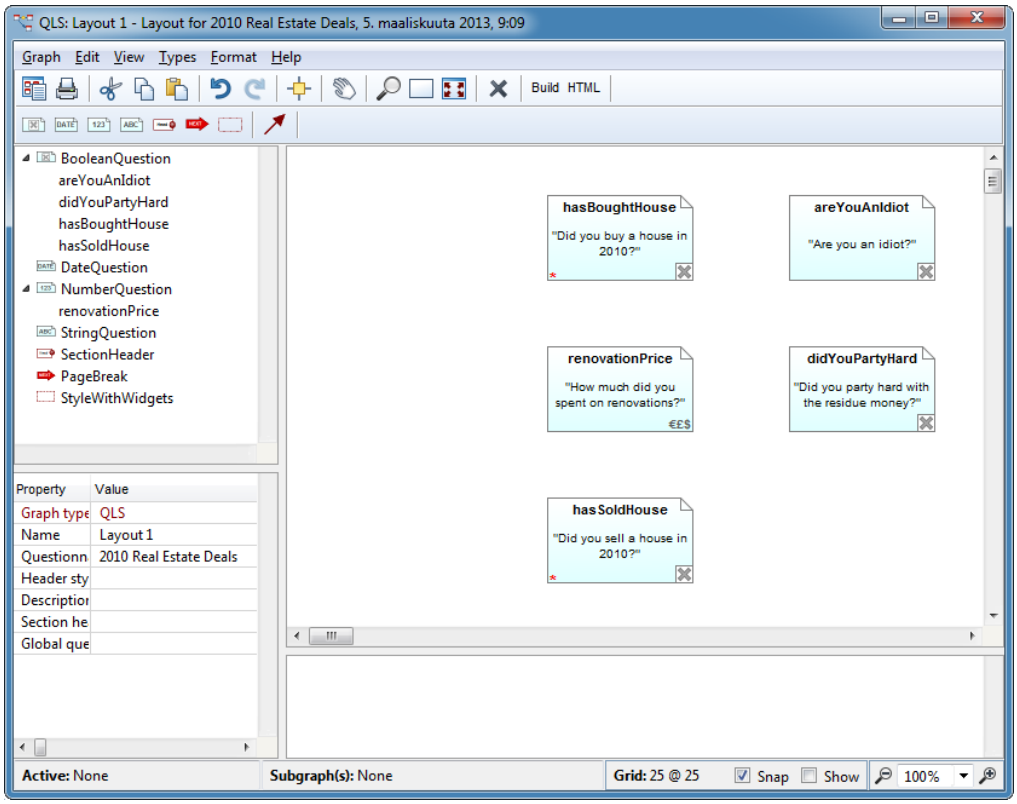
Working with the Questionnaire Layout Language

The first thing to keep in mind while working with QLS is its relationship to QL itself – a QLS model is just an extension of an existing QL model. The division of responsibilities between QL

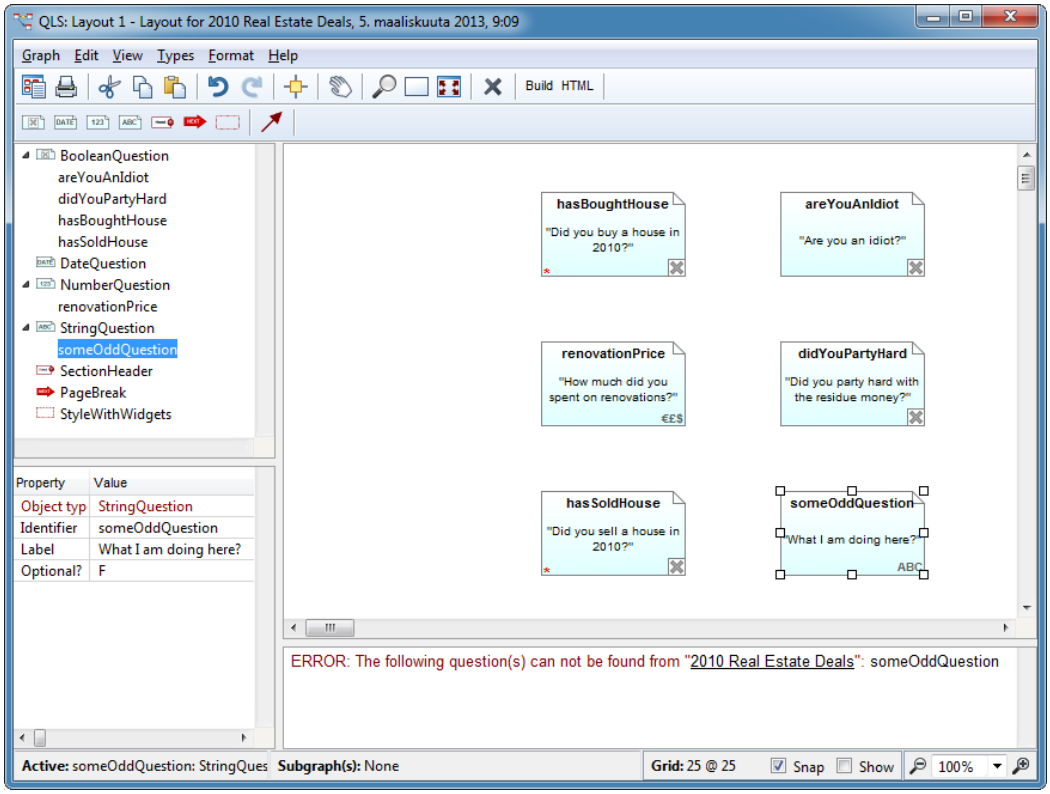
and QLS is clear: QL defines the questions, their sequence and other relationships, whereas QLS provides the layout and styling. This means that it is possible, for example, to create several different layouts for the same QL model without tampering with it.

The suggested workflow to follow with our solution is outlined below:

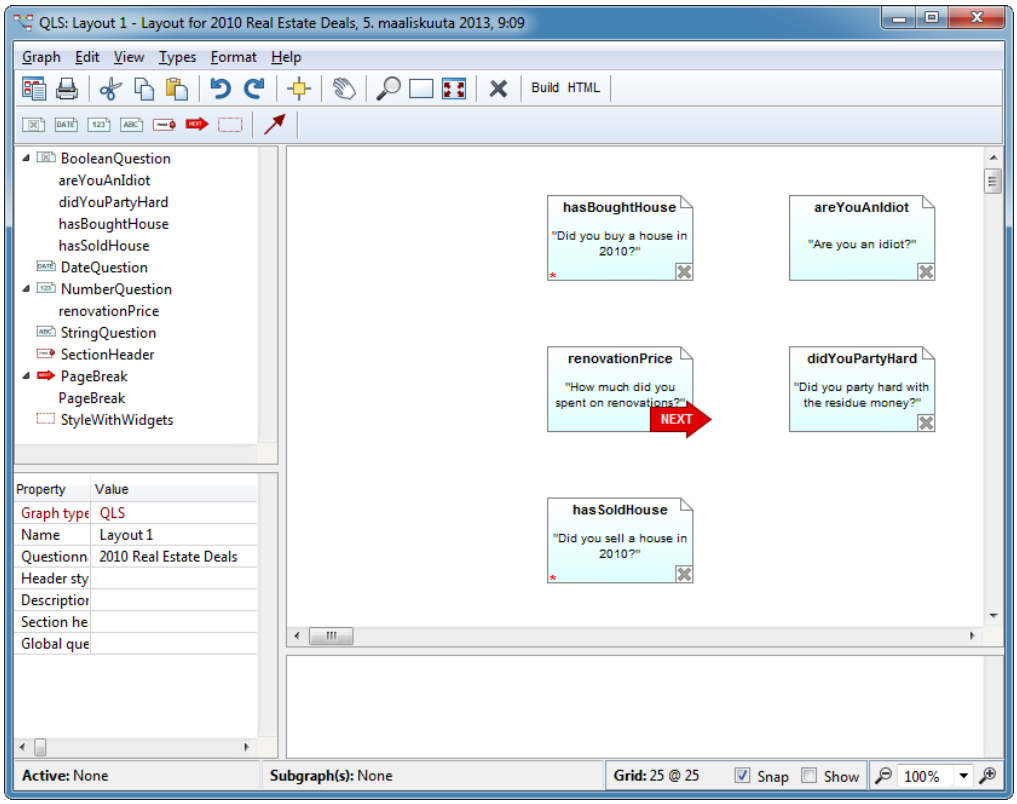
1) First, create the normal QL model for your questionnaire (we will stick with our previous example here).
2) Create a new QLS diagram and associate it with the underlying QL diagram by attaching it as the Questionnaire property for the new QLS model (select **Attach Existing Graph...** from field's popup menu).
3) Add existing Question objects from QL model into the diagram. This can be done by placing mouse cursor on the drawing (having no selections), pressing the right mouse button and selecting **Add Existing Object...** from the popup menu. After this, press 'Graphs' button in the dialog that opens. You will now see all available graphs in the 'Selection list' - double-click the one you have associated with this QLS model. You will now see all Questions present in the underlying QL model. Select the ones you want by double-clicking them and press **OK** once you are satisfied with your selection.



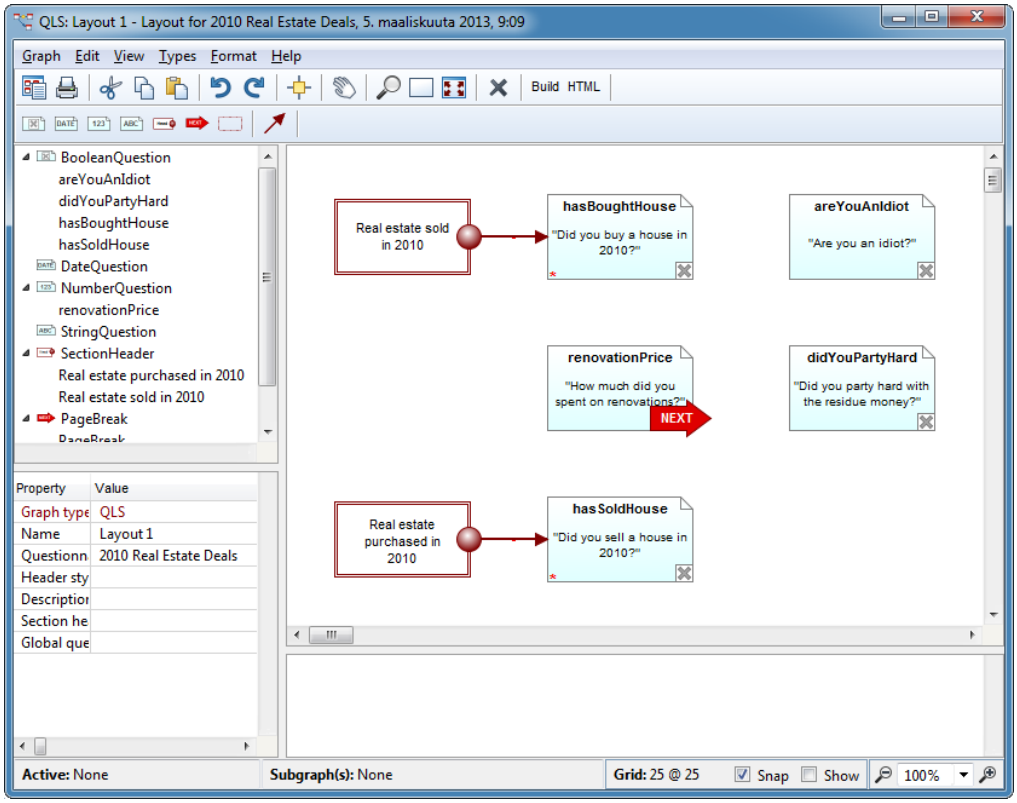
If you are trying to add a Question that does not exist in the original QL model, an error will be shown in the LiveCheck window at the bottom of the screen.



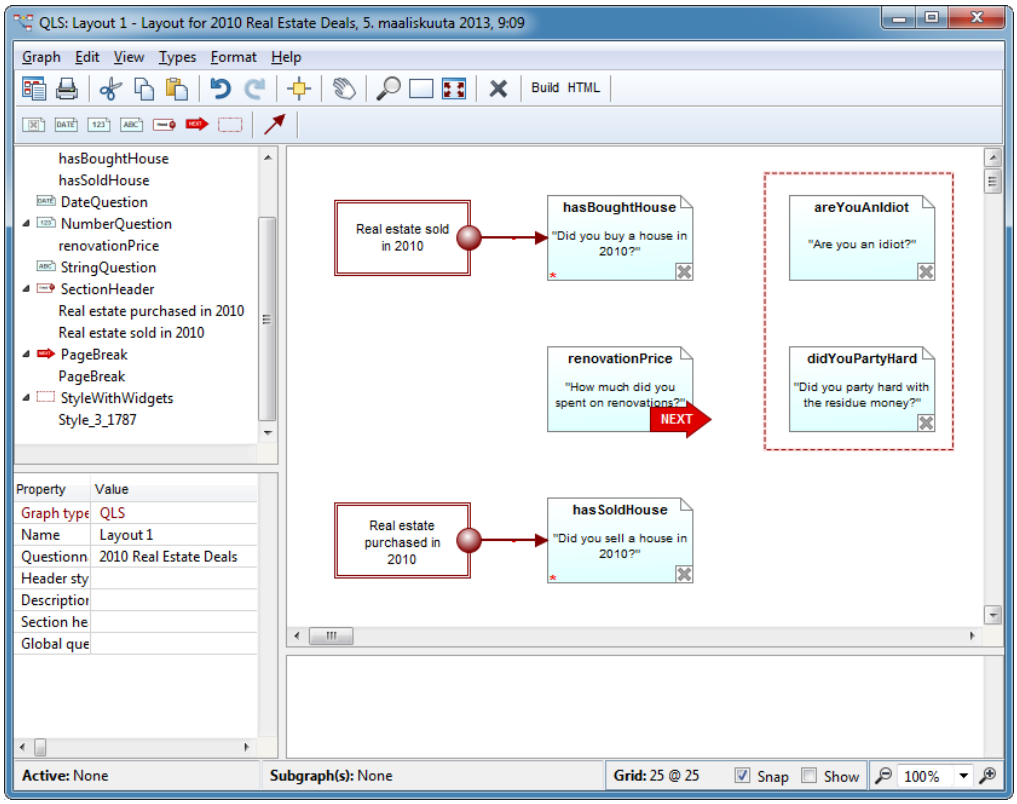
4) Add PageBreaks to define the page structure: drag a PageBreak on top of a Question to define a page break after this question:



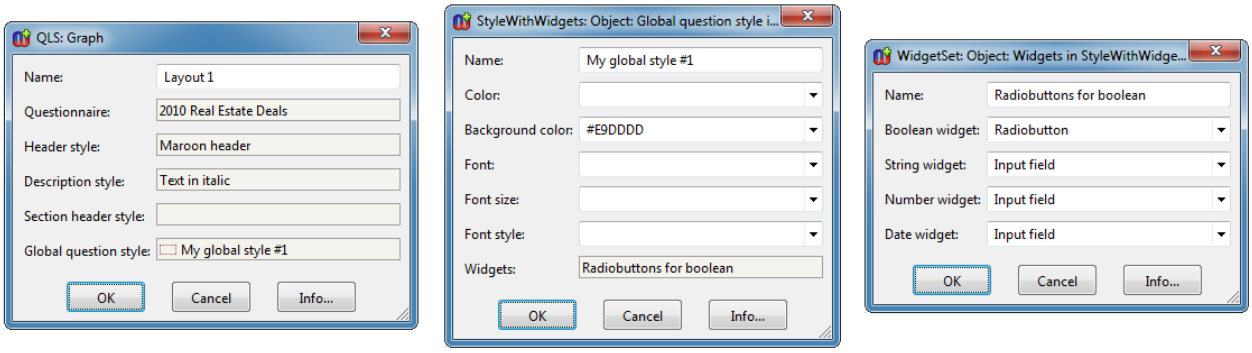
5) Define SectionHeaders if needed and connect them with SectionStart relationships to those questions that are supposed to start new sections:



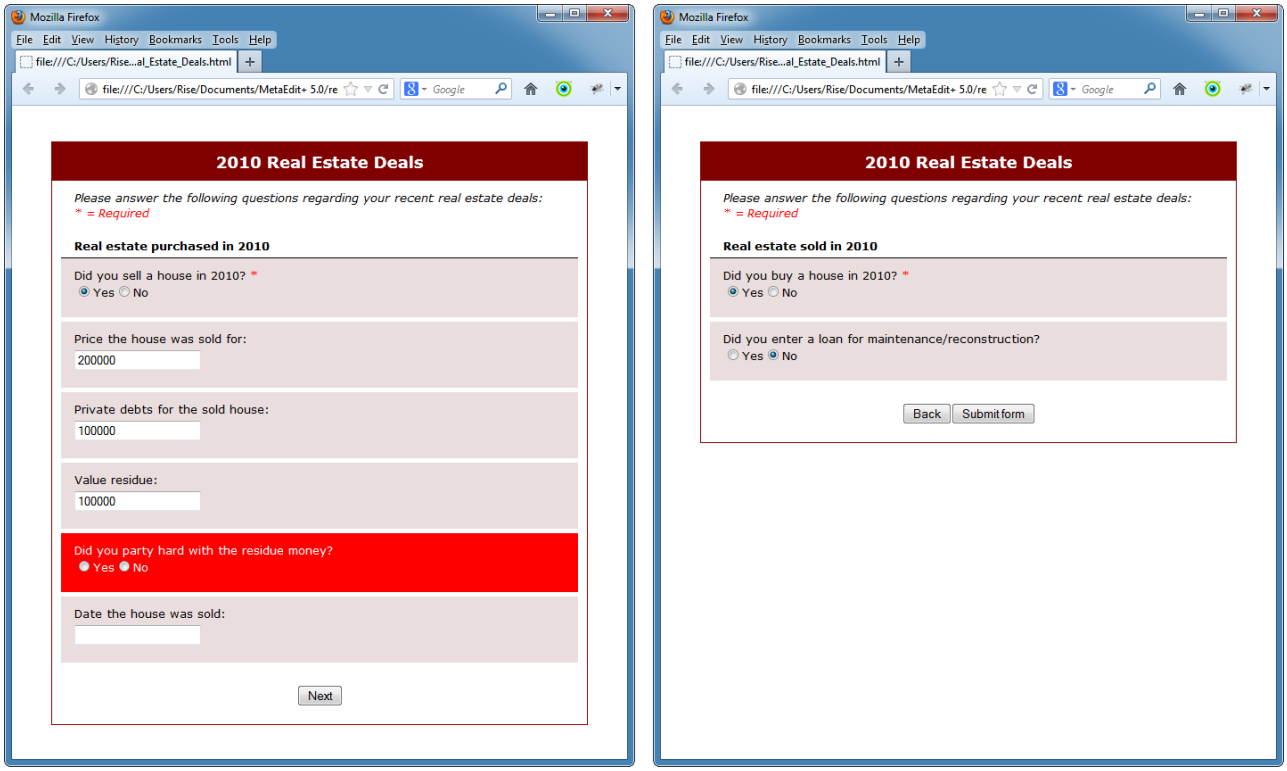
6) Styles can be added on several levels. It is possible to define a Style object around a set of Questions to tell the code generator to render these questions using the style attributes possessed by this Style object (or one of its nesting Styles):



Or you can add a global style definition for all questions as well as for various other page elements in the Graph properties:



7) Generate the JavaScript for the questionnaire by executing the 'Build' generator.
8) Test questionnaire in a browser:



At its present stage, our QLS implementation still remains a bit sketchy. It is provided as a prototype and example to illustrate some of the possibilities provided by MetaEdit+ and its code generator to tackle with visual design requirements. The main reason we rest our case here is the fact that all further development of this example no longer provides any important addition the original assignment itself. For the last changes we made, over 90% of our effort were spent adding features on the JavaScript domain framework to support various styling aspects and widgets – very little was added to the metamodel or general generator behavior or logic. Therefore we considered the requirements of the original assignment fulfilled far enough.

Questionnaire Analysis

In addition to the QL and QLS implementations the original assignment also included another optional requirement: analysis of the questionnaires. Most of these required analysis tasks were related to the sensibility, integrity or type safety of the QL models. In our implementation, the existing QL and QLS implementations already address some of these optional analysis requirements. For instance, as we have seen, the requirement for type checks was already fulfilled for the boolean, comparison and arithmetic operators either by our binding definitions or by visually denoting conflicting situations. Similarly, the requirement of ensuring that a question should only be asked once (if ever) is already fulfilled by the elementary QL implementation.

It is also worth pointing out that the nature of our base QL solution and having it as a graphical modeling language also renders some of the analysis task unnecessary in our case. For example, as all operators in our language require explicit connection with already existing question or value objects and as the code generator is promptly taking care of their implementations, we are well protected from the troubles brought by the undefined and uninitialized variables. In other words, our goal has been to empower the end user with a straightforward, flexible and trouble-free environment for quick creation and execution of the questionnaires so that “everything just works”.

MetaEdit+ also has some built-in features and predefined report generators that provide solutions for such model analysis and tracing requirements like finding the places where a certain object is referred or reused, or quick propagation of changes via shared properties.

Conclusion

As the 2013 edition of the LWC assignment obviously geared towards the implementation of the run-time rendering system of the questionnaire definition, it is hard to say how well our solution showcases the features and functionality of MetaEdit+ itself. With the current solution, where all moving parts - the language, code generator and run-time framework - were open for changes, there was no hitting the limits what comes to MetaEdit+ as a tool - unfortunately the same can not be said about our capability as JavaScript programmers! For more fruitful comparison and evaluation purposes, it would probably have been better idea to target for some existing run-time platform instead of coming up with our own - like was the case in LWC 2012. However, the graphical DSM environment provided several possible approaches for language definition. The current language solution that was picked up early on remained pretty much untouched through the whole development cycle - even if we had to heavily refactor our run-time system when support for QSL was added - which suggests that its level of abstraction is good enough to insulate us from the implementation details and the requirements for capturing the essential concepts of a questionnaire were met.