A Domain Specific Language for Statistical Sports Data Analysis

University of Oxford

MSc Software Engineering Dissertation

A dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering

Kellogg College
University of Oxford
2 March 2017

Colin Lomas

1 Contents

2 Abstract

This dissertation is to discuss and partially implement a Domain Specific Language to aid the statistical analysis of a large set of historical sports data. It is an opportunity to use the methodologies and material from the MSc Software Engineering course, augmented by the authors work experiences in the financial high frequency trading sector. It will discuss technical, process and business requirements to conclude whether such a complex raw data set can be enveloped into a flexible and extensible set of methods which allow both simplicity of use and fullness of requirements.

The author confirms that this dissertation does not contain material previously submitted for another degree or academic award; and the work presented here is the author's own, except where otherwise stated.

3 Acknowledgments

I would like to thank all the academics at the University of Oxford who provided the interesting and challenging topics, in particular my supervisors Professor Ralf Hinze and Professor Jeremy Gibbons.

I would also like to offer my appreciation to Olwen Astley and the team at Opta Sports for taking their time to talk to me about my project and provide the sources of data and information I required to undertake this dissertation.

4 Introduction

As the need for software increases in our everyday life, and the division between technical and non-technical staff begins to blur, it is becoming increasingly important that individuals employed in the business operations can take a more active role in the design and development of their business' computer systems.

Although most business users will not acquire the skills of experienced enterprise software engineers, many will be familiar with the basic concepts of coding, whether it be through Excel macros, scripting languages such as JavaScript or using graphical tools to program their combined AV remote controls. This level of technical ability is becoming increasingly important in the modern world, from setting up home cinema through to refining personal blog sites. That this enhanced ability and knowledge should finish at the office doors seems somewhat counter-intuitive. That these semi-technical business users should need to concern themselves with no more than word documents to define their critical computer systems.

Of course, different business users will have different levels of knowledge and specific areas of skills; a technical writer may know how to build complex macros in Word whereas financial traders may have a basic level of Java programming. It is important to attempt to build on user's particular skills to advance the design and development of their business' IT ecosystem.

Historically, specification documents would be discussed, compiled and signed off, often fundamentally flawed, before being passed on to the software developer in a darkened room to work on for three months. The dawn of Agile methodologies helped decrease the feedback loop and bring developers and business users closer together. However, the coding would still always be the remit of the software developers.

Domain Specific Languages attempt to merge these responsibilities. The experienced software developer will always be the best person to work on enterprise level coding, such as database design, performance, design principles and threading, yet there is no reason that business users cannot be involved in the final piece in the puzzle; the querying of data, reporting, top level control flow.

DSL's can be aimed at any level of experience and knowledge. Indeed, some DSLs may be designed for developers themselves, masking low level memory management to give a higher-level language specific to a given task. As we will discuss in more detail, there is no concrete definition of a DSL; XSLT and SQL could be considered to fall under the DSL umbrella.

For the purpose of this dissertation however, we will be concentrating on giving some programmatic control to our business users, namely traders in a sports betting environment. Financial traders have been using similar tools for many years; some high level for graphically defining columns in their risk tools, others writing embedded C# modules to calculate prices. With the dawn of data providers such as Opta supplying reliable and extensive data from the sports industry, it is the intention of this dissertation to create a simple DSL for traders to query this data for statistical analysis. These traders will understand basic programmatic concepts but will not want, nor have the ability to, create enterprise level infrastructures to reliably query the raw underlying data. The DSL should supply a

set of commands which give the traders the flexibility to achieve their analysis goals, while at the same time be simple and clear enough to not require a long educational process.

5 Background

5.1 A Very Brief History of Financial Trading

For many years, financial trading was undertaken by traders who made their decisions based on fundamental analysis. In other words, they knew the markets. Analysts would specialise in specific sectors; energy, pharmaceuticals, food, and their knowledge and predictions would be passed to traders, who would collate the information into real world trading decisions. These trades would be communicated via telephone through to a second set of 'pit traders' at the stock exchanges who would make the physical transactions.

During the computerisation boom of the 1970s, electronic communication was used to pass information regarding trades from the market directly to the trader's desks. NASDAQ was the first electronic stock market and opened in 1971. The New York Stock Exchange released DOT in 1976 which allowed the first direct electronic trading, albeit on a single stock level. However, it wasn't until the mid to late 1990s when direct trading (straight through processing (STP)) really started to build momentum, largely thanks to standardisation of protocols (FIX protocol released in 1992), increased computing power and the availability of a new breed of computer programmers and technically-minded traders having the skillset to bring all the technology and business requirements together.

Building on its own impetus, the increased volume, liquidity and volatility allowed traders to advance their trading strategies and increase frequency of their transactions. With computer power increasing, the idea of statistical analysis became more commonplace, allowing mathematicians to replace traditional fundamental market knowledge with pure mathematical algorithms based on the historic data available from the markets. Pairs trading was a simple early example of this which still exists today; trading based on the historic correlation between two stocks, no matter how tenuous the real-world association between the companies may be. Index Arbitrage was another; the ability to calculate the real price of an index based on its composition quicker than the exchange had time to print the index price, and trading on any disparities between the two.

Trading today is almost completely electronic, automated and increasingly fired by high frequency strategies; an estimated 80% of all foreign exchange futures are down to HF [1]). Apart from isolated exceptions (London Metal Exchange for instance [2]) the pits have closed, and although fundamental analysis still has its place on the trading floors, the role of a modern trader generally requires an advanced mathematical and/or scientific academic education [3].

5.2 A Very Brief History of Sports Betting

Although there are varying, mainly unsubstantiated, reports of the ancient Greeks betting on the Olympics and the Romans betting on Chariot races, the lasting image of modern day betting lies squarely with chain-smoking men at the sides of horse racing circuits and greyhound tracks. However, bookmakers eventually realised the interest in betting in other sports, in addition to the rapid demise of greyhound racing, so now you are much more likely to see an advert for football betting than you are for horse racing. This has piqued the interest of a younger audience who may not have the intrinsic interest in horse racing as the previous generations did.

The dawn of the internet and mobile apps have taken exclusivity away from the bricks and mortar high street bookmakers and into the homes and palms of a new generation of tech-savvy gamblers. This new ability to trade instantly, and in some cases trigger-based and automated, has led to a vast array of new trading opportunities. Spread betting is now commonplace and it is now not particularly exceptional for people to bet on the outcome of such things as the sum of shirt numbers of goal-scorers in the Scandinavian lower leagues.

This naturally relates us back to the way statistical analysis has taken over much of the fundamental analysis in the financial markets. It is probably not unfair to say most of the people betting on their phones while in a public house in central London are not experts of the Norwegian Division 2, nor do they have much idea how FC Molde II are progressing since changing their coaching structure. However, a few minutes of research will allow them to calculate that FC Molde II seem to have an exceptionally high percentage of goal scorers with high shirt numbers. Who the team are playing or where they are in the league is largely irrelevant, but a specific event based upon pure statistical facts, no matter how questionable it may seem to an outside observer, is key to the bet.

In a comparable manner to which financial trading somewhat self-prophesied its need and application of more and more data, sports data is now equally hungry for historical information. As this in-depth, reliable back data increases, sports betting is becoming increasingly institutionalised, with hedge funds using it as an alternative investment strategy. This also provides a natural alternative for previously financially driven quantitative traders; whether they are trading against IBM or Derby County is immaterial, as long as the historical data is there to work with.

5.3 Data Providers

The fundamental requirement for any statistical analysis is good quality, reliable data. In the financial markets, this data is mostly created automatically. As the product prices move or trades are made, the details are recorded and available for analysis immediately for anyone with the relevant licences; Bloomberg and Reuters are typical examples of providers of this data. For data that is not necessarily automatically recorded, static data such as index compositions, there are a number of companies who collate this information from the various sources into an easy to use, reliable feed. Traders can then combine the static and tick data together to provide a good source of back data for analysis.

Sports data is very different. None of the data regarding sports events is centrally recorded anywhere. This requires a data provider to collate all this information into a single source. When we think of sports data, we may initially think of the basics; scores, goals, goal-scorers, etc. However, some providers now record much deeper information such as the X, Y and Z co-ordinates of the ball when passed and received. The only current way to achieve this level of data is to manually record it, and data providers such as Opta (see section below) have teams of football experts watching and recording every single event while the game is being played.

Statistical data is expensive, and given the effort required to collate it, it is clear why this is the case. However, without this, it is impossible to even begin contemplating any trading strategy which is based on sound statistical analysis.

5.4 Opta Sports

Although we will discuss the abstraction of data sources within upcoming sections, this dissertation will concentrate on a particular data provider called Opta Sports. Opta collect more sports data than any other company and have incredibly detailed analysis of all events, which make it an excellent starting point for our project.

6 The Problem Context

6.1 What we are Trying to Achieve

One of the very first points to consider when designing any system are the user requirements. In short, we want to be able to create a way of querying the Opta data in a way in which traders can easily manipulate it without having to have any great software engineering or data science knowledge. This leaves them free to spend their time on their primary focus; to analyse the data and recognise any patterns which will allow them to make positive trading outcomes, whether that is manually analysing the results or pushing the results into statistical programmes such as MatLab, R, etc. The interface should be simple to use yet rich enough in functionality to give them flexibility around the groups of data they can retrieve.

One issue we must also contend with is that Opta's primary income stream is concerned with selling rights to their data. If we want to be able to roll this system out to several non-executive business units then it means that we need to keep a physical barrier between the users and raw underlying data. To achieve this, we need to create a layer of user interaction which is abstracted away from the data source but still allowing flexible, albeit controlled, reporting on it.

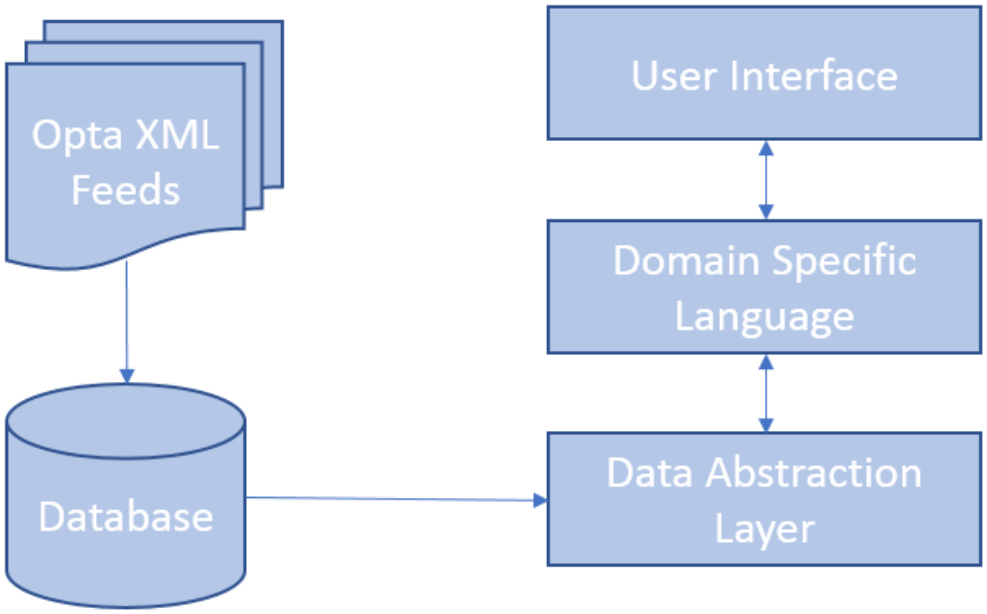
The solution here is to create a domain specific language (DSL) to provide this abstracted layer. This should allow us to give the users controlled access to the data and manage all error reporting, performance and extensibility within the control of our system, taking any systematic headaches away from the users.

One of the primary concerns here will be the compromise between the simplicity of interface and the flexibility and depth of functionality. It is likely the users will want to be able to have very high elasticity when it comes to the queries they can supply, and that could mean a large set of complex verbs for the DSL to provide. We will take a couple of examples and implement them, which will give us a good idea of just how viable this kind of project would be.

The project will therefore be split up into the following two areas of focus:

- Taking the Opta XML data files and inserting them into a database of our design
- Writing the DSL to communicate with the underlying data through a data abstraction layer

This can be seen diagrammatically below.



For the scope here, the user interface here will simply be a command line interface to the grammar of the DSL.

7 Preparing the Data

7.1 The Opta Data

7.1.1 Overview

The Opta data is supplied primarily as XML files, with some additional static data available through the appendix on the Opta website. There are a number of information sources but the three we are going to concentrate on for this dissertation are as follows:

- F40 – Player and Team information. This XML feed details the players involved in the match, along with the team they are representing.
- F24 – Event Details. This XML feed details all of the events for a given match.
- Static Event and Qualifier data. This information is available on the website and gives textual descriptions of all events and their related qualifiers, the relationship of which will be examined in the next section.

7.1.2 F40 Team/Player Feed

The F40 feed details the teams and players which will be referenced in the event feeds. A small example as follows:

```
<SoccerFeed timestamp="20131128T111924+0000">
  <SoccerDocument Type="SQUADS Latest" competition_code="EN_PR" competition_id="8"
competition_name="English Barclays Premier League" season_id="2013" season_name="Season
2013/2014">
    <Team city="London" country="England" country_id="1" postal_code="N5 1BU" region_id="17"
region_name="Europe" short_club_name="Arsenal" street="75 Drayton Park" uID="3"
web_address="http://www.arsenal.com">
      <Founded>1886</Founded>
      <Name>Arsenal</Name>
      <Player uID="59936">
        <Name>Wojciech aSzczesny</Name>
        <Position>Goalkeeper</Position>
        <Stat Type="first_name">Wojciech</Stat>
        <Stat Type="last_name">Szczesny</Stat>
        <Stat Type="birth_date">1990-04-18</Stat>
        <Stat Type="birth_place">Warszawa</Stat>
        <Stat Type="first_nationality">Poland</Stat>
        <Stat Type="weight">84</Stat>
        <Stat Type="height">196</Stat>
        <Stat Type="jersey_num">1</Stat>
        <Stat Type="real_position">Goalkeeper</Stat>
        <Stat Type="real_position_side">Unknown</Stat>
        <Stat Type="join_date">2008-07-01</Stat>
        <Stat Type="country">Poland</Stat>
      </Player>
      <Player loan="1" uID="17746">
        <Name>Emiliano Viviano</Name>
        <Position>Goalkeeper</Position> ….
```

Most of the data is self-explanatory so we will not go into too much detail, only to mention that:

- *SoccerFeed* is the root element for the entire document
- *SoccerDocument* gives the league or competition the subsequent teams are part of
- *Team* contains a number of attributes for a given team; unique ID (*uID*), name, address, etc and contains multiple *Player* elements
- *Player* elements contain information on the individual players

7.1.3 F24 Event Details Feed

The F24 feed is the document in which the main information regarding match events is provided. An sample as follows:

```
<Game id="695019" away_team_id="8" away_team_name="Chelsea" competition_id="8"
competition_name="English Barclays Premier League" game_date="2013-11-23T17:30:00"
home_team_id="21" home_team_name="West Ham United" matchday="12" period_1_start="2013-11-
23T17:30:27" period_2_start="2013-11-23T18:33:30" season_id="2013" season_name="Season
2013/2014">

    <Event id="1796423032" event_id="5" type_id="1" period_id="1" min="0" sec="5"
player_id="49413" team_id="21" outcome="0" x="32.8" y="58.1" timestamp="2013-11-
23T17:30:33.141" last_modified="2013-11-23T17:31:20">
      <Q id="319787581" qualifier_id="213" value="0.2" />
      <Q id="1003515826" qualifier_id="1" />
      <Q id="139030201" qualifier_id="140" value="74.1" />
      <Q id="1066858436" qualifier_id="212" value="44.7" />
      <Q id="1724945974" qualifier_id="157" />
      <Q id="225210807" qualifier_id="56" value="Center" />
      <Q id="1096438506" qualifier_id="141" value="73.9" />
    </Event>
```

This does require some explanation as this is the crux of the essential data the users will be querying.

- The *Game* element contains the basic information for each match, most importantly including the home and away teams and competition. Note that each team ID is supplied, which relates back to the F40 feed, but also includes the team name, presumably for human eye clarity.
- Each match element will include many *Event* elements. An *Event* can relate to any piece of information or event within the game that can be reported. This includes the team line-ups, corners, long balls and red cards, through to referee replacements and injury time announcements. These events are related to static data by the *type_id*. Note the *event_id* is actually a sequential ID for the event in the match, the *type_id* is specific to the type of event. In the example above *type_id* 1 relates to '*Pass*' for Team 21 (West Ham), player 49413 (Jamie Tomkins) in the fifth second of the first minute, an outcome of 0 (Unsuccessful) at pitch co-ordinates of x 32.8 and y 58.1.
- Each *Event* element can have a number of qualifiers. These qualifiers are specific to the reported event and are contained in *Q* elements within the parent *Event* element. Each qualifier may or may not have multiple related values, depending on the event type. The above example having the following qualifiers and values:
    - 213 – Angle the ball travelled in radians (value = 0.2)
    - 1 – Long Ball (pass over 32 metres)
    - 140 – The X pitch co-ordinate for the end of the pass (74.1)
    - 212 – Length of pass in metres (44.7)
    - 157 – Launch (Pass played up towards front players. Aimed to hit a zone rather than a specific player
    - 56 – Zone of pass (Centre)
    - 141 - The X pitch co-ordinate for the end of the pass (73.9)

This small example gives a excellent insight into the detail and scale of the information reported on each element of the match. It also indicates the amount of data that would be acquired over a

number of months, given that Opta report on almost every match in the top leagues throughout the world.

7.1.4 Static Data

As previously mentioned, there are various pieces of static data to be considered; mainly descriptive event and qualifier information. These are not currently available as an XML feed so to use these within a subsequent system, the information would have to be manually (or at least semi-automatically) scraped from Opta's appendices webpage.

7.2 Database Design

7.2.1 Overview

As we have previously mentioned, for the information in the XML document to be useful for querying in a performant manner, it should be transferred into a database.

7.2.2 Database Schema

To begin with, we shall look at the structure of the database schema. As the information contained within the XML documents is already well normalised and have sensible parent/child relationships, it makes sense to mimic this relational model in the database.

Firstly, let's put together a set of relational variables to make up the design of our database, along with basic foreign key constraints. Note that not all the fields from the original XML documents are being included – this is just for brevity in the project. In the real world, there would be no reason to leave out any information:

Team: Holds information for each team in the system

Team
teamID : TeamID name : String founded : String

Player: Holds information regarding the players. This includes a foreign key constraint to their relevant current team

Player
playerID : PlayerID teamID : TeamID name : String position : String

(Player, Team) $\mapsto$ {teamID $\mapsto$ teamID} $\in$ FK

Event: Holds information on the static data of each event (*Pass, Foul, Save*, etc). Each event has a name ("*Pass*") and a longer description ("*Any pass attempted from one player to another*"), information taken directly from Opta's appendices.

Event
eventID : EventID name : String description : String

Qualifier: Holds information on the qualifiers for events. A qualifier can relate to many different events ("*End X Co-Ord*" could relate to pass, free kick, save, etc) which is why a direct foreign key constraint in not included here. A mapping table (*QualifiersForEvent* (see below)) is added to facilitate this. The *Value* field is simply a textual descriptor with specific information regarding the associated values. Again, this information is taken directly from Opta's appendices.

Qualifier
qualifierID : QualifierID name : String value : String description : String

QualifiersForEvents: Mapping table to relate *Qualifiers* to their relevant *Events*. Includes two foreign keys; to *Event* and *Qualifier*

QualifiersForEvents
qualifierForEventID : QualifierForEventID eventID : EventID qualiferID : QualifierID

(QualifiersForEvents, Event) $\mapsto$ {eventID $\mapsto$ eventID} $\in \mathrm{FK}$
(QualifiersForEvents, Qualifier) $\mapsto$ {qualifierID $\mapsto$ qualifierID} $\in \mathrm{FK}$

Game: Main relational variable for a match. Has two foreign keys back to the *Team* variable, one for each team competing

Game
gameID : gameID homeTeamID : TeamID awayTeamID : TeamID

(Game, Team) $\mapsto$ {homeTeamID $\mapsto$ teamID} $\in$ FK
(Game, Team) $\mapsto$ {awayTeamID $\mapsto$ teamID} $\in$ FK

GameEvent: Holds information regarding all of the events reported for a game and contains several foreign keys back to *Game, Event* and *Team.* Note that *periodID* would generally relate to a mapping table but in this instance, we will simply record the value

GameEvent
gameEventID : GameEventID gameID : GameID eventID : EventID playerID : PlayerID periodID : N minute : N second : N teamID : TeamID outcome : N x : N y : N

(GameEvent, Game) $\mapsto$ {gameID $\mapsto$ gameID} $\in$ FK
(GameEvent, Event) $\mapsto$ {eventID $\mapsto$ eventID} $\in$ FK
(GameEvent, Team) $\mapsto$ {teamID $\mapsto$ teamID} $\in$ FK

GameEventQualifier: Holds all the qualifiers for a given event in a given game. For an *Event* which contains a single value then one record would be contained in here. For any *Qualifiers* which have more than one *Value* (*Team Line-up* for instance, which has a separate qualifier for each of the players on the pitch) multiple records would be held here for the *Event*

GameEventQualifier
gameEventQualifierID : GameEventQualifierID gameEventID : GameEventID gameQualiferID : GameQualifierID Value : String

(GameEventQualifier, GameEvent) $\mapsto$ {gameEventID $\mapsto$ gameEventID} $\in$ FK
(GameEventQualifier, GameQualifier) $\mapsto$ {gameQualifierID $\mapsto$ gameQualifierID} $\in$ FK

7.2.3 Normal Forms

It's worth having a short note regarding the normal form of our relational model here, and whether it adheres to the rules of the three primary areas.

First Normal Form

First normal form states that a single field cannot be allowed to contain multiple values, that it must be atomic. It's difficult to break this rule with a data model in general as the design simply can't be sensibly created. A theoretical example would be if the *Player* variable was merged into the *Team* variable, creating something as follows:

TeamID	**Name**	**Player**
1	Derby County	Will Hughes
		Tom Ince
2	Manchester United	Wayne Rooney
		Juan Mata

Each of the above *Teams* contain a non-atomic *Player* field which breaks 1NF. The data model we have created so far does not have such issues.

Second Normal Form

Second normal form states that no non-prime attribute is dependent on any proper subset of any candidate key.

In our data model, we don't have any relations that could fall into this trap. However, if we imagine a real world scenario where a player will have records for different clubs over time, then our data model would suddenly become problematic as the following scenario could occur:

PlayerID	**Name**	**TeamID**	**Position**
1	Bryan Robson	5	Midfield
1	Bryan Robson	10	Midfield

Because *Bryan Robson* moved from one club to another, suddenly we find that there would be two records in the *Player* relational variable. This would break second normal form. To fix this, a new variable would need to exist containing all the clubs a single player has played for. This would then make *Position* dependent solely on the candidate key *PlayerID* again.

Third Normal Form

For a relational model to adhere to third normal form it must not contain any data which is transiently dependant on any superkey.

Let's imagine that the *periodID* mentioned earlier in the *GameEvent* relational variable had an associated description in our model. If the description was added to the *GameEvent* variable, it would produce something like the following (some fields left out for conciseness):

GameEventID	**GameID**	**EventID**	**PeriodID**	**PeriodDescription**
1	1	1	1	First Half
1	1	2	1	First Half
1	1	3	2	Second Half

From this, the *PeriodDescription* field is a transient attribute of *PeriodID* and therefore introduces a redundancy. To fix this, the *Period* description would be pulled out to a separate *Period* relational variable and referenced at an atomic level.

7.2.4 Relations

A good way of checking the validity of a relational model is to produce one or more real-world relations against the model. An obvious candidate here is to select all qualifiers for a given event within a given game, relating all the relevant relational variable in the process.

It is possible to show this in either relational algebra or relational calculus. Below is the relational algebra solution. Firstly project the required fields from the cartisan product of the relevant relational variable into A. Then get the highest *GameID* from A, then restrict the results from A into C by only the highest *GameID* stored in B.

A == Project {GameID, EventID, QualifierID, Value} (Join GameEventQualifier, GameEvent, Game, Team, Team, Player, Event, Qualifier)
B == A Group (Max(GameID) as LatestGameID)
C == Restrict {GameID = LatestGameID} A

For the sake of completeness, the corresponding SQL query would be as follows:

```
SELECT          g.GameID, ge.EventID, geq.QualifierID, geq.Value
FROM            GameEventQualifier geq
INNER JOIN      GameEvent ge    ON ge.GameEventID = geq.GameEventID
INNER JOIN      Game g          ON g.GameID = ge.GameID
INNER JOIN      Team t1         ON t1.TeamID = g.HomeTeamID
INNER JOIN      Team t2         ON t2.TeamID = g.AwayTeamID
INNER JOIN      Player p        ON p.PlayerID = ge.PlayerID
INNER JOIN      Event e         ON e.EventID = ge.EventID
INNER JOIN      Qualifier q     ON q.QualifierID = geq.QualifierID
WHERE           g.GameID = (SELECT MAX(g.GameID)
FROM            GameEventQualifier geq
        INNER JOIN      GameEvent ge            ON ge.GameEventID = geq.GameEventID
        INNER JOIN      Game g                  ON g.GameID = ge.GameID
        INNER JOIN      Team t1                 ON t1.TeamID = g.HomeTeamID
        INNER JOIN      Team t2                 ON t2.TeamID = g.AwayTeamID
        INNER JOIN      Player p                ON p.PlayerID = ge.PlayerID
        INNER JOIN      Event e                 ON e.EventID = ge.EventID
        INNER JOIN      Qualifier q             ON q.QualifierID = geq.QualifierID )
```

7.2.5 CRISP-DM

Finally in the database section of this document, we should briefly look at the Cross Industry Standard Process for Data Mining (CRISP-DM) model, which will help us evaluate the approach we have taken so far.

The CRISP-DM Process suggests six steps to achieve efficient data mining:

1. Business Understanding
2. Data Understanding
3. Data Preparation
4. Modelling
5. Evaluation
6. Deployment

As we have designed the database schema ourselves, the Data Understanding part is a known. Due to the relational model chosen here, all Data Preparation, Modelling and Evaluation can be achieved by designing, running and evaluating SQL commands on the relational model.

The business understanding should be clear at this stage but let's choose the main requirements of the system and ascertain whether our model stands up to evaluation:

- Users require the ability to query on all parts of the underlying data, albeit in an abstracted, controlled manner, with the potential of grouping and aggregate functions. As the schema has been designed to closely mimic the structure of the original XML documents, SQL commands should allow us to return any combination of data available in the raw data set
- The system should be able to control the users level of querying as to not violate any business rules Opta set regarding intellectual property. This is not quite so obvious in the data model itself as, with knowledge of the underlying schema, users with direct access to the database would be able to pull out all of the raw data. However, this level of control and permissions will be regulated by the DSL and controlled access to the database.

This leaves us with Deployment. The deployment of the data would be controlled by jobs; firstly the initial loading of back data as described in previous sections, and secondly with incremental jobs to append new matches as and when required.

7.3 Populating the Database from the Opta XML Feeds

7.3.1 Overview

Now that we have our database model, the next stage is to populate it with the information held in the Opta XML documents. As previously mentioned we are primarily concerned with three areas of information; Player and Team data (F40 feed), Game Event data (F24 feed) and some static data. Over the next few sections we will discuss the options for implementing this along with the details of the chosen solution

7.3.2 Options for Data Population

To insert the information from the XML document into SQL Server, we have many options. In a real-life .NET situation, it is most likely that we would use a library within .NET itself (or a function rich 3rd party library) to bring the information across. This would give us good error handling and a reliable set of functions to use, bringing the amount of code we would have to write ourselves down to a minimum. However, as this project is about displaying the topics we have learnt on the course, I have decided to do this directly in XML, using the XSLT and XSD functions available to pure XML processing.

Within SQL Server there is a package called SQLXML which takes an XML document, along with a corresponding XSD from the command line and inserts into the database. This is how the data will be transferred into our data source.

7.3.3 Inserting Static Data

As previously mentioned, there is some data which is not available on the feeds but via the Opta website. Although there are possibilities of automatically scraping this from the site and having a process to pick this up and insert into the database, this is mostly out of scope of what we are trying to achieve here. This document won't go into any more detail about this data above saying that the information was manually picked up from the web site and inserted by hand in the database.

7.3.4 SQLXML Limitations

Although SQLXML is an excellent tool for inserting XML data into SQL Server, it has one serious drawback. SQLMXL will only take a flat XML structure, it cannot deal with any nested parent/child elements. As the Opta Xml structure is designed specifically as a hierarchical document, this means that we need to translate the original XML document into something that SQLXML can work with.

SQLXML also requires an XSD document to detail the mapping between the XML document and SQL Server tables it needs to map to.

The next sections show how we can transform the original XML documents into something SQLXML can work with and the XSD we would need to supply to map the relevant fields to the database

7.3.5 Team and Player (F40) Population

Let's look once again at an example of the F40 feed:

```
<SoccerFeed timestamp="20131128T111924+0000">
```

```
  <SoccerDocument Type="SQUADS Latest" competition_code="EN_PR" competition_id="8"
competition_name="English Barclays Premier League" season_id="2013" season_name="Season
2013/2014">
    <Team city="London" country="England" country_id="1" postal_code="N5 1BU" region_id="17"
region_name="Europe" short_club_name="Arsenal" street="75 Drayton Park" uID="3"
web_address="http://www.arsenal.com">
      <Founded>1886</Founded>
      <Name>Arsenal</Name>
      <Player uID="59936">
        <Name>Wojciech aSzczesny</Name>
        <Position>Goalkeeper</Position>
        <Stat Type="first_name">Wojciech</Stat>
        <Stat Type="last_name">Szczesny</Stat>
        <Stat Type="birth_date">1990-04-18</Stat>
        <Stat Type="birth_place">Warszawa</Stat>
        <Stat Type="first_nationality">Poland</Stat>
        <Stat Type="weight">84</Stat>
        <Stat Type="height">196</Stat>
        <Stat Type="jersey_num">1</Stat>
        <Stat Type="real_position">Goalkeeper</Stat>
        <Stat Type="real_position_side">Unknown</Stat>
        <Stat Type="join_date">2008-07-01</Stat>
        <Stat Type="country">Poland</Stat>
      </Player>
      <Player loan="1" uID="17746">
        <Name>Emiliano Viviano</Name>
        <Position>Goalkeeper</Position>
        <Stat Type="first_name">Emiliano</Stat>
```

As this document structure is too complex for SQLXML to work with, we need to apply some transformations to the document to bring it closer to the following:

```
<Teams>
  <Team>
    <ID>3</ID>
    <Name>Arsenal</Name>
    <Founded>1886</Founded>
  </Team>
  <Player>
    <TeamID>3</TeamID>
    <ID>59936</ID>
    <Name>Wojciech aSzczesny</Name>
    <Position>Goalkeeper</Position>
  </Player>
  <Player>
    <TeamID>3</TeamID>
    <ID>17746</ID>
    <Name>Emiliano Viviano</Name>
    <Position>Goalkeeper</Position>
  </Player>
```

This gives us the same information but allows SQLXML to work with a *Player* element with the TeamIDs provided as sub-elements, relating to *Team* elements which live at the same level as *Player*, rather than having to work with elemental ancestry.

To achieve this, an XSLT document can be created to transform the source into the destination.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:msxsl="urn:schemas-microsoft-com:xslt" exclude-result-prefixes="msxsl"
    xmlns:xalan="http://xml.apache.org/xalan">
    <xsl:output method="xml" indent="yes"/>
  <xsl:strip-space elements="*"/>

  <xsl:template match="SoccerDocument">

    <xsl:element name="Teams">
      <xsl:apply-templates select="Team"/>
    </xsl:element>
  </xsl:template>

  <xsl:template match="Team">
    <xsl:element name="Team">
```

```
      <xsl:element name="ID" >
        <xsl:value-of select="@uID"/>
      </xsl:element>
      <xsl:element name="Name" >
        <xsl:value-of select="Name"/>
      </xsl:element>
      <xsl:element name="Founded" >
        <xsl:value-of select="Founded"/>
      </xsl:element>
    </xsl:element>
    <xsl:apply-templates select="Player"/>
  </xsl:template>

  <xsl:template match="Player">
    <xsl:element name="Player">
      <xsl:element name="TeamID" >
        <xsl:value-of select="../@uID"/>
      </xsl:element>
      <xsl:element name="ID" >
        <xsl:value-of select="@uID"/>
      </xsl:element>
      <xsl:element name="Name" >
        <xsl:value-of select="Name"/>
      </xsl:element>
      <xsl:element name="Position" >
        <xsl:value-of select="Position"/>
      </xsl:element>
    </xsl:element>
  </xsl:template>

</xsl:stylesheet>
```

The *Team* and *Player* facets are split up into two separate templates for both clarity and reusability. XSLT sheets can quickly become complex and difficult to read so the more we can split the transformation into logical blocks, the better for both extending later and helping anyone else who may need to work on it further down the line. Note also that because the document is being flattened we need to include all relevant IDs within the element itself, so for the *Player* element for example, we need to include a *TeamID* as it has no direct hierarchical relationship to a corresponding *Team* element.

Running this against the Opta F40 document gives us exactly what we need:

```
<Teams>
  <Team>
    <ID>3</ID>
    <Name>Arsenal</Name>
    <Founded>1886</Founded>
  </Team>
  <Player>
    <TeamID>3</TeamID>
    <ID>59936</ID>
    <Name>Wojciech aSzczesny</Name>
    <Position>Goalkeeper</Position>
  </Player>
  <Player>
    <TeamID>3</TeamID>
    <ID>17746</ID>
    <Name>Emiliano Viviano</Name>
    <Position>Goalkeeper</Position>
  </Player>
  <Player>
    <TeamID>3</TeamID>
    <ID>37096</ID>
    <Name>Lukasz Fabianski</Name>
    <Position>Goalkeeper</Position>
  </Player>…
```

Next, we need to create the XSD document to allow SQLXML to validate the XML document and to work out the mapping between XML and SQL Server schema:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:sql = "urn:schemas-microsoft-
com:mapping-schema">
  <xsd:element name = "Teams" sql:is-constant = "1" >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name = "Team" type="Team" sql:relation = "Team" maxOccurs ="unbounded" />
        <xsd:element name = "Player" type="Player" sql:relation = "Player" maxOccurs =
"unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="Team">
    <xsd:sequence>
      <xsd:element name = "Name" type = "xsd:string" sql:field = "Name" />
      <xsd:element name = "ID" type = "xsd:integer" sql:field = "TeamID" />
      <xsd:element name = "Founded" type = "xsd:integer" sql:field = "Founded" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Player">
    <xsd:sequence>
      <xsd:element name = "Name" type = "xsd:string" sql:field = "Name" />
      <xsd:element name = "ID" type = "xsd:integer" sql:field = "PlayerID" />
      <xsd:element name = "TeamID" type = "xsd:integer" sql:field = "TeamID" />
      <xsd:element name = "Position" type = "xsd:string" sql:field = "Position" />
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

As well as validating the incoming XML document, the XSD document also includes information regarding mapping the elements within the document to the database tables and columns using the *sql:relation* and *sql:field* types. *Teams* is the root element and contains a sequence of *Team* and *Player* elements.

We use the Venetian Blind design for the document which allows each logical section (in our case, the three main elements; root element *Teams, Team and Player)* to be dealt with separately. This gives good reuse and a clear representation of the functionality. As our document is relatively small it could be argued a Russian Doll design could be used, whereby all of the elements are put into a single component. However, for the small overhead of breaking these we gain greater extensibility for any future additions or modifications.

Running a quick Powershell script as follows:

```
     $objBL = new-object -comobject 'SQLXMLBulkLoad.SQLXMLBulkLoad'
       $objBL.ConnectionString = 'provider = SQLOLEDB;data
source=COLIN-PC;database=Opta; integrated security = SSPI'
       $objBL.ErrorLogFile ='D:\XSLTF40Importerror.log'
       $objBL.Execute('D:\F40.xsd','D:\FlatternF40.xml')
       $objBL = $null
```

And the data is now available in the database:

```
SELECT      *
FROM        Team t
INNER JOIN  Player p on p.TeamID = t.TeamID
WHERE       t.TeamID = 3
```

100 %

Results | Messages

	TeamID	Name	Founded	PlayerID	Name	Position	TeamID
1	3	Arsenal	1886	8597	Tomas Rosicky	Midfielder	3
2	3	Arsenal	1886	8758	Mikel Arteta	Midfielder	3
3	3	Arsenal	1886	15943	Thomas Vermaelen	Defender	3
4	3	Arsenal	1886	17127	Per Mertesacker	Defender	3
5	3	Arsenal	1886	17733	Lukas Podolski	Forward	3
6	3	Arsenal	1886	17746	Emiliano Viviano	Goalkeeper	3
7	3	Arsenal	1886	18155	Mathieu Flamini	Midfielder	3
8	3	Arsenal	1886	19524	Santiago Cazorla	Midfielder	3
9	3	Arsenal	1886	20467	Theo Walcott	Midfielder	3
10	3	Arsenal	1886	27697	Nicklas Bendtner	Forward	3
11	3	Arsenal	1886	28566	Vassiriki Abou Diaby	Midfielder	3
12	3	Arsenal	1886	36968	Park Chu-Young	Forward	3
13	3	Arsenal	1886	37096	Lukasz Fabianski	Goalkeeper	3
14	3	Arsenal	1886	37605	Mesut Özil	Midfielder	3
15	3	Arsenal	1886	37748	Bacary Sagna	Defender	3
16	3	Arsenal	1886	38411	Nacho Monreal	Defender	3
17	3	Arsenal	1886	41792	Aaron Ramsey	Midfielder	3
18	3	Arsenal	1886	42427	Kieran Gibbs	Defender	3
19	3	Arsenal	1886	44346	Olivier Giroud	Forward	3

7.3.6 Game Event (F24) Population

The method for populating the database uses the same approach as for the F40 feed above. Looking at the original F24 example document, we have the same issue regarding multiple nested elements. For SQLXML to process this we need to flatten it out.

The original document is as follows:

```
<Games timestamp="2013-11-23T19:38:35">
  <Game id="695019" away_team_id="8" away_team_name="Chelsea" competition_id="8"
competition_name="English Barclays Premier League" game_date="2013-11-23T17:30:00"
home_team_id="21" home_team_name="West Ham United" matchday="12" period_1_start="2013-11-
23T17:30:27" period_2_start="2013-11-23T18:33:30" season_id="2013" season_name="Season
2013/2014">
    <Event id="330261085" event_id="1" type_id="34" period_id="16" min="0" sec="0" team_id="8"
outcome="1" x="0.0" y="0.0" timestamp="2013-11-23T16:37:53.628" last_modified="2013-11-
23T19:02:00">
      <Q id="410900332" qualifier_id="30" value="11334, 41135, 41328, 28495, 19419, 1718,
53392, 2051, 8438, 61262, 42786, 3785, 8442, 43670, 66842, 47412, 47431, 1827" />
      <Q id="1587667851" qualifier_id="194" value="1718" />
      <Q id="965417577" qualifier_id="44" value="1, 2, 2, 3, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, 5,
5, 5, 5" />
      <Q id="123726380" qualifier_id="131" value="1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 0, 0, 0,
0, 0, 0, 0" />
      <Q id="986536405" qualifier_id="59" value="1, 2, 28, 12, 24, 26, 7, 8, 29, 11, 17, 3, 5,
10, 14, 19, 22, 23" />
```

```
      <Q id="1387156569" qualifier_id="227" value="0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0" />
      <Q id="1993752334" qualifier_id="130" value="4" />
      <Q id="776522032" qualifier_id="197" value="404" />
    </Event>
    <Event   id="1498953578"   event_id="1"   type_id="34"   period_id="16"   min="0"   sec="0"
team_id="21"     outcome="1"     x="0.0"     y="0.0"     timestamp="2013-11-23T16:39:44.664"
last_modified="2013-11-23T17:31:11">
      <Q  id="1863063753"  qualifier_id="30"  value="1344,  10356,  19575,  18073,  8380,  49413,
2060, 49414, 5306, 90518, 12002, 5609, 18818, 42954, 60706, 3289, 28147, 12799" />
      <Q id="507092747" qualifier_id="131" value="1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 0, 0, 0,
0, 0, 0, 0" />
      <Q id="262221875" qualifier_id="227" value="0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0" />
      <Q id="1320194006" qualifier_id="194" value="5306" />
      <Q id="866232969" qualifier_id="130" value="8" />
      <Q id="1427015220" qualifier_id="59" value="22, 20, 17, 16, 19, 5, 26, 10, 4, 15, 23, 3,
7, 11, 13, 14, 21, 24" />
      <Q id="719838131" qualifier_id="44" value="1, 2, 2, 3, 2, 2, 3, 3, 4, 3, 3, 5, 5, 5, 5,
5, 5, 5" />
    </Event>
    <Event id="581712290" event_id="2" type_id="32" period_id="1" min="0" sec="0" team_id="8"
…
```

So we need an XSLT document to transform this into something which SQLXML can work with. The below shows a sample of this:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:msxsl="urn:schemas-microsoft-com:xslt" exclude-result-prefixes="msxsl"
    xmlns:xalan="http://xml.apache.org/xalan"
>
  <xsl:output method="xml" indent="yes"/>
  <xsl:strip-space elements="*"/>

  <xsl:template match="Games">
    <xsl:element name="Games">
      <xsl:apply-templates select="Game"/>
    </xsl:element>
  </xsl:template>

  <xsl:template match="Game">
    <xsl:element name="Game">
      <xsl:element name="ID" >
        <xsl:value-of select="@id"/>
      </xsl:element>
      <xsl:element name="HomeTeamID" >
        <xsl:value-of select="@home_team_id"/>
      </xsl:element>
      <xsl:element name="AwayTeamID" >
        <xsl:value-of select="@away_team_id"/>
      </xsl:element>
    </xsl:element>
    <xsl:apply-templates select="Event"/>
  </xsl:template>

  <xsl:template match="Event">
    <xsl:element name="Event">
      <xsl:element name="GameEventID" >
        <xsl:value-of select="@id"/>
      </xsl:element>
…
  </xsl:template>
</xsl:stylesheet>
```

Applying this to the original F24 XML document gives us a XML document in line with the requirements of SQLXML, with each *Game*, *Event and Qualifier* element as a direct sibling of the *Games* root element.

```
<Games>
  <Game>
```

```
    <ID>695019</ID>
    <HomeTeamID>21</HomeTeamID>
    <AwayTeamID>8</AwayTeamID>
  </Game>
  <Event>
    <GameEventID>330261085</GameEventID>
    <GameID>695019</GameID>
    <EventID>34</EventID>
    <PlayerID></PlayerID>
    <PeriodID>16</PeriodID>
    <Minute>0</Minute>
    <Second>0</Second>
    <TeamID>8</TeamID>
    <Outcome>1</Outcome>
    <X>0.0</X>
    <Y>0.0</Y>
  </Event>
  <Qualifier>
    <GameEventQualifierID>410900332</GameEventQualifierID>
    <GameEventID>330261085</GameEventID>
    <QualifierID>30</QualifierID>
    <Value>11334, 41135, 41328, 28495, 19419, 1718, 53392, 2051, 8438, 61262, 42786, 3785,
8442, 43670, 66842, 47412, 47431, 1827</Value>
  </Qualifier>
  <Qualifier>
    <GameEventQualifierID>1587667851</GameEventQualifierID>
    <GameEventID>330261085</GameEventID>
    <QualifierID>194</QualifierID>
    <Value>1718</Value>
```

Once again we use a Venetian Blind technique for clarity, noting that the added complexity of the F24 document makes the advantages of using this design much clearer:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:sql = "urn:schemas-microsoft-
com:mapping-schema">
  <xsd:element name = "Games" sql:is-constant = "1" >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name = "Game" type="Game" sql:relation = "Game" maxOccurs = "unbounded"
/>
        <xsd:element name = "Event" type="Event" sql:relation = "GameEvent" maxOccurs =
"unbounded" />
        <xsd:element name = "Qualifier" type="Qualifier" sql:relation = "GameEventQualifier"
maxOccurs = "unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="Game">
    <xsd:sequence>
      <xsd:element name = "ID" type = "xsd:long" sql:field = "GameID" />
      <xsd:element name = "HomeTeamID" type = "xsd:integer" sql:field = "HomeTeamID" />
      <xsd:element name = "AwayTeamID" type = "xsd:integer" sql:field = "AwayTeamID" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Event">
    <xsd:sequence>
      <xsd:element name = "GameEventID" type = "xsd:long" sql:field = "GameEventID" />
      <xsd:element name = "GameID" type = "xsd:long" sql:field = "GameID" />
      <xsd:element name = "EventID" type = "xsd:integer" sql:field = "EventID" />
      <xsd:element name = "PlayerID" type = "xsd:long" sql:field = "PlayerID" />
      <xsd:element name = "PeriodID" type = "xsd:integer" sql:field = "PeriodID" />
      <xsd:element name = "Minute" type = "xsd:integer" sql:field = "Minute" />
      <xsd:element name = "Second" type = "xsd:integer" sql:field = "Second" />
      <xsd:element name = "TeamID" type = "xsd:integer" sql:field = "TeamID" />
      <xsd:element name = "Outcome" type = "xsd:integer" sql:field = "Outcome" />
      <xsd:element name = "X" type = "xsd:float" sql:field = "X" />
      <xsd:element name = "Y" type = "xsd:float" sql:field = "Y" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Qualifier">
    <xsd:sequence>
```

```
      <xsd:element   name   =   "GameEventQualifierID"   type   =   "xsd:long"   sql:field   =
"GameEventQualifierID" />
      <xsd:element name = "GameEventID" type = "xsd:long" sql:field = "GameEventID" />
      <xsd:element name = "QualifierID" type = "xsd:integer" sql:field = "QualifierID" />
      <xsd:element name = "Value" type = "xsd:string" sql:field = "Value" />
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Once again, running a simple powershell script gives us the data we require in the database as follows:

```
select     t1.Name, t2.Name, ge.Minute, ge.Second, ge.EventID, gef.QualifierID, gef.Value from game g
inner join Team t1 on t1.teamid = g.awayteamid
inner join Team t2 on t2.teamid = g.hometeamid
inner join GameEvent ge on ge.gameid = g.gameid
inner join GameEventQualifier gef on gef.gameeventid = ge.gameeventid
```

100 %

Results | Messages

	Name	Name	Minute	Second	EventID	QualifierID	Value
1	Chelsea	West Ham United	70	14	5	233	716
2	Chelsea	West Ham United	40	17	74	56	Back
3	Chelsea	West Ham United	66	2	1	212	17.0
4	Chelsea	West Ham United	3	36	1	212	25.7
5	Chelsea	West Ham United	14	4	1	213	6.2
6	Chelsea	West Ham United	27	44	1	140	18.9
7	Chelsea	West Ham United	45	42	1	212	10.3
8	Chelsea	West Ham United	86	35	1	155	NULL
9	Chelsea	West Ham United	16	16	1	56	Back
10	Chelsea	West Ham United	89	39	1	212	8.1
11	Chelsea	West Ham United	91	23	43	56	Center
12	Chelsea	West Ham United	62	2	5	56	Center
13	Chelsea	West Ham United	87	34	1	56	Center
14	Chelsea	West Ham United	46	36	1	212	22.1
15	Chelsea	West Ham United	89	54	1	213	2.7

8 A Domain Specific Language (DSL) Solution

8.1 Introduction

Now we have the data structure in place, it is time to consider the method of querying and extracting data in a domain relevant way. To do this, we are going to partially develop a domain specific language called SportSL. Before we begin considering the specifics of SportSL, it is worth having some background discussions on the nature, properties and usage of Domain Specific Languages.

8.2 What is a Domain Specific Language?

A Domain Specific Language is fundamentally a language which is created with the sole purpose of providing functionality within a specific domain.

Whereas languages such as C, Java and Python are general purpose languages which can be used for an almost limitless amount of intentions, a DSL can only be used for its specific limited purpose. Suppose we needed to write a simple chess game using a command line interface. We could achieve this with a single verb; '*Move*'. At the player's turn, they type '*Move*' and supply two parameters; the pieces starting and destination squares. Once the move has been entered, the validity of the move is checked and reported back on. Obviously, this is a rather basic implementation of a chess game but technically it would work. This new language has only one function; to move a piece from one place to another. It cannot be used to write web applications, it has no conditional statements, it can only be used for its specified purpose.

DSLs can be technical or business focussed. Our chess example can be classed as a business focussed DSL, as it is solving a business domain issue; the rules of a game. SQL is an example of a technically focussed DSL. Although it doesn't solve any one particular business issue, it supplies a grammar to communicate with a database.

A good real-world example of a DSL is regular expressions. Checking the validity of a string in code can be an extremely cumbersome affair; we need to loop through each character, checking the conformity of the given match criteria, while keeping the history of previous characters in case we need to check against multiple instances of a character, etc. The complexity of verifying the format of an email address is often used as an example of this. The code needed to validate this be would large, easily inefficient and prone to error. Using regular expressions, this can be written concisely as follows:

```
@"^(?("")(""[^\r]+?(?<!\\)""@)|(([0-9a-z]((\.(?!\.))|[-!#\$%&'\*\+/=\?\^`\{\}\|~\w])*)(?<=[0-9a-z])@))
(?(\[)(\[(\d{1,3}\.){3}\d{1,3}\])|(([0-9a-z][-\w]*[0-9a-z]*\.)+[a-z0-9][\-a-z0-9]{0,22}[a-z0-9]))$ [7]
```

This also exposes a problem with DSLs, that the grammar needs to be documented, studied and understood. Regular expressions are a good illustration of this as the syntax is wonderfully concise and flexible yet can seem utterly inexplicable to the uninitiated.

Another potential issue is that the grammar of the language does not necessarily suggest the result. This is important when considering the users of the DSL may be non-technical people who may not

make the same assumptions around expected return values as a developer with twenty years experience. The following two pieces of grammar for instance:

```
Execute 5 + 5
Execute "5 + 5"
```

Would these two return the same value? Would the first return 10, the second "5 + 5", or both return the same? Without knowledge or documentation of the expected result, mistakes can happen. Again, not a problem that is unique to DSLs; documentation and semantic knowledge are always required in some form or another, but the training overhead should not be overlooked when considering the scale of the DSL that is being written.

It can sometimes be tricky to accurately dictate whether a system could be defined as a DSL or not. Excel is really just a graphical DSL for manipulating and displaying related data is it not? MatLab is a DSL concerned purely with statistical data. ASP.NET is a DSL to write web apps in. Obviously these are exaggerated arguments but it does ask the question 'How narrow does a language's function have to be to be considered a DSL?'. In fact, there are areas where the context of usage may define it over the nature of its initial purpose; Fowler argues that XSLT can be an DSL or not depending on its context [6]. Many descriptions of DSL's are concerned solely with the domain. "A class of closely related problems, a problem domain, can often be described by a domain-specific language, which consists of algorithms and combinators useful for solving that particular class of problems" [9] for instance is fine and accurate but then 'closely related problems' is only limited to our interpretation of the terms 'closely related' and 'problems'. To a developer working on high speed financial exchange connectivity, the difference between a mutex and a semaphore defines whether her contract is extended or not, but to others, they're just locks. Fowler himself gives a good summation: "a computer programming language of limited expressiveness focused on a particular domain" [8]. The reason why this should be considered a good synopsis is that it gives a DSL a combined boundary; of expressiveness and of domain specifics. SQL is very expressive yet is specific to data manipulation; our chess example was specific to our game yet woefully lacking in expressiveness, yet C# is expressive and non-domain specific; it is limited on neither front, therefore it doesn't fit with Fowler's rule.

One last general point on DSLs is that a well designed DSL, written in conjunction with the business users, can really help the business develop the model of future changes. They may not understand the underlying programmatical concepts, but a natural logical language which relates directly to their form of business can allow them to be much more expressive when discussing changes. "A primary source of accidental complexity is the large gap between the high-level concepts used by domain experts to express their problem statements and the low-level abstractions provided by general-purpose programming languages" [10] – this point has been used to promote the standardisation and globalisation of DSL standards across various business models. In general, the more involved the business users can be (and feel they are) the better, and providing a grammar that can be discussed in detail by both business and technical staff can be extremely beneficial.

8.3 Types of Domain Specific Languages

There are several different types of DSLs, the advantages and disadvantages of which need to be considered before deciding which to use for a specific project.

8.3.1 External DSLs

The definition of an external DSL is one that sits outside of any existing language. To implement an external DSL, the developer must be willing to create the compiler, define operator semantics, deal with memory management and all other considerations that would be necessary to create a general purpose language. A good example of this is SQL.

8.3.2 Internal DSLs

Internal DSLs are written in an existing language and expose domain specific grammar to the user. It can expose the grammar for the underlying language itself or restrict the grammar to a set of specific verbs which call underlying language methods, and is generally packaged up as a library. In many ways, it could be argued that an internal DSL is simply a user-facing API.

At first glance, it would seem that internal DSLs are a much more time efficient proposition, creating what is essentially a parser and a number of methods sitting underneath. However, an internal DSL is by definition tied to the underlying language it is written in, and that itself could be a problem. The semantics of the language may make translating the requirements of the user tricky; the developers could end up spending more time concentrating on making new features fit the abstraction (something Fowler calls Blinkered Abstraction [4]). Strongly typed languages such as C# and Java can be problematic because a DSL will be tied very closely to the type set available (although dynamic programming is now available in C#), which is why languages such as Ruby or Python are regularly used for implementing internal DSLs. Having full control of the entire language gives much greater flexibility for domain specific functionality. For instance, a DSL written in C# which is specific to low level network controllers may suddenly require levels of control which are unavailable in .NET. Of course, all of these arguments could be levelled at any software project. However, subsequent refactoring can be problematic if the DSL grammar has changed, which would require retraining for end users. As Kelker mentions “This means that your DSL will instantly adopt all the nice and not-so-nice syntactical features.” [5]. The underlying language must be chosen carefully to suit the needs of not only the domain but also of the skill-set available; in-depth compiler knowledge may not be something in supply in an application development team for instance.

8.3.3 Graphical DSLs

In theory, graphical DSLs are highly promising. Graphical user interfaces are ubiquitous with modern computing, they can double up as training tools with context sensitive help and people generally visually favour them. For simple domains, they can work well. The problem comes with complexity. Anyone who has used Biztalk or Windows Workflow will know how quickly the diagrams and flows become incredibly unwieldy and complex, and how easily a quick accidental hit of the delete key on 100% zoom mode can render the domain utterly irretrievable.

In addition, the overhead of creating the tools for a graphical user interface can be high. Unless it can be placed on top of an existing graphical modelling tool, you will essentially be rewriting Visio from scratch.

8.3.4 Fluent DSLs

It is worth mentioning Fluent DSLs at this point as they appear in many related texts. Fluent DSLs allow the chaining of methods to give a more concise visualisation of multiple method calls. A good

example of this is the way fluent syntax has been implemented in LINQ in .NET. However, I would argue that Fluent should be considered as an implementation design rather than a specific DSL type. External and Internal (and even Graphical at a push) DSLs can be written with fluent design in place.

8.4 SportSL Implementation

For SportSL, we will be creating an internal DSL in C#.Net. There are two factors which have this influenced this decision:

8.4.1 DSL Type

The decision to develop an internal DSL here is mostly influenced by the nature of intent. This dissertation is concerned with the implementation of a front to back sports data analysis system. It is not the intention here to consider compiler logic nor redevelop memory management processes. An internal DSL allows us to concentrate on the implementation of the business requirements in a technical environment.

In the real world, one of the primary concerns of a big data analysis DSL would be performance, and for this reason, an external DSL may indeed be a better decision, giving developers with the relevant level of expertise the tools to outperform a managed language such as C#. A graphical DSL is out of scope here as most time would be spent developing GUIs which, again, is not the intention of this project.

8.4.2 Base Language

There are many languages such as LISP and Ruby, which are more suited than .NET to create a DSL, due, not exclusively, to their flexibility around strong types. However, the author's (in the real world, this would be the development team) familiarity to C#, plus the availability of .NET's dynamic typing or reflection if necessary, have influenced the decision to use it.

8.4.3 Usage of Third Party DSL Frameworks

There are a number of third party DSL frameworks available for .NET such as Irony and Boo. The reason we are not going to use these here is simply because the majority of the work we need to discuss in this dissertation will already be provided and abstracted away from us. Writing the DSL from scratch will provide the opportunity to discuss each programmatic and functional decision in detail.

8.5 Project Scope of DSL

In a production environment, the SportSL language could contain a vast grammar. Eventually it would likely become a combination of SQL-like commands and statistics specific grammar, more akin to R, S+ or Matlab.

For the scope of this project, we will concentrate on building the technical requirements of a few examples, while always considering the extensibility of our technical design decisions. This is not unlike a proof of concept approach in an Agile environment; to prove that, end to end, useful, accurate information can be extracted in a manner which fulfils both technical and business needs. As the first pieces of functionality are built, consideration should be given to basic infrastructure needs such as error management and performance, and although neither of these might be of

absolute necessity to our proof of concept scope here, having a solid thoughtful technique to deal with such matters will allow subsequent development to flow more smoothly.

For the scope of this project, we will build grammar for the following commands:

- Bring back a list of players for a given condition
- Bring back some basic information of qualifiers and events for a given condition
- A helper function to list the available fields for the context

One of the main targets of this project is to define a DSL which is extensible and loosely coupled. It should be relatively easy to repoint any verbs to different classes or code, and even replace the underlying dataset to a new source. For example, the verb *where* could be a loose representation of a SQL Server *where* command, but we should be able to replace the entire SQL Server data source with, say, XML and reimplement the *where* clause to use some XPath code. This should be entirely invisible to the end user.

8.6 Defining the Grammar

A good starting point in the design of any DSL is to consider the expected grammar. In the instance of SportSL, we can look at this in a more abstract nature. For instance, we know we will require some type of ordering and grouping of our data. At this point, we can simply classify these together and consider the actual concrete grammar afterwards. As SportSL is fundamentally a data querying tool, there are several fundamental abstract types we need to consider:

8.6.1 Entity

The *Entity* grammar type represents a collection of related data. In the world of databases, this could be a single table or a related join of many tables. As any complexity of underlying data relationships should be shielded from the end user, an *Entity* should simply be something that represents a concept of data, rather than anything concrete. For example, an *Entity* called *Player* could point at the single underlying table *Players* in our data model. However, it may represent a join between the *Player* table and the *Teams* table to allow the user to retrieve useful related team information. It may even represent a totally different underlying data set altogether; an XML document, a JSON string, etc. This should be completely invisible to the user; the *Player* verb simply represents the *Player* data. This also gives the DSL full control over what data is available to the users. This is key to the business rule we have from Opta to abstract any user away from the full underlying data set; users will only see entities that represent tables or views of the data that are defined by the DSL.

8.6.2 Projection

The users will require the ability to select only the fields they are interested in. For some more complex entities there may be many fields available but only a small subset will be of interest. This becomes even more important when considering grouping of data. The Projection type is concerned with this very issue, and should implement some sort of filtering of the fields returned to the user.

8.6.3 Filter

The crux of any data query is the ability to filter the entire underlying data set. We represent this in SportSL as the Filter grammar type. This will allow the user to define what portions of the data set they require in a general query format. This can be seen as the *where* clause in SQL.

8.6.4 Sorting

Another fundamental action of any data query is the ability to sort and group the data. These functions are represented under a single grammar type in SportSL called Sorting.

8.6.5 Return Formats

It is very likely that the users of SportSL may use the returned query results as input into other statistical systems such as Matlab. This being the case, there needs to be some flexibility around the format of the data returned. In many cases, a simple delimited list may suffice, but in others an XML or JSON document may be required. The user doesn't want to have to create an intermediate parser to align the data with the expected input of those systems, either muddling around in Excel or using a scripting language such as Perl. The Return grammar type deals with this area. The user should be able to specify the return format from a given selection of types. Also, new return formats should be easy to implement and used in a common way from the parser.

8.6.6 Bespoke Functions

Dynamic help and documentation is always beneficial, especially in a DSL. Printed documentation is rarely read and is almost never kept up to date. Having the ability to write helper functions as part of the natural grammar of the language keeps help at the user's fingertips. The Bespoke grammar type is concerned with exactly that; to allow the creation of helper functions, which feel like normal DSL grammar but are not necessarily tied the underlying dataset. Simple examples of this may be to list all available fields for a given entity, to give examples of usage or more in-depth information of errors.

8.6.7 Examples of Expected Grammar

At this point, we can define some examples which the parser should be expected to parse and return appropriate data. Using a basic Agile approach, this allows us to work towards the implementation of commands which use a combination of the grammar types mentioned above, but simultaneously give us concrete examples of suitability.

Starting with a simple Entity type concerned with the *Player* data, a Bespoke grammar type called *allfields* should return all available field names for that context.

```
player();allfields()
```

The parser should also allow the user to perform some specific query constraints on the Player entity, including Projection, Filter and Order types. In the following, *show* is the Projection grammar type, *where* is the Filter and *orderby* is the Sorting type. Also, a custom Returner will be used to specify the format of the returned data, in this instance, the *list* command which is supplied with a delimiter character.

```
player();show(@name,@position);where(@name like 'John%');orderby(@name);list(|)
```

Finally, a more complex Entity should be created to prove the extensibility of the model. This will be the *Event* Entity, which will represent several related tables in the underlying database.

```
event();show(@hometeam,@awayteam,@minute,@second,@eventname,@qualifiername,@value);
where(@minute = 90);orderby(@minute,@second);list(|)
```

8.7 Creating the Grammar Interfaces

Let's start off with creating the interfaces for the abstract grammar types. A quick reminder of the grammar types we are looking to implement:

- Entity
- Projection
- Filter
- Sorting
- Return
- Bespoke

Three of these types are very similar in structure; Projection, Filter and Sort. All three of these will require a condition of some sort and a method to return a textual representation of the condition in a format usable by the executing method on the dataset. It is sensible then to create a super-interface for these three. We will call this IContextItem; the definition as follows:

```
public interface IContextItem
{
    string ExecuteCode();
    string Condition { get; set; }
}
```

Then three specific interfaces can extend this super-interface with no further implementation. It is possible that we could simply use the IContextItem interface as the main interface for all implementing classes. However, this will be discussed later.

Each one of the interfaces are defined as follows:

```
public interface IFilter : IContextItem {}
public interface IProjector : IContextItem {}
public interface ISorter : IContextItem {}
```

Another grammar type which may not initially seem to fit under the IContextItem interface is the Entity type. However, it too requires a method to return syntax to select its underlying entity, and may possibly need a condition of some sort, so implementing the interface seems to make sense. Unlike the other three interfaces mentioned, the Entity interface does require some extra methods specific to entities; namely column mapping. We will return to column mapping in the next section. For now however, the IEntity interface will be as follows:

```
public interface IEntity : IContextItem
{
    IColumnMapper ColumnMapper { get; set; }
    string ConvertMapping(string initial);
}
```

This leaves us with two grammar types which are notably different to the others; Return and Bespoke.

Return types act as repositories for the data to be returned in a given format, and therefore require three specific methods; an AddRow method to add a row of data to the repository, AllRows to return the data in its format, and a Condition property which may well be optional in many cases, but may be used for specifying delimiters, XML version types, etc. This can be represented as follows:

```
public interface IReturner
{
    void AddRow(List<string> fields);
    string AllRows();

    string Condition { get; set; }
}
```

The Bespoke interface is a little trickier, as the creation of any future concrete implementations are not as clear as the other interfaces. A concrete Bespoke type could be something in relation to the selected entity (a count of rows from a database), or it may be completely separate (a list of help functions), so flexibility is key. A Condition may still be required but a separate Execute method needs to be available, with an optional IEntity type. This will give anyone implementing a Bespoke type the ability to tie the execution method directly to the Entity or return data totally unrelated. The interface is as follows:

```
public interface IBespoke
{
    string Condition { get; set; }
    string Execute(IEntity entity);
}
```

8.8 Column Mappings

Before we delve into implementing our interfaces into concrete classes, it is worth discussing the topic of column mappings.

The column names we provide to the user should not necessarily reflect the exact underlying column names in the dataset. There are several reasons for this:

- Some column names in the underlying dataset may not be textually friendly to the user. As we have designed the SQL Server schema from scratch, this may not be a particular issue here. However, if the DSL were to switch to using a schema we had no control over, we may have column names which couldn't be easily interpreted.
- For entities which represent multiple joins, column names will require a table prefix (*t.Name* for example). Also, the column Name may represent multiple things within a single query; *Name* could be player name, home team name and away team name.
- Specifying a list of available column names gives the ability to hide any columns we do not wish the user to have access to. This gives us a level of control over the user's visibility of the underlying dataset; one of the pre-requisites of using Opta's data.
- Should the need to switch entire datasets ever happen, the users would need to relearn all the underlying schemas column names. Having a mapping allows us to give them the same data for the same columns in their saved queries.

There are also some general considerations for the implementation of column mappings:

- They must be specific to an Entity

- They relate to several grammar types; columns are available in Filter, Sorting and Projection types
- How do we recognise a column name in a Condition?

As we saw in the previous section, our IEntity interface includes support for column mappings; it contains a property of IColumnMapper and a method called ConvertMapping. The concept behind these two is that a column mapping class can be given to an IEntity, and ConvertMapping can take a string with the column names received from the parser and return a new string with the correct underlying columns included.

As we will see in upcoming sections, the way the Entity classes execute the query will convert the relevant columns for all three of the necessary grammar types. However, we do still need to tell the DSL what constitutes a column name within a condition. The easiest way to do this is to prefix them with a particular character, and taking the lead from SQL Server, we will use the @ character to indicate this. We will see how this is implemented in the following sections but an example of grammar usage is as follows:

```
player();show(@name,@position);where(@name like 'John%');orderby(@name)
```

The *show*, *where* and *orderby* methods all use columns which are pre-fixed with the @ symbol. The best example of why this is necessary is in the *where* method; how would the DSL know that *name* is a column name and not *like*? Giving the @ prefix makes this obvious and easy to parse.

To apply a column mapper to an Entity, we will create an IColumnMapper interface, which will consist of an indexer to retrieve a column given a key, and a helper method called AllFields, which will give access to all available fields to a Bespoke type if required:

```
public interface IColumnMapper
{
    string this[string name] { get; }
    List<string> AllFields { get; }
}
```

As we have already seen, the IEntity interface has a property of type IColumnMapper which it uses to apply the mapping to its execution code.

8.9 Concrete Implementations of the Grammar Interfaces

Now the interfaces have all been defined, and we've talked about the issues around column mappings, we can look at implementing some of the concrete types which will make up the physical grammar of our DSL.

Let's look back at the initial example we intend on implementing:

```
player();show(@name,@position);where(@name like 'John%');orderby(@name);list(|)
```

Here we have four grammar commands to implement:

- An Entity (IEntity) called *player*
- A Projector (IProjector) called *show*
- A Filter (IFilter) called *where*
- A Sorter (ISorter) called *orderby*

- A Return (IReturner) called *list*

8.9.1 Player Entity

The *player* entity is a simple map to the *Player* table in the database. We have four interface methods and properties to implement here. Firstly Condition and ColumnMapper will be taken care of by the creating class. The remaining methods are ExecuteCode and ConvertMapping.

The ExecuteCode method should contain the syntax specific to querying code from the underlying data source; in this case the SQL server table *Player*; '*from Player*'. The ConvertMapping method needs to take the code from the Controller (more of which later), inclusive of all columns prefixed with the @ character, and return a query suitable for the underlying SQL Server schema, based on the mappings available in the ColumnMapper object.

The ExecuteCode method is as follows and maps to the relevant SQL:

```
public string ExecuteCode()
{
    return " from Player ";
}
```

The ConvertMapping method uses the associated ColumnMapper to strip out the column names prefixed with @ based on a regular expression to return all words starting with @, then replace them with the associated column names:

```
public string ConvertMapping(string initial)
{
    string retMapping = initial;
    foreach (Match match in Regex.Matches(initial, @"(?<!\w)@\w+"))
    {
        string field = match.Value.Replace("@", "");
        retMapping = retMapping.Replace(match.Value, ColumnMapper[field]);
    }
    return retMapping;
}
```

8.9.2 Show Projector

The *show* projector is a simple filter which is designed to take a number of columns to allow the user to select a specific subset of the data set. The implementation of this is very simple; it implements the IContextItem.Condition property and simply returns the Condition as the ExecuteCode method's return string:

```
public class Show : IProjecter
{
    public string Condition
    {
        get; set;
    }

    public string ExecuteCode()
    {
        return Condition;
    }
}
```

8.9.3 Where Filter

In a similar manner to the *show* class, the w*here* Filter class simply implements the IFilter interface, implementing the Condition property and returning the Condition prefixed with the SQL *where* clause:

```
public class Where : IFilter
{
    public string Condition
    {
        get;set;
    }

    public string ExecuteCode()
    {
        return " where " + Condition;
    }
}
```

8.9.4 Orderby Sorter

The OrderBy class implements the ISorter class with a basic implementation of the Condition property along with the ExecuteCode method prefixing the Condition code with the SQL *Order By* clause:

```
public class OrderBy : ISorter
{
    public string Condition
    {
        get; set;
    }

    public string ExecuteCode()
    {
        return " order by " + Condition;
    }
}
```

8.9.5 List Returner

The List Returner is the most basic implementation of the Returner grammar type. It maps to a BasicListReturner class and simply returns the data as a delimited list of strings. The default delimiter is a comma but this can be overridden by supplying a string in its Condition (*list(|)*) for pipe delimited for example). The constructor sets the default delimiter, which would in reality be done via configuration settings, the AddRow method adds a List of strings to a private generic List of List of strings. The AllRows method is implemented by looping through the List of rows and returning them delimited and with new line characters.

```
  private List<List<string>> rows = new List<List<string>>();
    public BasicListReturner()
    {
        Delimiter = ",";
    }
    public void AddRow(List<string> fields)
    {
        rows.Add(fields);
    }

    public string Condition {
        get { return Delimiter.ToString(); }
        set
        {
            if (string.IsNullOrEmpty(value))
                Delimiter = ",";
            else
```

```
                Delimiter = value;
            }
        }
        private string Delimiter { get; set; }
        public string AllRows()
        {
            StringBuilder sb = new StringBuilder();
            foreach (List<string> row in rows)
            {
                foreach (string s in row)
                    sb.Append(s + Delimiter);
                sb.Append(Environment.NewLine);
            }
            return sb.ToString();
        }
}
```

8.9.6 AllFields Bespoke

One of the most useful helper methods for the SportSL DSL would be a list of available fields for a given entity. As this list could change as more (or less) fields are available via the column mapping classes, it makes sense to make this helper function dynamic.

The AllFields class does this by taking an instance of the IEntity interface and simply looping around the columns from the related ColumnMapper's AllFields method.

```
public class AllFields : IBespoke
{
    public string Condition
    {
        get;set;
    }

    public string Execute(IEntity entity)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string value in entity.ColumnMapper.AllFields)
            sb.Append(value + Environment.NewLine);
        return sb.ToString();

    }
}
```

8.10 Creating the Column Mappers

The IColumnMapper interface allows us to create classes which encapsulate the mapping of columns from an Entity instance to the underlying data source. For instance, the user may use a column called 'PlayerName' but the underlying column in the database query may be 'p.player'. As previously mentioned, this flexibility is especially useful when dealing with multiple table joins.

For example, the Player entity would have a ColumnMapper which looks like this:

```
public class PlayerColumnMapper : IColumnMapper
{
    private Dictionary<string, Tuple<string, string>> map = new Dictionary<string, Tuple<strin
g, string>>();
    public PlayerColumnMapper()
    {
        map["name"] = new Tuple<string,string>("name", "Player's Name");
        map["position"] = new Tuple<string,string>("position", "Player's Position");
    }

    public string this[string name]
    {
        get
        {
```

```
            return map[name].Item1;
        }
    }

    public List<string> AllFields
    {
        get
        {
            List<string> all = new List<string>();
            foreach(string key in map.Keys)
            {
                all.Add(key + " : " + map[key].Item2);
            }
            return all;
        }
    }
}
```

The *map* dictionary maps the column name the user will type in to a tuple containing the underlying data source's column name, plus a short description for the helper methods. Here we have only exposed two columns; name and position. These happen to map to columns in the database with the same name, but as we will see later when implementing a more complex example, this will not always be the case.

The AllFields method is implemented by looping around the map and returning a string containing the column and description.

8.11 Creating a Grammar Factory

Now we have a set of concrete grammar classes, we can look at exposing these to the user. As previously mentioned, a grammar type in code may not necessarily relate directly to a type the user is typing in via the UI. For example, we have implemented a *where* Filter grammar, but when a user types *where* into the parser, she has no idea (nor cares) what underlying class this is calling. The type the user is calling should not relate directly to the underlying class whatsoever, so we need to implement a mapping from verb to underlying class.

For this we need some kind of factory class to map the users inputted methods to the physical classes that it creates instances of, and this should be flexible enough to allow developers to switch in and out the mappings as required. GrammarFactory is the static class we use to map grammar to classes and then to create all of the relevant instances of classes.

The GrammarFactory class is as follows:

```
public enum GrammarType { Filter, Projecter, Entity, Returner, Sorter, Bespoke };
    public static class GrammarFactory
    {
        private static Dictionary<string, Tuple<Type, GrammarType>> typeMap = new Dictionary<string, Tuple<Type, GrammarType>>();
        private static Dictionary<Type, Type> entityColumnMappers = new Dictionary<Type, Type>();
        static GrammarFactory()
        {
            typeMap.Add("player", new Tuple<Type, GrammarType>(typeof(Player), GrammarType.Entity));
            typeMap.Add("event", new Tuple<Type, GrammarType>(typeof(Event), GrammarType.Entity));
            typeMap.Add("where", new Tuple<Type, GrammarType>(typeof(Where), GrammarType.Filter));
            typeMap.Add("orderby", new Tuple<Type, GrammarType>(typeof(OrderBy), GrammarType.Sorter));
            typeMap.Add("show", new Tuple<Type, GrammarType>(typeof(Show), GrammarType.Projecter));
            typeMap.Add("list", new Tuple<Type, GrammarType>(typeof(BasicListReturner), GrammarType.Returner));
            typeMap.Add("allfields", new Tuple<Type, GrammarType>(typeof(AllFields), GrammarType.Bespoke));

            entityColumnMappers.Add(typeof(Player), typeof(PlayerColumnMapper));
            entityColumnMappers.Add(typeof(Event), typeof(EventColumnMapper));
        }
        public static void AddContext(IController controller, string Name, string Condition)
        {
            Type typeObj = typeMap[Name].Item1;
            switch(typeMap[Name].Item2)
            {
                case GrammarType.Filter:
                    IFilter filter = Activator.CreateInstance(typeObj) as IFilter;
                    filter.Condition = Condition;
                    controller.Filter = filter;
                    break;
                case GrammarType.Bespoke:
                    IBespoke bespoke = Activator.CreateInstance(typeObj) as IBespoke;
                    bespoke.Condition = Condition;
                    controller.Bespoke = bespoke;
                    break;
                case GrammarType.Entity:
                    IEntity entity = Activator.CreateInstance(typeObj) as IEntity;
                    entity.Condition = Condition;
                    controller.Entity = entity;

                    Type columnMapperType = entityColumnMappers[entity.GetType()];
                    IColumnMapper columnMapperInstance = Activator.CreateInstance(columnMapperType) as IColumnMapper;
                    entity.ColumnMapper = columnMapperInstance;

                    break;
                case GrammarType.Projecter:
                    IProjecter projector = Activator.CreateInstance(typeObj) as IProjecter;
                    projector.Condition = Condition;
                    controller.Projector = projector;
                    break;
                case GrammarType.Returner:
```

```
                IReturner returner = Activator.CreateInstance(typeObj) as IReturner;
                returner.Condition = Condition;
                controller.Returner = returner;
                break;
            case GrammarType.Sorter:
                ISorter sorter = Activator.CreateInstance(typeObj) as ISorter;
                sorter.Condition = Condition;
                controller.Sorter = sorter;
                break;
        }
    }
}
```

This is fundamentally the core of the whole DSL and deals with the way the grammar maps to the underlying class structure of the system. It works in the following way:

- A GrammarType enum is set up containing the various abstract type of Grammars; entity, filter, returner, etc
- The *typeMap* dictionary holds the mapping between the grammar to a tuple containing the concrete class type and a reference to the GrammarType enum. For example, the grammar *where* links to a concrete type called *where* which is of abstract *GrammarType Filter*.
- The AddContext method takes an IController object and a pair of strings; Name and Condition. The method then maps the Name to the relevant type using the *typeMap* dictionary. Depending on the type of abstract grammar type the verb belongs to, the controllers context type properties are set. For example, the method may receive Name and Condition as '*where'* and *"@name = 'Colin'"* respectively. The *where* key will be looked up in the *typeMap* dictionary and, realising that is of a GrammarType.Filter, will set the controller's Filter property as a new instance of the Where class via reflection, setting its Condition property to the Condition passed in.
- For any Entity types, there is an additional supporting dictionary called *entityColumnMappings*. This dictionary maps together two Types, one relating to an IEntity implementing class and the second an IColumnMapping. When an IEntity type is received in the AddContext method, a corresponding instance of a IColumnMapping class is created and set as the IEntity's ColumnMapper property.

It's worth mentioning here that in a production environment, both mapping dictionaries would be populated via a configuration file rather than hardcoded in the GrammarFactory. This way, any required changes could be done in configuration, removing the need for code changes, compilation or deployment.

The advantage of using a factory such as this is that any commands can be switched on at the backend to work in completely different ways without the user having any concerns about learning new grammar. Should it be decided in the future that the Player entity requires a link to a second table, then a new concrete implementation of the IEntity can be created, and with a quick configuration change, can instantly be available to the user. Similarly, if it was decided to pull out the entire backend data source and replace SQL Server with something different, the configuration could be changed to point at a variety of new classes which implement the same functionality for the new data source, again with no worry to the user; their queries will still work, the results will be the same.

8.12 Executing the Query

Now all classes have been set up to create a query against the database, we need a class which executes it; this what the IExecutor interface is designed for.

The IExecutor interface contains a single method Execute, which takes a string of execution code (SQL for instance) and a delegate to call with each row returned. For the SQL Server data source, we need to create a connection to the database, a command with the SQL statement then loop around the result set, calling the *addMethod* delegate for each one.

This is done as follows:

```
public class Executor : IExecutor
{
    public void Execute(string ExecuteCode, Action<List<string>> addMethod)
    {
        using (SqlConnection connection = new SqlConnection("Data Source=.;Initial Catalog=Opt
a;Trusted_Connection=true;"))
        {
            connection.Open();

            SqlCommand command = new SqlCommand(ExecuteCode, connection);
            SqlDataReader reader = command.ExecuteReader();
            while(reader.Read())
            {
                List<string> fields = new List<string>();
                for(int i=0; i<reader.FieldCount; i++)
                {
                    fields.Add(reader[i].ToString());
                }
                addMethod(fields);
            }
        }
    }
}
```

The delegated method will generally relate to an IReturner implementing class (normally its AddRow method) but we will see this in more detail in upcoming sections.

8.13 The Controller Class

A Controller class is the main class with which the parser will communicate and ties everything together. They are based on the IController interface which contains a reference to each of the abstract grammar types as well as the IExecutor and a single method *Execute*. As previously demonstrated, the GrammarFactory sets all of these properties apart from the IExecutor, which we will come to shortly.

An implementation of this can be seen below:

```
public class Controller : IController
    {
        public Controller()
        {
            // set the default Returner
            Returner = new BasicListReturner();
        }
        public IProjecter Projector { get; set; }
        public IFilter Filter { get; set; }
        public ISorter Sorter { get; set; }
        public IReturner Returner { get; set; }

        public IEntity Entity { get; set; }
        public IExecutor Executor { get; set; }

        public IBespoke Bespoke { get; set; }

        public string Execute()
        {
            if (Bespoke != null) // overrides all others
            {
                return Bespoke.Execute(Entity);
            }
            else
            {
                Executor.Execute(ExecuteString, Returner.AddRow);
                return Returner.AllRows();
            }
        }
```

```
        private string ExecuteString
        {
            get {
                string mapped = Entity.ConvertMapping(Projector.ExecuteCode() + Entity.Execute
Code() + Filter.ExecuteCode() + Sorter.ExecuteCode());
                return "select " + mapped;
            }
        }
    }
```

The main discussion points here are:

- The private ExecuteString creates a full valid SQL statement from the provided grammar classes. It firstly calls the IEntity's ConvertMapping method to replace all column names with the correct mappings and prefixes the result with the SQL select command.
- The Execute method takes the converted ExecuteString and passes it, along with the IReturner's AddRow method as a delegate, to the IExecutor's Execute method, then returns the result from the IReturner's AllRows method to the caller.
- As Bespoke objects work in very different ways than the rest of the system, a conditional statement in the Execute method looks to see if the Bespoke property has been set, and if so, ignores everything else and calls the IBespoke.Execute method, passing the provided IEntity.
- The Controller's constructor creates a default Returner as we will always require a Returner of some sort

8.14 Creating the Parser

One of the main advantages of this entire solution is that, being as the mapping between user's entry and the underlying classes is completely dynamic and handled by the factory and controller, the Parser class will be very straight forward. Not only that, but it's quite difficult to imagine a situation where the Parser class is ever changed. This advantage should not be overlooked. Janota, Fairmichael, Holub, Grigore, Charles, Cochran, and Kiniry state a valid point: '*Programmers often write custom parsers for the command line input of their programs. They do so, in part, because they believe that both their program's parameterization and their option formats are simple. But as the program evolves, so does the parameterization and the available options. Gradually, option parsing, data structure complexity, and maintenance of related program documentation becomes unwieldy*.' [11]. In the case of SportSL, it is very unlikely we will ever need to change the Parser as any new grammar will become available automatically when new classes are created and mappings put in place.

All that is required are two methods; Parse and Execute:

```
class Parser
    {
        IController controller;
        public void Parse(string queryString)
        {
            List<Tuple<string, string>> operations = new List<Tuple<string, string>>();

            controller = new Controller(); // would be from config
            controller.Executor = new Executor(); // would be from config

            foreach (string element in queryString.Split(';'))
            {
                string[] components =
                       element.Split(new string[] { "(", ")" }, StringSplitOptions.None);
```

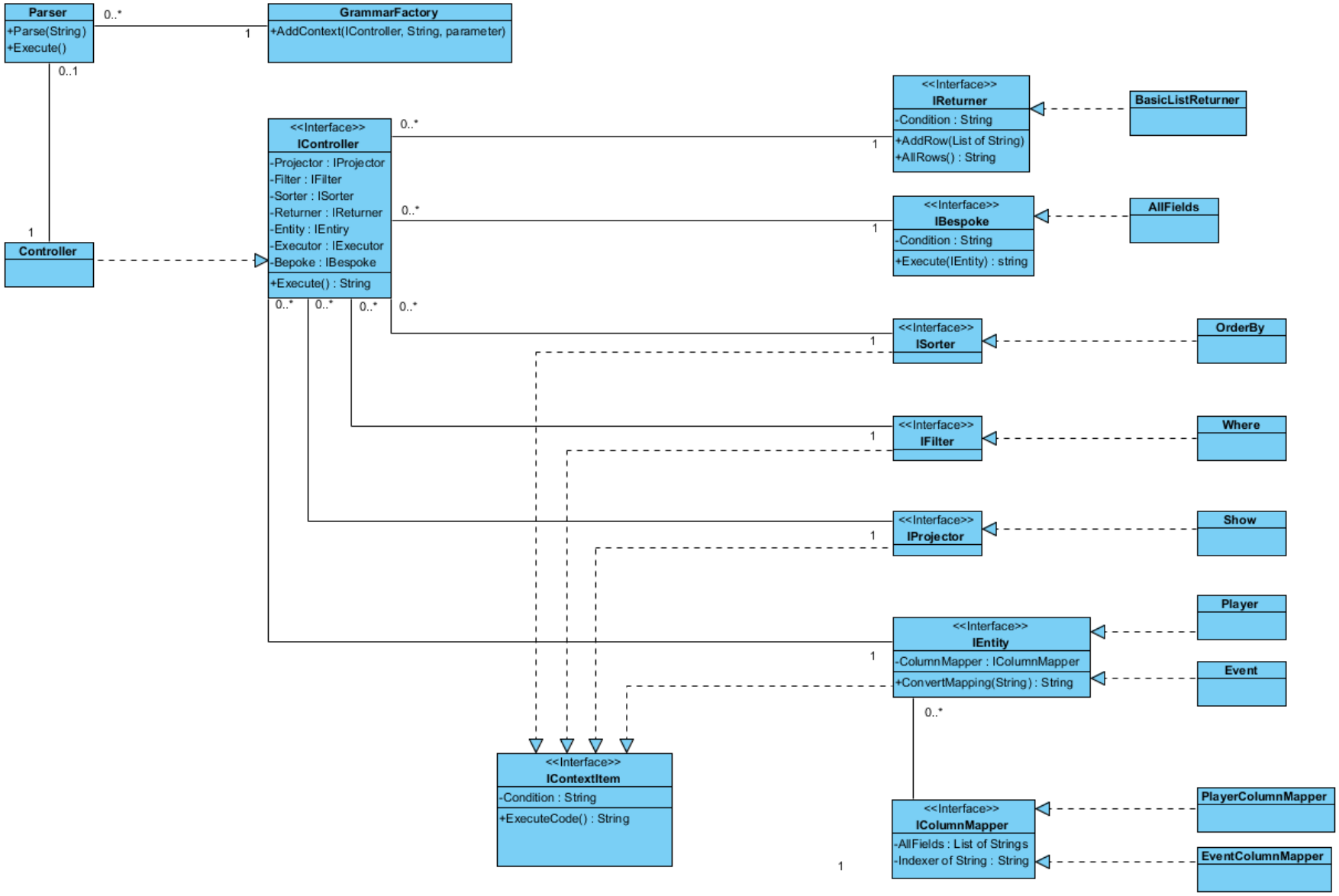
```
                GrammarFactory.AddContext(controller, components[0], components[1]);
            }
        }

        public void Execute()
        {
            Console.WriteLine(controller.Execute());
        }
    }
```

The Parser firstly sets up instances of relevant IController and IExecutor implementing classes (as mentioned in code these would be held in configuration), then loops around the input string, splitting it out via our semi-colon grammar delimiter. For each piece of grammar, it then pulls out any condition from supplied parentheses and calls the GrammarFactory's AddContext.

By the end of the *Parse* method, we should have a Controller fully set up with all the relevant supplied grammar. This is now ready for the Execute method, which in this case simply displays the return value of the Controller's Execute method.

8.15 Class Diagram

Before we step through an example, it seems a good time to look at how all of the classes and interfaces fit together on a class diagram:


Parser
+Parse(String)
+Execute()
0..*
1
GrammarFactory
+AddContext(IController, String, parameter)
0..1
1
Controller
<<Interface>>
IController
-Projector : IProjector
-Filter : IFilter
-Sorter : ISorter
-Returner : IReturner
-Entity : IEntiry
-Executor : IExecutor
-Bepoke : IBespoke
+Execute() : String
<<Interface>>
IReturner
-Condition : String
+AddRow(List of String)
+AllRows() : String
BasicListReturner
<<Interface>>
IBespoke
-Condition : String
+Execute(IEntity) : string
AllFields
<<Interface>>
ISorter
OrderBy
<<Interface>>
IFilter
Where
<<Interface>>
IProjector
Show
<<Interface>>
IEntity
-ColumnMapper : IColumnMapper
+ConvertMapping(String) : String
Player
Event
<<Interface>>
IContextItem
-Condition : String
+ExecuteCode() : String
<<Interface>>
IColumnMapper
-AllFields : List of Strings
-Indexer of String : String
PlayerColumnMapper
EventColumnMapper

8.16 Running an Example

Let's look back at the examples we gave for perceived usage in a previous section. The first example was of a bespoke grammar type called AllFields on the Player entity, which would return all available columns when using the player entity:

```
player();allfields()
```

The way this command is parsed and executed is as follows:

- A Controller is created by the Parser
- The Parser splits out the input into two commands; player() and allfields(), each being passed to the GrammarFactory
- The GrammarFactory maps the player command to the Player class, which has a related context type of IEntity and sets the controller's IEntity property
- The GrammarFactory maps the allfields command to the AllFields class, which has a related context type of IBespoke
- The Parser calls the IBespoke's Execute command which loops around the Player object ColumnMapper instance and returns the name and description for each available column

The results are as follows:

```
name : Player's Name
position : Player's Position
```

This is as expected.

Now let's look at a more complex example, which will query the data source:

```
player();show(@name,@position);where(@name like 'John%');orderby(@name);list(|)
```

This command should select the name and position columns for all rows from the *player* entity whose name column starts with '*John*', ordering by the *name* column and returning the data in a pipe delimited list.

The programmatic flow of this command is as follows:

- A Controller is created by the Parser
- The Parser splits out the input into five commands; player(), show(@name,@position), where(@name like 'John%'), orderby(@name) and list(|), each being passed to the GrammarFactory, which in turn:
    - Maps the *player* command to the Player class, which has a related context type of IEntity and sets the controller's IEntity property
    - Maps the *show* command to the Show class, which has a related context type of IProjector, and sets its Condition property to '*@name,@position*'.
    - Maps the *where* command to the Where class, which has a related context type of IFilter, and sets its Condition property to '*@name like 'John%"*
    - Maps the *orderby* command to the OrderBy class, which has a related context type of ISorter, and sets its Condition to '*@name*'

    - Maps the *list* command to the BasicListReturner class, which has a related context type of IReturner, and sets its Condition to '|'
- The Parser then calls the Controller's Execute method. This calls the ConvertMapping method on the Entity, passing through the ExecuteCode string of each context type to the Entity's ConvertMapping method to replace the provided column names with the correct SQL Server ones. It then passes this SQL statement through the Executor's Execute method, along with the Returner's AddRow method.
- The Executor populates the Returner with the data and passes back the result of the Returner.AllRows back to the Parser to display

The results are as follows:

```
John Arne Riise|Defender
John Brayford|Defender
John Guidetti|Forward
John Lundstram|Midfielder
John Obi Mikel|Midfielder
John O'Shea|Defender
John Ruddy|Goalkeeper
John Stones|Defender
John Terry|Defender
Johnny Heitinga|Defender
```

8.17 A More Complex Example

So far we have shown how a very simple single table entity can be wired up to produce results. However, a good test of the flexibility of the solution is to create a more complex example.

One the main areas of the database concerns itself with events, qualifiers and the related team and player information. If we are to expose this information in a useful and concise way, then we need to create grammar which follows the same pattern as the player example but with more advanced table and column mappings.

Let's return to the complex example we wrote initially:

```
event();show(@hometeam,@awayteam,@minute,@second,@eventname,@qualifiername,@value);where(@minu
te = 90);orderby(@minute,@second);list(|)
```

This looks very similar to our previous player example, and of course it should, as the grammar should be easy to use, remember and recreate, no matter the difference in entity. However, under the hood there are a couple of major differences:

- The *event* Entity is a multi-table join
- The column names used in the command do not match the column names in the database

To allow this query to work, we need to create two new classes; an instance of an IEntity implementing class, and a related IColumnMapper.

The event entity would look as follows:

```
public class Event : IEntity
    {
        public IColumnMapper ColumnMapper
        {
            get;set;
        }
```

```
        public string Condition
        {
            get;set;
        }

        public string ConvertMapping(string initial)
        {
            string retMapping = initial;
            foreach (Match match in Regex.Matches(initial, @"(?<!\w)@\w+"))
            {
                string field = match.Value.Replace("@", "");
                retMapping = retMapping.Replace(match.Value, ColumnMapper[field]);
            }
            return retMapping;
        }

        public string ExecuteCode()
        {
            return " from game g " +
                "inner join Team t1 on t1.teamid = g.awayteamid " +
                "inner join Team t2 on t2.teamid = g.hometeamid " +
                "inner join GameEvent ge on ge.gameid = g.gameid " +
                "inner join  Event e on e.EventID = ge.EventID " +
                "inner join GameEventQualifier gef on gef.gameeventid = ge.gameeventid " +
                "inner join  Qualifier q on q.QualifierID = gef.qualifierid";
        }
    }
```

The core difference here between this and the Player entity is simply the ExecuteCode implementation, linking seven tables together.

A related IColumnMapper is also required:

```
public class EventColumnMapper : IColumnMapper
    {
        private Dictionary<string, Tuple<string, string>> map = new Dictionary<string, Tuple<s
tring, string>>();
        public EventColumnMapper()
        {
            map["hometeam"] = new Tuple<string, string>("t1.name", "Home Team Name");
            map["awayteam"] = new Tuple<string, string>("t2.name", "Away Team Name");
            map["minute"] = new Tuple<string, string>("ge.minute", "Event Minute");
            map["second"] = new Tuple<string, string>("ge.second", "Event Second");
            map["eventname"] = new Tuple<string, string>("e.name", "Event Name");
            map["qualifiername"] = new Tuple<string, string>("q.name", "Qualifier Name");
            map["value"] = new Tuple<string, string>("gef.value", "Value");
        }

        public string this[string name]
        {
            get
            {
                return map[name].Item1;
            }
        }

        public List<string> AllFields
        {
            get
            {
                List<string> all = new List<string>();
                foreach (string key in map.Keys)
                {
                    all.Add(key + " : " + map[key].Item2);
                }
                return all;
            }
        }
```

Here we can see why the Column Mappers are useful. The resulting column names from the database will all be prefixed with the table name (or acronym) and the two teams names representing the home and away teams are actually the same column in the same table, linked via

two separate joins so are both called *Name*. By mapping the column names to something more user friendly and logical such as HomeTeam and AwayTeam, the user will get a clearer idea of what data they are receiving.

The flow for this command is very similar to the previous example, but using the Event entity rather than the Player, and produces the following results:

```
Chelsea|West Ham United|90|5|Clearance|Out of play|
Chelsea|West Ham United|90|5|Clearance|Head|
Chelsea|West Ham United|90|25|Pass|Throw-in|
Chelsea|West Ham United|90|25|Pass|Angle|1.3
Chelsea|West Ham United|90|25|Pass|Length|21.3
Chelsea|West Ham United|90|25|Pass|Pass End X|69.1
Chelsea|West Ham United|90|25|Pass|Long ball|
Chelsea|West Ham United|90|25|Pass|Pass End Y|27.9
Chelsea|West Ham United|90|29|Pass|Pass End X|83.4
Chelsea|West Ham United|90|29|Pass|Pass End Y|32.1
Chelsea|West Ham United|90|29|Pass|Angle|5.9
Chelsea|West Ham United|90|29|Pass|Length|7.7
Chelsea|West Ham United|90|30|Clearance|Head|
Chelsea|West Ham United|90|35|Pass|Pass End X|44.6
Chelsea|West Ham United|90|35|Pass|Angle|5.0
Chelsea|West Ham United|90|35|Pass|Pass End Y|35.1
Chelsea|West Ham United|90|35|Pass|Length|11.1
Chelsea|West Ham United|90|39|Tackle|Defensive|
Chelsea|West Ham United|90|39|Take On|Offensive|
Chelsea|West Ham United|90|42|Pass|Angle|0.2|
Chelsea|West Ham United|90|42|Pass|Long ball||
Chelsea|West Ham United|90|42|Pass|Pass End X|92.6|
Chelsea|West Ham United|90|42|Pass|Length|53.3|
Chelsea|West Ha……
```

9 Reflection

Our initial goal was to create a basic DSL to query the Opta data source, and for this we have succeeded. All the main considerations have been met as follows:

9.1 Flexibility

As the solution has separated the grammar from the concrete underlying classes using mappings, we have added a good level of flexibility. A command entered in the parser can be switched out to a completely new concrete class without the need for any compilation or change to the users interface. In fact, the entire data source could be switched out, only requiring a new Executor and a few simple classes to apply the data source specific query commands.

The same is true for the columns and their associated names. It is possible to give user-friendly names to any of the columns, which becomes particularly useful when dealing with multi-table joins as we saw in the Event Entity example.

Also, with the IBespoke interface, we have given a way of dynamically creating helper functions, which will grow as the system becomes more complex. Developers can develop the functions as they include functionality, which takes away from the job of creating and maintaining help WIKIs or webpages.

However, this does lead to an interesting question; has the abstract degree of functionality required to give this level of flexibility lead to a DSL which is too general? In the grammar created as part of this dissertation, none of the verbs are specific to sports data. The *Entity* type could apply to any underlying data set, regardless of its contents; sports data or otherwise. The *Projection*, *Filter* and *Ordering* types are very typical of any data querying tool. It could be argued that this DSL is more akin to a general SQL or LINQ-type system. This doesn't necessarily invalidate it for our intended usage, as new verbs or abstract types subsequently added could tie it directly to sports data. For instance, it may be considered that the flight of the ball is so important to the users that a whole new type is created specific to the data involved in predicting the ball's trajectory in certain instances. This new type would then make the DSL very specific to the underlying sports data. It is noteworthy though that by increasing flexibility we may have moved away from the definition of a language specific to our domain.

9.2 Control of Data Visibility

One of the main business considerations was Opta's concerns over exposing the raw underlying data set directly to the users. As we have used controlled Entities, similar to a SQL Server view, on the underlying data, along with a controlled list of columns stored in configuration, we have full control over what data the user can view. In fact, this model could be very advantageous to Opta as the different views of data could be sold to the end user; they simply pay for the Entities they require. Should the user ever wish to upgrade, a configuration change to add the new Entities in is all that would be required to expose them.

9.3 Simple Parser

DSL parsers can become cumbersome over time as more grammar is added. Rather than an increasingly unwieldy switch statement, requiring a new compilation every time new functionality is

required, the parser here is very simple. As all the commands are passed to a factory which uses configuration to create instances, the parsers job is simply to split up the input string and pass each item to the factory method. In theory, no changes would ever be required to the parser, no matter how much new functionality was added.

9.4 Further Considerations

When designing an internal DSL, it is important to consider the base language for suitability. As previously mentioned, C# was used because of the authors familiarity in it. However, due to the config-based nature of the design, the strict type-checking of .NET has had to be worked around using reflection, thus moving away from some of the most useful parts of the .NET platform. This is not necessarily a problem but when a system must start working around the fundamental basis of a language, then it may be worth considering other options.

Naturally, there are some major pieces of functionality missing from the solution which would be essential in a production quality system. It will be obvious that there is currently no error checking at all in the code. This was intentional as to not make the code snippets too verbose and to concentrate on the job at hand. However, error handling would be of utmost importance here. The user would expect a good level of error reporting to indicate to them what they had done wrong, not just a lazily thrown stack trace. As reflection is at the core of the program, we cannot rely on compile time type warnings. If the mappings are wrong, the user types in an invalid piece of grammar, invalid columns are used, the condition given results in invalid SQL, we would need to send this information back to the parser in a helpful format, suggesting how to fix the issue.

Also, we have not gone as far as to implement aggregate functions, which would be required for a query language. These should just require a new type of abstract grammar type (such as having an IAggregate interface) for functions such as max, min, average, etc and a new implementation of the ISorter interface for grouping.

Our design currently only allows one instance of each abstract grammar type for a controller. In our examples, this is all that is necessary. However, it may be the case that more than one is required. For instance, in our aggregate functions, we may require two ISorter interfaces; one for ordering, one for grouping. We may also decide that an IFilter should be split into separate logical *where*, *and* and *or* functions. If this is the case then we would need to change the IFilter property on the Controller to a List. This should be relatively simple but we would need to take care that the concatenation of commands fits together when creating the underlying executing query. Currently, the grammar can be sent in any order; the entity doesn't not need to be first, as we have a natural flow of commands; entity, projector, filter and finally sorter, so can assume this in code when creating the execution string. However, if we had two filters then we would need a way of knowing which filter came first to stop getting underlying SQL of '*and X = Y where Z = A*' and suchlike.

The only other consideration is that it is possible to pull out even further into config. With the IEntity implementations for example, the only difference is the SQL command in the ExecuteCode method. It would be possible to create a single Entity class and pull the relevant execution code from config, therefore never requiring a compilation when changing the queries.

10 Conclusion

Domain Specific Languages can be highly useful in abstracting the complexity of a system away from a user and allowing them to use business grammar they understand to complete tasks that would otherwise be potentially out of their technical ability.

In the case of SportSL, this would allow traders to concentrate on their primary skills; to identify statistical patterns and create algorithms, without having to spend time learning a complex underlying data set and writing code or SQL. The abstraction has also proved useful in isolating the data provider's intellectual property away from the end user in a configurable manner.

Whether it be developing systems for household appliances for the internet of everything, communicating with autonomous cars or querying complex data sets for trading purposes, the need for semi-technical business personnel to be involved in the development of systems will vastly increase in the future. The overhead of training these individuals to be enterprise level developers or employing more and more experienced developers will be high, so simple, tried and tested DSLs could provide solutions for many of these issues.

Over time, we will no doubt see some standardisations across related industries. A standard DSL for developing against all refrigerators would make perfect sense. However, although DSLs have a long history, general adoption still seems in its infancy.

I believe the journey taken while compiling this dissertation has produced an interesting insight into how useful a thoughtful domain specific language can be in a relatively complex scenario.

11 References

[1] Congressional Research Service: High Frequency Trading: Overview of Recent Developments. https://www.fas.org/sgp/crs/misc/R44443.pdf

[2] London Metal Exchange: The Ring. https://www.lme.com/en-gb/trading/venues-and-systems/ring/

[3] A New Breed of Trader on Wall Street: Coders with a Ph.D. http://www.nytimes.com/2016/02/23/business/dealbook/a-new-breed-of-trader-on-wall-street-coders-with-a-phd.html?_r=0

[4] Domain Specific Languages; Fowler 2010

[5] Clojure for Domain-specific Languages; Kelker, 2013

[6] Domain Specific Languages; Fowler 2010

[7] How to: Verify that Strings Are in Valid Email Format (MSDN): https://msdn.microsoft.com/en-us/library/01escwtf(v=vs.110).aspx

[8] Domain Specific Languages; Fowler 2010

[9] Domain-Specific Languages IFIP TC 2 Working Conference, DSL 2009 Oxford, UK, July 15-17, 2009 Proceedings: Generic Libraries in C++ with Concepts from High-Level Domain Descriptions in Haskell A Domain-Specific Library for Computational Vulnerability Assessment; Lincke, Jansson, Zalewski, Ionescu 2009

[10] Domain-Specific Languages IFIP TC 2 Working Conference, DSL 2009 Oxford, UK, July 15-17, 2009 Proceedings. CLOPS: A DSL for Command Line Options: Janota, Fairmichael, Holub, Grigore, Charles, Cochran, Kiniry