## 1 Introduction

The demand for gathering data by electronic means, such as web-based surveys or feedback forms for mobile applications, faced an important increase in the recent years [6]. The increase in demand is also reflected into the evolution in the socially-oriented technology, such as mobile technology, social media, etc. With the evolution of the socially-oriented technology, the design complexity of the surveys also increased. Survey creators want to express surveys that interact with the users in complicated ways [6], such as forking/branching, different ways to answer questions, different scales used in questions, etc. However, there are not too many solutions that can help survey creators express surveys in the way they want [6]. The already existing Domain-Specific Languages (DSLs), such as SuML [6] and QPL [6] offer limited possibilities regarding design flexibility.

The objective of this project is to define and implement a textual DSL for generating surveys to be used on two different platforms. Moreover, this project highlights the development and implementation workflow as well as generating surveys for the Android platform and for web application.

This project addresses the online survey and surveys presented as a mobile application, using e-mail as an electronic data reporting method.

The domain of survey research has a broad coverage and it can be employed by almost any discipline or field [5]. Therefore, it is beneficial to design a tool, such as a language, that can generate surveys for different environments as easy as possible.

## 2 Domain Analysis

A common type of social science and quantitative research method is represented by surveys. In a survey research, the researcher defines a relevant collection of individuals as a sample that represents the target population as a whole [3]. The surveys can be presented in different formats and environments such as a written document, an online survey, a face-to-face interview, electronic data reporting or a telephone interview [4]. This tool should allow the users to express surveys in a clear and fast way while not having to write code that is not specific to the domain [6].

Surveys include two types of questions; open questions represent the type of questions where the respondent is not provided with possible response categories and the respondent has to input his/her own answer. The closed question type is the type of question where the possible response categories are presented to the respondent along with the question. Moreover, the closed questions type is also divided into several sub-types, most commonly used being *Dichotomous*, *Multiple Choice*, *Constant Sum*, *Rank Order Scaling*, and *Staple Scale* [4, 6]. Moreover, every particular question type previously stated has the feature to be set either optional or mandatory to be answered by the user.

In order to address the survey design complexity addressed in the previous chapter, an important aspect of the domain is *forking*/*branching* [8]. The *branching* aspect of the survey helps social scientists to define a survey workflow according with what the respondent answers a question [8].

## 3 Ask

Ask is the language we have designed, that enables the user to write out surveys for websites and android phones. In this section we will cover the Ask DSL.

### 3.1 Language requirements

From the analysis of surveys we found a range of question types most commonly used, i.e. Multiple Choice, Constant Sum, *Rank Order Scaling* and *Staple scale* in surveys and decided that the language should support these.
A considerable number of online surveys also have optional questions and allows forking questions. This lead us to the following language requirements

- Full support for all question types found during language analysis
- A question can be mandatory or optional in a survey
- Where possible a question can lead to different questions depending on the answer

### 3.2 Language Design

The meta-model of our language was realized iteratively, refining it at each iteration step. The changes made through the iterations  and the reasons why are described in

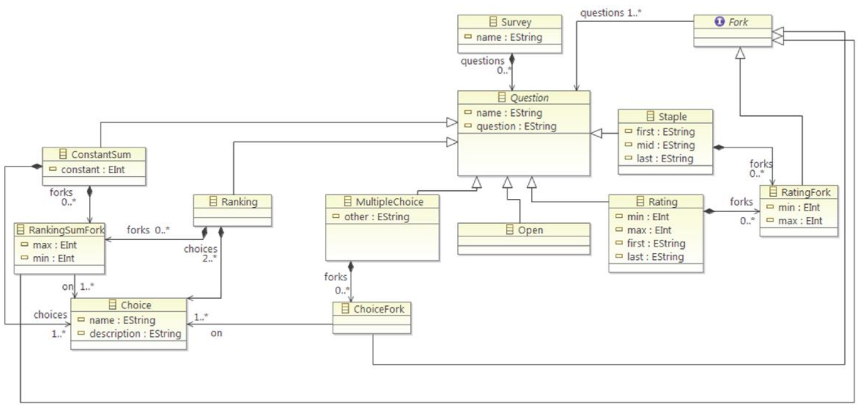


**Fig. 1.** Meta-model of our language

The attributes of each question type are explained in Appendix 1. The question types our language needs to support is open ended, multiple choice, ranking, ranking sum, staple and rating. They all share some common attributes; An identifier(*Name*), the

question they ask(*Question*) and whether it is required or not(*isRequired*). So all questions are subclasses of the **Question** class that defines these three attributes for them. Instances of **Question** is contained by by a **Survey** class to be able to create a series of questions of different types.
Each question type, except **Open**, contains an attribute named *Fork*. Each of these attributes are a subclass of the **Fork** class. The purpose of using forks is to allow branching into different paths of questions, depending on the answer. So each **Fork** subclass models one or more possible answers to the question they are contained in. If one of these modelled answers is answered, the survey will continue along the path of the questions defined in the **Fork**'s attribute **Questions**. For example for the **Rating** question a **RatingFork** can be specified; if the question is answered with a number within the range stated in **RatingFork**'s *min* and *max* the survey will continue with the questions specified in **RatingFork**'s *Questions*. Once the questions in *Questions* has been answered the survey will continue with the normal questions in the survey.

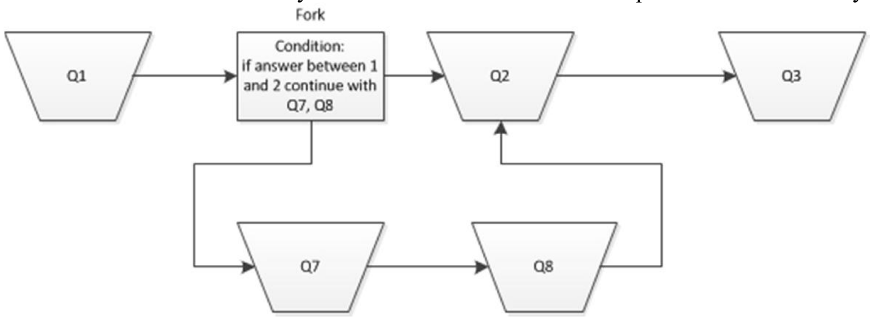


**Fig. 2.** Question flow diagram with the use of a fork

## 3.3 Design principles

The final meta-model and grammar was created over a couple of iterations, each time refining the meta-model. Each iteration was started when we noticed that our design was violating a design guideline, a *what-not-to-do* (WNTD) or if we found an error. On our first iteration of the grammar we tried applying Design Guideline 3 [7], make your language consistent. For us this meant making the internal keywords, grammar keywords and grammar layout consistent; Making the non-terminals, terminals and the language structure the same. E.g. using the non-terminal "if" in all of the forks. After our first iteration of the meta-model and grammar, our DSL was "Duplicating the concepts and semantics of traditional programming languages" [9] which is a WNTD. At that time, the **Fork** class contained a question instead of referencing it. The effect was that the user would have to nest all the questions inside a fork. At the next iteration we modified it to correctly reference **Question** creating a more flat structure in the grammar.
In the first iterations of the meta-model the Dichotomous question type was still present. However, in order to avoid conceptual redundancy [7], we decided to remove it. It is possible to create a Dichotomous question by using a Multiple Choice type of question instead.

## 3.4 Grammar and Concrete Syntax

The grammar is a core part in Ask, making it produce a readable and straight forward concrete syntax. Taking into consideration how surveys are usually written, such as in a simple text editor or having graphical languages aid in visualizing the domain, we decided to strive for a flat language.

```
TestSurvey
  MultipleChoice Q1:
    the user
    is required
    to answer "Was today a good day?"
    picking up to 1 these A : "Yes", B : "No", C : "Maybe"
    if A then continue with Q2
    if B, C then continue with Q2, Q3
  Ranking Q2:
    the user
    may rank "Favorite food"
    these  A : "Pizza", B : "Salad", C : "Cookies"
  Rating Q3:
    "Today's weather" 1 10 "Terrible" "Great"
```

**Fig. 3.** Concrete Syntax

We only allow a single survey for each file, to reduce the complexity both on the back end as well as for the user. The survey will be declared with a name, followed by the declaration of the questions. Each question will be declared by choosing the desired type such as **MultipleChoice** displayed in Figure 2 as well as assigning an identifier. Inside the body of a question we have a number of optional terminals to increase the readability of the grammar. This is instead of simply stating the question, choices and conditions in one go and increase readability. It is possible to use keywords such as the user and may, but these can be omitted. The actual question string can be prefixed with answer, rate, or rank depending on the question type. A question can be marked as mandatory by using the is required keyword.

**MultipleChoice** and **ConstantSum** will contain a comma separated list of **Choice**. Each **Choice** is defined by an identifier follow by a colon as well as the **Choice** description. Each **Choice** can be referenced in a Fork of the parenting question. Instances of **Fork** are declared with if followed by the choice identifiers, as well as a comma separated list of the **Question** identifiers that are to be referenced to. The **fork** can reference several choices and can lead to one or more questions. If the answers are within an integer range, such as in **Rating** questions, the fork will be declared with an integer range instead of choices.

The final goal of the grammar is to allow a concrete syntax that can display explicit surveys. As well as provide the option to write dense surveys by removing the terminals which are not required.

## 3.5 Use of models

With the desire of having the flat structure discussed in the grammar and syntax section, a couple challenges arose. One challenge was the default implementation of the identifiers in the *Xtext* language, where it is possible to have the same identifier for choices in the same scope. With this ambiguity, a choice in a fork will be referenced to the first occurrence of that identifier. To circumvent the ambiguity of identifiers we implement a model transformation which replaces the choices in the forks with the choices of the local question if they match.

## 3.6 Constraints.

We have to introduce a number of constraints that will help in avoiding both user introduced pitfalls as well as the language's shortcomings. These constraints include checking for unique identifiers for specific questions, checking that forks do not reference either the same choice or question twice. Additionally, looping needs to be avoided so a question cannot fork to a previously defined question, but needs to reference a later one. We also do not allow empty question or choice strings, but do allow empty strings in the bounds of **Rating** and **Staple** questions.

## 3.7 Code Generation

After completing the model transformation and passing the constraints the parsed model will be put through the code generators. Assuming at the first question of the concrete syntax in Figure 3 the output would look like the code snippet in Figure 4.

```
ArrayList<Choice> arrRankQ2 = new ArrayList<Choice>();
   arrRankQ2.add(new Choice ("A","Pizza"));
   arrRankQ2.add(new Choice ("B","Salad"));
   arrRankQ2.add(new Choice ("C","Cookies"));
  RankingQuestion Q2 = new RankingQuestion
("Q2","Favorite food",false, arrRankQ2);
questions.add(Q2);
```

**Fig. 4.** Generated android code

## 4 System Architecture

The final implementation of the architecture capitalizes on all of the previously outlined components, allowing the straightforward creation of Model-to-Text (M2T) transformations. The construction of an external DSL allowed us to capture essential knowledge in the specific domain and to have full control over the implementation, syntax and logic of the language. This made the construction of the code generators relatively a simple task due to the unambiguous language design. We decided to generate code for two back-ends: Web application and Android Mobile Platform.

One of the most crucial points before producing the code generators was to implement a "dummy" instance, depicting the desired outcome for each of the back-end platforms. This allowed us to consider different ways to generate the code and further pointed us in the right direction when it comes to easy and clean implementation of the generators due to the already gained backend knowledge. We used *Xtend* templates to create two code generators. The first generates Java class for Android. The second generates HTML and JS for a web application. In both cases our main focus was to ensure efficiency of the generators and readability of the produced back-end code.

The creation of various constraints to check the DSL instances for issues was important as well. The constraint checker is executed before the code generation and will report errors if there are problems with the validity of the DSL instance.

Code generation has various advantages that convinced us that it is the best choice for this project. Among the most important ones are code reuse, system verification and maintainability. The automation process ensures correctness of the generated code, save a lot of development hours and allow for faster time to market and overall lowers cost of the architecture in the long run [10].

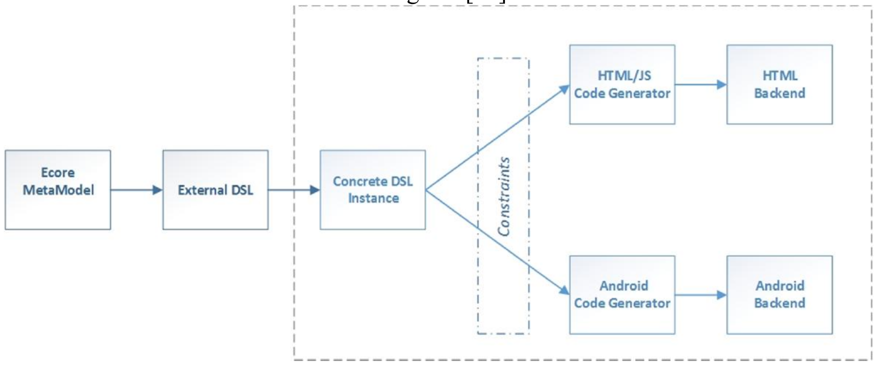


**Fig. 5.** Architecture Diagram

## 5 Testing

Ensuring the validity and the correct behavior of a system requires the implementation of testing procedures. Moreover, it is often an invaluable asset to introduce testing mechanism in the development process as early as possible[11]. We did individual component testing on each of the system parts: meta-model, external DSL and code generation.

As soon as we did the first implementation of the meta-model, we started creating dynamic instances to test its validity, see appendix [reference to tests].This helped us in finding flaws in our design. Testing our external DSL and the code generation was done by implementing concrete syntax instances and ensuring verification of the constraints. The instances and the constraint verification allowed us to consider better ways to express the language and to introduce more constraints to prevent the incorrect use of the DSL.

Once we finished implementing each of the system components we performed system testing by running the already established testing procedures in subsequent order. Due to the limited project time frame we did not perform any unit testing via a framework such as JUnit.

## 6 Conclusion

Overall the requirements of the DSL have been met through the meta-model as well as the grammar. We have avoided false referencing of choices by introducing a model transformation, as well as checking whether question names are unique. Concerns about looping, as well as issues with double references on choice and questions in forks have been resolved. The survey can both contain optional questions and also answer depending forks that allow for more specific surveys.

We have implemented two code generators for HTML and Android that generate platform specific code based on concrete instances of our external DSL. They ensure correctness of the generated code, lower development cost and overall faster time to market.

the meta-model of the designed language only has one way of expressing each type of question.
[we have done everything needed]
[worth creating a DSL or not?]
requirements met

## References

1. Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats L., Visser, E., Wacsmuth G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages, http://voelter.de/dslbook/markusvoelter-dslengineering-1.0.pdf
2. Fowler, M.: Domain Specific Languages, pp 33-102. Addison-Wesley Professional, Boston, USA (2010)
3. Sapsford, R.: Survey Research Second Edition. Sage Publications Ltd., London, United Kingdom (2006)
4. Leeuw, E. D., Hoh, J. J., Dillman, D. A.: International Handbook of Survey Methodology. The European Association of Methodology (EAM), http://www.eam-online.org
5. Campbell, A., Katona, G.: The Sample Survey: A Technique for Social Science Research. The Dryden Press, New York, USA (1953)
6. MacHenry, M., Matthews, J.: Topsl: A domain-specific language for on-line surveys. Northeastern University & University of Chicago, USA (2004)

7. Karsai, G., Rumpe, B., Krahn, H., Schindler, M., Pinkernell, C., Völkel, S.: Design Guidelines for Domain Specific Languages, IEEE Software, 2003.

8. Battey K., QuestionPro, http://www.questionpro.com

9. Kelly, S., Pohjonen, R.: Worst Practices for Domain-Specific Modeling, IEEE Computer Society (2009)

10. Bran Selic. The pragmatics of model-driven development. IEEE Software, 20(5):19–25, 2003

11. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Addison-Wesley Signature Series (Fowler))

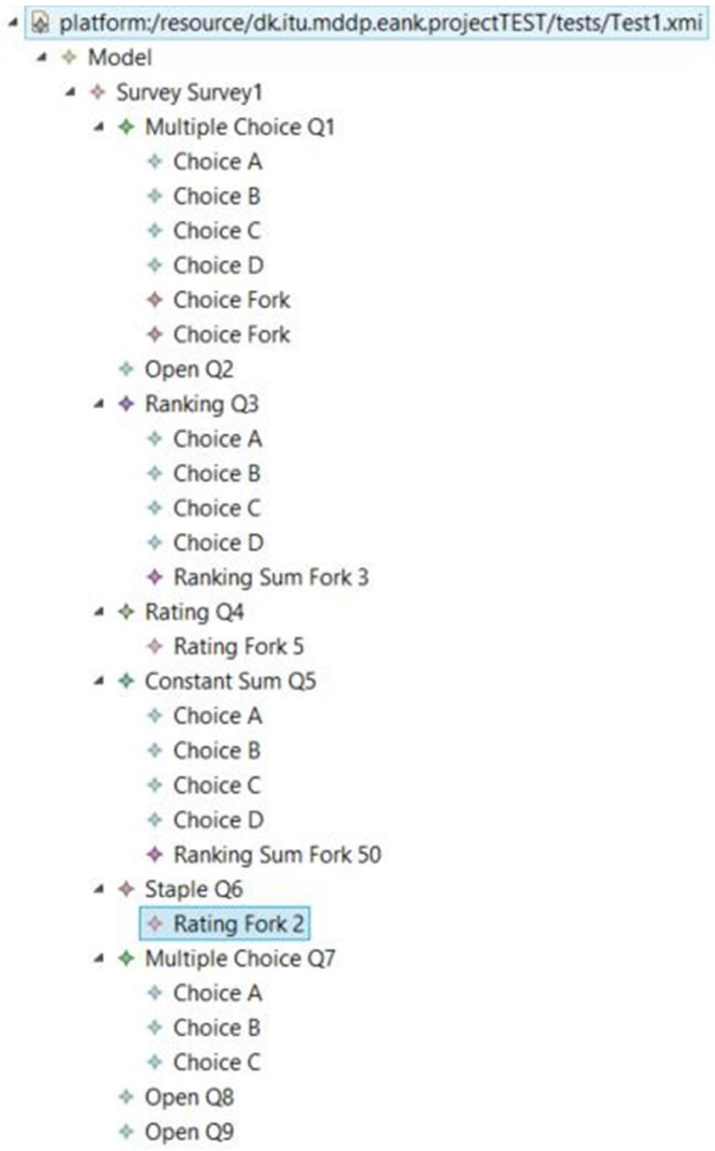
12. Implementing Domain-Specific Languages with Xtext and Xtend (Lorenzo Bettini)

**Appendices**
**Appendix 1**

| Question Sub-classes | Attributes |
|---|---|
| Ranking | *min* -The possible answers lower bound<br>*max* -The possible answers upper bound<br>*first* - A string stating what should be stated at the minimum number in the question<br>*last* - A string stating what should be stated at the maximum number in the question |
| Staple | *first* - A string stating what should be stated at the minimum number in the question<br>*last* - A string stating what should be stated at the maximum number in the question<br>*mid* - A string stating what should be stated at the middle of *min* and *max* |
| Open | |
| Multiple Choice | *other* - Whether its possible to input another choice not stated<br>*choices* - The choices that can be chosen to answer the question.<br>*answerCount* - The amount of choices that can be chosen to answer. |
| Ranking | *choices* - The choices that can be chosen to answer the question. |
| Constant Sum | *choices* - The choices that can be chosen to answer the question.<br>*constant* - The integer that the inputted numbers has to add up to. |

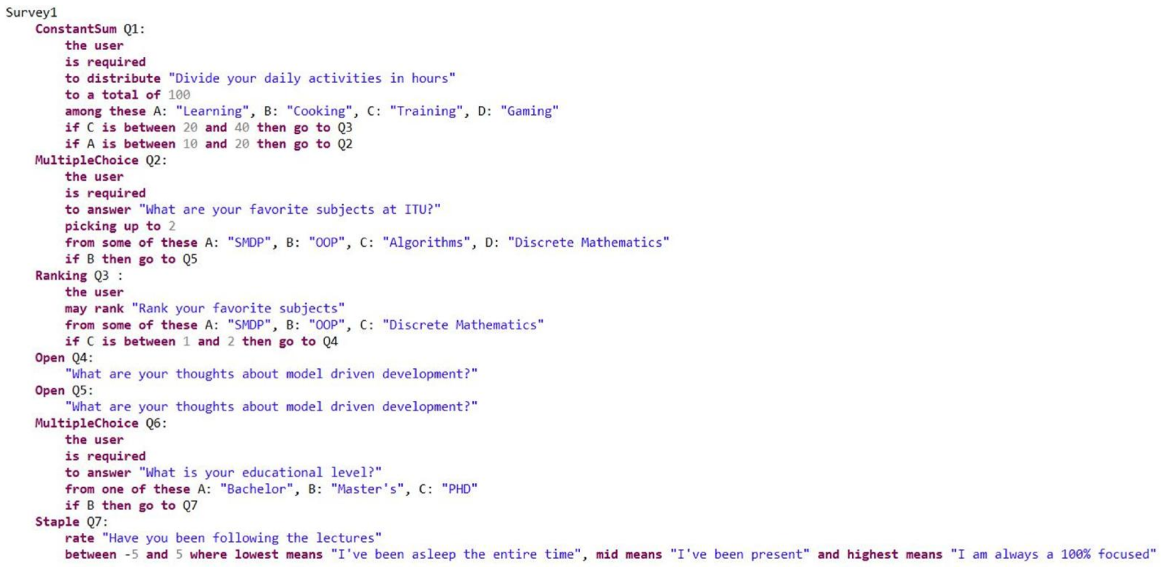
- Dynamic Instance Testing

\



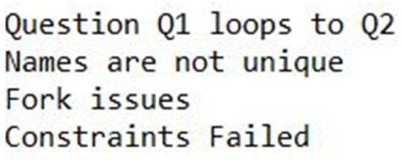
```
platform:/resource/dk.itu.mddp.eank.projectTEST/tests/Test1.xmi
  Model
    Survey Survey1
      Multiple Choice Q1
        Choice A
        Choice B
        Choice C
        Choice D
        Choice Fork
        Choice Fork
      Open Q2
      Ranking Q3
        Choice A
        Choice B
        Choice C
        Choice D
        Ranking Sum Fork 3
      Rating Q4
        Rating Fork 5
      Constant Sum Q5
        Choice A
        Choice B
        Choice C
        Choice D
        Ranking Sum Fork 50
      Staple Q6
        Rating Fork 2
      Multiple Choice Q7
        Choice A
        Choice B
        Choice C
      Open Q8
      Open Q9
```

- Concrete Syntax Instance



```
Survey1
    ConstantSum Q1:
        the user
        is required
        to distribute "Divide your daily activities in hours"
        to a total of 100
        among these A: "Learning", B: "Cooking", C: "Training", D: "Gaming"
        if C is between 20 and 40 then go to Q3
        if A is between 10 and 20 then go to Q2
    MultipleChoice Q2:
        the user
        is required
        to answer "What are your favorite subjects at ITU?"
        picking up to 2
        from some of these A: "SMDP", B: "OOP", C: "Algorithms", D: "Discrete Mathematics"
        if B then go to Q5
    Ranking Q3 :
        the user
        may rank "Rank your favorite subjects"
        from some of these A: "SMDP", B: "OOP", C: "Discrete Mathematics"
        if C is between 1 and 2 then go to Q4
    Open Q4:
        "What are your thoughts about model driven development?"
    Open Q5:
        "What are your thoughts about model driven development?"
    MultipleChoice Q6:
        the user
        is required
        to answer "What is your educational level?"
        from one of these A: "Bachelor", B: "Master's", C: "PHD"
        if B then go to Q7
    Staple Q7:
        rate "Have you been following the lectures"
        between -5 and 5 where lowest means "I've been asleep the entire time", mid means "I've been present" and highest means "I am always a 100% focused"
```

- Constraints



```
Question Q1 loops to Q2
Names are not unique
Fork issues
Constraints Failed
```

- Example of Code Gen data
  code gen data should go here

Things to keep?

Old Grammar Section:

The grammar is one of the more important parts when it comes to end user interaction with the DSL. Here it is important to have a language that is not too complicated, while still being comprehensive enough to express all the domain specific needs. One way of achieving this is to recognize the common user's proficiency with programming languages, or lack thereof. Taking into consideration how surveys are usually written, such as in a simple text editor or having graphical languages aid in visualizing the domain, we decided to strive for a flat language, which would suit the expected end user in swiftly understanding the structure.

This means that our grammar focuses on having descriptive keywords for the question type, which also aids in identifying when the declaration of a question begins, as well as when it ends with the next question type keyword. Additionally, we have added several keywords that could help in cognitive understanding of how one would formulate a question. However, these keywords are mostly optional, with a few being mandatory for overall easier understanding and reading of the concrete syntax.

Another requirement of the DSL was to allow traversal of the survey depending on previous question input, which we refer to as 'forking'. The challenge here was to avoid having nested questions, and to keep the concrete syntax flat. To handle this we allow for top level question declaration, and the ability to reference these questions in another question's fork. Finally the questions can be defined as optional by the use of or absence of the defined keywords. This means that some regular syntax would look something like this:

2.4
Design guideline 8 (Karsal et al, 4), reflect only the necessary domain concepts. When we decided on what the DSL would support of survey concepts we tried keeping in mind what was essential and what was not. In this line of thought we might have been able to cut out forking questions, but not having them would leave resulting surveys unable to respond to the users input and in turn drop the usefulness of having a DSL to create surveys.

The benefits that DSLs bring to software engineering include a more efficient development process than the General Purpose Languages (GPLs) [1]. DSLs can reduce the complexity of an application logic by replacing parts of the code using GPL with less DSL code[1].
Fowler (2010) divides DSLs into three main categories:

- external DSLs, represented by languages different and independent from the language the application works with [2]
- internal DSLs, which implies the use of a GPL and a small part of the DSL's features in order to address a small aspect of the application [2]
- the workbench of the DSL, or Integrated Development Environment (IDE), used to define, build and edit the language [2]

Compared to the capabilities of GPLs, DSLs provide limited possibilities of expressiveness. While the learning curve of GPLs is more steep than the one of DSLs due to the variety of capabilities (e.g. variety of data types support, abstraction structures), DSLs provide the minimum capabilities to support its domain [2]. Thus, a DSL is more facile to learn and more accessible than a GPL.

As a consequence of the DSLs limited expressiveness, a DSL is efficient when used on a relatively small domain [2]. Therefore, a DSL is not suitable for building large scale systems, rather to define segments of a system or small sized systems [2].
In order to start defining a DSL, the first tool a programmer needs is a language workbench, such as *Xtext*. A language workbench is used to create a meta-model and to define the grammar of the language [2]. *Xtext* is integrated with Eclipse and offers an IDE for the developed language, as well as code generation..

[**Figure 1**] illustrates the flow model of the final version of our architecture, outlining each and every component and the logical steps of implementation. We are confident that it follows outlined design principles, allowing simple to create, yet powerful instances. Furthermore, the back-end specific code can be generated within seconds and does not require any back-end knowledge, due to its plug-and-play nature.