Diploma Thesis

January 6, 2007

Data Mining within Eclipse

Building a Data Mining Framework with Weka and Eclipse

Julio Gonnet

of Kitchener-Waterloo, Ontario, Canada (01-710-672)

supervised by

Prof. Dr. Harald Gall

Patrick Knab

University of Zurich

Department of Informatics

s.e.a.l.

software evolution & architecture lab

Diploma Thesis

Data Mining within Eclipse

Building a Data Mining Framework with Weka and Eclipse

Julio Gonnet

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Diploma Thesis

Author: Julio Gonnet, julio.gonnet@access.unizh.ch

Project period: 20.05.2006 - January 6, 2007

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank all of the people that helped me throughout the framework's development and especially in the writing of my diploma thesis. Furthermore, I find the Weka developers deserve a special thanks for developing, documenting and most of all sharing this excellent data mining facility. Last but not least, I would also like to thank any future developers extending the framework and so contributing to its functionality.

Abstract

In the past years, there has been a great interest in the field of data mining. All around the world, larger companies have been investing vast sums of money in enormous data-warehouses and powerful data mining facilities, in the hope of extracting new information and so attain an economic advantage over other companies. With today's fast-growing technology, interoperability and tendencies for just-in-time systems, it is becoming more likely that one will use or depend on data that does not yet exist or belong to one's self. Furthermore, from a software engineering point of view, direct access to an application's database is not recommended, due to the entailing dependencies and coupling to the application. Ultimately, we will want to do a lot more than just mine a set of data from our local database. Be it a more powerful pre-processing of data, the integration with other business applications or the automatic creation of a report for management, we will not get around having to integrate data mining solutions in order to solve more complex problems.

In our specific case, we are especially interested in the analysis of software evolution and require a data mining framework that will seamlessly integrate with an IDE, an integrated development environment such as eclipse, already offering a large variety of components that produce software-related data.

In this thesis, we present the design and development of a data mining framework, integrating arbitrary data sources, existing data mining facilities and potential data consumers. In the first two chapters, we provide a brief introduction to the world of data mining, explain the need for integration and outline the framework's requirements. The tool's functionality is presented as a guided tour of the framework, followed by an in-depth technical look at the framework's main components. We then discuss the various highlights and problems encountered, present a simple proof of concept and round it off with our conclusions and an outlook to the framework's future development.

Zusammenfassung

In den letzten Jahren hat das Interesse an Datamining stark zugenommen. Überall auf der ganzen Welt haben grössere Firmen hohe Geldbeträge in umfassende Data-Warehouses und leistungsfähige Datamining Applikationen investiert, in der Hoffnung, dadurch neue Informationen zu erhalten und so einen ökonomischen Vorteil gegenüber anderen Firmen zu erzielen. Mit der heutigen, sich schnell entwickelnden Technologie, Interoperabilität und der Tendenz zu Just-In-Time-Systemen wird es immer wahrscheinlicher, Daten zu benutzen oder von Daten abhängig zu sein, welche noch nicht existieren oder uns schlichtweg nicht gehören. Ferner ist der direkte Zugriff auf die Datenbank einer Applikation vom Standpunkt des Software-Engineerings aus wegen der damit verbundenen Abhängigkeit und Koppelung zur Applikation nicht zu empfehlen. Letzendlich werden wir viel mehr unternehmen wollen, als lediglich Daten aus unserer lokalen Datenbanken zu extrahieren. Egal ob es sich dabei um die leistungsfähigere Vor-Verarbeitung der Daten handelt, die Integration anderer Business-Applikationen oder die automatische Erzeugung eines Reports für das höhere Management, wir kommen letztendlich nicht darum herum, bestehende Datamining Lösungen integrieren zu müssen, um komplexere Probleme zu lösen.

In unserem spezifischen Fall sind wir speziell an der Analyse der Software Evolution interessiert und benötigen dafür ein Datamining-Framework, welches sich nahtlos in eine IDE, ein Integrated Development Environment, wie Eclipse integrieren lässt, welches bereits eine grosse Auswahl an Komponenten anbietet, welche softwarebezogene Daten produzieren.

In dieser Diplomarbeit präsentieren wir das Design und die Entwicklung eines Datamining-Frameworks, welches frei wählbare Datenquellen, bestehende Datamining-Lösungen und potenzielle Datenkonsumenten integriert. In den ersten zwei Kapiteln stellen wir die Welt des Dataminings kurz vor, erklären die Notwendigkeit der Integration und umreissen die Anforderungen an das Framework. Die Funktionalität des Frameworks wird in Form einer geführten Tour durch das Framework vorgestellt, gefolgt von einem tiefgehenden technischen Blick auf dessen Hauptkomponenten. Danach diskutieren wir die verschiedenen Besonderheiten und Probleme, auf welche wir gestossen sind, präsentieren ein schlichtes *Proof of Concept* und schliessen mit den Schlussfolgerungen und einem Ausblick auf die zukünftige Entwicklungsmöglichkeiten des Frameworks ab.

Contents

1 Introduction **1**
1.1 What is Data Mining? 1
1.2 Why Mine Source Code? 1
1.3 Purpose of this Thesis 2

2 Integrating a Data Mining tool **3**
2.1 Why Integrate it? 3
2.2 Experimental vs. Productive Use 4
2.3 Data Mining on the fly 4
2.4 Goals 5
2.5 Requirements 6
2.6 Why Eclipse? 9
2.7 Why Weka? 9
2.8 Why Java? 9

3 A Brief Tour of the Framework **11**
3.1 Installation & Requirements 11
3.2 Startup 12
3.3 Data Providers 13
3.4 Sessions 15
3.5 Attributes 16
3.6 Probing the Data 18
3.7 Using the Weka Bridge 19
3.8 Results 21

4 Developing a Data Mining Plug-in **23**
4.1 The General Idea 23
4.2 A Brief Peek at Weka 25
4.3 The Key Elements and how they work 28
4.4 Packaged Providers 41
4.4.1 The SQL DataProvider 42
4.4.2 The HQL DataProvider 43
4.4.3 The ARFF DataProvider 44
4.5 A Data Mining View 45

5 SE Aspects & Problems encountered **47**
5.1 A Pseudo-Singleton Pattern 47
5.2 Lightweight XML Persistence 49
5.3 The Need for a Global ID 49
5.4 Preprocessing when Classifying 50
5.5 Exception Handling 50
5.6 Transitively Dependent Data 51
5.7 Memory - A Limited Resource 51
5.8 Other Usefull Stuff Worth Knowing 53

6 Proof of Concept **55**
6.1 Mixing Providers 55
6.2 The Metrics DataProvider 56
6.3 A Bridge to Weka 57
6.4 Transferability 57
6.5 Running Data Mining Example 57

7 Conclusions & Future Development **59**
7.1 Conclusions 59
7.2 Future Development 60

A Lightweight XML Persistence **61**

B Weka Hacks **71**

C DataObjectDefinition Annotations and Source Code **77**

List of Figures

3.1 Eclipse's "Show View" Dialog with the framework's view selected. 12
3.2 The framework's view is split into the depicted tabbed panes. 12
3.3 A screenshot of the *Providers* tabbed pane. 13
3.4 The tree-like representation of the framework's available DataProviders, displaying the creation of an instance through a DataProvider-implementation's context menu. 14
3.5 The merging of a DataSessions datasets provided by different DataProviders to a new single dataset. 14
3.6 The *Sessions* tabbed pane with the available sessions and their assigned DataProviders on the left and session-specific settings on the right. 15
3.7 The assignment of DataProvider instances to a given session. 16
3.8 The *Attributes* pane displaying the framework's attributes from several perspectives along with a selected attribute's details. 17
3.9 The *Probe Data* pane showing a session's resulting data 18
3.10 The *Probe Data* pane showing a DataProvider's data-contribution to a Session . . . 18
3.11 The *Probe Data* pane displaying the merging of two sets of metrics. 19
3.12 A screenshot of the *WekaBridge* pane with SessionRun information on the left and Classifier details on the right. 20
3.13 The top left corner of the Explorer's *Preprocess* panel showing the framework's WekaBridge button. 21
3.14 A built classifier's context menu from within the Explorer's *Classifier* panel, showing a menu item that passes the selected classifier back to the WekaBridge. 21
3.15 A screenshot of the *Results* pane, displaying the classification results of our Iris example. 22

4.1 The framework, its immediate context and the interactions or rather flows of objects. 24
4.2 The framework represented as layered plug-ins. 25
4.3 The assignment of DataProviders to DataSessions 29
4.4 A simple diagramm of the DataSet's internal structure 31
4.5 DataSessions, SessionRun and the Weka Explorer in context. 33
4.6 A UML Class Diagram of the DataObjectDefinition Wrapper 36
4.7 The Framework, WekaBridge and its tool instances in context 40
4.8 The WrappedWekaObject's composition, export and description possibilities and immediate context. 41
4.9 The HQL DataProvider as plug-in layers 43
4.10 The three levels of the HQL DataProvider 44

5.1 A UML Diagram of the Pseudo-Singleton Pattern 48
5.2 A dataset depending on another to merge with the rest, basically requiring more than one merging cycle. 51
5.3 An extension to Figure 4.3, displaying the assignment of not only DataProviders, but also DataSessions to a given DataSession. 52

6.1 The mixing respectively merging of DataProvider's DataSets 56

Chapter 1

Introduction

1.1 What is Data Mining?

In an ever so growing world of technology, where hardware becomes cheaper and better by the day, new opportunities arise, that were previously unimaginable or simply not feasible. The same applies to data mining. With the increasing possibility of storing information at a reasonable price wherever it is produced, we now have a means to collect and track nearly every conceivable event that takes place, such as a weather phenomenon, an earthquake or a heart attack. More importantly, we can store and track any decision taken, such as a parliamentary vote, the sale of stock options or the purchase of a product, such as milk or a vehicle.

Unfortunately, as our technology and databases grow, so does the gap between creating the data and actually understanding it. There is an increasing shortage of capacity to process and interpret all this data. Companies are spending a lot of money to store information, without it ever paying off. Hence the need for the automatic or at least semi-automatic processing and interpretation of data, to ultimately attain new information, that will hopefully be of use and lead to an economic advantage.

Data mining is defined as the process of discovering patterns in data [IHW05]. Despite the recent interest in data mining, the underlying idea is not really new. The search for patterns dates back to the dawn of mankind, where hunters sought patterns in animal behavior. What is new, is the ever increasing size, diversity and easy accessibility of these vast worlds of data. Similar to the domain of physics, we strive to come up with a new theory or law explaining an observed phenomenon, ultimately allowing us to make non-trivial predictions for new, unprecedented situations. The more information we store, the better our chances are to explain an observation.

1.2 Why Mine Source Code?

I guess it is safe to say, that not all software is perfect. In the contrary, I find that the quality of today's software is deteriorating, as we attempt to solve increasingly more complex problems through the use of more complex software. This observation is not new and in fact, has already been observed and documented in [LRW$^+$97] and is generally known as *Lehman's Laws of Software Evolution*. Up and until now, these laws have proved to be rather accurate. Unfortunately, Lehman's laws remain rather vague, indicating that a deterioration will occur, but we are still oblivious, as to what exactly causes or precedes this deterioration and what can be done to prevent it.

The analysis of both software and software evolution present excellent candidates for the search of patterns governing the quality and evolution of software. Discovering such a pattern would provide software producing companies with an enormous commercial advantage over their competition. Wouldn't it be great to be able to accurately predict which source code is likely to cause the most problems, prior to releasing it?

Recently, the Software Engineering and Architecture Lab (S.E.A.L) of the Department of Informatics at the University of Zürich has invested much time and effort in collecting, processing and interpreting data, related to Software Analysis and Software Evolution Analysis, such as the Mozilla project's source code repositories [FPG03]. Furthermore, the group has already successfully applied data mining techniques to predict defect densities in source code files [Kna05] and later even published their findings in [KPB06]. At present, the S.E.A.L. group is working on an integrated environment aimed at supporting the analysis of software evolution, called *evolizer*. One of evolizer's many goals is, to extend the work done in [Kna05] and integrate a data mining framework, allowing a more comfortable means to process and data mine any data produced and managed by the evolizer environment. Hence the need for a data mining framework that can seamlessly integrate with evolizer.

1.3 Purpose of this Thesis

The motivation behind this thesis is to actually implement a framework for the evolizer environment, integrating existing data mining tools and facilities and so allowing for a flexible, yet comfortable means to data mine any data provided by evolizer, or for that matter, any other component. The objective of this thesis is not to discover new cutting-edge patterns related to the evolution of software, but more to generally encourage and allow the discovery of these patterns. In a simple sentence, this thesis' objective is to generally *make it happen*.

Hence, the core of this thesis is its underlying implementation, which is not an actual part of this document. Most of the time invested on this thesis, was spent on designing and implementing a first release of our data mining framework, that is capable of integrating and interoperating with evolizer's environment. This document represents the written part of the thesis, a documentation of the framework's architectural design and development. As this thesis was written for the S.E.A.L. group and was under its supervision, it obviously focuses on software engineering aspects, such as the framework's architecture, its vital components and their interoperability, the problems encountered, etc. As the thesis aims at integrating existing data mining facilities, it will assume the reader is current in the field of data mining and thus not further describe data mining algorithms, as they are not relevant to the framework's development.

Furthermore, this thesis is thought as a general documentation to the framework and is structured as follows. After providing a general motivation and explaining the reasons behind this implementation, we shall move on to a brief tour of the framework's first release, providing an insight to its functionality and structure. Following this tour, we will get down to business and have a rather detailed look at the framework's internal components and functionalities. With that out of the way, our next chapter will detail a selection of highlights and problems encountered throughout the development. Last but not least, we will round it all off with a brief proof of concept, our conclusions and an outline of possible future work.

Chapter 2

Integrating a Data Mining tool

2.1 Why Integrate it?

In a simple sentence - because eventually you are going to want to do a lot more than just data mine. I don't pretend to know what you are going to want to do with the raw data, the results or byproducts, but I am confident you *will* want to do something more. Therein lies the beauty of the problem. It is easy to integrate A with B and C, if you know that they all are, but it becomes quite a challenge when you want to integrate A with anything. So in a manner, I am not really integrating, as I do not have a tangible B and C, but merely preparing or rather enabling an easier integration of an existing component.

Real world data mining applications greatly differ from the simple examples presented in data mining books, such as [IHW05]. You will find yourself spending an extensive amount of time finding out what data you or your company has, where it is "burried", how to access it and how to preprocess it so that your data mining tool can recognize it. Building a classifier or actually classifying new data will only take a fraction of that time. If you intend to optimize the data mining process, you are going to need something more than just a mere tool. Your options are to either build a comprehensive data warehouse or integrate a data mining tool with any other component that can help you data mine.

With time, you will find yourself accessing data through defined interfaces (Web Services, data producing applications, etc.) instead of a direct access to a database. With today's technology, interoperability and tendencies for just-in-time systems, there is a good chance that the data you intend to mine is either not yours (example: analyzing some other company's web-shop product stocks and prices) or does not reside on a database at all (data might be created on demand). Considering the well tried data hiding principles in software engineering, it is neither recommended nor wise to directly access an application's database for information. Especially not if you are going to base strategic company decisions on that data.

As Weka [IHW05] accept its data as a single relation, the preparation of this data will have to take place outside of the tool and is thus neither supported, nor automated. The same applies for any tool-generated results. You are likely to require them as an input for an arbitrary application, in a report or at least for in a database. They will be of least use to you on your screen, where they are out of context[1] and the only means to reuse them is via the copy-paste command. Why execute multiple steps in sequence, when you can connect them all together and run them as one?

In the long run, you either acquire an elaborate data mining suite, build a data warehouse or integrate your tool of choice with your company's remaining IT.

[1]At best you will have an ID and its corresponding class, whereas you will still be missing a semantic meaning to that class and an adequate visualization of the object or result at hand.

2.2 Experimental vs. Productive Use

While dwelling over the many requirements and architectural aspects, one prominent question did arise - Is this tool thought for experimental or productive use. Generally, the answer to this question is rather simple. This is a diploma thesis - of course it is experimental. But there is quite a difference in the approach. At a first glance, we are building a means to easily data mine release history. Basically all we need is the data, a data mining tool and something to connect both together. Get it working for a few algorithms and we can take it from there. Giving it a bit more thought, we soon realized, that the solution at hand was neither experiment-friendly nor flexible. Sure, you could run a set of data mining algorithms on some predefined data, but it will probably all be hardcoded - painfully configurable at best. What if you wanted to do more?

Personally, I consider a tool that does rather little in a limited and restricted fashion to be productive - one should not require a Ph. D. to operate it. Productive tools are designed to reduce the variables and complexity in order to increase productivity. And herein lies the problem: we do not know which variables to reduce or how complex the tool's domain will become. Building a tool that connects data with a few data mining algorithms may sound great at first, but fades when we realize that we have just reinventing the wheel - there are plenty of data mining tools out there that already do it better than whatever we can implement in six months. You can run several "experiments" on it, but the tool itself is not experimental. Today's fancy cutting edge experimental feature will be "water under the bridge" by tomorrow. It is very likely that during the first experiments, somebody is going to come up with a more suitable data mining algorithm or presentation of the results - and our tool isn't going to support it. It is per se defined to solve a given set of problems.

So what defines an experimental approach? A trivial answer would be the opposite to my definition above: do a lot in an unlimited and unrestricted fashion - basically keep as many variables as possible, as somebody might want to use them. It may be impossible to allow an unlimited and unrestricted use, but a bit of flexibility will take us most of the way. The ability to bend without breaking is key to an experimental approach. Experimental is when a user can actually *contribute* to the tool with new cutting-edge features and actually make a difference.

So which approach should we pursue? Well, ideally both. As part of a the evolizer project the experimental approach seems obvious. As there is no general experience in the mining of release history, most non-experimental users will not know what to mine for - one less reason to support the productive use. On the other hand, researchers outside of the project might want to reproduce the results to verify them. Over time, thanks to the results gained through experimental use, software companies might seek this new knowledge and want to apply it. After all, we see no reason not to increase productivity. Minimizing any tedious steps taken in preparation of the data is already a great relief.

2.3 Data Mining on the fly

One of the main challenges in data mining is the preparation of the data as a single relation. This often requires extensive knowledge of the existing data (schemas, login data, accessibility) and on relational databases (SQL, relational algebra, etc.). As the preparation of data usually leads to a new relation, you are very likely to require *write* access to a database to create it. If you are lucky, your company will already have a data warehouse at your disposal - if not, you are looking at some serious trouble.

Within larger companies, acquiring the services of a database can be quite tricky and tedious. You can try installing a database application on your workstation or contact your company's database support group. The former will probably result in a nightmare of bureaucracy, licenses, security, access rights, support, etc. As to the later, unfortunately, most database administers are very conservative when it comes to creating new databases and granting a *write* access - with good reason. There is a good chance you will be replicating very sensitive data onto your own database. And it is unlikely that you will be able to solve this issue by creating views on existing tables. It seems that the people with the domain knowledge to data mine are unfortunately not the ones with all the technical liberties, such as an unlimited access to open-source and commercial tools, bandwidth, processing power, etc.

All in all, there is no obvious reason as to why we use a relational database to prepare the data. It is not even comfortable (SQL, bureaucracy, access rights, network bandwidth, etc.). You will find that in the domain of software analysis, most of the data must first be created by the appropriate tools. If you are lucky, the data will be persisted onto a database, but probably not the way you would like it.[2]. So yes, you can *dump* the produced data into a database and process it, but why would you do that? All you are doing is storing very short-lived snapshots. Every time a parameter changes, you have to reproduce either fragments or the entire data, store it appropriately, preprocess it and rerun the data mining algorithms. You will basically be dropping the old data and reproducing it time and again - quite a lot of work for so little flexibility.

So why not skip the persisting of snapshot data, that is only good for a session and instead collect respectively produce all the data required and merge it to a single relation, all within the workstation's memory - basically doing it *on the fly*. Persist the setup used to produce the relation, not the relation itself!

As a matter of fact, the data mining tool of choice for this framework, Weka, loads *all* the data into memory before mining it. All our framework would be doing, is basically hold a second pointer per data element. Thanks to technological advances, memory has become a relatively cheap resource. Memory is likely to be easier and cheaper to upgrade, than to get your own database instance. And if your workstation's memory is not sufficiently large, you could still apply the concept of windowing or lazy loading. Regarding security issues, building the relation in memory is just as risky as viewing the data in a commercial database-client. When mining *on the fly*, any changes to access rights on sensitive data will apply immediately. Furthermore, you avoid the replication of data and the problems[3] associated to it. If you need to persist the processed relation, you are still at liberty to export it to any format you like.

2.4 Goals

Before we dive into the flood of requirements one should consider, we would like to abstract a little and point out the main goals of this project - basically describe what we intend to achieve. we will do so by answering the following four questions:

Where and how do we get our data? To be honest, I do not pretend to know how much data there will be, where it will come from, what format it will be in, how to combine it with other data and most importantly, how to access it all. But we do not really need to know all this. What we do know, is the format expected by most data mining tools - namely a single relation as defined by Codd [Cod70]. So what we really need is an *extendible* architecture that will accept an arbitrary number and variety of data sources (files, Objects, relations, streams, etc.) and merge them to that single relation.

And so, we can only partially answer this question - we transform an arbitrary number of homogenous representations of the data to a single relation and pass it on to the data mining tool.

[2]It is unlikely that you will have direct access to the database. Not to mention an understanding of the schema.
[3]There is a risk that the data may be forgotten and so cause problems in the future.

We delegate the rest of the question to those with an extensive knowledge of the data - the ones transforming the arbitrary number and variety of data to the framework-specific representation we are expecting.

In order to put the framework at test, we have provided framework extensions that implement the access to relational (via SQL) and object-relational (HQL[4]) databases.

How do we process our data? Again, I do not pretend to have an extensive knowledge of the data mining domain. And as before, we do not need to - Weka and other data mining tools have done a wonderful job putting their domain knowledge into practice by implementing extensive facilities with rich intuitive user interfaces. There is no need nor sense in us "reinventing the wheel". As a user, you will be much more efficient building classifiers through the tool's original user interfaces, than through our "graphical interpretation" of the tool's libraries. This project is about integration, not re-implementation. If the tools at hand do not suit your needs, either extend the tool[5] or add a new tool as an extension to the framework.

What to do with the results? As you may have already suspected, I have no clue as to what you might want to do with the results. But one thing is for sure, you are going through a lot of trouble to produce them, so it would be beneficial if you could use them a bit more efficiently. Sure, It is great to see the results on a screen, but most data mining tools can do that. It would be better, if you could seamlessly process the results and any by-products[6] without having to copy-paste them to their destination.

Why are we doing all this? It basically all turns around integration. This is not about creating something radically new, but rather simply *making it happen* - putting it all together, without really knowing what *all* really is or will be. In a simple sentence - this framework is designed to ease the task of data mining.

2.5 Requirements

This section will look into the following selection of requirements that came to mind prior to and during the development of this framework. The requirements listed are neither complete nor formal. They are intended to point out certain aspects and features of the framework and so provide an outline before going into more detail in the chapter 4.

Independence to the eclipse platform After extensive experimental research, one should be able to productively apply the acquired knowledge outside of the eclipse platform without having to re-implement the framework or data producers. This of course only applies to data producers that are not dependent of the eclipse platform to produce their data. The data mining project should first be configured within eclipse and then run outside of eclipse (with the stored configuration), on a different platform, such as an application server.

Allow any data that can map to a relation As one of the goals is to ease the task of data mining, the framework should abide to it by providing data provider extensions that access both relational

[4]Hibernate Query Language is a powerful, fully object-oriented SQL-like query language.

[5]In the case of Weka, you can extend the tool by implementing your own classifier or filter.

[6]Anything that was produced as a result of or simply during the data mining process, such as graphs, charts, statistics, detailed classifier evaluation results, etc.

and object-oriented data (persisted through Hibernate [KB04]. Both providers should be configurable and allow access to any database using SQL respectively HQL. All other data should be accessible through customized `DataProvider` objects, homogenizing the data for later processing. The idea is to tap any conceivable data source whose data can ultimately map to a relation.

Inspection of Data It would be beneficial to view the data as it is being merged to a final relation.This encourages a better understanding of the data and helps reduce configurational mistakes. It is much easier to spot a mistake in the data of a single data provider than to look for it in the final relation. This allows for a simplified verification of a single data provider's configuration.

Independence of a data mining tool Keep in mind that a data mining tool may be great today, but might not suit your needs by tomorrow. A project should not only be able to choose between data mining algorithms, but also between tool implementations. The framework's success should not depend on the underlying data mining libraries it cannot control nor change. This calls for properly defined interfaces between the framework and the data mining libraries.

Do not change other source code or data The framework's acceptance depends on its ability not to interfere with its context. Wherever possible, the framework should discourage changes to other application's source code or data models.

Translation As a result of the latter requirement, the framework needs a means to translate values from their original context to a new context, seeing that we should not change them. This requirement is key for the definition of attributes and the identification of data tuples. Back in the day, when memory and storage space were rare, developers optimized by using short cryptic names. As data will normally outlive us, there is a good chance you will be working with very old data, full of acronyms and cryptic names. As if that was not bad enough, each data provider will probably support its own primary key domain, which will be incompatible with other primary keys.

In avoiding replication and changes to the database, are only option is to translate, thus allowing a merging of two differently named attributes (from two different data provider) to a single one or the joining of two tuples with different primary key domains.

Traceability It would be good to know who defined a given attribute, who provided data to that attribute and where it can be found within its original data source. For example, it would be rather difficult to apply a rule based classifier built on translated names to the original data. When translating attribute names and primary keys, it is important to keep track of what exactly took place. Ideally, translations should be reversible to guarantee traceability. Translations should also be made visible to the user. In terms of data mining, despite the primary key's redundant character, it is vital to keep an association between tuples and their corresponding id. Knowing that five objects have been classified as a certain type is interesting, but not very useful if you do not know which five they are.

Verification and Reproducibility It is vital that all results can be reproduced and thus verified with the same or another data mining tool outside of the framework. Due to the integration of only one data mining tool at the time of this writing, the framework must be able to export the data to a Weka-specific format (for example the ARFF format) in order to verify the results. Furthermore, the framework should be able to store and export its current configuration. This allows the transfer of a data mining project both over systems[7] and time. The exported configuration should be comprehensible outside of the framework's context.

Attribute and Object Definitions Most sources of data (relational databases, Hibernate, etc.) offer limited information on attributes that is difficult to retrieve and parse. The framework should therefore support the definition of attributes and data objects (HQL can return a column of Java objects). Attribute definitions linked to actual data are unlikely to change. Manually defining each attribute can be tedious and time-consuming. As required, to ease the task of data mining, the framework should infer an attribute definition (according to given heuristics), if it is not explicitly defined. Data providers should be allowed to provide any definitions they use for their data. Definitions should be persisted between framework sessions.

Programmatically extendible If the framework does not support a given feature, it should be possible to add the feature via an extension. The idea is to extend the framework with new data producers (plug-ins with data to mine), data processors (data mining tools and libraries) and data consumers (plug-ins that are interested in results and by-products).

Non-programmer friendly A user with little or no knowledge in programming should be able to configure the framework and both extensions packaged with it, namely the SQL and HQL DataProviders. The framework should therefore permit the declaration of attribute and object definitions, jdbc drivers, hibernate configurations, data providers, etc. through the eclipse plug-in extension mechanism.

Configuration persistence A data mining project is likely to require more than one session of work. The ability to automatically store the framework's current configuration is key to the previously described "easing the task" goal. Nobody likes losing several hours of work and having to manually restore an application's state, just because the application could not save it.

Should not rely on a database It is unreasonable to assume that a user will install a database instance on his or her workstation just to run the framework. The framework is meant to be lightweight and easy to use. Running it off a database will bring upon many complications a user should not have to deal with. Having to install a database instance for a plug-in is not exactly considered to be lightweight. On the other hand, a database instance can solve certain problems associated to the framework's limited memory. The framework should therefore not rely on it, but rather support it as an optional feature.

Must run on eclipse As this framework is thought to be the data mining back-end of S.E.A.L.'s evolizer project, the framework must run on the eclipse platform. Most of evolizer's components are designed to run on eclipse technology and are potential data-providing candidates for our framework.

Must look like eclipse If the framework is destined to run on eclipse, it should at least try to look and feel like eclipse. It is much easier for new users to adjust to a new application if the user interface looks and feels familiar. User interfaces are often neglected, despite their enormous impact on user satisfaction. People do not buy cars because of their technical specifications and great tools are not worth much if one cannot use them.

[7]Provided both systems have the same setup (data providers)

Account for third party user interfaces Data providers could be rather complex to configure and may require a significantly larger graphical user interface than assumed. The framework should account for this and grant provider-specific user interface space whenever requested.

2.6 Why Eclipse?

Apart from the explicit requirement related to S.E.A.L.'s evolizer project, we would like to point out a few other reasons why we chose the eclipse platform for our data mining framework.

First and foremost, eclipse is the IDE[8] of choice at the Department of Informatics of the University of Zürich. There is not much point in developing a framework that runs on a platform nobody uses. As a student of this department, I am unfortunately only familiar with this environment. A proper analysis of other all possible platforms or the acclimatization to a new environment would have consumed to much valuable time.

The basic idea is to extend a set of data mining tools by integrating them into a larger extendible platform, such as eclipse, which is based on the OSGI framework), thus allowing our tools to interoperate with almost anything. As an IDE, eclipse is loaded with potential and future data producers of interest to the evolizer project - yet another reason to go with eclipse. Furthermore, the eclipse technology is widely accepted, also used commercially (as IBM's WebSphere), has a great community for support and is very well documented [EC06]. By choosing eclipse, we save ourselves the trouble of implementing an extendible framework's base code, the problems associated with interoperability, updates and releases and the persistence of the framework's state.[9]

Last but not least, the platform's PDE (Plugin Development Environment) is a great infrastructure for the development, testing and running of potential data providers and consumers as eclipse plug-ins. The framework's success greatly depends on the multitude and choice of data providers and consumers. A developer-friendly environment is thus key to the framework's positive outcome. Personally, I find it will be difficult to find more flexibility and extensibility elsewhere.

2.7 Why Weka?

For the purpose of testing the framework, it was necessary to provide at least one data mining tool to work with. Ironically, we did not choose Weka because of its data mining features, but rather because it is GPL licensed, has stable releases, is well documented (see [IHW05]), widely accepted and familiar to the S.E.A.L group (see [Kna05]). Furthermore, it is experimental in nature (it offers the ability to be extended), is programmed in Java (it is likely to run without problems within eclipse) and provides an excellent graphical user interface. The availability of its source code and JavaDocs ensures an easier integration to the framework.

2.8 Why Java?

I guess it is safe to say, that Java is by far not the best language for "crunching numbers" - so why use it to data mine? As with eclipse, Java is the language of choice of the Department of Informatics. As a student of this department, I am sufficiently proficient in Java, but not really in any other

[8]Integrated Development Environment. Basically an environment intended to support the development of software.
[9]Automatically persisted upon shutdown through eclipse's IMemento-structure.

language. I imagine the development of a framework in a language I do not comprehend to be rather difficult.

Regarding speed, we are willing to bet that you will be spending significantly more time getting your data together than actually building a classifier with it. The popularity, interoperability, wide support, the multitude of libraries all contribute to the extendibility of the framework and thus aid the task of getting your data together. Yet, if you are in need for speed, you probably already know what you are mining and are not really into experiments or integration. You are most likely better off with a powerful standalone data mining tool.[10]

[10]Oracle and IBM provide the following integrated tools optimized for data mining:
http://www.oracle.com/technology/products/bi/odm/index.html
http://www-306.ibm.com/software/data/iminer/

Chapter 3

A Brief Tour of the Framework

Before taking a closer look at the framework's architecture and components, we would first like to introduce the framework from a user's perspective. In order to do so, we shall run one of the many classical data mining examples featured in [IHW05]. It may seem a little unusual to go ahead and present the final result, a working version of our framework, without at least having presented the *big picture* or its architecture in general. We find that the following chapters are rather technical and thus easier to understand, if the reader has at least seen the framework in action and was able to get a first impression. Hence, this chapter is to be considered an informal tutorial.

3.1 Installation & Requirements

This chapter's tutorial does feature adequate screenshots, but for a better viewing and understanding of the framework, we recommend you install the plug-ins required by the framework and run this tutorial on your own. Unfortunately, due to the constraints of paper and printer resolutions, the presented screenshots are not nearly be as good as their counterpart in eclipse. Furthermore, you may want to switch between tabs or change certain configurations to better understand their effects. In order to run this tutorial, you will require the following:

- Eclipse 3.2 running on Java 1.5
- An installed version (preferably v8.1) of PostgreSQL
- access to org.evolizer.iris_dataprovider-plug-in's source code for the following:
  - `org.evolizer.hibernate_dataprovider.util.CreateIrisDataSet.java`
  - `org.evolizer.hibernate_dataprovider.util.iris.data`
  - `hibernate.cfg.xml`
- and the following framework plug-ins:
  ```
  org.evolizer.iris_dataprovider_1.0.0.jar
  org.evolizer.metrics_dataprovider_1.0.0.jar
  org.evolizer.sql_dataprovider_1.0.0.jar
  org.evolizer.weka_plugin_1.0.0.jar
  org.evolizer.gplweka_1.0.0.jar
  ```

For information on how to install the plug-ins, please consult [EC06], respectively [Dou05] for the installation of PostgreSQL. To run the iris example, you will have to populate a database with the iris dataset. To comfortably do so, we have packaged executable code that will read the data from the `iris.data` file, create `Iris` objects and persist them with Hiberante [KB04]. Prior to executing `CreateIrisDataSet.main()`, you should reconfigure the `hibernate.cfg.xml` configuration file, as described in [KB04], to match your database (user name, password, etc.). It is generally advisable to re-deploy the `org.evolizer.iris_dataprovider`-plug-in, as described in [EC06], containing these modifications. This will grant you more flexibility, such as the use of a database instance other than PostgreSQL.

3.2 Startup

The framework's graphical user interface consists of a single eclipse view, accessible through eclipse's Window - Show View - Other menu as depicted in Figure 3.1. At present, the framework and its view are both named after its underlying data mining libraries - Weka. Opening the view will initialize the framework and load any default settings or previous states. Because the user interface is an eclipse view, any changes done to the view do not need to be saved. Instead, they are immediately effective.

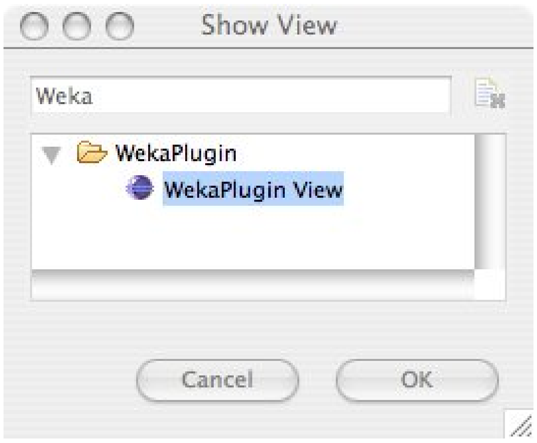


Figure 3.1: Eclipse's "Show View" Dialog with the framework's view selected.

As seen in Figure 3.2, the framework's view consists of seven tabbed panes, each providing a different view and so supporting a different part of the data mining process. As the current release does not yet support the notion of a configuration, its corresponding tabbed pane is still empty.

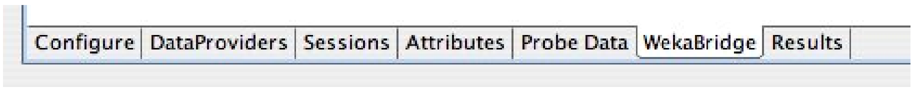
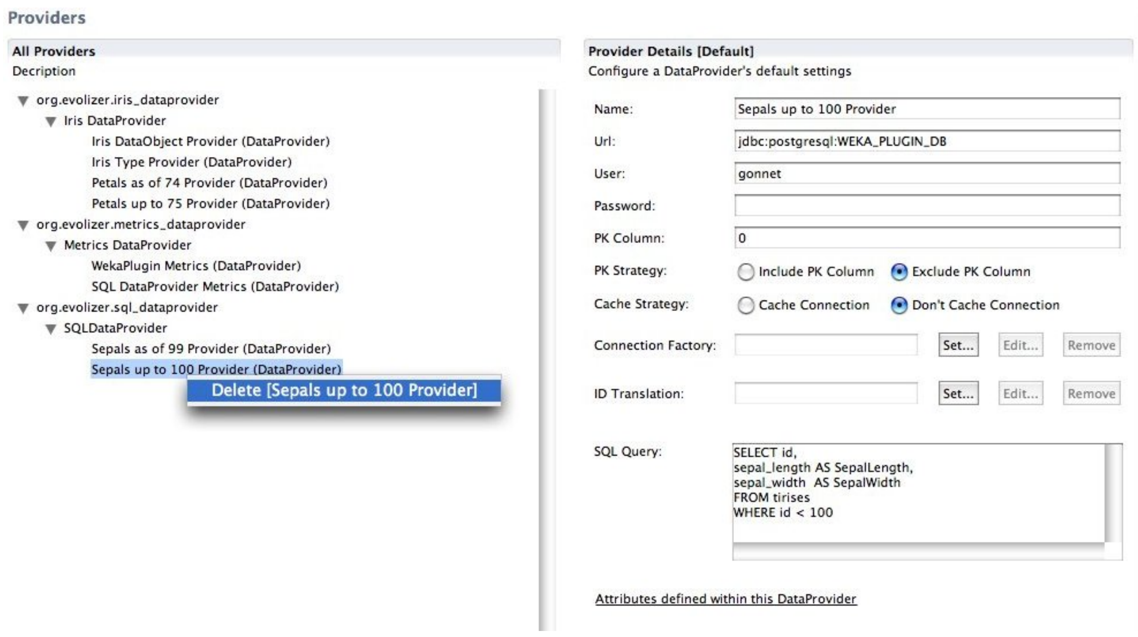


Figure 3.2: The framework's view is split into the depicted tabbed panes.

3.3 Data Providers

The next tab and data mining step in line is the configuration of DataProvider objects. A DataProvider is, as the name already gives away, a domain-specific means to feed data to the framework. Each DataProvider implementation represents an access to a potential data source. DataProviders are declared and packaged as plug-in extensions to the framework's plug-in. Upon startup, the framework will automatically load all available DataProviders and preconfigured DataProvider instances. In our case, it will automatically create the DataProviders necessary to run the tutorial's iris example. Figure 3.3 shows a screenshot of the Providers pane. As you can see, the pane is divided in two, providing general information on available DataProvider implementations and instances on the left and a selected DataProvider's configuration on the right. Each DataProvider is designed to represent its configuration in its own way, so you may encounter a different image, depending on which DataProvider instance you select.



Providers

All Providers
Decription

- org.evolizer.iris_dataprovider
  - Iris DataProvider
    - Iris DataObject Provider (DataProvider)
    - Iris Type Provider (DataProvider)
    - Petals as of 74 Provider (DataProvider)
    - Petals up to 75 Provider (DataProvider)
- org.evolizer.metrics_dataprovider
  - Metrics DataProvider
    - WekaPlugin Metrics (DataProvider)
    - SQL DataProvider Metrics (DataProvider)
- org.evolizer.sql_dataprovider
  - SQLDataProvider
    - Sepals as of 99 Provider (DataProvider)
    - Sepals up to 100 Provider (DataProvider)

Delete [Sepals up to 100 Provider]

Provider Details [Default]
Configure a DataProvider's default settings

Name:	Sepals up to 100 Provider
Url:	jdbc:postgresql:WEKA_PLUGIN_DB
User:	gonnet
Password:	
PK Column:	0
PK Strategy:	Include PK Column [✓ Exclude PK Column]
Cache Strategy:	Cache Connection [✓ Don't Cache Connection]
Connection Factory:	Set... Edit... Remove
ID Translation:	Set... Edit... Remove

SQL Query:

```
SELECT id,
sepal_length AS SepalLength,
sepal_width  AS SepalWidth
FROM tirises
WHERE id < 100
```

Attributes defined within this DataProvider

Figure 3.3: A screenshot of the *Providers* tabbed pane.

As depicted in Figure 3.3, the left-hand side is organized as a tree, with plug-in-names as the top-level nodes. Nearly everything in eclipse is provided by a plug-in, so why not a DataProvider? Each Plug-in can provide several DataProvider implementation, each of them represented as a named child of their providing plug-in. An implementation can be instantiated several times, each one being individually configurable and appearing as a named child to the implementation. Figure 3.4 shows the implementation's context menu (accessible via a right-mouse-click) providing a means to instantiate it. The instance's context menu is visible in Figure 3.3, where it allows the removal of the selected instance.

The right-hand side provides fields and buttons to configure the DataProvider-instance. In our case, it is an SQL DataProvider responsible for providing the sepal-data of the tuples with an id up to 100. The name field allows you to name the instance to your liking, the url, user and password field are required to establish a JDBC-connection to the relational database. You can also declare an implementation of the ConnectionFactory-interface, if you require programmatic control over the connection. Each DataProvider is required to map each data-tuple with an id. The pk-column-

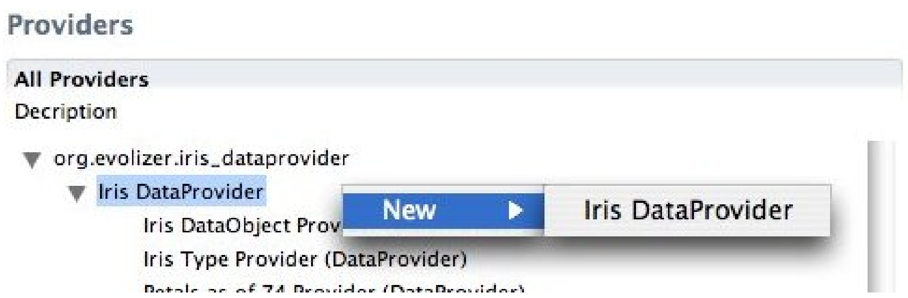


Figure 3.4: The tree-like representation of the framework's available DataProviders, displaying the creation of an instance through a DataProvider-implementation's context menu.

field identifies the column containing the id (either by column-name or index, starting with 0) and the pk-strategy determines, whether the id is included into the data or simply paired with it, to identify the tuples (exclude). As DataProviders are developed for different domains, each id will have its own domain. When merging data, the framework requires a means of translating an id-object from one domain to the other. The id-translation field and buttons are intended solve this. Last but not least, a user can view and modify the query used to extract the data.

For the purpose of merging DataProvider-data, we will run three separate examples. As a first, we will access all of the iris data from a single DataProvider, the *Iris DataObject Provider*. It is an extension of the HQL DataProvider, designed to extract data stored with the object-oriented Hibernate persistence layer [KB04]. To demonstrate the merging of data from different providers, we will access fragments of the same data through different DataProvider instances and implementations. The petal an iris type information will be provided by three HQL DataProvider extensions, the sepal information through two SQL DataProvider instances. Figure 6.1 graphically represents what we intend to achieve - the merging of datasets originating from different DataProviders to a single coherent dataset. Our third example uses the Metrics DataProvider to merge the Metrics corresponding to two separate Java Projects to a single relation.

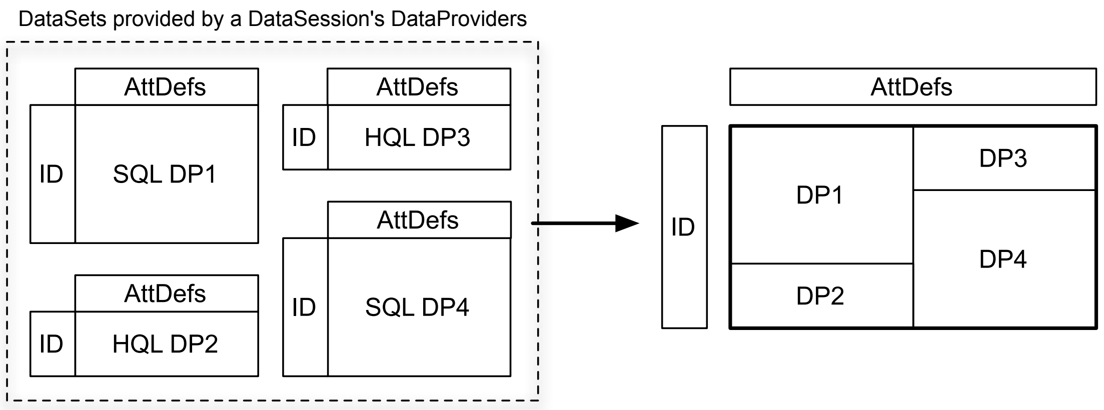
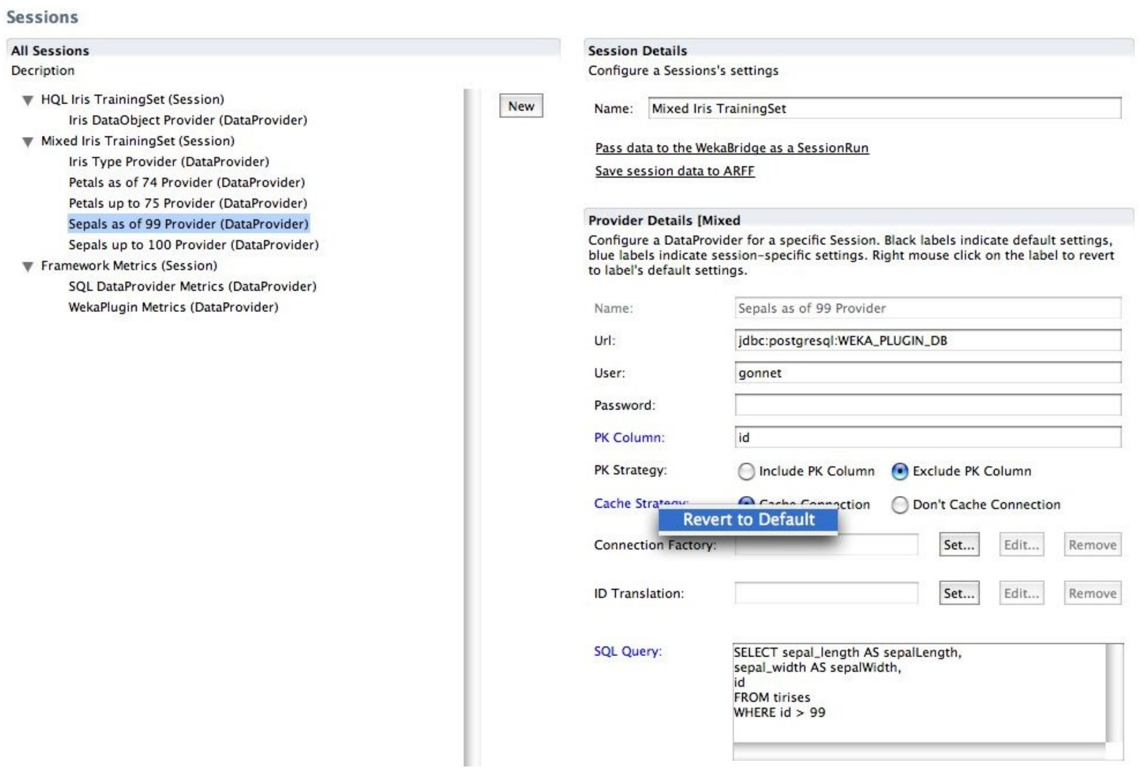


Figure 3.5: The merging of a DataSessions datasets provided by different DataProviders to a new single dataset.

So now that we have all of our DataProvider's configured, we can move on with our examples and merge the underlying data.

3.4 Sessions

Because we have three examples to run and a bunch of configured DataProviders to do so, we need a means to group them and so represent smaller data mining projects or work-units. Hence the use of Sessions. A Session ultimately defines the final relation to be used by the underlying data mining tool. So in our case, with our three examples, we will require three Sessions. Figure 3.6 shows the *Sessions'* pane, consisting of three sections. The left-hand section provides information on the available sessions and assigned DataProviders. The right-hand upper section permits the naming of a session and offers certain session-specific functionality. The right-hand lower section provides session-specific configurations of a DataProvider.


Sessions

All Sessions
Decription

- HQL Iris TrainingSet (Session)
  - Iris DataObject Provider (DataProvider)
- Mixed Iris TrainingSet (Session)
  - Iris Type Provider (DataProvider)
  - Petals as of 74 Provider (DataProvider)
  - Petals up to 75 Provider (DataProvider)
  - Sepals as of 99 Provider (DataProvider)
  - Sepals up to 100 Provider (DataProvider)
- Framework Metrics (Session)
  - SQL DataProvider Metrics (DataProvider)
  - WekaPlugin Metrics (DataProvider)

Session Details
Configure a Sessions's settings

Name: Mixed Iris TrainingSet

Pass data to the WekaBridge as a SessionRun
Save session data to ARFF

Provider Details [Mixed
Configure a DataProvider for a specific Session. Black labels indicate default settings, blue labels indicate session-specific settings. Right mouse click on the label to revert to label's default settings.

Name: Sepals as of 99 Provider
Url: jdbc:postgresql:WEKA_PLUGIN_DB
User: gonnet
Password:
PK Column: id
PK Strategy: ( ) Include PK Column (•) Exclude PK Column
Cache Strategy: (•) Cache Connection ( ) Don't Cache Connection

Connection Factory:

ID Translation:

SQL Query:

```
SELECT sepal_length AS sepalLength,
sepal_width AS sepalWidth,
id
FROM tirises
WHERE id > 99
```

Figure 3.6: The *Sessions* tabbed pane with the available sessions and their assigned DataProviders on the left and session-specific settings on the right.

It is important to understand, that a DataProvider may be used by more than one session and can be individually configured, depending on its use. The configuration accessible via the *Providers* pane is the DataProvider's default configuration, which generally applies unless otherwise specified by a given session-specific configuration. DataProviders allow for session-specific setting, which override its default counterpart, but only for that given session. Any changes to the configuration from the session-pane are considered session-specific and are visible by a blue coloration of the setting's label. A session-specific setting can be reverted through the blue-label's context menu (as shown in Figure 3.6). This construct allows for the flexible reuse of DataProviders and encourages experimental behavior.

Sessions alone will not get us very far. A right-click on a session will reveal its context menu, permitting the assignment of DataProviders as depicted in Figure 3.7. A DataProvider can only be assigned once and the list will only contain the available DataProvider that have not yet been assigned. This permits the comfortable mixing and merging of data provided from several Data-Providers, such as our fragmented iris dataset example. Now that we have our data-sources and sessions configured, let us move on to the selection and definition of attributes.

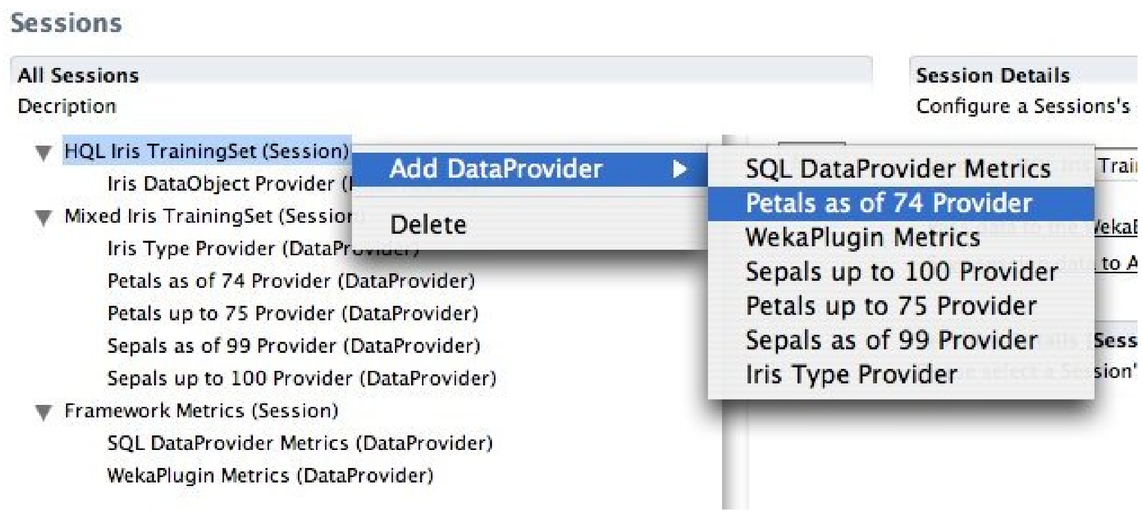

Sessions
All Sessions
Decription
HQL Iris TrainingSet (Session)
Iris DataObject Provider
Mixed Iris TrainingSet (Session)
Iris Type Provider (DataProvider)
Petals as of 74 Provider (DataProvider)
Petals up to 75 Provider (DataProvider)
Sepals as of 99 Provider (DataProvider)
Sepals up to 100 Provider (DataProvider)
Framework Metrics (Session)
SQL DataProvider Metrics (DataProvider)
WekaPlugin Metrics (DataProvider)
Add DataProvider
Delete
SQL DataProvider Metrics
Petals as of 74 Provider
WekaPlugin Metrics
Sepals up to 100 Provider
Petals up to 75 Provider
Sepals as of 99 Provider
Iris Type Provider
Session Details
Configure a Sessions's

Figure 3.7: The assignment of DataProvider instances to a given session.

3.5 Attributes

The next tab in line is the *Attributes* pane, providing a comprehensive view of all attributes in their context, allowing for their selection and definition within a session. Figure 3.8 depicts the attribute pane, split in two, just like the other panes, providing a tree-like view of all the attributes on the left-hand side and a selected attributes details on the right. The tree's root nodes present the following views of the available attributes:

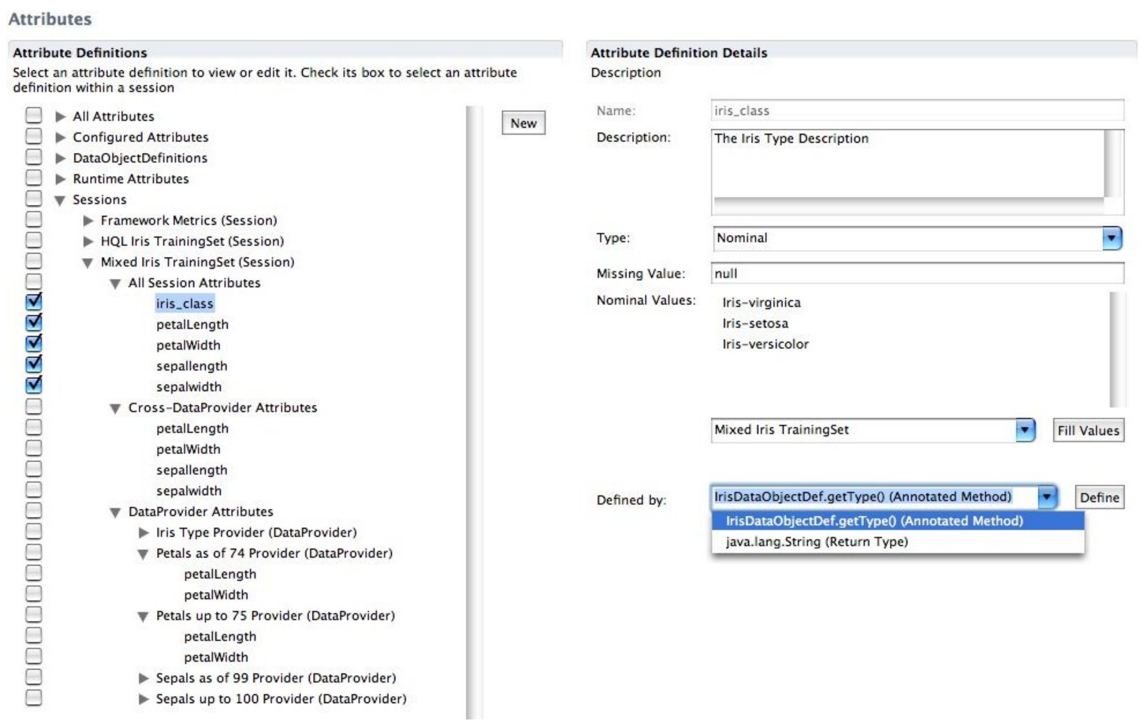
All Attributes will plainly list every attribute encountered throughout the framework

Configured Attributes will only show attributes that were programmatically defined through a DataProvider. This features is currently not supported and does not show any attributes.

DataObjectDefinitions will show attributes defined by DataObjectDefinitions (described in 4.6). The attributes are grouped according to their defining parent, the DataObjectDefinition.

Runtime Attributes lists all the attributes manually created and defined during the framework's session (through the "New" button in the upper right corner). This feature is currently not supported and does not show any attributes.

Sessions will show all the attributes currently used within a session and is visible (fully extended) in Figure 3.8. The root-node's direct children represent a session with attributes. Each of its children, a node representing an active session, further contain three child nodes. The first contains all of the session's attributes, the second only showing attributes defined and used by more than one DataProvider and the third node grouping the session's attributes according to their DataProvider.



Attributes

Attribute Definitions
Select an attribute definition to view or edit it. Check its box to select an attribute definition within a session

```
☐ ▶ All Attributes
☐ ▶ Configured Attributes
☐ ▶ DataObjectDefinitions
☐ ▶ Runtime Attributes
☐ ▼ Sessions
☐     ▶ Framework Metrics (Session)
☐     ▶ HQL Iris TrainingSet (Session)
☐     ▼ Mixed Iris TrainingSet (Session)
☐         ▼ All Session Attributes
☑               iris_class
☑               petalLength
☑               petalWidth
☑               sepallength
☑               sepalwidth
☐         ▼ Cross-DataProvider Attributes
☐               petalLength
☐               petalWidth
☐               sepallength
☐               sepalwidth
☐         ▼ DataProvider Attributes
☐             ▶ Iris Type Provider (DataProvider)
☐             ▼ Petals as of 74 Provider (DataProvider)
☐                   petalLength
☐                   petalWidth
☐             ▼ Petals up to 75 Provider (DataProvider)
☐                   petalLength
☐                   petalWidth
☐             ▶ Sepals as of 99 Provider (DataProvider)
☐             ▶ Sepals up to 100 Provider (DataProvider)
```

New

Attribute Definition Details
Description

Name:	iris_class
Description:	The Iris Type Description
Type:	Nominal
Missing Value:	null
Nominal Values:	Iris-virginica Iris-setosa Iris-versicolor
	Mixed Iris TrainingSet — Fill Values
Defined by:	IrisDataObjectDef.getType() (Annotated Method) — Define

IrisDataObjectDef.getType() (Annotated Method)
java.lang.String (Return Type)

Figure 3.8: The *Attributes* pane displaying the framework's attributes from several perspectives along with a selected attribute's details.

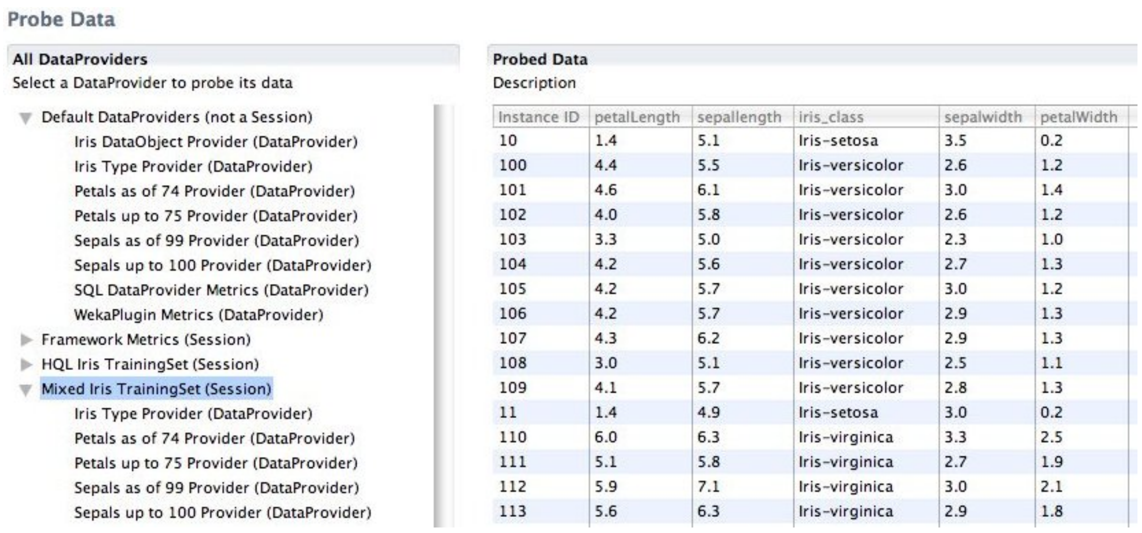
Besides providing a multi-perspective view of the attributes, the left-hand side also permits the inclusion respectively exclusion of an attribute and its values from a session's resulting data. Unfortunately, the displayed tree is not able to suppress checkboxes at certain nodes. Only the attributes contained in a session's all-attributes node are checkable. All other checkboxes will ignore any mouse-clicks.

The right-hand section of the pane, as shown in Figure 3.8, displays an attribute's details and allows for its proper definition. Unfortunately, not all data sources contain enough information to define an attributes to a data mining tool's liking. A DataProvider can either programmatically add this information or leave it to the user. This section allows a user to add an attribute description, define its type (nominal, numeric or string) and define the representation of an attribute's missing value. Furthermore, it can automatically look for a nominal attribute's possible attribute values or have the user enter them manually. Normally the user will not have to define attributes, as their definition is automatically guessed, according to certain defining objects provided upon their creation. A user can manually override a guessed definition or revert back to it through the *Defined by:* pull-down menu.

In our example, the attributes are defined by SQL-return-values and by a DataObjectDefinition (used to power the HQL DataProvider, which in turn is used to retrieve the iris data as a whole). To verify our DataProvider respectively Session configurations and attribute definitions, we can view the data prior to processing it from within the the *Probe Data* pane.

3.6 Probing the Data

The intention behind this pane is to rule out configuration mistakes and to encourage a better understanding of the data. As depicted in Figure 3.9, on the left-hand side, the user can see all of the active sessions with their underlying DataProvider instances. The righ-hand side will display a table representing a selected element's data. As DataProvider instances also offers a default configuration, we have added a *Default DataProviders* node to the tree to represent the default-configuration-specific data, permitting the validation of DataProviders not assigned to a session.



Probe Data

All DataProviders
Select a DataProvider to probe its data

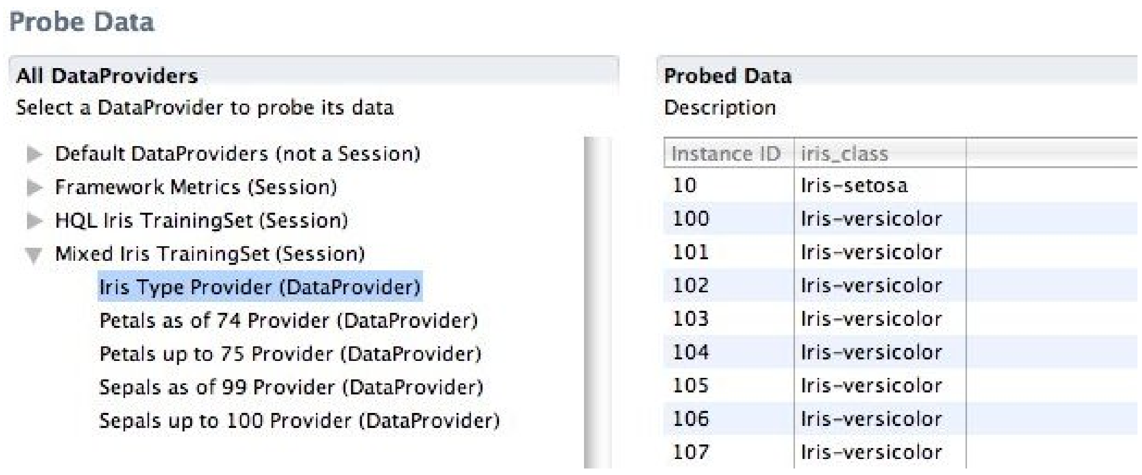
- ▼ Default DataProviders (not a Session)
  - Iris DataObject Provider (DataProvider)
  - Iris Type Provider (DataProvider)
  - Petals as of 74 Provider (DataProvider)
  - Petals up to 75 Provider (DataProvider)
  - Sepals as of 99 Provider (DataProvider)
  - Sepals up to 100 Provider (DataProvider)
  - SQL DataProvider Metrics (DataProvider)
  - WekaPlugin Metrics (DataProvider)
- ▶ Framework Metrics (Session)
- ▶ HQL Iris TrainingSet (Session)
- ▼ Mixed Iris TrainingSet (Session)
  - Iris Type Provider (DataProvider)
  - Petals as of 74 Provider (DataProvider)
  - Petals up to 75 Provider (DataProvider)
  - Sepals as of 99 Provider (DataProvider)
  - Sepals up to 100 Provider (DataProvider)

Probed Data
Description

Instance ID	petalLength	sepallength	iris_class	sepalwidth	petalWidth
10	1.4	5.1	Iris-setosa	3.5	0.2
100	4.4	5.5	Iris-versicolor	2.6	1.2
101	4.6	6.1	Iris-versicolor	3.0	1.4
102	4.0	5.8	Iris-versicolor	2.6	1.2
103	3.3	5.0	Iris-versicolor	2.3	1.0
104	4.2	5.6	Iris-versicolor	2.7	1.3
105	4.2	5.7	Iris-versicolor	3.0	1.2
106	4.2	5.7	Iris-versicolor	2.9	1.3
107	4.3	6.2	Iris-versicolor	2.9	1.3
108	3.0	5.1	Iris-versicolor	2.5	1.1
109	4.1	5.7	Iris-versicolor	2.8	1.3
11	1.4	4.9	Iris-setosa	3.0	0.2
110	6.0	6.3	Iris-virginica	3.3	2.5
111	5.1	5.8	Iris-virginica	2.7	1.9
112	5.9	7.1	Iris-virginica	3.0	2.1
113	5.6	6.3	Iris-virginica	2.9	1.8

Figure 3.9: The *Probe Data* pane showing a session's resulting data

As depicted in Figure 3.9, the selection of a session-node will display the session's resulting data on the right-hand side, whereas a selection of a DataProvider as shown in Figure 3.10, will only display the DataProviders contribution to the session.



Probe Data

All DataProviders
Select a DataProvider to probe its data

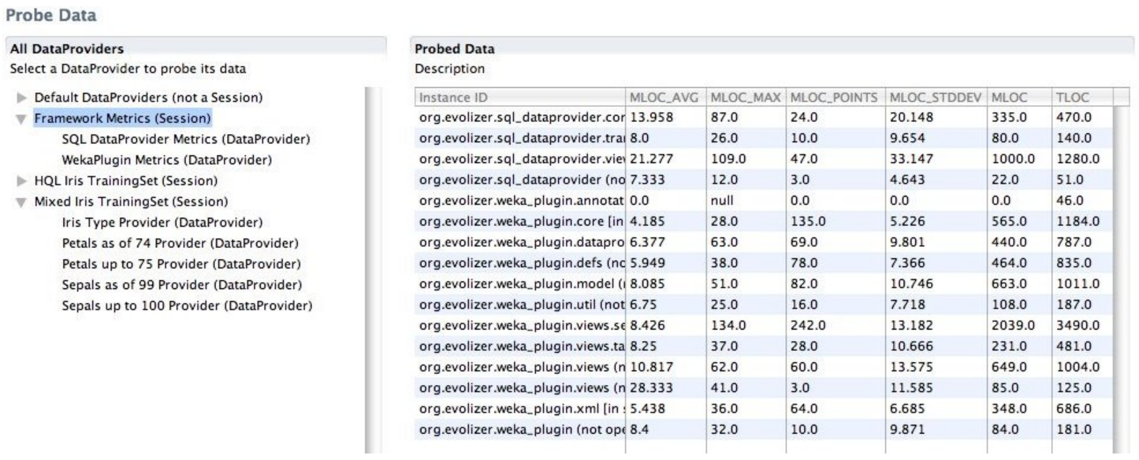
- ▶ Default DataProviders (not a Session)
- ▶ Framework Metrics (Session)
- ▶ HQL Iris TrainingSet (Session)
- ▼ Mixed Iris TrainingSet (Session)
  - Iris Type Provider (DataProvider)
  - Petals as of 74 Provider (DataProvider)
  - Petals up to 75 Provider (DataProvider)
  - Sepals as of 99 Provider (DataProvider)
  - Sepals up to 100 Provider (DataProvider)

Probed Data
Description

Instance ID	iris_class
10	Iris-setosa
100	Iris-versicolor
101	Iris-versicolor
102	Iris-versicolor
103	Iris-versicolor
104	Iris-versicolor
105	Iris-versicolor
106	Iris-versicolor
107	Iris-versicolor

Figure 3.10: The *Probe Data* pane showing a DataProvider's data-contribution to a Session

Ultimately, this view provides us with a means to verify the correct merging of our fragmented iris dataset before sending it off to our data mining tool. As for our metrics-example, Figure 3.11 clearly shows the merging of two different sets of metrics (indicated by the package names in the *Instance ID* column). Once we verify that our sessions and providers are correctly configured, we can move on to actually working with the data.


Probe Data
All DataProviders
Select a DataProvider to probe its data
Default DataProviders (not a Session)
Framework Metrics (Session)
SQL DataProvider Metrics (DataProvider)
WekaPlugin Metrics (DataProvider)
HQL Iris TrainingSet (Session)
Mixed Iris TrainingSet (Session)
Iris Type Provider (DataProvider)
Petals as of 74 Provider (DataProvider)
Petals up to 75 Provider (DataProvider)
Sepals as of 99 Provider (DataProvider)
Sepals up to 100 Provider (DataProvider)
Probed Data
Description

Instance ID	MLOC_AVG	MLOC_MAX	MLOC_POINTS	MLOC_STDDEV	MLOC	TLOC
org.evolizer.sql_dataprovider.cor	13.958	87.0	24.0	20.148	335.0	470.0
org.evolizer.sql_dataprovider.tra	8.0	26.0	10.0	9.654	80.0	140.0
org.evolizer.sql_dataprovider.vie	21.277	109.0	47.0	33.147	1000.0	1280.0
org.evolizer.sql_dataprovider (no	7.333	12.0	3.0	4.643	22.0	51.0
org.evolizer.weka_plugin.annotat	0.0	null	0.0	0.0	0.0	46.0
org.evolizer.weka_plugin.core [in	4.185	28.0	135.0	5.226	565.0	1184.0
org.evolizer.weka_plugin.datapro	6.377	63.0	69.0	9.801	440.0	787.0
org.evolizer.weka_plugin.defs (nc	5.949	38.0	78.0	7.366	464.0	835.0
org.evolizer.weka_plugin.model (i	8.085	51.0	82.0	10.746	663.0	1011.0
org.evolizer.weka_plugin.util (not	6.75	25.0	16.0	7.718	108.0	187.0
org.evolizer.weka_plugin.views.se	8.426	134.0	242.0	13.182	2039.0	3490.0
org.evolizer.weka_plugin.views.ta	8.25	37.0	28.0	10.666	231.0	481.0
org.evolizer.weka_plugin.views (n	10.817	62.0	60.0	13.575	649.0	1004.0
org.evolizer.weka_plugin.views (n	28.333	41.0	3.0	11.585	85.0	125.0
org.evolizer.weka_plugin.xml [in	5.438	36.0	64.0	6.685	348.0	686.0
org.evolizer.weka_plugin (not ope	8.4	32.0	10.0	9.871	84.0	181.0

Figure 3.11: The *Probe Data* pane displaying the merging of two sets of metrics.

3.7 Using the Weka Bridge

The WekaBridge is the component connecting Weka's data mining libraries to the framework. Its primary function is to feed Weka with data, collect any produced results and apply them to new data. Figure 3.12 shows a screenshot of the *WekaBridge* pane, consisting of the following four sections:

Weka Tool Controls allows for the initiation of Weka's tools, such as the Weka Explorer, Experimenter or KnowledgeFlow. These tools open up in a separate window and are implemented by Weka, not the framwork. The framework might hack into the implementation to enable an interoperability with the framework. At present, only the Explorer is supported. The *Loaded Session Run* label and field represent a sort of data "drop-box", where the user can load *Session Runs* for a Weka tool to pick up. A *Session Run* is basically a snapshot of a Session and is labeled with the session's name and a timstamp. A click to the *Explorer* button will open a new Weka Explorer instance and automatically load it with the data in the drop-box.

Weka Classifier Controls control the classification of data outside of Weka's tools using its libraries instead. It comprises two fields, one to store the Session Run to be data mined, the other to store the classifier to data mine it with and a button. This *Classify* button will classify the loaded Session Run with the loaded classifier and automatically jump to the *Results* pane to present its findings.

Session Runs manages the creation, use and destruction of session runs. All existing session runs are displayed in a tree-like view as children of their parent session. A double-click on a session-element will create a new session run according to the session's current configuration, add it as its child and load it both into the WekaBridge's drop-box and the Session-Run-to-classify. A right-click on a the session element or the session run itself offers more options as depicted in Figure 3.12, such as the opening of a Weka tool with the selected session run.

Classifier Instances manages any data mining products, such as in our case Weka Classifiers, Clusterers or Associators, produced by the Weka tool instances and fed back to the WekaBridge. The view groups its data mining products according to their type, as can be seen on the right-hand side of Figure 3.12.

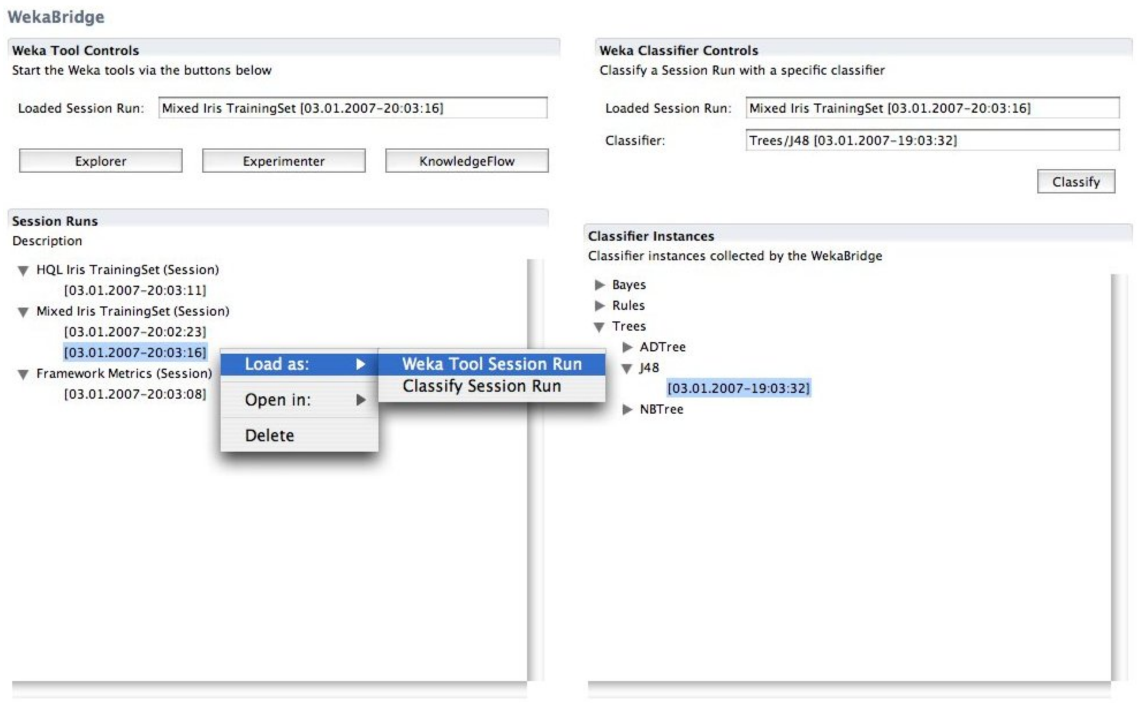
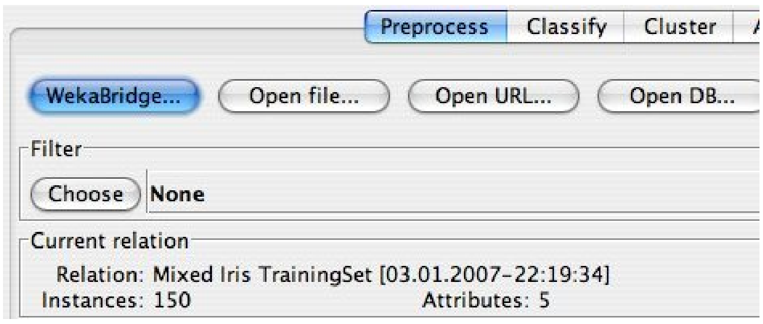

WekaBridge
Weka Tool Controls
Start the Weka tools via the buttons below
Loaded Session Run: Mixed Iris TrainingSet [03.01.2007-20:03:16]
Explorer
Experimenter
KnowledgeFlow
Session Runs
Description
HQL Iris TrainingSet (Session)
[03.01.2007-20:03:11]
Mixed Iris TrainingSet (Session)
[03.01.2007-20:02:23]
[03.01.2007-20:03:16]
Framework Metrics (Session)
[03.01.2007-20:03:08]
Load as:
Open in:
Delete
Weka Tool Session Run
Classify Session Run
Weka Classifier Controls
Classify a Session Run with a specific classifier
Loaded Session Run: Mixed Iris TrainingSet [03.01.2007-20:03:16]
Classifier: Trees/J48 [03.01.2007-19:03:32]
Classify
Classifier Instances
Classifier instances collected by the WekaBridge
Bayes
Rules
Trees
ADTree
J48
[03.01.2007-19:03:32]
NBTree

Figure 3.12: A screenshot of the *WekaBridge* pane with SessionRun information on the left and Classifier details on the right.

In an effort to avoid having to re-instantiate Explorer instances for each and every session run, the Explorer has been modified to load new data from the WekaBridge's drop-box whenever desired, permitting the reuse of Explorer instances. Figure 3.13 shows a fragment of the Explorer's *PreProcess* panel with the new "WekaBridge..." button. A simple click to this button will automatically replace any of the Explorer's loaded data with a copy of whatever is in the drop-box. Note, that the *Current relation* area in Figure 3.13 contains the session run's label, consisting of the session's name and the session run's creation-timestamp.

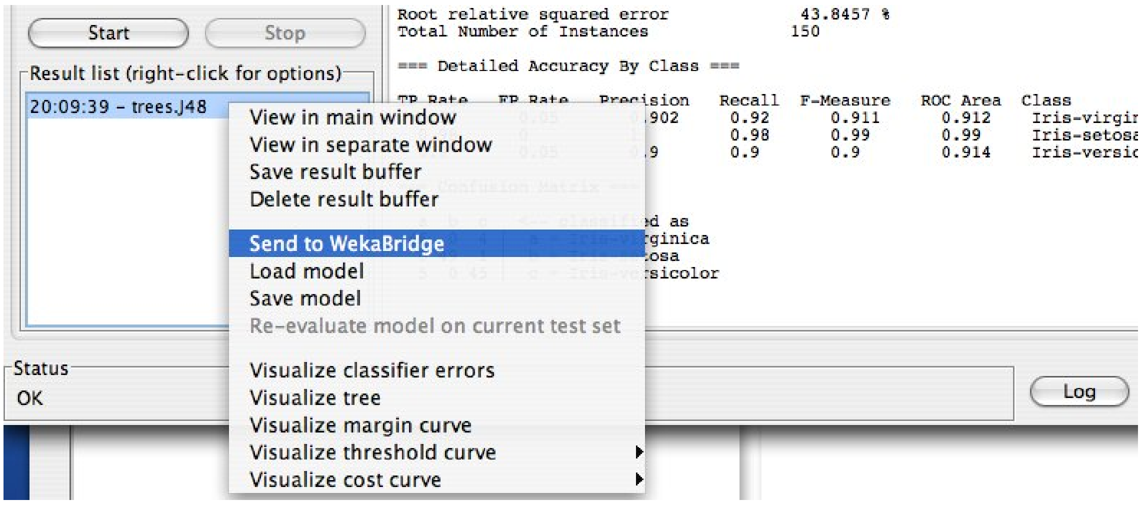
Figure 3.14 shows the second modification to the Weka Explorer, permitting the use of built classifiers outside of the Weka tool. Upon building a classifier within the Explorer's *Classify* panel, one can select and, for example, visualize it as a tree. In our case, we are only interested in passing it back to the WekaBridge for further use. Selecting the *Send to WekaBridge* menu-item will send all the related classifier information back to the WekaBridge, where it will appear in the *Classifier Instances* section.

As for our fragmented iris example, we first created a session run, then loaded it into the drop-box and instantiated Weka's Explorer. We then built a J48 decision tree classifier within the Explorer, using the "Mixed Iris TrainingSet" data. Upon construction, we passed the built



Relation: Mixed Iris TrainingSet [03.01.2007-22:19:34]
Instances: 150 Attributes: 5

Figure 3.13: The top left corner of the Explorer's *Preprocess* panel showing the framework's WekaBridge button.



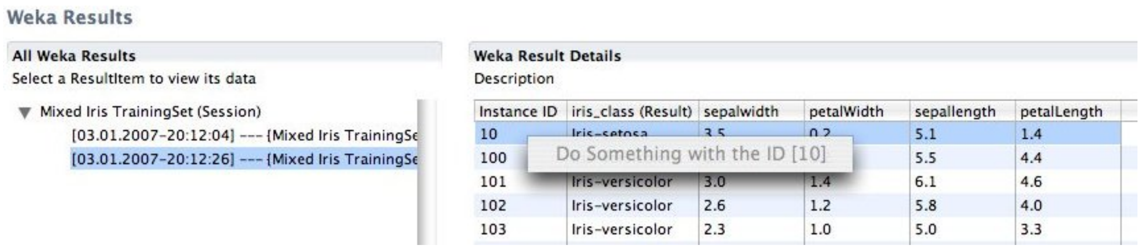
```
Root relative squared error            43.8457 %
Total Number of Instances             150

=== Detailed Accuracy By Class ===

TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                     .902        0.92     0.911       0.912    Iris-virgir
                                 0.98     0.99        0.99     Iris-setosa
                     .9          0.9      0.9         0.914    Iris-versic

                     d as
                     ginica
                     osa
                     sicolor
```

Figure 3.14: A built classifier's context menu from within the Explorer's *Classifier* panel, showing a menu item that passes the selected classifier back to the WekaBridge.

classifier back to the WekaBridge. To test the classification of new data, we switched back to the *Attributes* pane and de-selected the iris-type attribute, so as to exclude it from our next session run. A quick look at the *Probe Data* pane confirmed the attribute's exclusion. Back within the *WekaBridge* pane, we double-clicked on our modified session (excluding the iris-type attribute) and on our exported J48-classifier to load them both for classification. A simple click of the classify button classifies the loaded data with the loaded classifier and automatically jumps to the *Results* pane, presenting the results as a table.

3.8 Results

Last, but not least, we have the *Results* tab, which basically presents and manages the classification results produced by the WekaBridge. This view consists of two sections, one managing all previously acquired results, grouped according to their corresponding session, the other presenting a selected element's results as a table, with the tuple's id in the first column and the tuple's *class* or result in the second. Figure 3.15 displays the results of our previously classified iris dataset. A right-click to a row reveals its context menu, allowing third party contributors to implement additional functionality related to a given result. In our case, the context menu merely consists of a disabled menu-item, proving that the *Result* pane is actually in possession of the tuple's id after classification, thus guaranteeing the tuple's traceability.


Weka Results

All Weka Results
Select a ResultItem to view its data

▼ Mixed Iris TrainingSet (Session)
[03.01.2007-20:12:04] --- {Mixed Iris TrainingSe
[03.01.2007-20:12:26] --- {Mixed Iris TrainingSe

Weka Result Details
Description

Instance ID	iris_class (Result)	sepalwidth	petalWidth	sepallength	petalLength
10	Iris-setosa	3.5	0.2	5.1	1.4
100				5.5	4.4
101	Iris-versicolor	3.0	1.4	6.1	4.6
102	Iris-versicolor	2.6	1.2	5.8	4.0
103	Iris-versicolor	2.3	1.0	5.0	3.3

Figure 3.15: A screenshot of the *Results* pane, displaying the classification results of our Iris example.

This concludes our brief tour of the framework.

Chapter 4

Developing a Data Mining Plug-in

4.1 The General Idea

Before focussing on the integration of Weka's tools and the interesting components that make up the framework, we would like to briefly "zoom out" and take a look at the big picture - what does this framework intend to achieve?

All in all, our goal is to build a framework that eases the task of data mining through integration. Hence the need to integrate foreign, third-party implementations, represented by the entities surrounding the framework-entity in 4.1. As a first, the framework requires a means to retrieve data from any possible source in any format. The DataProvider implementation is responsible for acquiring a specific set of data and intended to serve it to the framework. The framework on the other hand, is designed to accept data from more than one DataProvider, process and merge it to a final relation, and store it within the framework, as the result of a DataSession. The rectangle on the left-hand side represents one or more DataProviders connected to the framework. Figure 4.1 only represents the framework's interaction with its context. The internal storage of the final relation and the DataSessions are not displayed.

The framework then converts the data to an acceptable format and feeds it to a data mining tool of choice for processing. The tool processes the raw data (builds a classifier, filters data, visualizes it, etc.) and returns a result or product (a WDMO, a tool-specific wrapped data mining object) back to the framework. These two steps are represented by the two directed arrows connecting the framework with the *Data Mining Tool* entity. The returned objects are wrapped to include data mining functionalities, such as the correct classification of fresh data.

Note, that as depicted, the framework will pass the WDMO object on to the data mining libraries along with another DataSet to mine the data, expecting a set of results in return. Data mining results and by-products are stored by the framework and are made accessible to registered compenents interested in consuming them. These components, namely the DataConsumers, are depicted as a rectangle on the right-hand side, representing one or more DataConsumers, which have access to the framework-managed results.

An important and obvious decision was to separate and abstract the framework from any possible data sources. There is a practically infinite supply and variety of potential data sources, each accessible in a unique manner. The framework should expect a uniform homogenous representation of the data, such as the DataSet, and leave the localization, retrieval and possible transformation of the data to domain-specific implementations, such as the DataProvider. A user

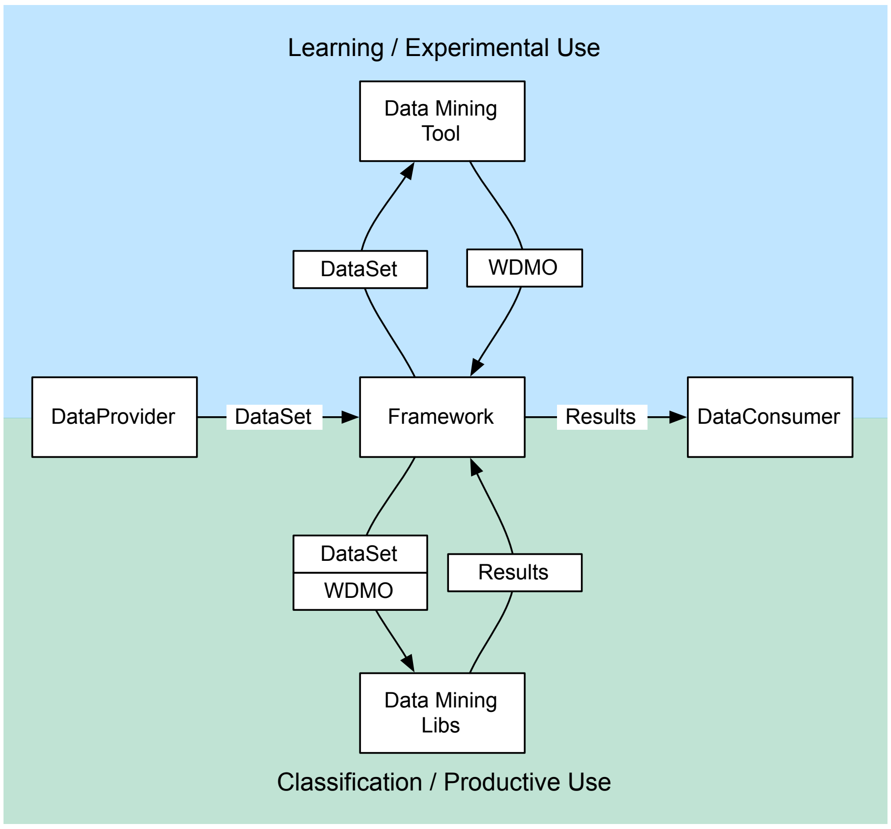


Figure 4.1: The framework, its immediate context and the interactions or rather flows of objects.

interested in data mining should only have to worry about the transformation of his or her data to the framework's DataSet model, nothing more.

It quickly becomes apparent, that the data mining framework itself does not offer much functionality. It merely provides a data-trading platform, allowing for anybody interested in sharing or consuming data to parttake. The idea is to leave the fancy data mining functionality to those, that know it best - the data mining tools - and focus on the simple task of collecting, merging and sharing data. By doing so, the framework no longer consists of a single plug-in, but rather of a collection of plug-ins, each serving a distinct purpose.

At present, the framework's current release consists of five plug-ins, each responsible for a specific task. Figure 4.2 depicts the current plug-in layering. The framework's base consists of the framework's base plug-in, providing extension-points for other plug-ins to extend the framework and so provide their functionalities. The *Weka Plugin* for example, extends the framework, offering it its data mining facilities. Both the *SQL* and *HQL DataProviders* extend the framework and provide it with DataProvider instances. Both SQL and HQL offer extension-points themselves, allowing a more specific use of their services through a higher-level plug-in, in this case a custom implementation, namely the *Custom Iris DataProvider*. On the right-hand side, we have the integration of an existing plug-in, the *Metrics Plugin*, which is extended by the *Metrics DataProvider* offering its facilities to the framework.

Originally, our intention was to support a project-like structure within eclipse for data mining projects. Each project would then contain the code required to access the data and process the

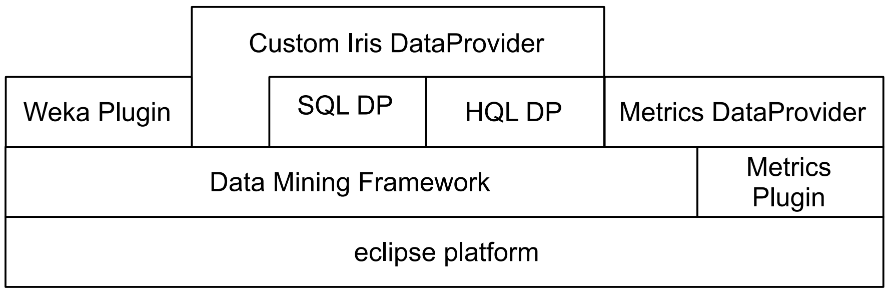


Figure 4.2: The framework represented as layered plug-ins.

results. Unfortunately, integrating a project's source code into the framework is not easily done or rather impossible. The only means to actively change a DataProvider's implementation and run it along with the remaining framework plug-ins, is by developing it within eclipse's Plugin Development Environment (PDE) and running it in a separate eclipse instance. The source code used to access data is thought to be configurable and thus reusable and its use is not restricted to a data mining project. It can therefore be individually distributed as a plug-in extending the framework, incorporating the concept of the DataProvider. The same goes for DataConsumers.

Seeing that DataProviders are no longer project specific, but rather general contributions to the framework, the framework will require a means to group and configure DataProviders for a given data mining work-unit - namely the DataSession. A DataSession represents a specific group of configured DataProviders and is basically responsible for the collection and merging of data before it is mined. As an internal model to the framework, the DataSession is not visible in the figures presented here.

4.2 A Brief Peek at Weka

Originally, this framework was thought to only integrate Weka into the eclipse framework. Throughout the development, it became apparent, that the framework could just as well be tool independent. Unfortunately, most of the framework's code was built on this original assumption and as a result, the framework's implementation is strongly influenced by Weka's processes and is dependent of the Weka libraries. Due to this, it might prove to be difficult to add other data mining tools (that use an approach different to Weka's) to the framework. We plan to decouple the framework from Weka and provide Weka's facilities as an extension to the framework by the next release. To better understand the framework, we shall first take a brief look at Weka's provided tools and libraries. The following information is based on [IHW05] and the JavaDocs provided along with its tools.

The Weka Workbench

The workbench is best described by citing the following paragraph from [IHW05]: "The Weka workbench is a collection of state-of-the-art machine learning algorithms and preprocessing tools. It is designed so that you can quickly try out existing methods on new datasets in flexible ways. It provides extensive support for the whole process of experimental data mining, including preparing the input data, evaluating learning schemes statistically, and visualizing the input data and the result of learning. As well as a wide variety of learning algorithms, it includes a wide range of preprocessing tools. This diverse and comprehensive toolkit is accessed through a common interface so that its users can compare different methods and identify those that are most appropriate for the problem at hand."

Explorer The Explorer, Weka's main graphical user interface, basically offers all of Weka's facilities in a comprehensive tool with five tabbed panes. It allows the loading of data from several sources, the manipulation of its attributes, the preprocessing of the data (application of filters), the building of Classifiers, Clusterers or Associators and provides for multiple visualizations of both data and results. As the name already describes, this tool is intended to explore the world of data mining, respectively Weka's facilitities. Despite all its great features, the Explorer does have a significant shortcoming - it loads all of the data directly into memory. Hence, the Explorer might not be the tool of choice for very large datasets, especially if it's wrapped by a resource-hungry framework. As there is no *silver bullet* for every conceivable problem, the workbench does include the following tools specialized in solving the problems the Explorer cannot handle.

Experimenter The Experimenter was designed to answer a rather important question: "Which methods and parameter values work best for a given problem?" Technically, the Explorer could handle this question, but the experimenter goes a bit further and aims at supporting the comparison of learning techniques. By automating many of the processes, it eases the running of classifiers and filters with different parameter settings on a multitude of datasets. This idea inspired the framework's architecture consisting of individually configurable and assignable DataProviders and DataSessions. Due to the greater workload associated with the experiments, the Experimenter also offers the ability to distribute the processing to an arbitrary amount of machines using Java RMI[1].

KnowledgeFlow The KnowledgeFlow supports the design of streamed data-processing configurations. Again, technically the Explorer could solve this problem. The KnowledgeFlow goes a bit further by configuring all the necessary steps taken in the Explorer into a streamed data-processing flow that can be executed as a whole. Weka's support for certain incremental algorithms in combination with this streamed data processing, permits the mining of the very large datasets that do not "fit" in the Explorer. This tool offers an excellent graphical user interface where the user can design the data-processing flow by connecting drag-and-dropable processing steps and so define the desired data stream. This facility not only graphically represents data mining workflows, but also permits their storage and thus the transferability between different environments and throughout time - yet another reason to choose Weka as the primary data mining tool for our framework.

Commmand Line Interface (CLI) One of Weka's various features, is it's ability to be executed via the operating systems command line interface. Weka takes this a step further and offers a custom command interface to access Weka's facilities in a raw form, without having to load the classpath manually. This user interface requires a rather thorough knowledge of Weka's internals (Classifiers, Filters, the package structure, etc.) and unfortunately is of little use to the framework, as it does not integrate very well.[2]

[1]Java Remote Method Invocation, a programming interface for performing the object equivalent of remote procedure calls.

[2]Interoperability is probably only possible through customized Loaders. All other data would be transferred via the file system

The ARFF Format

ARFF stands for Attribute-Relation File Format and was developed for use with the Weka machine learning software and consists of two distinct sections - the *Header* Information and the *Data* information. The former contains both the relation and attribute declarations. The latter consists of the data declaration line and the actual data as comma-separated instance lines (missing values as question marks).

The ARFF format is logically paired with the Instances object described further below. ARFF files can be loaded into Instances and an Instance is converted to an ARFF through its `Instances.toString()`-method. Unlike the Instances object described further below, the ARFF format merely represents the dataset and does not specify a class attribute. The following ARFF-snippet is an example taken from [IHW05]

```
@relation weather

@attribute outlook {sunny, overcast, rainy}
@attribute temperature numeric
@attribute humidity numeric
@attribute windy {true, false}
@attribute play? {yes, no}

@data
sunny, 85, 85, false, no
overcast, 83, 86 false, yes
rainy, 70, 96, false, yes
rainy, 65, 70, true, no
...
```

Classifiers, Clusterers, Associators

Weka implements the various data mining algorithms by extending the Classifier, Clusterer or Associator Class. These base classes guarantee a standardized access respectively configuration of the corresponding algorithm, depending on what you intend to do. The implementations and their corresponding base classes can be found in the following packages (including subpackages):

```
weka.classifiers
weka.associations
weka.clusterers
```

If Weka does not support your desired algorithm, you are free to implement it on your own by extending one of the base classes and loading it into Weka's classpath. Weka has a unique way of discovering algorithm implementations upon startup. This "disovery algorithm", while permitting the flexible expansion of the provided algorithm suite, caused certain problems during integration, which we shall further explain in section 5.8 on page 54.

Weka's basic building blocks

In order to seamlessly integrate Weka into the eclipse environment as an extension to our framework, we had to "hack" into Weka's code and apply a few changes. Most of the WekaBridge's code basically translates between the framework and Weka's basic building blocks listed below. The Instances, Instance and Attribute classes are located in the `weka.core`-package, whereas the Loader is located within the `weka.core.converters`-package.

Instances As already mentioned above, the Instances Object is basically the programmatic counterpart of an ARFF file, with the exception that the class attribute can be specified. It is designed to handle an ordered set of weighted instance objects and hence consists of a vector-like container for both instance objects and the data's corresponding attributes (as a header). Instances objects are Weka's denomination in terms of data. As Instances objects do not track ids, it is recommended to limit their use to merely build classifiers and not to classify actual data.

Instance The Instance class represents a tuple. To optimize for speed, all Instance values are stored internally as floating-point numbers (`java.lang.Double`) - Nominal values are stored as indexes to the corresponding nominal value in the Attribute's definition. This poses a problem to our ResultListeners (described in section 4.3 on page 38). A result is simply a double value and requires the class Attribute for interpretation. Implementations of the ResultListener interface are not expected to reference class Attributes nor have to deal with Instances-header information. The Instance object holds a reference to its parent Instances object in order to gain access to the header, allowing the Instance to interpret Attribute objects.

Attribute The Attribute object provides a means to declare attributes (nominal, numeric, relational or String) and is basically self explanatory. It contains all of a nominal or string column's possible values. An Instance either holds the actual value (if it is numeric) or a reference to the value contained within the Attribute object.

Loader The Loader is an interface specifying an object capable of loading an Instances object from an arbitrary input source. At a first glance, the Loader concept is the defined way to load data from my framework into Weka. Throughout the development, we found that it was quicker and easier to manually build the Instances objects than to register and connect implementations of the Loader interface. Loaders do come in handy when working with the KnowledgeFlow. When processing very large datasets, it is recommended to implement an incremental loader and flag it a such by implementing the IncrementalConverter interface.

4.3 The Key Elements and how they work

As the title already gives away, this section will take a closer look at the framework's key components, explain their purpose and functionality and focus on the architectural respectively software engineering factors considered during their development.

DataProviders

As already described above, the DataProvider is the means to access a data source of interest and convert it into an instance of the framework's internal DataSet model (described further below). Ironically enough, DataProviders are probably the most valuable component in this data mining framework, and are neither implemented nor provided by the framework. The framework merely supplies base classes, such as the `AbstractDataProvider` class, to simplify the DataProvider's development.

DataProviders are intended to be independent of a data mining project. They are constructed to access a given data source, not as a means to preprocess data for a given purpose. Due to this independence, they must be configured prior to their use, depending on how one intends to use them. The SQL DataProvider for example, requires a url, user name and password to establish a connection and a query-string to access data. A DataProvider answers the question as to *how* a data source is accessed, leaving the *what* to the user's configuration. In an effort to avoid replicating DataProviders for similar data mining projects, the DataProvider is thought to support a *default* configuration next to a set of *project-specific* configurations, each mapped to a DataSession, defining their immediate purpose. In order to use a DataProvider, it must first be assigned to a DataSession. The DataProvider's behavior and functionality thus depend on the DataSession passed as a parameter upon invocation. If a DataSession does not define a specific setting, the DataProviders default will apply. This construct allows for changes to either affect a single DataSession, or the DataProvider's general behavior.

Figure 4.3 shows the cross product between DataProviders and DataSessions. As depicted, a DataProvider can be assigned to more than one DataSession and may have DataSession-specific settings that override the DataProvider's default settings. Session-specific settings only apply to the session and cannot conflict or influence each other.

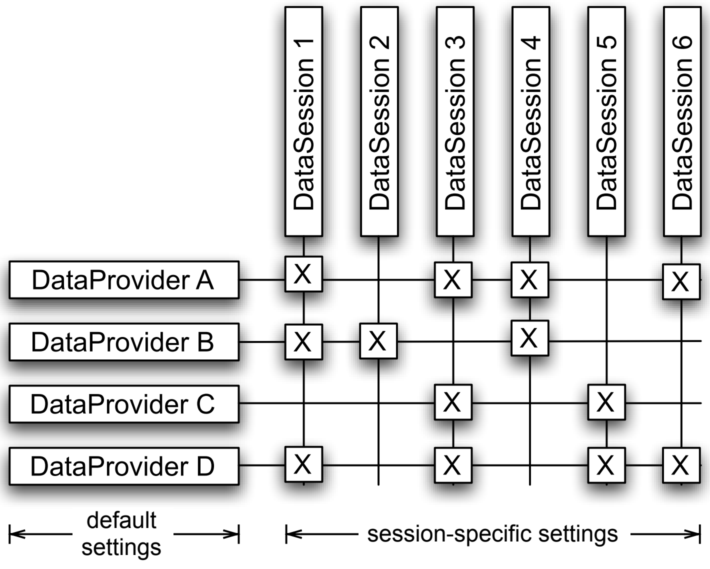


Figure 4.3: The assignment of DataProviders to DataSessions

At present, all DataProviders must be declared as extensions to the framework, which in return passes this information to a `DataProviderManager` instance, responsible for instantiating, managing and destroying any DataProvider isntances. In an effort to avoid a repeated configuring of DataProviders, a DataProvider implementation can automatically provide a set of programmatically pre-configured DataProvider instances. Certain DataProviders, such as the SQL DataProvider, may choose to themselves support extension-points permitting a declarative-configuration as an alternative to the programmatic approach.

Besides providing a simplified means of accessing data, the DataProvider also accounts for the translation of id objects and attribute names. The former is necessary when merging several DataSets from different DataProviders to a single relation. The latter comes in handy, when renaming attributes to better understand them - not everybody likes to work with acronyms. Furthermore, if the data is stable, then so will the resulting attributes be. A DataProvider could then enrich the data by providing its own programmatic definitions of the attributes likely to be used. As an optimization, a DataProvider will cache its data and and only refresh it upon a change in a DataSession's configuration. This is quite helpful, when working with larger datasets, a busy or slow database or simply a limited bandwidth. On the downside, it does unnecessarily accumulate data and will thus compete with the remaining framework for memory.

Upon closure or shutdown of the framework, its state is persisted. All DataProvider instances are then stored using XML and eclipse's IMemento facilities. A DataProvider will only store its default configuration, ignoring all DataSession-specific configurations, leaving their persistence to the corresponding DataSession object.

DataSessions

A DataSession represents a sort of "work in progress" and as such must define the collection and merging of different DataSets to a final relation. A DataSession is a simple construct used to create SessionRuns (described further below), a type of tool-specific data snapshot. Apart from providing information for the framework's view, its functionality basically breaks down to the following.

A user can assign any DataProvider to a DataSession and select which of its attributes to include or exclude. Upon creating a SessionRun, the DataSession will collect the DataSets produced by its assigned DataProviders, merge them to a single DataSet and remove all non-selected attributes.[3]. At present, DataSession objects are included into the storage of the framework's state. The DataSession maps its settings to an XML file (`DataSession.toXML()`), mapping any session-specific settings to a representation of their corresponding DataProvider objects. The DataSession restores its state by implementing the lightweight XML framework's `AbstractXMLFilter` as a nested class. Appendix A provides more details and code snippets on the XML framework.

DataSets

A DataSet is an implementation of the "homogenous" relational data model mentioned in section 2.4. It is the framework's denomination in terms of data. All data providers are required to provide their data as a DataSet object in order for the framework to automatically process it. The DataSet is a result of a DataSession and DataProvider cross-product and thus contains a reference to both its parent (DataSession) and producer (DataProvider). Figure 4.4 shows both references as rectangles in the top-left corner.

Structure The DataSet's internal model is derived from Codd's definition of the relational model [Cod70]. A relation is defined as a set [4] of tuples. A relation will consist of a header and body. The former is a set of attributes, the latter a set of tuples. A tuple is an ordered set of attribute values.

The DataSet is constructed in a similar way. Figure 4.4 shows the body consisting of a `Hashmap` mapping (dotted line) an `Object` (the ID_1 rectangle) as a key to the corresponding tuple (the Val_1, Val_2 and Val_3 rectangles) represented by an array of objects. Originally, the DataSet's first version consisted of a Hashmap mapping ID-Strings to a Hashmap (representing the tuple) mapping AttributeDefinitions to Objects (the attribute values for the tuple). The advantage was that the DataSet could grow in both directions (more tuples or more attributes) without restrictions, allowing for a very flexible population of the DataSet.

Due to the overhead and redundancy of each tuple having its own `Hashmap`, it was re-engineered to a single `Hashmap` per DataSet, known as the DataSet's header. The header maps AttributeDefinition objects to Integers, depicted in Figure 4.4 as the boxes connected by the dotted line in the header-rectangle, indicating the array-position of a given AttributeDefinition. The header

[3]The merging and selection of attribute is actually implemented by the DataSet class. The DataSession merely invokes this functionality.

[4]A set is an unordered collection of distinct objects called elements or members. Each element must be unique within a set.

is extended by an array of AttributeDefinitions, corresponding to the Att1, Att2 and Att3 rectangles, mapping an array-position to its corresponding AttributeDefinition. This permits the easy identification of and access to a value's corresponding AttributeDefinition.

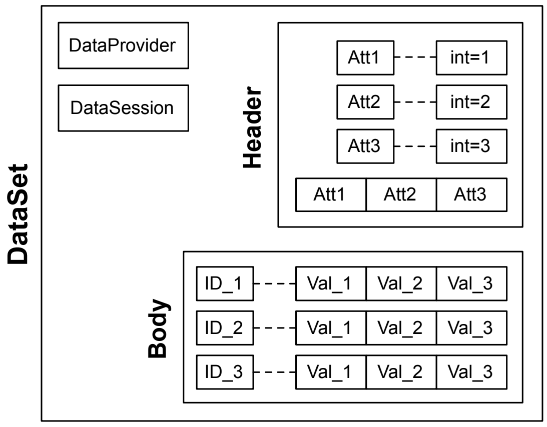


Figure 4.4: A simple diagramm of the DataSet's internal structure

This new construct ensures Codd's *ordering of relation attributes* requirement, but drastically reduces the flexibility in populating the DataSet. The ordering is not intended to distinguish between two attributes with the same name as described by Cobb, but rather required to disconnect the header (AttributeDefinitions) from the body (Hashmap of IDs to `Object`-arrays) and still be able to interpret each single value. AttributeDefinitions are designed to be distinct in and accessible via their names, as more than one DataProvider might share the same AttributeDefinition. The actual ordering is performed after all the AttributeDefinitions have been registered (stored into the Hashmap), but prior to input of the first tuple and thus depends on the Hashmap's iterator behavior. The ordering only applies to the DataSet instance, not the configured DataProvider. As the ordering has no apparent heuristics (basically depends on hash values) and DataSets are dropped and rebuilt with every DataProvider's configuration change, the ordering will not remain constant and can therefor not be relied upon.

The ID is generally separated from the data. It is important to keep track of the tuple's id when classifying data[5]. An alternative would be to include the id with the data. Unfortunately, the user must be aware of this and manually remove it from within the data mining tool, as the tool might mistakenly consider the id for classification. The id is hence irrelevant to any data mining and is as such to be excluded. If for any reason the user chooses to explicitly data mine the id, he or she can do so by simply adding it to the rest of the data.[6]

Functionality A DataSet can be constructed from scratch and populated with data or created by merging several existing DataSets to a single new DataSet. When creating from scratch, all the AttributeDefinition objects must be known in advance and registered prior to populating the DataSet. The population is done via the following methods and is per attribute value or per tuple.

```
public void addDataElement(Object instanceID,
        AttributeDefinition column, Object data) throws Exception
public void addDataRow(String instanceID,
```

[5]It is not so important to keep track of the ID when building a classifier, as we already know the results. In order to use new classification results, we must be able to assign them to something, hence the constant mapping to an ID-object.

[6]The id is then duplicated - contained within the tuple and again in the tuple's Hashmap-key.

```
        HashMap<AttributeDefinition, Object> row) throws Exception
```

Note that the DataProvider is required to pair each attribute value with the corresponding AttributeDefinition - an absolute attribute position or ordering is not necessary. As the DataSets are invoked by third-party software, the DataSet is rather pessimistic and will throw Exceptions if it cannot process the data. When creating a DataSet from existing DataSets, only the selected[7] AttributeDefinitions are registered to the new DataSet. Each selected attribute-value is then copied to the new DataSet. It is important to understand, that the id translation only takes place when merging DataSets. The original id is still valid for the DataProvider's original DataSet and only requires a translation when moved to a new context (merged tuples). With respect to transitively dependent data, it would be wise to offer a means to reassign the id of an existing DataSet.[8] Transitively dependent data is discussed in more detail in section 5.6 on page 51. The remaining functionality basically retrieves information on the DataSet (headers, tuple count, etc.) or the underlying data.

SessionRuns

The general idea behind the SessionRun is to provide a snapshot of a given DataSession that also serves as input to a specific data mining tool. It basically decouples a DataSession's produced data (in the form of DataSets) from the DataSession, by storing it in a format that a data mining tool of interest accepts. A SessionRun enables changes to the DataSession's configuration without damaging previous states (SessionRuns). It is intended to be implemented by a framework extension integrating a specific data mining tool (like the org.evolizer.weka_plugin.gplweka-plugin). The SessionRun is currently not defined as an interface, but rather as a concrete implementation based on Weka. It is provided by the framework, not the Weka extension and consists of an empty Instances object (header information) and a collection of Instance objects mapped to their corresponding id object. Because Instances do not keep track of ids, the instance objects are held separately and only added to the Instances upon its retrieval (when building a classifier). We plan to reengineer the SessionRun in order to decouple it from Weka by the next release.

So the question arises, as to why use a SessionRun object instead of just using Weka's Instances data model? Because a SessionRun should be reusable and reproducible. The former requirement is rather straightforward. As we have no knowledge or control on who will use, or worse, modify the Instances object, we cannot assume that it will remain untouched. A SessionRun is a snapshot, and must thus remain immutable. The latter requirement is a bit more complicated. A snapshot does not necessarily have to be a one-way disposable object, but could rather be a persistable DataSession-state. A DataSession is intended as a mere aid to build SessionRuns, nothing more. It could prove useful to store sufficient information in a SessionRun to reproduce the DataSession's configuration and so produce an equivalent SessionRun at a later moment. This metadata would then serve as a documentation to the DataSession configuration respectively development. Provided the metadata is human-readable and the underlying data sources haven't changed, even a non-framework user could reproduce the data and classifier to validate a result. An Instances object can be stored and restored, but it is inadequate to represent a DataSession's state.

Figure 4.5 shows the DataSession's evolution over time (from left to right) at the bottom of the figure. As the DataSession object is part of the framework, it lies outside of the data mining tool bounds (dotted line). The figure shows four distinct SessionRuns, representing the DataSession's state at a given time. SessionRuns are provided by the data mining tool extension and stored within the WekaBridge, hence their position. The Weka Explorer is loaded with data, by exporting a SessionRun's data to a Weka-friendly format, in this case an Instances object. The Explorer can then process this data and pass it back to the WekaBridge as a WrappedWekaObject. The

[7] The selection is passed as a Hashmap-parameter mapping each AttributeDefinition to a boolean.

[8] Example: merge A with B on _a, reassign the resulting DataSet's id to _b and merge it with C on _b.

WekaBridge can then data mine any other SessionRun with the stored WrappedWekaObject from within the framework.

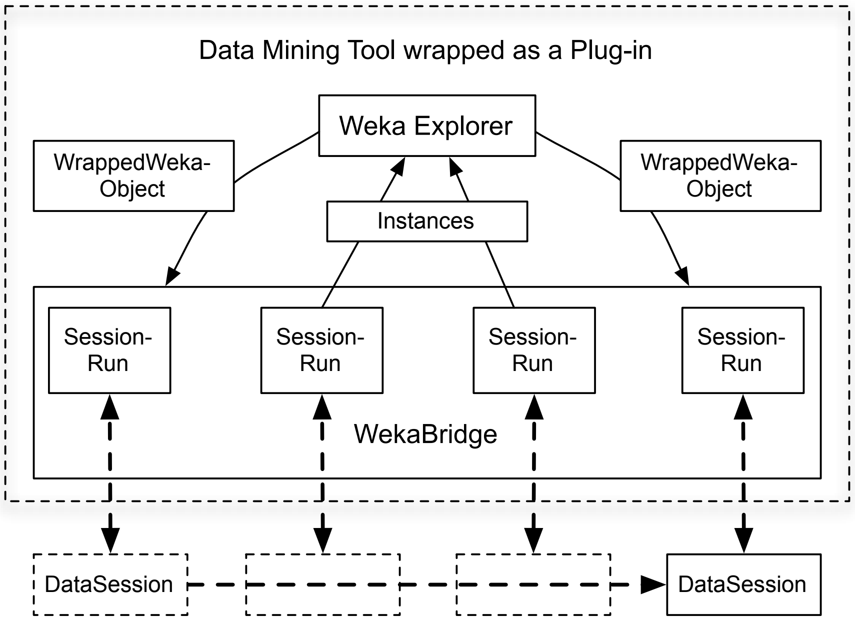


Figure 4.5: DataSessions, SessionRun and the Weka Explorer in context.

A SessionRun will store the data based on its header information. When classifying new data, it may be advisable to use the classifier's header information to guarantee a smooth execution. The header's size or ordering may have changed in the meantime, causing an exception at best or wrong results at worst. Hence, the SessionRun must be able to transpose its data to a new set of Instance objects based on the classifier's header information.

AttributeDefinitions

For the purpose of avoiding confusion in this section between terms, I will abbreviate the framework's AttributeDefinition to *AttDef* and Weka's Attribute to *WekaAtt*. The AttDef is an attempt to simplify the tedious task of defining and redefining attributes in preparation of the data. The idea is to not have to define an attribute at all, and whenever this is not possible, one have to define it once and hopefully never again. The AttDef is based on Weka's ARFF format and influenced by the WekaAtt. Other data mining tools might require more information than the AttDef currently offers. As with other objects, the AttDef will be subject to reengineering in order to decouple the framework from Weka.

At a first glance, the definition of an attribute appears to be rather straightforward. It basically consists of a name, a position in a table or relation, an attribute type[9] and may hold predefined nominal values. We added an attribute description to encourage a better understanding of the data and the ability to define missing values[10]. The AttDef translates to a WekaAtt when a DataSession's state is stored as a SessionRun. Until then, the defining parameters are of no impor-

[9]Weka knows the following four types: nominal, numeric, relational and String

[10]A null-value does not necessarily have to be interpreted as a missing value. Missing values are defined as String representations. If not explicitly defined, the AttDef will default to null as a missing value.

tance. The real problems arose with the following questions: Who can define attributes? What uniquely identifies an attribute? What if two entities define the same attribute in different ways? Must every entity define its attributes? When processing data paired with AttributeDefinitions by third-party providers, we cannot just assume that everything will go fine.

In a first attempt, the AttDefs were unique to the DataProvider declaring them, as only the DataProvider had exact knowledge of the data in use. We soon noticed, that depending on how the data is organized, more than one DataProvider may be referencing the same AttDef. For example, the Mozilla software repository used in [Kna05] is stored as multiple database instances - one per release, all with the same schema. We could easily merge the data into a new relation, but this is not really an option, due to the framework's requirement to be independent of a database instance. Hence, the framework must permit DataProviders to share or rather reference the same AttDef. Because of this, DataProviders are no longer permitted to instantiate AttDefs, as they are oblivious to their neighboring DataProvider's presence and can therefore not share its AttDefs with others. Instead, AttDefs are to be created and distributed by a centrally controlled entity - the AttributeDefinitionFactory.

With more than one DataProvider using the same AttDef, the question arises as to which DataProvider actually defines the attribute and how far will that definition go? It turns out, that the easiest way to handle the attribute problems is to make them globally accessible by name. Accessing an AttDef will create the AttDef, but not define the attribute, allowing it to be shared by many, without having to define it each time. Once initialized, the AttDef object can be defined an arbitrary amount of times. The definition requires a *defining object* as parameter. At present, the AttDef accepts `java.lang.Class` and `java.lang.reflect.Method` objects as defining objects. The Class object represents the attribute value's type and is used to infer or rather guess[11] a minimal definition. The Method object is intended when working with object-oriented data instead of the classical relational databases. The AttDef will parse the Method's annotations to retrieve additional information (as described further below in DataObjectDefinitions) if present and otherwise resort the Method's return type analogue to the Class object. If the AttDef was defined manually (not through the `defineAttribute(Object def)`-method), the defining object is set to the AttDef itself. This flexible means of defining attributes allows users to extend or override existing attribute definitions and later return to the original definitions.

An AttDef stores all defining objects encountered, so that the user may return to a previous definition. The definition of AttDef objects occurs through the DataProviders and there is no specific order, in which the DataProviders are invoked. Thus, there is no predictable order in the defining objects. Generally, an AttDef will define itself by using the most recently added Method object, if existent, or resort to the most recently added Class object[12]. A manual definition supersedes all other defining objects. We intend to reengineer the AttDef to only consider defining objects that were actually provided by a DataProvider assigned to the DataSession of interest.

As for traceability, an AttDef translates to a WekaAtt using the same name. As AttDef names are immutable, it is always possible to find a WekaAtt's creating AttDef. We are currently unable to track the defining object used to create a given WekaAtt. A simple Hashmap with the WekaAtt as key and defining object as value could solve this, but would collect junk over time, as the WekaAtts are created from scratch for every SessionRun. As defining objects should be mapped to their DataProviders[13] and are practically immutable (try changing a Method or Class object at runtime), if we can track the defining object, we can definitely link a WekaAtt to its corresponding DataProviders. As DataProviders allow for an attribute name translation, we can track the attribute's original name and look it up, if we wish to do so. All this comes in quite handy, when

[11]If the value is numeric (long, int, short, byte, double and float) or a String, then so is the attribute. If the Class is boolean, then the attribute is nominal (with true and false as values).

[12]A Method object is likely to have more information and always has a return value.

[13]In order to only apply a defining object that actually originated from a DataProvider that is assigned to the DataSession.

wondering where that darn attribute came from that is being so readily used by our rule-based classifier.

DataObjectDefinitions

Now that we have relational data sources covered (by mapping columns to attributes via an AttributeDefinition object), what do we do when tapping object-oriented data? What if the column is not a "simple" value (String, double, integer, date), but rather an arbitrary domain-specific java object? S.E.A.L's release history *evolizer* project is currently being developed using Hibernate [KB04] as a persistent layer. It's rather simple to convert a JDBC ResultSet to a DataSet, but what do you do with a hibernate query result that mixes simple types along with slightly more complex objects? Sure, you can use Java's Reflexion classes to retrieve the object's methods or parse the corresponding hibernate mapping files or annotations, but that will not get you very far. Hibernate annotations and mapping files were designed and are intended for hibernate, not for our data mining framework. Even so, it would be rather complicated and tedious to parse and you would still be lacking a missing value declarations, descriptions and the tuple's id. As with Java's Reflexion, you will be getting a lot more than you asked for - not every method or variable will be of interest. We need a better means to extract attribute information from object-oriented data.

A possible solution to this problem would be to "flatten" the complex java objects upon retrieval by rewriting the query. Instead of extracting whole objects as a column, hibernate should extract an object's values as separate columns as in the following example:

```
SELECT iris.id           AS ID,
       iris.petal_length AS PetalLength,
       iris.petal_width  AS PetalWidth,
       iris.sepal_length AS SepalLength,
       iris.sepal_width  AS SepalWidth
       FROM Iris         AS iris
```

as opposed to

```
SELECT iris FROM Iris AS iris
```

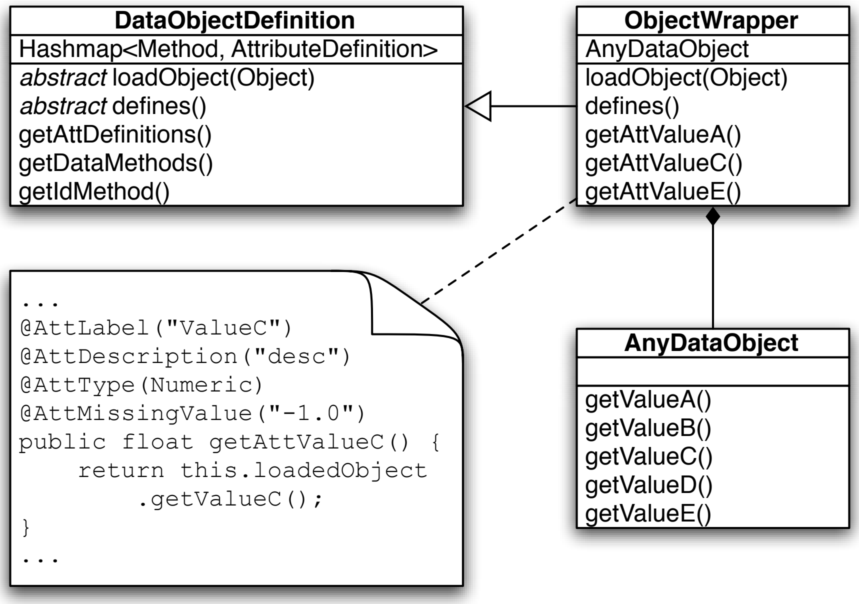
Unfortunately, this requires detailed knowledge on all of data and would clutter the query text, obliterating it and thus complicating any attempts to modify it. Besides all that, an OQL[14] query returning flattened objects as an endless list of values kind of defeats the purpose of querying an object-oriented persistence layer.

Ideally, I would have liked to enrich the data with the required attribute information, but as a requirement is not to change existing data or code, this is not really an option. Originally, the DataObjectDefinitions were declared in an XML configuration file. This construct inspired the Lightweight XML Persistence mini-framework described in appendix A on page 61 in order to easily process any framework or data provider related configurations. In an effort to keep the attribute definitions as close as possible to the data respectively their corresponding java objects without actually modifying them and to spare the user the task of writing raw xml, I resorted to the more elegant option of wrapping the objects and using framework-specific annotations, resulting in the DataObjectDefinition.

As depicted in Figure 4.6 and defined in the code snippet further below, defining a data object's (represented in our example by the AnyDataObject class) attribute information is achieved by subclassing the DataObjectDefinition class, represented by the ObjectWrapper-class, and declaring the *wrapper* methods of interest, corresponding to the `getAttValueA()`,`getValueC()` and

[14]Object Query Language - A query language similar to SQL designed for object-oriented databases.

`getValueE()` methods in Figure 4.6. These wrapper methods in return simply invoke the data object's original method and return its return-value, as can be seen in the *UML-Note*. When gathering the attribute information, the DataObjectDefinition will, through the use of Java Reflexion, per default consider all public, zero-parameter methods declared within the wrapping subclass. Note that the developer has full control over the methods as only the public methods of the object's dynamic type will be regarded. Any inherited (not overridden) or non-public methods providing a functionality or processing of the data (for example a translation or filtering) are consequently ignored. Additional information, like the id-attribute, the descriptions or missing values are optional and declared by annotating the methods. If for some odd reason, the wrapping class still declares public methods that are of no interest or use to the data mining community, they can be excluded by annotation.



```
...
@AttLabel("ValueC")
@AttDescription("desc")
@AttType(Numeric)
@AttMissingValue("-1.0")
public float getAttValueC() {
    return this.loadedObject
        .getValueC();
}
...
```

Figure 4.6: A UML Class Diagram of the DataObjectDefinition Wrapper

```
public class ObjectWrapper extends DataObjectDefinition {
    ...
    private Object wrappedObject = null;
    ...
    public void loadObject(Object wrappedObject) {
        this.wrappedObject = wrappedObject;
    }
    ...
    public Class defines() {
        if(this.wrappedObject != null)
            return this.wrappedObject.getClass();
        else
            return null;
    }
    ...
    @AttObjectID
    @AttExclude
    public int getID() {
        return this.wrappedObject.getId();
    }
    ...
    @AttLabel("IntValue")
    @AttDescription("No description")
    @AttType(AttributeType.NUMERIC_ATTRIBUTE)
    @AttMissingValue("-1.0")
    public int getIntegerValue() {
        return this.wrappedObject.getIntegerValue();
    }
    ...
}
```

This simple yet very flexible construct permits the enrichment of object-oriented data with metadata without having to touch the original data or corresponding code, allowing for a more or less automatic flattening of object-oriented data to their corresponding AttributeDefinitions. Furthermore, it allows a user to easily select and rename the attributes or rather methods of interest and discard the rest, thus providing a clean, crisp and simple *view* of the data to the data mining community. As the code associated with the object-oriented data is not likely to be subjected to rapid changes, the framework does not require a means to change a DataObjectDefinition at runtime. It would be therefore safe to package the DataObjectDefinition-wrappers along with the DataProvider tapping the object-oriented data. Further information on the framework-specific annotations and snippets of the underlying DataObjectDefinition source code can be found in Appendix C on page 77.

The DataObjectDefinition implementation does not actually belong to the framework's immediate domain, as it only processes DataSets. Instead, it can be considered an object-oriented provider-specific add-on, belonging to the Hibernate DataProvider. As it is a base class and generally supports the retrieval of data through java objects, it may be applied to any DataProvider and is thus provided by the framework itself. I intend to offer the ability to declare a DataObject-Definition as an extension, thus providing an alternative to actually programming it in Java.[15].

Despite its simple and flexible characteristics, the DataObjectDefinition does actually have a severe limitation - all object methods *must* either return a String object or a primitive type[16]. The user is encouraged to work around this by using powerful object-oriented query language (for example HQL). Furthermore, the wrapper is designed as a singleton, providing a sliding window view on a per object basis. Objects must be loaded prior to their processing, thus opening a door for synchronization problems when the object is simultaneously invoked for more than one data object.

DataConsumers

DataConsumers are the opposite to the DataProviders - components that are eager to process any data mining results or byproducts. A DataConsumer is basically the other side of the integration process, providing more functionality and so answering the question as to what one might want to do after mining a dataset. Unfortunately, due to the time constrains of this thesis, I was only able to define a single interface (ResultListener) for the consumption of results. Ideally, the

[15]Not everybody is comfortable or experienced in writing java code.
[16]In Java, this would correspond to a double, float, long, int, short, byte or boolean.

framework would provide any consumable product through a wide set of defined interfaces. A DataConsumer would then implement one or more interfaces and register the implementations as an extension to the framework.

Unfortunately, this rather flexible solution does entail certain complications. For one, the framework itself does not really produce any results or byproducts. Currently, the only results it produces is a mapping of an id-object to a double value, which is actually produced by Weka - other tools might produce different results, making it rather difficult to standardize a result retrieval. An approach would be to differentiate the different types of consumable products. On the one hand side, you have data-mining-specific results and products. These consist of built classifiers, filters, classified datasets and statistics or details on classifiers and can be considered rather close to the framework. Hence, it may be possible to squeeze them into an internal model, just as I did with the DataSet and offer them through the framework itself. All other consumable are specific to the data mining tool in use. Weka, for example, offers a a set of visualizations. A possible DataConsumer might tap the visualization itself or simply use its underlying data to produce its own visualization. As the framework has no control or knowledge over the data mining tool's products, it cannot really publish them. Hence, a DataConsumer would require extensive knowledge of the tool extension and thus become rather dependent of its facilities.

Last but not least, we are again confronted with the id-problem. Most of the data mining results will be paired to a possibly translated id-object. In order to correctly interpret and use the results, the DataConsumer would have to know the id's domain. The id's domain is not considered to be stable, is defined by and can vary between DataSession's and is not readily accessible as such.

ResultListeners

As already described above, the ResultListener is an interface providing a means for potential DataConsumers to access data mining results as they are produced. A ResultListener is constantly fed with new results (no specific ordering, as the tuples themselves have no order) as they come in. As the framework currently only supports Weka as a data mining tool, an implementation of this interface is registered directly with the WekaBridge (described further below) and not via the framework. I intend to decouple it from Weka and register it through the framework, instead of a tool-extension. Furthermore, it should support a more distinguished listening, for example by focusing on a single DataSession's results, instead of anything and everything. The following code snippet details the ResultListener's current interface.

```
public interface ResultListener {
    public void newResult(Object id, weka.core.Instance instance,
            double result, weka.core.Instances header);
}
```

The idea is that a ResultListener only react to a given type of result. Besides the already described id-problem, the ResultListener will have difficulties interpreting the results, as it may have little knowledge of the class attribute in use. An implementation would have to access the header to retrieve the class attribute, read its nominal values and compare them to what it is looking for in order to know which results are of interest. Any discrepancy in the attribute's nominal values and the value of interest will result in a "deaf" ResultListener. The ResultListener is a great means to collect and pass on tuples of a given class to any other plug-in.

WekaBridge

Originally, the WekaBridge was thought as a means to transport objects back and forth between the underlying data mining tools (Weka) and the framework. It has significantly grown and is now an interface to access all data mining facilities provided by a given integrated data mining tool. As the framework was originally built on Weka (basically using its basic building blocks), it was already referencing the weka libraries. In order to avoid cyclic dependencies, the WekaBridge was defines as an interface within the Weka plugin avoiding the use of framework-specific classes as can be seen in the following code snippet.

```
import weka.associations.Associator;
import weka.classifiers.Classifier;
import weka.clusterers.Clusterer;
import weka.core.Instances;

public interface WekaBridge {

    public boolean hasData();
    public Instances getWekaInstances();

    public void addClassifier(Classifier classifier,
            Instances header);
    public void addClusterer(Clusterer clussterer,
            Instances header);
    public void addAssociator(Associator associator,
            Instances header);
}
```

The interface's implementation on the other hand requires references to the framework's classes. It is therefore packaged with the framework. A decoupling of Weka and the framework will remove all references from the framework to weka's libraries and thus eliminate that dependency. A framework extension integrating a mining tool is then intended to reference the framework's classes and implement its interfaces in order to integrate smoothly. Tool-specific objects (for example weka.core.Instances, weka.classfiers.Classifier, etc.) are to be provided by the extending plug-in as wrapped objects in order to implement framework specific interfaces without changing the underlying data mining tool's original code. The intention is that a tool's bridge (such as the WekaBridge) can feed its own implementation of a SessionRun to their own wrapped objects (such as classifiers, filters, etc.) without exposing tool-specific classes to the framework.

Upon successful decoupling, the WekaBridge interface will sort of change directions and have to provide adequate means to integrate the bridge into the framework. With respect to each data mining tool having its own set of special features to differentiate itself from the rest, it would be a waste to downsize each tool to a standardized interface. Instead, each tool should be allowed to offer its specialties in the way they see fit, by offering them ample graphical user interface space. I would like to point out, that the framework really only integrates the tools, provider and consumers and doesn't actively control them, unless it a standardized functionality. The framework merely provides a *data-trading* platform and (graphical) space for each component. Hence, a bridge may be full of functionality, but that functionality will probably be accessed by the bridges graphical user interface. To the framework, a bridge remains a simple means to transport data back and forth.

The WekaBridge's current implementation as depicted in Figure 4.7 enables the startup of Weka instances, such as the Weka Explorer, Experimenter and KnowledgeFlow and can be armed

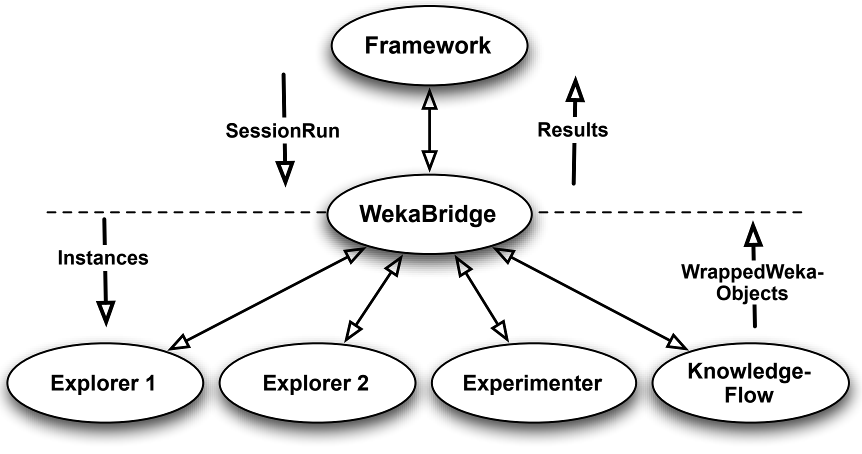


Figure 4.7: The Framework, WekaBridge and its tool instances in context

with a SessionRun and WrappedWekaObject object. The Bridge supports an arbitrary amount of Weka instances, but can neither control nor differentiate them. A SessionRun can be opened with a Weka instance, automatically loading the armed SessionRun into the tool upon startup. A Weka instance is only aware of its creating bridge, not the framework. Once armed, any Weka instance is able to access the armed SessionRun and thus acquire new data. WrappedWekaObjects are passed back to the bridge anonymously. This construct permits, for example, the running of similar experiments by feeding the same data to several individually configured Weka instances. Data, results or wrapped objects are pushed onto the bridge and fetched by the Weka instances respectively by the framework.

WrappedWekaObject

Throughout the framework's development, it became increasingly difficult to ensure the correct export and use of a Classifier, Clusterer or Associator built within Weka and thus guarantee correct results. A framework that garbles the results without the user knowing it is of little use to anyone. A Weka Classifier, for example, not only represents the learned data, but is also dependent of the dataset's header used to build the classifier. If the data used to build the classifier was preprocessed, for example the discretization of numeric values, the classifier will be of no use, if it cannot reproduce the preprocessing on a fresh dataset. What if our discretizing filter suddenly uses a different classification of numeric values, different to the one used when building the classifier? Or what if the attribute order or count changes between learning and classifying? What if our classifier is dependent of yet another piece of information we are unaware of?

In order to guarantee correct classification results, we can either invest a great deal of time understanding how Weka and or that fact, any other data mining tool works, or just leave it to the third-party tool providers extending the framework. The easiest solution is to require tool extensions to wrap their internal data mining object representations into an object that will implement a set of minimal framework interfaces. This ensures the proper separation between data mining tools and our framework. In the case of Weka, the tool offers three types of data mining products, the Classifier, the Clusterer and the Associator, all with similar means of classifying data, but no common base class or interface. Instead of integrating all three into the framework, we simply wrapped them in to an abstract WrappedWekaObject, along with the dataset's header, the applied filters and any other functionality or information required, as depicted in Figure 4.8 (bottom).

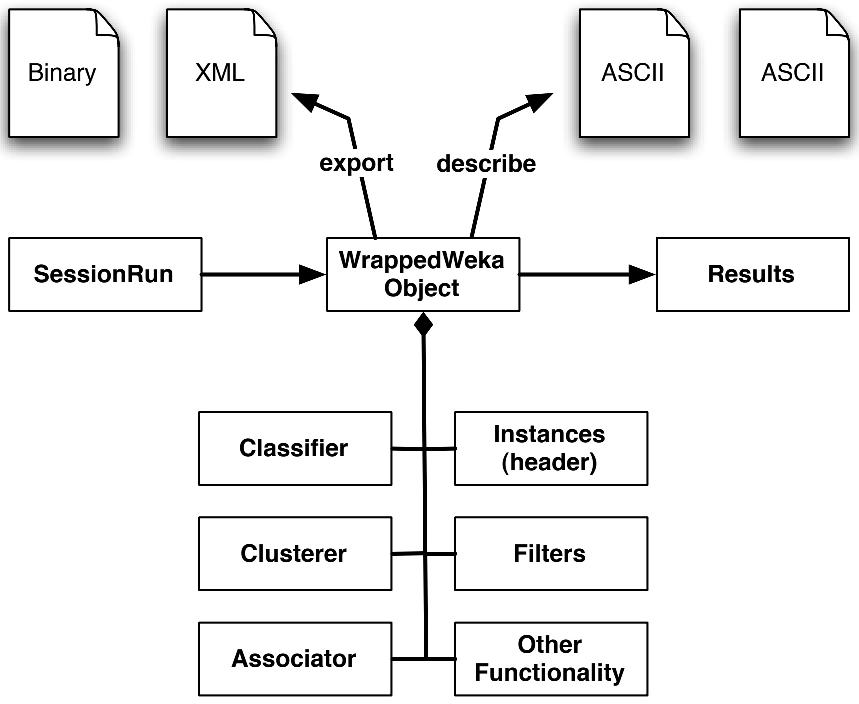


Figure 4.8: The WrappedWekaObject's composition, export and description possibilities and immediate context.

Upon building the classifier, the tool in use should contain all the information necessary to correctly classify a new dataset. The WrappedWekaObject would then store all this tool-specific information required and provide a simplified interface to the standard classification facilities offered by the tool, but invoked by the framework. Any additional wrapped functionality remains accessible through the corresponding bridge object and its graphical user interface. All in all, the framework cannot and does not need to know what is actually behind the WrappedWekaObject, as long as it does whatever it is meant to do correctly.

Currently, the WrappedWekaObject only supports Weka Classifiers and their corresponding header information, but does not regard any preprocessing of the dataset used to build the Classifier. This would require a more severe and riskier hack into Weka's source code. I intend to support KnowledgeFlow-models and look into converting any produced classifier into a KnowledgeFlow-model, as it is likely to automatically solve the problem of pairing the preprocessing information along with the classifier. Furthermore, the WrappedWekaObject should implement methods to persists itself to XML or a binary format and describe its contents as ASCII text or some other standardized format, as represented by the four file objects on the top of Figure 4.8.

4.4 Packaged Providers

Obviously, the framework itself does not perform much functionality. Its usability and success greatly depends on the integrated data mining tools and the available DataProvider and DataConsumer framework extensions. In order to test and validate the framework's features, it requires functioning DataProvider implementations. I intended to provide the following three DataProviders, which I will describe in more detail further below. Unfortunately, due to time constraints, the ARFF DataProvider has not been realized. All of the DataProviders extend the AbstractDataProvider base class, which provides a minimal functionality and so eases the development task.

4.4.1 The SQL DataProvider

The SQL DataProvider was a rather simple and obvious choice of which DataProvider to implement. Most of the data found today is stored in a relational database and at the least accessible via SQL. The idea behind the SQL DataProvider is to automate the transformation of a JDBC[17] ResultSet into the framework's DataSet, converting column metadata to AttributDefinition-objects. This simple, yet flexible construct allows for the integration of any data accessible via SQL into the framework, with little to no restrictions. I would like to point out, that some users may not only want to retrieve the data, but also process it along the way, such as the selection, projection or aggregation of data. The goal is to provide the developer with the full power and flexibility of JDBC and SQL. If the SQL DataProvider does not support what you are looking for, simply extend the provider and override whatever you dislike.

The current version of the SQL DataProvider offers a set of JDBC Drivers to access today's commonly found database instances.[18]. The DataProvider can thus be directly instantiated and used without further extending it, provided it includes the JDBC Driver of interest. If the it does not support a desired JDBC driver, a developer can easily package the desired drivers in a Plug-in extending the SQL DataProvider. Besides offering a means to modify the SQL query and set the connection's user name, password and url, the SQL DataProvider further offers a means to cache a connection between retrievals and include, exclude and identify the data's primary key column(s). All of these properties are accessible via the provider's graphical user interface, but can just as well be defined as a plug-in extension (extending the SQL DataProvider's plugin, and thus appearing as a SQL DataProvider) or programmatically (extending the SQL DataProvider classes and registering itself as an independent DataProvider.

The programmatic approach, for example, allows a developer to better control the management of a database connection. This comes in very handy, when working with databases of larger organizations with more complex IT landscapes. It is not unusual that a developer will have great difficulty in establishing a direct JDBC connection with the database. Often, database instances used in productive environments are only accessible via technical user accounts assigned to designated applications running on secure application servers. In most cases, the application server will be the only entity with access to the login data, manage the connections and offer them to its applications as a connection pool. A programmatic approach therefore gives you the flexibility to work around these obstacles while still serving its core purpose - the mapping of a ResultSet object to a DataSet object.

Last but not least, the SQL DataProvider is the provider of choice to validate the framework, by reproducing the data mining results published in [Kna05]. Most of the Mozilla project's data and related FAMIX data has been stored in relational database instances. Furthermore, the SQL DataProvider is an ideal candidate, in combination with the HQL DataProvider, to test the merging of different DataProviders, as described later in section 6.1 on page 55. Due to Hibernate's object-relational characteristics, data stored with Hibernate is also accessible via SQL.

[17]Java Database Connectivity - an API oriented towards relation databases for the Java programming language, defining how a client may access a database.

[18]Database instances, such as DB2, Oracle, SQL Server, MySQL, PostgreSQL and HSQLDB

4.4.2 The HQL DataProvider

The HQL DataProvider is the object-oriented counterpart to the SQL DataProvider. As this framework is primarily being developed to data mine source code in the context of S.E.A.L.'s release history analysis project, I have chosen to implement a generic DataProvider capable of accessing any data persisted with Hibernate. Since the analysis of source code can be considered a rather new development in the field of Software Engineering, it is safe to assume that the corresponding tools and produced data will be rather new as well. A good part of these tools are still in development and experimental in nature. They are therefore likely to use newer technologies, such as Hibernate's object-oriented persistence layer.

From a Software Engineering point of view, it is not really wise to access another application's data model, despite its object-oriented nature. The application, in our case an eclipse plug-in, should instead offer adequate interfaces, allowing for a controlled and stable access of the plug-in's underlying data. The general idea of the framework is not to hack into or simply tap other databases, but rather integrate data producing tools and plug-ins into the framework. The HQL DataProvider should therefore only be applied when accessing data that has no immediate producer that can be integrated.

As with the SQL DataProvider, its main purpose is to automatically map a data-object's method return values to a DataSet attribute values, using the DataObjectDefinition as a primary means to accomplish this. The challenge here lies in the processing of a query result consisting of Strings, primitive data types and arbitrarily complex java objects.

The provider's functionality is rather straightforward. When processing a result, it checks whether the value is anything other than a String or a primitive type. If so, it looks for the object's DataObjectDefinition. The DataObjectDefinition contains information pairing the object's methods of interest with the corresponding AttributeDefinition objects. Upon populating the DataSet, the object is automatically flattened by invoking the object's methods and storing their return values. If the id column's value is a complex object, the DataObjectDefinition must be consulted to identify the id-methods. All String values and primitive types are processed analogue to the SQL DataProvider. Unlike the SQL DataProvider, the HQL DataProvider *must* always be extended in order to be instantiated. This is why it is completely covered by the Domain-specific DataProvider in Figure 4.9, whereas the SQL DataProvider is not. A HQL query requires JDBC drivers, Hibernate configuration information (either as an XML file or as a `java.util.Properties` object), mapping files and the model classes to execute correctly. All of these must be declared by and provided as plug-in-extensions to the HQL DataProvider plug-in (represented by the Domain-specific DataProvider).

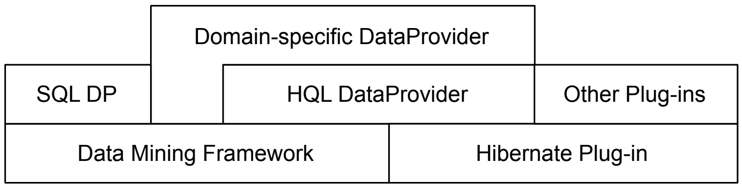


Figure 4.9: The HQL DataProvider as plug-in layers

The HQL DataProvider currently consists of two plug-ins. One performing the functionality described above, the other offering Hibernate's facilities (represented by the Hibernate Plug-in) to any other plug-ins, as depicted in Figures 4.10 and 4.9. The JDBC Drivers, Hibernate configuration information and model classes are intended for the Hibernate plug-in, but are passed through the HQL DataProvider (dotted line on the left), as the plug-in providing these objects cannot know which Hibernate plug-in is currently in use. At present, the framework provides its own

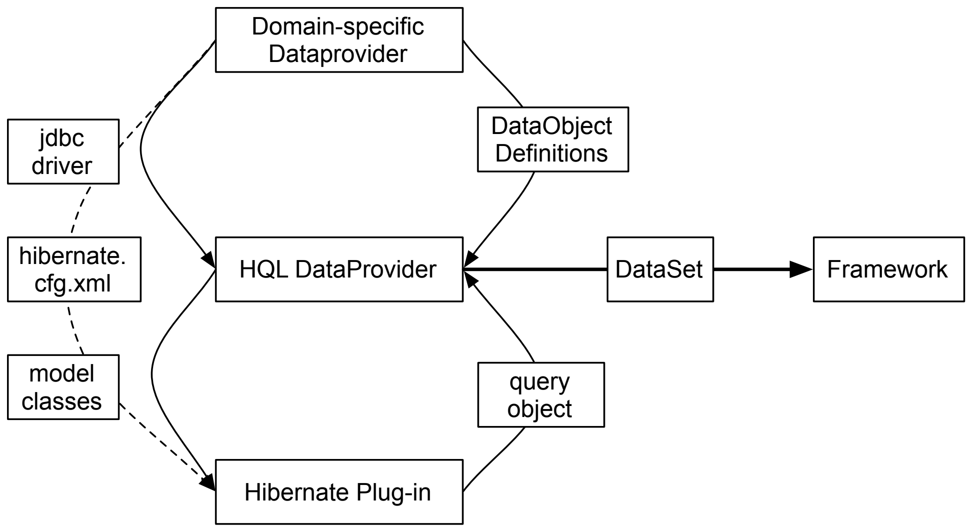


Figure 4.10: The three levels of the HQL DataProvider

implementation of a Hibernate Plug-in, but is ultimately intended to run from any other plug-in offering Hibernate facilities. It would be a waste of resources if every plug-in had its own Hibernate implementation. The HQL DataProvider then retrieves a Hibernate Query object and maps its contained values, using DataObjectDefinitions wherever required, to a DataSet destined to the framework.

The JavaObjectDataProvider

After refactoring the HQL DataProvider for the n-tieth time, it became obvious, that the facilities allowing the use of Java Objects next to Strings and primitive types could be of use to other DataProvider implementations. This ultimately resulted in the development of the DataObjectDefinition class. As the framework is intended to integrate other data producers as eclipse plug-ins, it is extremely likely that these other data producer's data will be in the form of Java Objects. It would therefore be wise to provide developers with a base implementation that will automatically convert collections of objects to DataSets. Unfortunately, I haven't gotten around to extracting a fifth, more generic DataProvider, that accepts collections of objects as input, from the HQL DataProvider (it basically already has all the necessary code). The HQL DataProvider would then obviously extend this new *JavaObjectDataProvider*.

4.4.3 The ARFF DataProvider

It is rather unfortunate, that Weka is being integrated into a data mining framework that does not accept data in its native format, the ARFF format. A newcomer would have a lot of difficulties understanding the framework and data mining itself, if it cannot run simple examples as described in the many data mining books (such as [IHW05]). Experienced users that invested much time and effort in creating datasets in ARFF will find little use in such a framework. Unfortunately, due to time constraints, I was not able to implement a DataProvider that accepts an ARFF file or stream as input, but plan to do so by the next release.

I would like to note, that despite the ARFF format being the closest we can get to the actual

DataSet used by the framework (the DataSet was actually influence by the ARFF format), it does not support the concept of a tuple id, entailing a potential problem that cannot be ignored. The primary purpose of an ARFF DataProvider is to feed its data to the framework and later merge it with other data - not really feasible when the ARFF's id column is missing.

4.5 A Data Mining View

Another considerable aspect of the framework's development is the design of a graphical user interface. Easing the data mining task also includes providing a clean, simple, comprehensive and intuitive user interface that seamlessly integrates with its environment, in this case the eclipse platform. A feature rich plug-in or application is of little use, if you cannot work with its interface. I personally find that the user interface is often neglected, despite it being the only component an average user will actively use.

A major challenge to the framework is that it is expecting feature-rich third-party implementations that require rich user interfaces to control the provided features. Each tool knows best how to represent itself, so the framework is better off providing graphical user interface space to each component. Unfortunately, each third-party developer will have his or her personal style and will incorporate it into the extensions user interface. This will result in a wild diversity of look and feels, where users are likely to spend more time learning how to use each user interface than actually performing data mining tasks. Familiarity is necessary, but must not apply for everything - only recurring user interfaces should look the same. It would be a waste of time to reengineer Weka's user interfaces to look and feel the same as the framework or eclipse. Our goal is to integrate, not reengineer. Instead, any original Weka user interfaces should be left untouched and displayed in a separate window, to stress its independence of the framework. Any framework-specific tool configurations should look and feel like eclipse, using its set of graphical components. Any recurring user interfaces, such as the minimal configuration requirements of a DataProvider are to be implemented by the framework, as it is ultimately the framework specifying these interfaces.

The problems arise when for example combining the framework's user interface and the DataProvider's domain-specific user interface in a single DataProvider configuration. Either the framework passes its interface as a parameter to the DataProvider, which in turn will blend it into whatever it is producing, or the DataProvider returns its contribution as user interface fragments which are then put together by the framework. At present, all existing DataProvider's implementations consist of a single user interface, not two. This is not a problem as I developed them along with the framework, considering their look and feel. The framework's next release does intend to support separate user interfaces and relieve the developer from having to re-implement a user interface for the DataProvider's minimal configuration.

Currently, the framework is controlled by a single eclipse view, the *WekaPlugin*[19] View, consisting of six framework-specific tabbed panes and another to control the WekaBridge. Integrated data mining tools are offered a separate tab, while DataProvider are only offered space within a tab (as `org.eclipse.ui.forms.widgets.Section` objects). If an integrated tool or DataProvider require more space, they can either work with dialogs or request space in the configuration tab. The configuration tab is intended to configure the framework or any other components and is organized like a notebook, showing a single page at a time, mapping each page to a component. Unfortunately, there was insufficient time to implement the configuration tab, but this description should apply by the framework's next release.

The framework's user interface is implemented as a view, but looks like an editor. The difference lies in the graphical components used and the fact that changes to a view apply immediately,

[19]Will be renamed, once the framework is decoupled from Weka.

whereas changes in an editor only apply when they are saved to its underlying resource. The reason for not choosing an editor, is that eclipse automatically associates an editor to a resource. The framework does not support the notion of a framework-resource, although it would technically be possible to persist the framework's state as a resource. Instead, the framework's state is automatically persisted upon closure or shutdown using eclipse's IMemento structure.

I would like to add, that it was rather difficult to mimic the look and feel of an eclipse editor. Unfortunately, expensive books such as the one I used, "eclipse, Building Commercial-Quality Plug-ins", do not really mention how to make a plug-in look and feel like eclipse. I personally find this a rather important requirement for a commercial-quality plug-in. Anybody interested in mimicking eclipse's look and feel is welcome to use the framework's code as an example.

Chapter 5

SE Aspects & Problems encountered

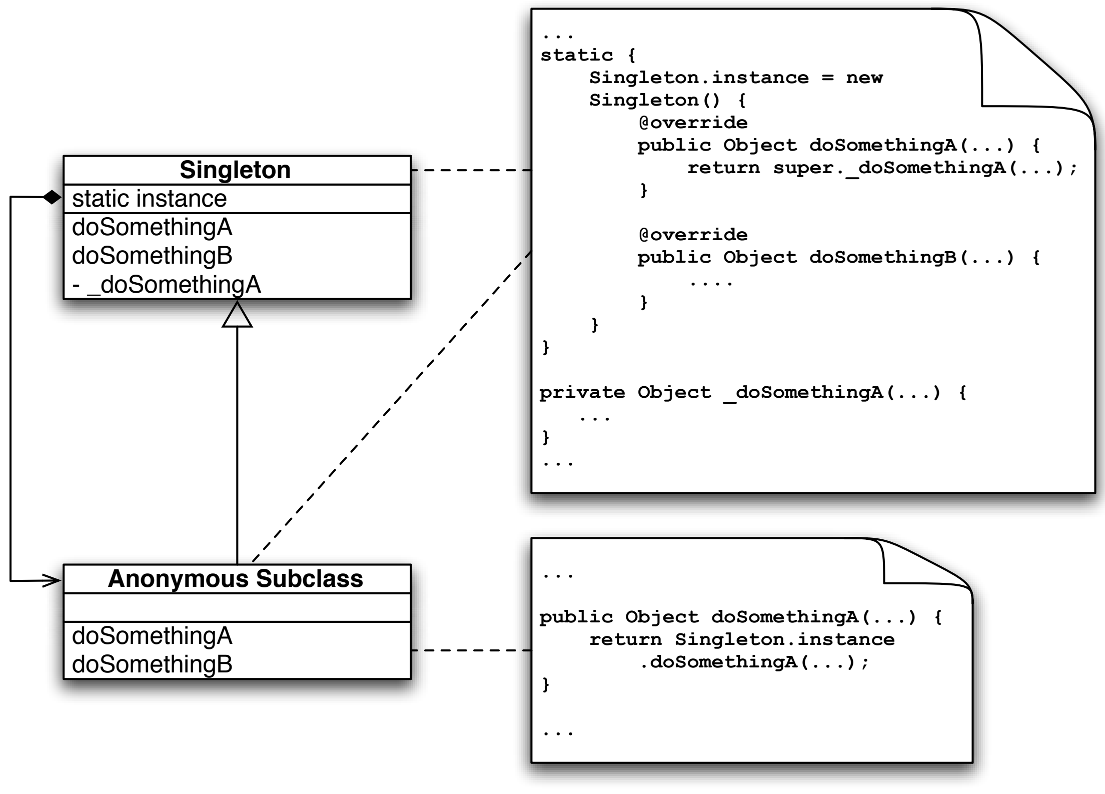
Throughout the framework's development, we have come across much source code, had to deal with various architectural decisions, stumbled on several problems and had a good look at the many possible approaches and solutions. This chapter will first point out a few of the software-technical highlights encountered and later go through the more complex problems that arose.

5.1 A Pseudo-Singleton Pattern

While developing the framework for eclipse and other software, we have often encountered situations, where we would have liked to use the Singleton pattern [GHJV95], but were unable to do so, as we had no control on the object is accessed. The idea behind the Singleton Pattern is to hide the constructor, declaring it as private, and so control its instantiation. The class then instantiates itself from within and stores the instance as a static variable, accessible through a static accessor-method. There are situations though, were the hiding of a constructor will not do the trick. When for example, declaring executable Java classes in an eclipse extension, the extension-point's source code will instantiate the declared classes through reflexion or by simply invoking the `Class.newInstance()`-method. This is a very common way to publish and use interface implementations between plug-ins. If the constructor is hidden, the instantiation will fail, just as it was designed to, resulting in a possible failure or erroneous behavior of the extension. A work-around would be to use the Factory Pattern [GHJV95], encapsulating the instantiation in a factory class and declare the factory's implementation instead of the singleton class itself. But ultimately, we are just shifting the problem to another class. We still cannot control the factory's creation and risk ending up with more than one factory object. Ironically, the factory is intended to be a single point of entry and is as such an ideal candidate for the Singleton Pattern.

All in all, we will not be able to implement the Singleton Pattern if we have no means of preventing a component to access a class via its zero-parameter constructor, such as the eclipse platform does. The Java language does not offer a means to modify a constructor to return a static variable of the same type instead of creating new one. Simply restricting a constructor's accessibility might enforce a controlled instantiation, but will severely limit its accessibility - the class becomes useless to any consumer unable to locate or use the static accessor-method. We can either change the consumer's code or come up with a more flexible solution.

Throughout the development of this framework, we came up with a neat, yet simple approach that mimics a singleton's behavior, while still permitting access to the object via its zero-parameter



```
...
static {
    Singleton.instance = new
    Singleton() {
        @override
        public Object doSomethingA(...) {
            return super._doSomethingA(...);
        }

        @override
        public Object doSomethingB(...) {
            ....
        }
    }
}

private Object _doSomethingA(...) {
    ...
}
...
```

```
...

public Object doSomethingA(...) {
    return Singleton.instance
        .doSomethingA(...);
}

...
```

Figure 5.1: A UML Diagram of the Pseudo-Singleton Pattern

constructor. Like the singleton, our approach will hold the unique instance as a static variable, allowing the developer to instantiate it upon its first retrieval or within the class' static constructor, as depicted in Figure 5.1. What is important, is that the constructor be publicly accessible and the singleton instance be an anonymous subclass of the singleton class. This anonymous subclass would then implement all of the singleton's functionality, leaving the singleton class itself with a mere delegation of method invocations to its static singleton instance.

What is special about this construct, is that it controls the instantiation without limiting it - anybody can instantiate the singleton class and will always end up invoking the same single object's methods. This construct provides singleton-characteristics for the functionality, as it is locked up in the anonymous class, but not for the object itself, permitting an arbitrary amount of instances. The comparison of two instances (`object1.equals(object2)` or `object1 == object2`) will normally return false if the comparison is not delegated to the anonymous subclass as well. But this should not bother us, as we often apply the Singleton Pattern to ensure singleton-characteristics on the functionality, not on the objects. This approach is not that different to using a factory class, only that everything is encapsulated in a single class, the objects are available through the singleton's zero-parameter constructor and generally very light (basically consist of little code and no variables).

We decided to refer to this construct as the "Pseudo-Singleton Pattern", as it ironically allows for more than one instance. We are pretty sure that many other developers must have come up with such a solution, but could not find any publications to reference. Considering this thesis' software engineering nature and focus on eclipse plug-in developments, we found it worthy of mentioning. Developers should take special care in overriding all of the singleton's public methods, so as to avoid an endless recursion.

5.2 Lightweight XML Persistence

A further highlight is the creation of a very lightweight XML persistence framework, primarily used to persist the framework's state, but also generally used throughout the framework to comfortably parse any XML encountered, such as the Metric-exports described in section 6.2. The framework basically consists of two classes, comprising about 220 lines of code, all based on SAX[1] [ERH04]. The beauty of it, is that it supports the notion of XPaths[2], can record arbitrary document fragments or XML elements, parses a document in as little as two Java statements while remaining relatively light, extendible and most of all simple. Unlike other XML persistence frameworks, it provides absolute control over the underlying XML, allowing the comfortable and easy parsing of any well-formed XML snippet. Appendix A describes the *Lightweight XML Persistence Framework* in more detail, providing an example and code-snippets for a better understanding.

5.3 The Need for a Global ID

Considering that the framework's functionality basically breaks down to providing a sort of data-trading and -merging platform, the framework itself will be of little use, if it cannot interpret or identify the underlying data. This would not be a problem, if the framework provided or produced the data itself, but as it is designed to accept data from any given source, it will have to find a means to litteraly "connect the dots".

At present, a basic requirement to all DataProviders, is that they uniquely identify each tuple, pairing it to an id-object. Due to the framework's hashmap-specific implementation, two keys are considered the same, if the statement `key1.equals(key2);` returns true. The framework imposes this requirement for two reasons: First, it attempts to merge data originating from different DataProviders by matching the tuple's id objects. Secondly, the framework must fullfil the traceability-requirement described earlier on in section 2.5. It is not of much use to know a tuple's classification, if we cannot identify the tuple.

Unfortunately, when it comes to merging data on their id-objects, most DataProviders will pair the data with a domain-specific id, requiring the prior translation of the id-objects before merging. A translation must take place within the framework, as we should not assume a user is able to change the original data. Fortunately, the framework requires DataProviders to support the tranlation of id-objects, respectively the assignment of an IDTranslation implementation to a DataProvider per DataSession. But herein lies a futher problem: A DataProvider cannot know, which ids are going to be used by other DataProviders, nor can it guess the final relation's id-domain. If the DataProvider is lucky, then it will provide data for a specific domain, such as software engineering. Providing data for a specific domain significantly reduces the possible id-domains. A DataProvider could then implement translations to match a set of possible ids. In the case of software engineering, candidates for an id-domain range from whole Java projects to methods, but at least they are finite.

Ideally, a DataSession would publish its resulting relation's id-domain and have DataProvider's automatically match it. Unfortunately, in reality this is a lot more complicated than it sounds and is probably impossible to realize. We are basically stuck with translating id-objects back and forth. Our only hope is that, at least for the software engineering domain, somebody

[1]Simple API for XML - an event-based API for reading XML Documents

[2]A non-XML language for identifying particular pars of XML documents.

come up with a uniform way of identifying data-minable software resources and have DataProviders adhere to this type of identification.

5.4 Preprocessing when Classifying

This is an unfortunate shortcoming that we only noticed close to the end of the development. Upon reproducing the results published in [KPB06], we realized that any preprocessing of the data within Weka's tools, such as the discretization of numeric values, is not incorporated into the classifier and is lost as such. When attempting to classify fresh data through the WekaBridge, we were actually feeding it with unprocessed data, ultimately producing wrong results.

The question arose, as to where the preprocessing information should be stored and who's responsibility it would be to ensure this. A simple work-around would be to allow a user to pass data through Weka, to pre-process it accordingly, before classifying it from within the WekaBridge. Unfortunately, this entails manual work and it would not be reasonable to assume a user could remember and correctly execute all of the required processing step. Seeing that a classifier ultimately depends on the data used to build it, we find that it is also greatly dependent of the filters used to pre-process it. A user should be working with several classifiers and not several versions of the data. Hence, the classifier should be aware of this pre-processing and also be able to automatically reproduce it on fresh data before classifying it. Our solution is to wrap more information, such as the applied filters, and more functionality, such as the application of these filters on fresh data, into a data mining object returned by a tool. Hence the importance of the WrappedWekaObject described earlier on in section 4.8.

Despite it not being an actual part of our framework, we found this shortcoming worth mentioning, as it entailed an important architectural decision - a returned data mining object should know exactly what it is intended to do and how to do it. The framework should assume that a data mining object be autonomous, producing results on its own and not be part of a larger process of which the framework has no knowledge.

5.5 Exception Handling

This is a problem that is most likely to arise throughout time and should be considered when further designing and developing the framework. Our framework basically relies on third-party implementations that will sooner or later fail for whatever reason and start throwing exceptions. Unfortunately, our framework has no clue as to where these problems occur, if they are critical or ignorable and worst of all, how to adequately inform the user about their existence. At present, the framework simply catches any exception and prints it to the user interface, hoping the user will understand what is going on. For example, if we wish to view our data in the *Probe Data* pane, and by chance forgot to provide the database password, the framework will provide us with a beautiful representation of the thrown exception instead of the desired data. Addmittedly the current solutions does work, but it is far from being user-friendly. If the user is not familiar with SQL or the concept of an exception he or she will be lost.

All in all, the message must get accross to the user, that something is not working correctly. The framework can either implement a minimal interface for exceptions, enforcing the use of exceptions enriched with user-specific information, or attempt to delegate the exception handling to third-party contributions, giving them more control over the framework's user interface, to adequately publish and handle their exceptions. The former, while rather simple for the framework to implement, might entail a lot of work for DataProvider developers. The enriched exception object might not suite every DataProvider's needs. A simple message might not suffice to adequately inform the user on what is going wrong. The latter options seems more promising and deserves more attention when further developing the framework.

Another problem to consider, is when third-party components trigger a framework functionality, which in turn invokes other third-party code, that will end up throwing an excpetion. In the worst case the exception will be thrown all the way back to the initial third-party contribution, who will be unable to handle it. Special care must be taken to catch and adequately handle all exceptions "passing" through the framework.

5.6 Transitively Dependent Data

At present, the framework can only correctly merge data whose id can be translated to the id-domain used by the final relation. But what if some data's id cannot directly translate to the final relation's id-domain? What if, as depicted in Figure 5.2 (left image), `ABC` has no means of matching the `D`-id to directly merge with the other data, and must instead merge with `CDE` on `C` (middle image) to then merge as a whole with the rest of the data on the `D`-id (right image).

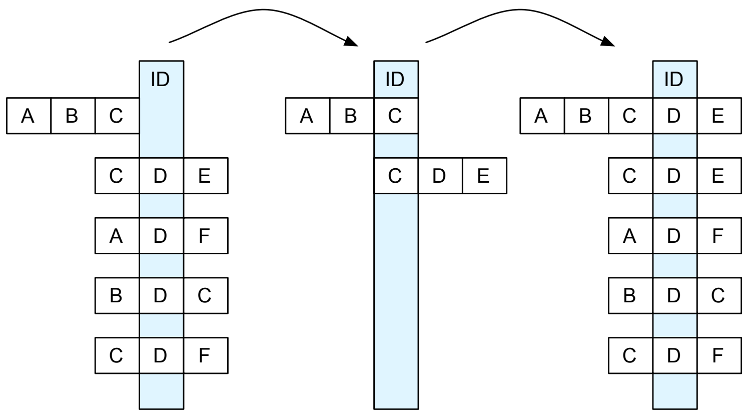


Figure 5.2: A dataset depending on another to merge with the rest, basically requiring more than one merging cycle.

The only means to support transitively dependent data, is to also allow the assignment of one DataSession to the other. Figure 5.3 depicts an example, where DataProviders A, B and C first merge in DataSession 1, before merging as a whole to DataPovider C in DataSession 2. This would technically be possible, as the DataSession class already implements the most important method of the DataProvider interface, namely the `getDataSet(DataSession session)` method. A DataSession respectively its underlying DataSet would then require a means to redefine the id paired to each tuple. Furthermore, seeing that DataProviders are required to provide an IDTranslation implementation, a DataSession would be required to do the same, if it is to be merged with other data.

5.7 Memory - A Limited Resource

Throughout the development of the framework, it became apparent, that the framework itself could potentially consume a lot of memory and eventually run out. Up until now, the only data

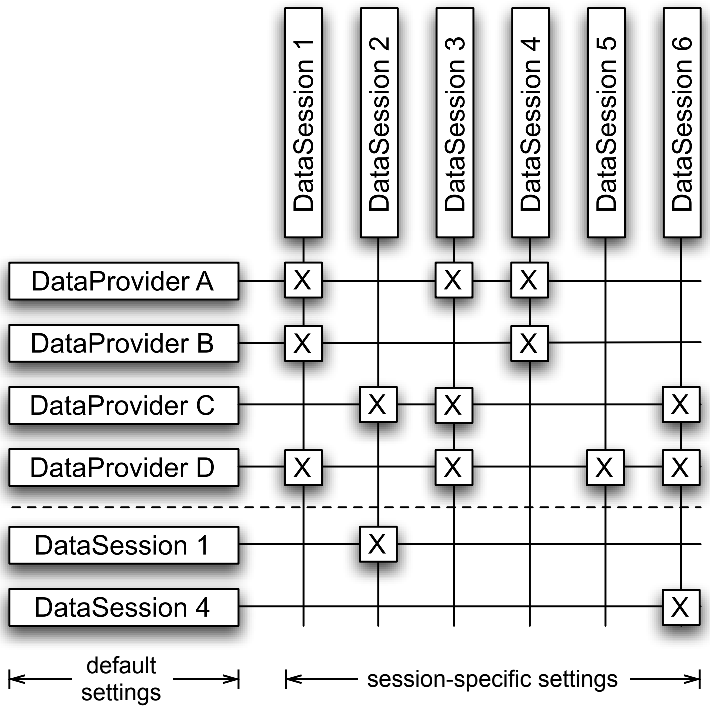


Figure 5.3: An extension to Figure 4.3, displaying the assignment of not only DataProviders, but also DataSessions to a given DataSession.

we have ever loaded, were the ones described in [IHW05] and a set of the framework's metrics. And up until now, it all worked at a good speed without memory problems. But when taking a closer look at the implementation, we realize that there are many copies of the data lying around, ranging from cached DataSet objects to SessionRuns to classification results. If we are not careful with memory, the framework will only be useful for smaller data mining projects.

We find there are two possible solutions to the problem. We can either offer the optional backing of the framework with a database instance, allowing it to automatically persist unused data to clear up memory, or support incremental DataProviders, retrieving data as it is consumed. The former option, while conflicting with our requirements to not depend on a database, offers a greater flexibility and is probably easier to realize. Furthermore, a remote database instance would technically permit an easy and flexible sharing of a data mining project. The latter option is likely to fail, as a good amount of data mining algorithms are not incremental in nature. On the brighter side, the classification of new data is usually incremental and does not require many resources. Either way, it is advisable to consider memory consumption when further developing the framework.

5.8 Other Usefull Stuff Worth Knowing

Conflicting SWT and AWT Threads

We came across this problem when integrating Weka's native user interfaces into the framework. From the *WekaBridge* pane, a user can open instances of Weka's Explorer, Experimenter or KnowledgeFlow. Reengineering them to actually look and feel like our framework would have been a lot of work and have little effect. It made a lot more sense to keep Weka's user interface as it is and so offer a familiarity to its existing users. Furthermore, Weka's user interface is written using Java Swing [GRV03], while our framework's user interface is based on SWT [EC06], which caused the following problem.

When passing classifier objects back to the WekaBridge, its user interface is designed to react to the new input and display it accordingly. To do so, we defined a WekaBridgeListener interface supporting the notification of WekaBridge-relevant events. The WekaBridge's *Classifier Instances* section for example, implements this interface and updates its internal `TreeViewer` object whenever a classifier is passed to the WekaBridge. Upon doing so, the whole plug-in would irrecoverably crash.

It turns out, that the Explorer's thread, being a Swing-thread, ended up invoking a SWT-component-refresh, creating a conflict. Apparently, both SWT- and Swing-threads are not compatible and should not invoke each other's user-inface-related methods, such as the repaint or refresh of a graphical component. As the following code snippet shows, we solved this problem by encapsulating the conflicting invocation into a runnable thread and having it run outside of the main SWT or Swing thread.

```
ClassifierSection.this.getSection().getDisplay()
        .asyncExec(new Runnable() {
            public void run() {
                treeViewer.refresh();
            }
        });
```

ClassLoaders in a Plug-in Environment

This problem was encountered when migrating our development from a standalone version to an integrated eclipse plug-in. To avoid complications with eclipse and SWT, the framework was initially developed as a standalone version with most of Weka's code packaged as a jar file.[3] Upon migration, the code started to crash, throwing multiple NullPointerExceptions from everywhere. It turned out, that several components, especially Weka's libraries, were trying to access packaged resources, but somehow were unable to do so. The problems were obviously somehow associated to the plug-in architecture, as they only started to appear once we moved the implementation to the eclipse platform.

As we later found out, there are several ways to access resources with Java. The most commonly known is the `java.io.FileInputStream` with a `java.io.File` object representing the resource. This will normally not be a problem, as the file is normally something located by the user through a FileChooser[4] dialog with an absolute or relative path within the user's filesystem. It becomes a bit trickier, when accessing resources "bundled" within your code, such as when exporting an application to a jar or war file.[5] This is when you start working with

[3]Java Archive, used to distribute Java libraries and applications.
[4]A simple mechanism allowing the user to visually select a file from the filesystem.
[5]A Java Web Archive. Similar to jar, only that it contains a complete web-application according to the Java-Servlet-Specification.

`java.lang.ClassLoader` objects. Both our framework and Weka were accessing bundled resources through Classloader objects, but apparently the wrong ones.

It turns out, that a standalone application is rather forgiving when using ClassLoaders, as it will normally provide one single ClassLoader allowing any component to successfully access any bundled resource. Using ClassLoaders in a plug-in evironment is by far not as forgiving. Unlike the standalone environment, a plug-in environment such as eclipse, is likely to provide a ClassLoader per plug-in. You are likely to encounter serious problems, if you are not careful when accessing resources. To solve our problem and to better understand how ClassLoader's function, we ran a series of tests, accessing the same resource, each time with a different ClassLoader. Surprisingly, all but one ClassLoader failed! What really amazed us, was that the `new String() .getClass.getClassLoader();` statement actually returned null, when run from within a plug-in. It seems that the one and only reliable way to get always get the correct ClassLoader, is to retrieve it from a static class bundled with the resource of interest. The following code snippet may help understand:

```
PluginClassName.class.getClassLoader().getResource("...");
PluginClassName.class.getClassLoader().getResourceAsStream("...");
```

Note, that "PluginClassName" has to be a class bundled along with the resource. As shown in the following snippet, a good practice would be to encapsulate all access to a bundle's resources into a single class and provide the resources via static accessor-methods.

```
public class ResourceAccessor {
    public static InputStream getXYZResource() {
        return ResourceAccessor.class.getClassLoader()
                .getResourceAsStream("...");
    }
}
```

Weka's "Discovery" Algorithms

Another problem encountered when migrating the implementation to the eclipse platform, was that Weka suddenly did not know any its filters, classifiers, clusterers or associators. At first, we thought it had something to do with the ClassLoader problem, but soon realized that it still did not load any classifiers, even after we solved the ClassLoader problem.

It turns out, that Weka does not readily load the objects from a file as we first assumed, but actually discovers them upon startup. It seems that Weka's implementation uses the Java classpath information to search for objects subclassing given base classes, such as Weka's Classifier class. As with the ClassLoader problem, this one only appeared when running the code in eclipse's plug-in environment. It seams that Weka's implementation accesses the classpath information via `System.getProperty("java.class.path");`. When running in a plug-in environment, each plug-in will have its own classpath information. Accessing the System's property will get you the framework's classpath information, but not the one you will be looking for. Unfortunately, we have not yet found out, how to access a plug-ins individual classpath information. We solved this problem by overriding the discovery algorithm and now force it to load the classes from a properties file. The related code *hacks* are available in Appendix B.

Chapter 6

Proof of Concept

As a form of validation, we decided to not only test the implementation for correctness or failures, but also subject it to a general *proof of concept*. We find it rather difficult to formally test requirements, such as the easing of the data collection process or the ability to load piece of data from any conceivable source. In conclusion, we have decided to focus on the following five distinct features of the framework, namely the merging of data originating form different sources, the extension and integration of an existing plug-in as a DataProvider to the framework, the integration of a data mining tool along with its user interfaces, the transferability of a data mining project and the actual mining of a given dataset.

6.1 Mixing Providers

Due to the framework's ability to integrate any possible data source, it is vital that it can correctly merge data provided by not only different DataProvider instances, but also different implementations. In order to test this features, we decided to break up an existing dataset and its fragments through different DataProviders. As already described in section 3.3, we stored the iris dataset provided by Weka as Iris-objects using Hibernate's object-oriented persistence layer. The advantage of using Hibernate, is that it actually stores the information in a relational database, allowing it to be accessed by our SQL DataProvider, and so simplifying the test. We then configured the DataProviders to ensure a fragmentation of the data's retrieval. Figure 6.1 depicts the data fragments from each DataProvider (left-hand side) and the resulting merged relation (right-hand side).

Notice, that the data is fragmented in both dimensions (tuples and attributes). SQL DP1, for example, will be referencing respectively defining the same attributes as HQL DP2, while sharing tuples with both HQL DP3 and SQL DP4. Upon creation of a SessionRun, the DataSession will merge all values using id-objects and AttributeDefiniton as keys to identify them, producing a single, coherent DataSet as can be seen on the right-hand side of Figure 6.1. As already presented in our *tour of the framework*, the DataSession was able to automatically merge the data correctly. This example was rather simple, as the id-objects were compatible and therefore did not require a translation. But none the less, it demonstrated the framework's ability to merge arbitrary data. Difficulties might arise, when working with more complex data sources, that use incompatible id-objects.

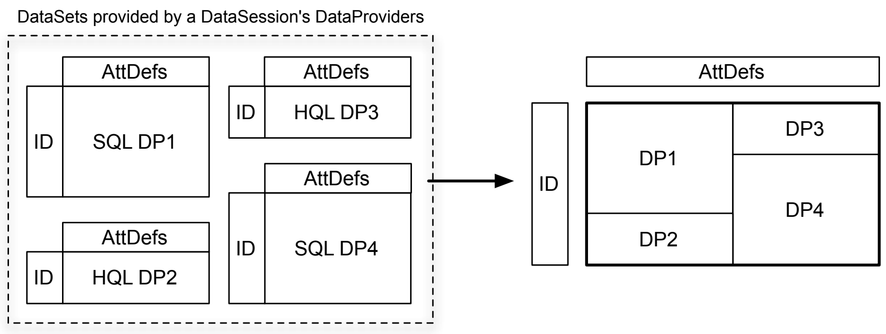


Figure 6.1: The mixing respectively merging of DataProvider's DataSets

6.2 The Metrics DataProvider

A personal goal of mine was to implement a DataProvider that actually uses an existing eclipse plug-in as a data source. For this purpose and in the spirit of data mining source code, we decided to integrate the net.sourceforge.metrics plug-in, which will produce an extensive set of source code metrics for a given Java project. We were interested in how much work it would actually involve to extend plug-ins to function as DataProviders and if it was possible to do so without entailing extensive modifications to the existing plug-in. Furthermore, the Metrics DataProvider should also serve as an example and reference for any developers attempting to integrate an existing, data-producing plug-in. The Metrics Plug-in turned out to be an ideal candidate for integration, as it already provides a means of exporting metrics to XML and defines an eclipse-extension-point accounting for custom exports.

The Metrics DataProvider currently employs a DataProviderFactory as a singleton to collect the metrics exports and create DataProvider instances. Each instance has a reference to the singleton to access the exported metric data. Instead of exporting the metrics to a file, as the Metrics plug-in would normally do, it instead passes the metrics as a XML-String to the singleton class for storage. The exported object is labeled after what would have been the file's name (entered in the FileChooser-dialog) followed by a timestamp. A DataProvider instance is then assigned an XML-String as source, which it later parses using the lightweight XML persistence framework (described in Appendix A) to populate the resulting DataSet. The lightweight XML framework proved to be quite useful and radically simplified the parsing of the exported metrics.

Unfortunately, the stored XML-Strings can be rather large in size (2.5 Mb for the framework's metrics) and if not properly managed, result in a waste of memory. As they are the DataProvider's internal data model, it is the DataProvider's responsibility to keep itself clean respectively persist and cache any large objects.

All in all, the Metrics DataProvider was rather simple to implement and, as seen in our *Tour of the Framework* example, works surprisingly well. It comprises four packages, 15 classes, one view and about 1500 lines of code, whereas most of the code is related to the parsing of XML. Furthermore, it defines two extensions, one for the framework, the other for the metrics-plug-in and offers access to any loaded Java Project's metrics.

6.3 A Bridge to Weka

We successfully managed to open one or more instances of Weka's Explorer tool from within the framework and pass a DataSession's snapshot to any of the open instances, without severely impairing the Explorer's functionality or garbling its graphical user interface. To our surprise, we were able to integrate Weka's tools and their corresponding user interfaces with minimal changes to the source code and without entailing limitations to its functionality. Furthermore, we were able to extract classifier objects built within Weka's tools and successfully use them to classify new data from within our framework.

6.4 Transferability

An important feature of the framework, is that it can transfer data mining project from one system to the other and through time as well. The current implementation does not yet support the export or import of the framework's state, but does persist it through eclipse sessions, guaranteeing a transferability through time. This comes in very handy, as data mining usually consumes a lot of time and is seldom performed in one clean go.

Upon shutdown, the framework automatically persists all configurations to XML and passes them to eclipse as the view's state prior to shutdown. Upon startup, the framework is passed its previous state, which it then parses using the lightweight XML persistence framework and so restores itself. The transferability between systems is not really a challenge, as we are already persisting the state in XML. All we need to do, is provide a user interface that will invoke the persistence to XML and export it to a file. The importing framework instance would then restore the framework's state, as if the XML had been passed on by eclipse. The lightweight XML persistence framework proved itself easy to use and configure, requiring a minimal amount of code to persist and restore the framework's state.

6.5 Running Data Mining Example

This feature, despite it being very important to the framework, has already been partially tested. The *Mixing Providers* section proved, that the collection and preparation of data works without flaws. The *A Bridge to Weka* section describes how well the Weka tool integrated into the framework, allowing the proper transfer of data to Weka and the return of any produced classifiers. As the integrated version of Weka is equivalent to its standalone counterpart, there are no real framework-specific limitations as to what you can data mine. Our test basically consisted of the simple steps demonstrated in our *Tour of the Framework* in chapter 3.

Chapter 7

Conclusions & Future Development

7.1 Conclusions

All in all, we are happy to say, that the framework does contribute in easing the task of data mining. The ability to tap any conceivable data source without restrictions at the cost of a bit of programming does seem appealing and did prove itself useful when implementing the Metrics DataProvider. But as somebody once said: “A fool with a tool, is still a fool”. Ironically, the framework does not really provide much functionality and it is definitely not a *silver bullet* or revolution of the data mining process. Good results still depend on the user’s knowledge and proficiency in the domain. On the other hand, the functionality that it does provide, despite being simple, is a key element and thus a vital foundation for the integration of third-party contributions. The framework is merely designed to serve as a data-trading and -merging platform, connecting data providers (sources), data processors (tools) and data consumers (sinks). And admittedly, it does its task very well.

With just a few hundred lines of code, you can programmatically declare a new DataProvider, a few hundred more and you are already producing DataSets. Add another few hundred lines of code and you will end up with a fully integrated and configurable DataProvider. Changes to the data or its underlying source will only require a change to the corresponding DataProvider, not the rest of the framework or any other DataProviders. Furthermore, the framework avoids the production and management of redundant copies of data and does an excellent job in leaving existing code and data untouched. Thanks to the integration of the metrics plug-in, we can now data mine any Java project loaded into eclipse.

As for the whole development process and the thesis as such, I am especially proud to have created a not only working, but thanks to the Metrics DataProvider, also readily useable piece of code. I am especially fond of the thesis’ spin-off products, namely the Pseudo-Singleton Pattern and the Lightweight XML Persistence Framework. I hope the data mining framework will be of use to the S.E.A.L. group and integrate well with the oncoming evolizer environment. I look forward to seeing some interesting DataProvider and DataConsumer contributions from their part.

7.2 Future Development

As already pointed out in various sections of this thesis, many of the described features are still far from being usable. There is still much to do. The following is a selection of the most important or interesting features the framework should consider implementing.

Decouple from Weka

Personally, I find that the framework's future development should first focus on decoupling the framework from the Weka libraries. This modification is likely to entail crucial changes to the framework's architecture and inner components. It is better to solve this issue as soon as possible, not only to allow the integration of other data mining tools, but also to stabilize the framework's inner structure. Third-party contributors should be able to rely on stable interfaces, otherwise they are unlikely to commit to the framework.

Database Backing

Memory consumption may not be an issue at this stage of development, but it will start to cause problems as soon as the framework is released and used for larger data mining projects. Future development should consider an optional backing of the framework's internal models through the use of a local database instance. The idea would be to temporarily persist all larger data objects that are currently out of focus, such as an older SessionRun or an unused DataProvider's cached DataSets.

Exporting Projects

Sooner or later, the framework will have to support the notion of a "data mining project" and support it adequately, providing a means to import, export, share or maybe even version such a project.

Ability to Run outside of Eclipse

Currently, the framework's entire functionality is only accessible via the eclipse platform. At present, the valuable DataProvider implementations and configurations cannot be used in a productive environment, such as a server application. Obviously, a server application cannot support the same user interface as does the eclipse platform. Hence, a framework instance running outside of eclipse would have to be limited to productive use and would only support a very restrictive user interface. The framework should look into the possibility of running on powerful applications servers, without requiring changes to a given project or its underlying configurations.

DataSession Cloning

Say a user would like to run a series of test, which basically use the same data, but with small variations. The user can either produce a SessionRun per test and re-configure the DataSession after each run, or create a DataSession per test. A large set of SessionRuns will use up a lot of memory and (re-)creating the same or equivalent DataSession per test is not only demotivating, but simply a waste of time. The framework should therefore support the ability to clone a DataSession and so encourage experimental behavior, without destroying existing DataSession settings.

Appendix A

Lightweight XML Persistence

This section presents a lightweight XML persistence framework, developed along with the data mining framework to ease the persistence of the framework's state upon shutdown and to generally support the framework's configuration via XML.

The Beauty of using SAX

I personally favor the SAX parser [ERH04] because it is extremely lightweight and only reads a document once. It does not require much memory and is ideal to extract multiple small fragments of information. Furthermore, the SAX parser allows the use of multiple SAX filters, which, if correctly configured, will individually react to certain SAX-events, such as the start or end of an XML element. An obvious drawback to the SAX filter is, that it has no means of accessing the document or larger fragments as a whole. It also does not provide for a tracking of the parsers current location within the document. SAX filter implementations are either configured to work incrementally or have to temporarily record the collected data and process it at the end of the recording.

This is not a real problem when using the lightweight XML framework, as the framework's AbstractSAXFilter implementation supports the recording of XML fragments. A user can start multiple recordings, link them to any object, later stop the recording and access the equivalent XML fragment with the corresponding key-obects. The AbstractSAXFilter also tracks the parser's current location within the document as a namespace-insensitive xpath. A filter can easily verify the parser's current location, by comparing the current xpath-string with what it is expecting. The framework offers yet another helpful feature, combining both preceding features in one. Instead of manually controlling the recording of elements, a user can define the elements to be recorded by passing their xpath values to the AbstractSAXFilter upon instantiation. The filter will then automatically record all elements encountered whose xpath matches one of the defined values. The recordings can then be retrieved after having parsed the document of interest.

The real beauty of SAX is, that a properly implemented extension of the AbstractSAXFilter will be small and independent enough to be declared as an anonymous nested class inside of the object to be configured, having complete access to the object's privately-declared variables.

The following code snippets and example code will demonstrate the beauty and simplicity of this lightweight XML framework. AbstractSAXFilter.java has already been described and requires no further introduction. The DefaultParser is designed to simplify the entire parsing process. Example.java contains an implementation of the AbstractSAXFilter as a nested anonymous class and demonstrates how the DefaultParser is to be used. Example.xml is the XML document to be parsed.

AbstractSAXFilter.java

```
package org.evolizer.weka_plugin.xml;

import java.util.*;
import org.evolizer.weka_plugin.core.AppContext;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.XMLFilterImpl;

public abstract class AbstractSAXFilter extends XMLFilterImpl {

    private static final String             LE         = System.getProperty(
                                                         "line.separator");
    private Stack<String>                   xPath      = null;
    private Hashtable<String, List<String>> recordings = null;
    private Hashtable<Object, StringBuffer> recorders  = null;
    protected AppContext                    appContext = null;

    public AbstractSAXFilter(AppContext appCtx) {
        this.appContext = appCtx;
    }

    public AbstractSAXFilter(AppContext appCtx, List<String> paths) {
        this(appCtx);

        this.recorders = new Hashtable<Object, StringBuffer>();
        this.recordings = new Hashtable<String, List<String>>();

        for (Iterator<String> i = paths.iterator(); i.hasNext();)
            this.recordings.put(i.next(), new ArrayList<String>());
    }

    public void startDocument() throws SAXException {
        this.trackDocumentStart();
        this.recordXML("<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + LE);
        super.startDocument();
    }

    public void endDocument() throws SAXException {
        this.recordXML(LE);
        this.trackDocumentEnd();
        super.endDocument();
    }

    public void startElement(String namespaceURI, String localName,
                             String qualifiedName, Attributes atts)
                             throws SAXException {
        this.trackElementStart(localName);

        this.recordXML("<" + this.encodeXML(qualifiedName));
        if (atts != null)
            for (int i = 0; i < atts.getLength(); i++)
                this.recordXML(" " + this.encodeXML(atts.getQName(i)) + "=\""
                    + this.encodeXML(atts.getValue(i)) + "\"");
        this.recordXML(">");

        super.startElement(namespaceURI, localName, qualifiedName, atts);
    }

    public void endElement(String namespaceURI, String localName,
    String qualifiedName) throws SAXException {
        this.recordXML("</" + this.encodeXML(qualifiedName) + ">");
        this.trackElementEnd();
        super.endElement(namespaceURI, localName, qualifiedName);
    }

    public void characters(char[] text, int start, int length)
    throws SAXException {
        this.recordXML(this.encodeXML(new String(text, start, length)));
        super.characters(text, start, length);
    }

    public void ignorableWhitespace(char[] text, int start, int length)
    throws SAXException {
        this.recordXML(this.encodeXML(new String(text, start, length)));
        super.ignorableWhitespace(text, start, length);
    }

    public void processingInstruction(String target, String data)
    throws SAXException {
        this.recordXML("<?" + this.encodeXML(target) + " "
                + this.encodeXML(data) + "?>" + LE);
        super.processingInstruction(target, data);
    }

    private void trackElementStart(String element) {
        this.xPath.push(element);

        String path = this.getCurrentXPath();
        for (Iterator<String> i = this.recordings.keySet().iterator(); i
                .hasNext();)
            if (path.endsWith(i.next()))
                this.startRecording(path);
    }

    private String trackElementEnd() {
        String path = this.getCurrentXPath();
        for (Iterator<String> i = this.recordings.keySet().iterator(); i
                .hasNext();) {
            String searchPath = i.next();
            if (path.endsWith(searchPath)) {
                this.recordings.get(searchPath).add(this.stopRecording(path));
            }
        }
        return this.xPath.pop();
    }

    private void trackDocumentStart() {
        this.xPath = new Stack<String>();
    }

    private void trackDocumentEnd() {
        this.xPath = null;
    }

    protected String getCurrentXPath() {
        if (this.xPath == null)
            return new String();

        StringBuffer buffer = new StringBuffer("/");
        for (Enumeration e = this.xPath.elements(); e.hasMoreElements();) {
            buffer.append(e.nextElement());
            if (e.hasMoreElements())
                buffer.append("/");
        }
        return buffer.toString();
    }

    private void recordXML(String xml) {
        for (Iterator<StringBuffer> i = this.recorders.values().iterator(); i
                .hasNext();) {
            StringBuffer recorder = i.next();
            if (recorder != null)
                recorder.append(xml);
        }
    }

    protected void startRecording(Object key) {
        this.recorders.put(key, new StringBuffer());
    }

    protected String stopRecording(Object key) {
        StringBuffer recorder = this.recorders.remove(key);
        if (recorder != null)
            return recorder.toString();
        else
            return null;
    }

    private String encodeXML(String xml) {
        xml = xml.replace("&", "&amp;");  // must be encoded before others
        xml = xml.replace(">", "&gt;");
        xml = xml.replace("<", "&lt;");
        xml = xml.replace("'", "&apos;");
        xml = xml.replace("\"", "&quot;");
        // add your own encodings
        return xml;
    }

    public Hashtable<String, List<String>> getRecordings() {
        return this.recordings;
    }
}
```

A can be seen in the following snippet, a user can basically add his or her own encodings by overriding this method and invoking the original method before executing the custom code.

```
@Override
public String encodeXML(String xml){
    xml = super(xml);
    xml = xml.replace("XXX", "&YYY;");
    // xml = CustomEncodings.encodeXML("XXX", "&YYY;");
    return xml;
}
```

DefaultParser.java

```
package org.evolizer.weka_plugin.xml;

import java.io.IOException;
import java.io.StringReader;
import java.util.Properties;

import javax.xml.parsers.*;
import org.evolizer.weka_plugin.core.AppContext;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;

public class DefaultParser implements XMLConstants {

    private static Logger logger = Logger.getLogger(DefaultSetupParser.class);
    private AppContext appContext = null;
    private SAXParser filterParser = null;
    private XMLReader armedParser = null;
    private XMLFilter[] filters = null;

    public static void parse(AppContext appCtx, XMLFilter[] filters, String xml)
        new DefaultParser(appCtx, filters).parse(xml);
    }

    public DefaultParser(AppContext appCtx, XMLFilter[] filters) {
        this.appContext = appCtx;
        this.filters = filters;
        this.initParsers();
    }

    private void initParsers() {
        SAXParserFactory factory = SAXParserFactory.newInstance();
        Properties xmlProps = this.appContext.getXMLProperties();
        try {
            factory.setValidating(false);
            factory.setNamespaceAware(true);
            factory.setFeature(xmlProps.getProperty(SAX_VALIDATION), false);
            factory.setFeature(xmlProps.getProperty(SAX_NAMESPACES), true);
            factory.setFeature(xmlProps.getProperty(SAX_SCHEMAVALIDATION),
                    false);
            factory.setFeature(xmlProps.getProperty(SAX_SCHEMAFULLCHECKING),
                    false);
            factory.setFeature(xmlProps.getProperty(SAX_DYNAMICVALIDATION),
                    false);
            factory.setFeature(xmlProps.getProperty(SAX_NAMESPACEPREFIXES),
                    true);

            this.filterParser = factory.newSAXParser();
        } catch (SAXNotRecognizedException e) {
            logger.error("SAX-Feature not recognized", e);
        } catch (SAXNotSupportedException e) {
            logger.error("SAX-Feature not supported", e);
        } catch (ParserConfigurationException e) {
            logger.error(e, e);
        } catch (SAXException e) {
            logger.error(e, e);
        }
    }

    private void armParser(XMLReader parser, XMLFilter[] filters,
            ContentHandler handler) {
        if (filters.length > 0) {
            filters[0].setParent(parser);
            for (int i = 1; i < filters.length; i++) {
                filters[i].setParent(filters[i - 1]);
            }
            filters[filters.length - 1].setContentHandler(handler);
            this.armedParser = filters[filters.length - 1];
        } else {
            parser.setContentHandler(handler);
            this.armedParser = parser;
        }
    }

    public void parse(String contents) {
        try {
            this.armParser(this.filterParser.getXMLReader(), filters,
                    new DefaultHandler());

            this.armedParser.parse(new InputSource(new StringReader(contents)));
        } catch (SAXException e) {
            // log & handle exception
        } catch (IOException e) {
            // log & handle exception
        }
    }
}
```

Example.java

```
import java.io.InputStream;
import java.util.*;
import org.evolizer.weka_plugin.core.AppContext;
import org.xml.sax.*;

public class Example {

    private AppContext appContext = null;
    private String     xml        = null;

    public static void main(String[] args) {
        new Example().run();
    }

    public Example() {
        this.appContext = AppContext.getAppContext();
        this.xml = this.convertStream2String(
            Example.class.getClassLoader()
                .getResourceAsStream("Example.xml"));
    }

    public void run() {
        List<String> xpaths = new ArrayList<String>();
        xpaths.add("/person");

        AbstractSAXFilter filter = new AbstractSAXFilter(
                this.appContext, xpaths) {
            private Object person = null;

            public void startElement(String uri, String localName,
                    String qName, Attributes atts)
                    throws SAXException {
                String xpath = this.getCurrentXPath() + "/" + localName;

                if (xpath.equals("/group/person"))
```

```
                    this.startRecording(this.person = new Object());

                super.startElement(uri, localName, qName, atts);
            }

            public void endElement(String uri, String localName,
                    String qName) throws SAXException {
                String xpath = this.getCurrentXPath();

                super.endElement(uri, localName, qName);

                if (xpath.equals("/group/person"))
                    System.out.println("Person: ["
                            + this.stopRecording(this.person) + "]");
            }
        };

        System.out.println("\nManual recording results:");
        DefaultSetupParser.parse(this.appContext, new XMLFilter[] { filter },
                this.xml);

        System.out.println("\nAutomatic recording results:");
        Hashtable<String, List<String>> recordings = filter.getRecordings();
        for (Iterator<String> i = recordings.keySet().iterator(); i.hasNext();) {
            String path = i.next();
            List<String> recordings2 = recordings.get(path);
            for (Iterator<String> i2 = recordings2.iterator(); i2.hasNext();)
                System.out.println("Found [" + i2.next() + "] " +
                                   "to xPath[" + path + "]");
        }
    }

    private String convertStream2String(InputStream stream) { ... }
}
```

Example.xml

```
<group>
    <person name="Bart"/>
    <person name="Homer">
        <children>
            <person name="Bart"/>
            <person name="Lisa"/>
            <person name="Maggie"/>
        </children>
    </person>
    <person name="Lisa"/>
    <person name="Maggie"/>
    <person name="Marge">
        <children>
```

```
            <person name="Bart"/>
            <person name="Lisa"/>
            <person name="Maggie"/>
        </children>
    </person>
</group>
```

Console Output

```
Manual recording results:
Person: [<person name="Bart"></person>]
Person: [<person name="Homer">
        <children>
            <person name="Bart"></person>
            <person name="Lisa"></person>
            <person name="Maggie"></person>
        </children>
    </person>]
Person: [<person name="Lisa"></person>]
Person: [<person name="Maggie"></person>]
Person: [<person name="Marge">
        <children>
            <person name="Bart"></person>
            <person name="Lisa"></person>
            <person name="Maggie"></person>
        </children>
    </person>]

Automatic recording results:
Found [<person name="Bart"></person>] to xPath[/person]
Found [<person name="Bart"></person>] to xPath[/person]
Found [<person name="Lisa"></person>] to xPath[/person]
Found [<person name="Maggie"></person>] to xPath[/person]
Found [<person name="Homer">
        <children>
            <person name="Bart"></person>
            <person name="Lisa"></person>
            <person name="Maggie"></person>
        </children>
    </person>] to xPath[/person]
Found [<person name="Lisa"></person>] to xPath[/person]
Found [<person name="Maggie"></person>] to xPath[/person]
Found [<person name="Bart"></person>] to xPath[/person]
Found [<person name="Lisa"></person>] to xPath[/person]
Found [<person name="Maggie"></person>] to xPath[/person]
Found [<person name="Marge">
        <children>
            <person name="Bart"></person>
            <person name="Lisa"></person>
            <person name="Maggie"></person>
        </children>
    </person>] to xPath[/person]
```

Appendix B

Weka Hacks

This section will briefly describe the vital hacks required to integrate the weka libraries and tools into our data mining framework as an eclipse plugin. As with most software today, the weka libraries and tools are subject to evolution and are bound to change. Since we have no control over weka's development, we have to re-implement the various hacks into the newest release. In order to identify the changes and thus ease the transition between versions, I have added the `/* WEKABRIDGE */` comment to every "hack". A full-text search will quickly reveal all changes performed.

Below you will find most of the changes as code snippets with a short description. The missing snippets are related to weka having to load resources as a plugin bundle instead of a standalone application and are variations of the following snippet:

```
WekaClassName.class.getClassLoader().getResource("...");
WekaClassName.class.getClassLoader().getResourceAsStream("...");
```

whereas "`WekaClassName`" will represent a class' name located within the resource's bundle. These resource-loading changes basically ensure that the plugin will load the corresponding resource via a static class' ClassLoader as described in section 5.8 on page 53. A full-text search for "classloader" and "getResource" within the `org.evolizer.weka_plugin.gplweka`-plugin project will reveal the resource-loading code.

org.evolizer.weka_plugin.xml.Explorer

The Explorer class controls the initialization and functionality of the Weka Explorer (described in 4.2 on page 25). The `WekaBridge` invokes the Explorer through this class and thus has no access to it's variables. This hack basically enables the passing of `weka.core.Instances` objects to the explorer without exposing it's variables while permitting the WekaBridge to only access a single Class/Object.

```
...
    /* WEKABRIDGE*/
    public void setInstances(Instances inst) {
        this.m_PreprocessPanel.setInstances(inst);
    }
...
```

weka.gui.explorer.ClassifierPanel

The WekaBridge not only passes `weka.core.Instances` objects to the weka tools, but should also be able to return an object wrapping the learned classifier and any other information required to classify new data (applied filters, attribute information, class attribute, etc.). This snippet inserts a `JMenuItem` into the classifier context menu that, upon triggering, will pass the required information back to the `WekaBridge`. As we have little knowledge on or control over the `Classifier`'s and `Instances`'s life cycle[1], it is safer to pass copies rather than the objects themselves.

The attribute information (header) is copied by constructing a new *empty* `Instances` object with the original `Instances` object as a parameter.[2] Prior to classification, each row is transformed into a `weka.core.Instance` object using this attribute information, thus ensuring the correct attribute count and ordering. Obviously, the classifier object should be cloned as well. Due to time constraints this currently isn't the case, but should be corrected in the following release.

```
...
import org.evolizer.weka_plugin.gplweka.WekaBridgeFactory;
...
    protected void visualize(String name, int x, int y) {
        ...
        /* WEKABRIDGE start */
        JMenuItem sendToBridge = new JMenuItem("Send to WekaBridge");
        if (classifier != null) {
            sendToBridge.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    try {
                        Instances header = new Instances(trainHeader, 0);
                        header.setClassIndex(trainHeader.classIndex());

                        WekaBridgeFactory.getWekaBridge().addClassifier(
                                classifier, header);
                    } catch (Exception e1) {
                        e1.printStackTrace();
                    }
                }
            });
        } else {
            sendToBridge.setEnabled(false);
        }
        resultListMenu.add(sendToBridge);
        resultListMenu.add(loadModel);
        /* WEKABRIDGE end */
        ...
    }
...
```

[1]There is a risk that either object could be modified after being passed to the WekaBridge, but prior to its use to classify new data.

[2]The underlying weka.core.Attribute objects are not copied. It is very unlikely that they will change, as Attribute objects are not reused across Instances objects.

weka.gui.explorer.PreprocessPanel

```
...
import org.evolizer.weka_plugin.gplweka.WekaBridge;
import org.evolizer.weka_plugin.gplweka.WekaBridgeFactory;
...
    public PreprocessPanel() {
        ...
        /* WEKABRIDGE */
        m_WekaBridgeBut.setToolTipText(
                "Load a set of instances from the WekaBridge");
        ...
        /* WEKABRIDGE */
        m_WekaBridgeBut.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                loadInstancesFromWekaBridge();
            }
        });
        ...
        /* WEKABRIDGE */
        buttons.setLayout(new GridLayout(1, 7, 5, 5));
        // buttons.setLayout(new GridLayout(1, 6, 5, 5));
        buttons.add(m_WekaBridgeBut);
    }
...
    /* WEKABRIDGE */
    public void loadInstancesFromWekaBridge() {
        try {
            addUndoPoint();
        } catch (Exception ignored) {
            ignored.printStackTrace();
            m_Log.statusMessage("Problem loading from WekaBridge");
            JOptionPane.showMessageDialog(PreprocessPanel.this,
                    "Couldn't load Instances from the WekaBridge",
                    "Load Instances", JOptionPane.ERROR_MESSAGE);
        }

        try {
            WekaBridge bridge = WekaBridgeFactory.getWekaBridge();
            if (bridge.hasData())
                this.setInstances(bridge.getWekaInstances());
        } catch (Exception e) {
            e.printStackTrace();
            m_Log.statusMessage("Problem loading from WekaBridge");
            JOptionPane.showMessageDialog(PreprocessPanel.this,
                    "Couldn't load Instances from the WekaBridge",
                    "Load Instances", JOptionPane.ERROR_MESSAGE);
        }
    }
...
```

weka.gui.GenericPropertiesCreator

As described in section 5.8 on page 54, I was unable to access the plugin bundle's classpath, thus disabling weka's discovery algorithm. A workaround is the declaration of `Classifier` implementations as an extension. Again, due to time constraints, I simply overrode Weka's discovery algorithm and loading the classes listed in the `weka/gui/GenericObjectEditor.props` file. In future releases, the `org.evolizer.weka_plugin.gplweka` plugin should load `Classifier` implementations from extension declarations.

```
...
    protected void generateOutputProperties() throws Exception {
        m_OutputProperties = new Properties();
        Enumeration keys = m_InputProperties.propertyNames();

        while (keys.hasMoreElements()) {
            String key = keys.nextElement().toString();
           /* StringTokenizer tok = new StringTokenizer(m_InputProperties
                    .getProperty(key), ",");
            String value = "";
            // get classes for all packages
            while (tok.hasMoreTokens()) {
                String pkg = tok.nextToken().trim();

                Vector classes = new Vector();
                try {
                    classes = ClassDiscovery.find(Class.forName(key), pkg);
                } catch (Exception e) {
                    System.out.println("Problem with '" + key + "': " + e);
                }

                for (int i = 0; i < classes.size(); i++) {
                    // skip non-public classes
                    if (!isValidClassname(classes.get(i).toString()))
                        continue;
                    // some classes should not be listed for some keys
                    if (!isValidClassname(key, classes.get(i).toString()))
                        continue;
                    if (!value.equals(""))
                        value += ",";
                    value += classes.get(i).toString();
                }
                // m_OutputProperties.setProperty(key, value);
            }
            * WEKABRIDGE */
            Properties goeProps = new Properties();
            goeProps.load(GenericPropertiesCreator.class.getClassLoader()
                    .getResourceAsStream("weka/gui/GenericObjectEditor.props"));
            if (goeProps.getProperty(key) != null)
                m_OutputProperties.setProperty(key, goeProps.getProperty(key));
            else
                m_OutputProperties.setProperty(key, "");
            }
        }
...
```

Appendix C

DataObjectDefinition Annotations and Source Code

AnnotatedDefinition

```
package org.evolizer.weka_plugin.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.evolizer.weka_plugin.defs.AttributeType;

public class AnnotatedDefinition {

    /* used to define an attribute’s type */
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.METHOD)
    public @interface AttType {
        AttributeType value() default AttributeType.NOMINAL_ATTRIBUTE;
    }

    /* per default the method name is used to name the attribute,
     * unless otherwise specified. this annotation is used to rename
     * the corresponding attribute */
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.METHOD)
    public @interface AttLabel {
        String value();
    }

    /* used to describe an attribute */
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.METHOD)
    public @interface AttDescription {
        String value();
    }

    /* used to declare the String representation of a missing value */
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.METHOD)
    public @interface AttMissingValue {
        String value();
    }

    /* used to declare an enumeration of values to a nominal attribute */
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.METHOD)
    public @interface AttEnumeration {
        String[] value();
    }

    /* used to mark the method(s) responsible for identifying the tuple */
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.METHOD)
    public @interface AttObjectID {}

    /* used to exclude methods that are public but should not belong */
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.METHOD)
    public @interface AttExclude {}
}
```

AttributeType

```
package org.evolizer.weka_plugin.defs;

public enum AttributeType {
    NOMINAL_ATTRIBUTE, NUMERIC_ATTRIBUTE, STRING_ATTRIBUTE;

    public static AttributeType[] getTypes() {
        return new AttributeType[] { NOMINAL_ATTRIBUTE,
                NUMERIC_ATTRIBUTE, STRING_ATTRIBUTE };
    }

    public String toString() {
            switch (this) {
        case NOMINAL_ATTRIBUTE:
            return "Nominal";
        case NUMERIC_ATTRIBUTE:
            return "Numeric";
        case STRING_ATTRIBUTE:
            return "String";
        }
        return "";
    }
}
```

DataObjectDefinition

```
package org.evolizer.weka_plugin.defs;

import java.lang.reflect.Method;
import java.util.*;
import org.evolizer.weka_plugin.annotations.AnnotatedDefinition;

public abstract class DataObjectDefinition extends AnnotatedDefinition {
    ...
    public abstract void loadObject(Object dataObject);
    public abstract Class defines();
    ...
    private void processDeclaredMethods() throws Exception {
        Method[] methods = this.getClass().getDeclaredMethods();
        for (int i = 0; i < methods.length; i++) {

            /* implementation of the abstract loadObject() method */
            if (methods[i].getName().equals("loadObject")
                    && methods[i].getParameterTypes().length == 1
                    && methods[i].getParameterTypes()[0] == Object.class)
                continue;

            /* implementation of the abstract defines() method */
            if (methods[i].getName().equals("defines")
                    && methods[i].getParameterTypes().length == 0)
                continue;

            /* ignore all methods requireing parameters */
            if (methods[i].getParameterTypes().length > 0)
                continue;

            /* check for the ID-method */
            if (methods[i].getAnnotation(AttObjectID.class) != null)
                this.idMethod = methods[i];

            /* ignore excluded Methods */
            if (methods[i].getAnnotation(AttExclude.class) != null)
                continue;

            /* sets the attribute's label */
            String label = null;
            AttLabel attLabelAnn = methods[i].getAnnotation(AttLabel.class);
            if (attLabelAnn != null && !attLabelAnn.value().trim().equals(""))
                label = attLabelAnn.value();
            else
                label = methods[i].getName();
            AttributeDefinition attDef = AttributeDefinitionFactory
                    .getGlobalFactory().getAttDef(label);
            attDef.addDefiningObject(methods[i]);

            this.id2AttDefs.put(label, attDef);
            this.id2Methods.put(label, methods[i]);
            this.methods2AttDefs.put(methods[i], attDef);
            this.attDefs2Methods.put(attDef, methods[i]);
        }
    }
    ...
}
```

References

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[Dou05] Korry Douglas. *PostgreSQL*. Sams Publishing, 2nd edition, August 2005.

[EC06] Dan Rubel Eric Clayberg. *eclilpse, Building Commercial-Quality Plug-ins*. Pearson Education, Inc., 2nd edition, 2006.

[ERH04] W. Scott Means Elliotte Rusty Harold. *XML in a Nutshell*. O'Reilly, 3rd edition, September 2004.

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*, pages 23–32, Amsterdam, The Netherlands, September 2003. IEEE, IEEE Computer Society.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1st edition, 1995.

[GRV03] James Gosling, Mathew Robinson, and Pavel A. Vorobiev. *Swing Second Edition*. Manning Publications Co., 2nd edition, 2003.

[IHW05] Eibe Frank Ian H. Witten. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann Publishers, 2nd edition, 2005.

[KB04] Gaving King and Christian Bauer. *Hibernate in Action: Practical Object/Relational Mapping*. Manning Publications Co., 1st edition, 2004.

[Kna05] Patrick Knab. Mining release history. Master's thesis, University of Zürich, Department of Informatics, 2005.

[KPB06] Patrick Knab, Martin Pinzger, and Abraham Bernstein. Predicting defect densities in source code files with decision tree learners. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 119–125, Shanghai, China, May 2006. ACM Press.

[LRW$^+$97] M M. Lehman, J F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. Metrics and laws of software evolution - the nineties view. In *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*, page 20, Washington, DC, USA, 1997. IEEE Computer Society.