Archie

A Statistics Framework For Elexis

Technical Report

Dennis Schenk
Peter Siska

March 2009

Supervised by
Prof. Dr. Oscar Nierstrasz
David Röthlisberger

University of Bern, Switzerland
Software Composition Group

Abstract

Archie is a statistics framework for the electronic medial records system Elexis. Archie empowers Elexis to generically create anything from simple overviews to complex statistical reports about any data found within the Elexis system. Depending on which plug-ins are installed, an Elexis installation contains data about patient demographics and history, consultations, drug administration, practice management and inventory, finances and accounting, laboratory, etc. Archie provides a platform for Elexis and for all installed plug-ins to easily and rapidly create statistical reports without having to be concerned with recurring aspects such as data input and output, form validation, result presentation, or the user interface in general. Data visualization is handled entirely by Archie, it just requires the raw data to adhere to a defined standard.

Contents

1 Introduction **3**
- 1.1 Problem Statement . . . 3
- 1.2 Goals and Challenges . . . 4
- 1.3 EMR Systems and Elexis . . . 5
- 1.4 Archie . . . 6
- 1.5 Structure of this Document . . . 6

2 Domain **7**
- 2.1 Terminology . . . 8
- 2.2 Status Quo . . . 10
- 2.3 Elexis . . . 12

3 Design and Evolution **13**
- 3.1 Building A Prototype . . . 13
- 3.2 Ideas for Architecture . . . 14
- 3.3 Architectural and Design Decisions . . . 16
- 3.4 Evolution . . . 18
- 3.5 Visualization and Export . . . 20

4 Implementation **21**
- 4.1 Architecture . . . 21
  - 4.1.1 Structure of an Eclipse Plug-in . . . 21
  - 4.1.2 Archie Plug-in Definition . . . 21
  - 4.1.3 Extension Point Definition . . . 22
  - 4.1.4 Initial Extension Schema . . . 23
  - 4.1.5 Improving Usability . . . 23
  - 4.1.6 Extension Point Implementation . . . 25
- 4.2 Data Providers and Datasets . . . 26
  - 4.2.1 Content and Label Providers . . . 29
  - 4.2.2 Parameterization through Annotations . . . 29
  - 4.2.3 Provider Implementations . . . 32
- 4.3 Controllers . . . 32
  - 4.3.1 “New Statistics” Action . . . 34
  - 4.3.2 Additional Actions . . . 36

4.4 Managing Classes . . . 36
4.5 Factories . . . 37
4.6 Charts . . . 39
4.6.1 Pie Charts . . . 39
4.6.2 Bar and Line Charts . . . 40
4.6.3 The Chart Model . . . 40
4.7 Helper Classes . . . 42
4.8 User Interface . . . 43
4.9 Elexis User Interface Contribution . . . 46
4.10 Dashboard Charts . . . 47
4.11 Limitations . . . 49
4.11.1 Internal . . . 49
4.11.2 External . . . 50

5 Team Organization **52**

6 Requirements Engineering and Validation **54**

7 Conclusions **56**
7.1 Lessons Learned . . . 56
7.2 Future Work . . . 57

A Licensing of Archie **59**

B User Manual **60**
B.1 Quick Start . . . 60
B.2 Usage . . . 61
B.2.1 Dashboard View . . . 61
B.2.2 The Sidebar View . . . 63
B.2.3 The Output View . . . 63

C Developer Manual **68**
C.1 Extending AbstractDataProvider . . . 68
C.1.1 Constructor . . . 69
C.1.2 getDescription . . . 69
C.1.3 createHeadings . . . 69
C.1.4 createContent . . . 70
C.2 Adding Additional Parameters . . . 71
C.3 Registering with the Archie Extension Point . . . 73
C.4 Where to Get Additional Help . . . 74

List of Figures **75**

Bibliography **77**

Chapter 1

Introduction

1.1 Problem Statement

Elexis is an open source Electronic Medical Records system (detailed information about Elexis and its history is provided in Chapter 2). It has grown into a complex software solution with about 850 classes, around 100k lines of code, and many features. There are aspects that are yet not very mature though. Despite a large and broad functionality and some crude and not very accessible statistical functions, Elexis does not include the possibility for users to generate system-wide and customizable statistical reports about patients, drugs, consultations, or any similar aspect of recorded data. Important analyses about state and development of the practice using Elexis are missing. There is, for example, no easy way for doctors to see how many consultations they have given over a certain time frame or how this correlates with the practice's income and spending.

What Elexis misses is the ability to inform its users about the state a practice is in and also to visualize this data. It should help in making predictions about future developments and to tell what services were provided. Elexis needs the ability to inspect flow of money and inventory, provide key facts about patients in the system. Also it needs to be able to compare different aspects of such data.

This lack of basic overview and data correlation motivates the idea to develop a statistics framework for Elexis — a platform that helps to gather, compose and also visualize overviews of any data in the system rapidly and easily.

Another aspect requiring improvement is usability and screen design. We also contributed in this area to the project (see Section 4.9)

1.2 Goals and Challenges

Goals. First of all, we want to create a framework for developers allowing them to *quickly and easily assemble their own statistics*. Programmers still have to write Java code, but besides getting the data of interest out of the system, the underlying framework takes care of the rest: handling user interaction and preparation of data and its presentation.

Second, we want to give users the possibility to quickly generate and export textual and graphical reports about any data in Elexis. An example of a user requirement is to answer the question "How much money have I earned over the last six months and what were my expenses over that time period". The generated reports will provide answers to such questions.

Third, we want to create a framework and accompanying statistics that are widely used by both Elexis developers and its users. To accomplish this we work as close as possible with both sides. We gather requirements, feature requests, ideas and also critique from the people who are using Elexis in practices.

Besides these three goals, the framework has to be intuitive, easy-to-use and understand, and come with a nice looking user interface.

Challenges. One challenge to achieve these proposed goals is the structure of the Elexis application. It has grown into a large system with a lot of interdependent classes and methods. The source code is often hard to read and understand because of bad formatting and naming, mixing of German and English vocabulary, just a few JavaDoc and other comments, etc. For these reasons, it is difficult to find ones way through the program code and to find the right starting points.

Another challenge is to find a solution to enable plug-ins that use the framework to be generically parametrizable. This means a plug-in can have an unknown number of attributes — with unknown types and names — and the framework must be able to present these attributes as editable parameters in the correct form to the user. It also must know how to get the information from the user interface back to the plug-in.

Each plug-in needs to be able to programmatically define validation procedures for its parameters without having to concern itself with the user interface.

Moreover, the user interface has to be able to show all available statistics to the user and help him in finding what he is looking for.

1.3 EMR Systems and Elexis

In the following paragraphs we shortly explain the context in which Elexis and thus also Archie is situated.

Most physicians still use paper and file based systems to manage their patients. Adoption of Electronic Medical Record Systems (EMR) is slow. Switzerland is still one of the least advanced countries when it comes to distribution of EMR. There are various reasons for the slow adoption at which we take a closer look in Chapter 2. Nevertheless, the number of deployed EMR systems is slowly rising, and there are various efforts being made to speed up the process. Most notable is Switzerland's ambitious "eHealth" strategy[1] which wants to break ground for a wide deployment of EMR: its goals are to define standards, looking at financial feasibility, law foundations, basic education and online service possibilities. It also wants to start pilot projects and in general speed up deployment of EMR dramatically on a pretty tight schedule.

There is not much diversity in currently available EMR systems in Switzerland: there are a few and most of these software solutions available are either expensive or require the physicians to have their own large information system which most physicians don't consider an option as they don't have the necessary knowledge about deployment and/or maintenance. Also the costs that such systems impose are high

Elexis is an alternative to proprietary EMR solutions: it is based on the open source Eclipse Platform[2] and can be used with various open source database systems, such as MySQL[3] or PostgreSQL[4].

[1] http://www.ehealth.admin.ch
[2] http://www.eclipse.org
[3] http://www.mysql.com
[4] http://www.postgresql.org

1.4 Archie

At the beginning we were searching for a name for our project. We settled on Archie — after Professor Archie Cochrane (1908-1988) [Coch89]. He was a British general practitioner and originator of evidence-based medicine, which is every form of medical treatment where patient oriented decisions are made specifically on grounds of proven effectiveness which in turn is given through the means of statistical data.

1.5 Structure of this Document

In Chapter 2 we give a short overview of the problem domain, clarify terms and definitions and report about the status quo of EMR. In Chapter 3 we describe how we were going on about the tasks at hand, how we designed the architecture of our plug-in, what technical challenges we faced, and how we resolved them. In Chapter 4 we present our final implementation and explain the main technical solutions as well as the architecture. Chapter 5 shortly describes how we were organized as a team and how we split up work. In Chapter 6 we present how we communicated with the users and developers of Elexis, and how we incorporated their feedback into our work. Chapter 7 provides information on what we learned from the project and delivers a short outlook on future activities. In Appendix A we explain under which license Archie will be released and why. In Appendix B and Appendix C we provide manuals for Archie users and for developers, respectively, on how to write new statistics on top of the framework.

Chapter 2

Domain

In this chapter we take a closer look at the domain Elexis and Archie belong to: the domain of electronical medical records. Besides the technological aspects there are numerous other aspects, mostly political, that we find are important and worth a short analysis.

As mentioned in the introduction most physicians in Switzerland still use paper-based patient records. EMR adoption is slow but gaining ground. In the following we study the disadvantages of paper-based systems and the advantages of EMR .

With paper-based systems patient information and health records are distributed over many locations and are usually not readily available. This leads to multiple records about the same aspect of patient health being kept at different facilities, like hospitals, doctors practices etc. Redundant medical tests have to be made for which results are probably already stored somewhere else. Another factor is that patients most of the times can't provide full medical history about themselves. Most people don't remember or don't know all their medical details. If they think they do they maybe remember them wrong. All this can lead to loss of time and money through inefficiency, unnecessary administrative costs, false diagnosis, and errors in treatment.

Patient records should be available quickly and from all locations, particularly for chronic patients or in case of emergencies. This leads to less spending of valuable time on tests and clarifications about a patients condition and medical history.

Systems which meet such requirements lead to better treatments, less errors, instantly transferable records, complete information, better reactivity, and less administrative costs.

The numbers of doctors with own practices using EMR vary between 8% and 10% in Switzerland. In the mentioned e-Health Strategy of Switzerland, it is discussed, that by the year 2010 about 50% of doctors with own practice should deploy EMR systems and in 2015 every citizen should have his own patient record [Gesu07]. How this goal should be accomplished is not drawn out clearly yet, but the example of Australia shows that it is feasible: There the percentage of doctors using EMR ascended from 8% in 2000 to over 50% in 2006 [Bhen06].

About 90% of practices in the Netherlands, Great Britain and Scandinavia use EMR. The quality of these systems varies strongly though. Some nations advance implementation of EMR through subsidies and/or regulations. The USA are currently in a similar situation as Switzerland, with a distribution of EMR of about 15% [Bhen06].

2.1 Terminology

If one starts to engage himself with the domain of EMR he faces confusion and finds no general consensus on key aspects. There are many abbreviations for different concepts in the world of electronic patient records and many have disputed, conflicting or overlapping definitions. Some of these problems occur when businesses try to sell their products to customers and their marketing departments create new buzzwords to stress uniqueness. Another reason is that standardization efforts in the field are still in an early phase and sometimes uncoordinated, which is reflected in the jungle of different words with different meanings. In the following paragraphs we try to flesh out some of the core concepts we already used so far to make them clearer.

An Electronical Medical Record (EMR) is a repository of demographic and health data of a patient. It offers clinical decision support, controlled medical vocabulary, computerized provider order entry, pharmacy, and clinical documentation applications. It supports the patients electronic medical record across inpatient and outpatient environments, and is used by healthcare practitioners to document, monitor, and manage health care delivery within a care delivery organization (CDO). The data in the EMR is the legal

record of what happened to the patient during his encounter at the CDO and is owned by the CDO. A Schematic concept of a sample EMR can be seen in Figure 2.1.

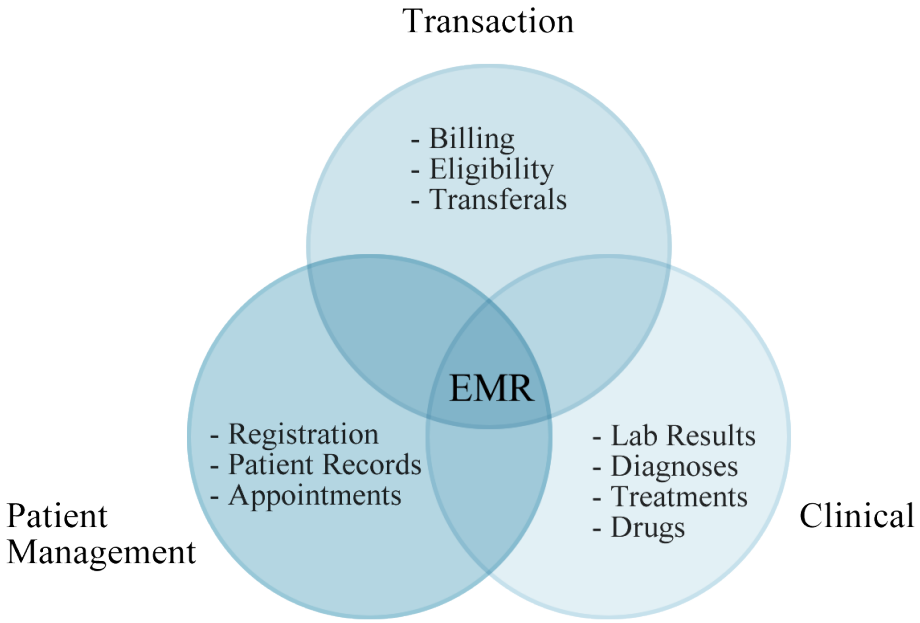


Figure 2.1: Schematic concept of a sample EMR.

An Electronic Health Record (EHR) is a longitudinal record of a patients health over time and can be seen as a subset of each care delivery organizations EMR. It provides clinician and also consumer access to clinical details captured from one or more encounters, from different facilities. It mostly spans over a full lifetime from birth to death. An EHR itself requires the presence of EMRs, because the EMRs are the actual data providers for EHRs, in the form of EMR summaries. EHR data is owned by the patient. EHR contains data from episodes of care across multiple CDOs within a community, region, or in some instances the entire country.

A Clinical Information System (CIS) is a clinical repository of patient data. CIS is a form of EMR but it is more specialized to the environment of clinics. Sometimes the term is used interchangeably with EMR. A CIS typically covers: pathology and radiology order entry and results reporting,

medication prescription and administration, clinical work lists, decision support, etc.

Figure 2.2 shows a sample relation between these concepts. Different EMRs respectively CIS' are plugged into an EHR which accompanies a patient over his whole life.

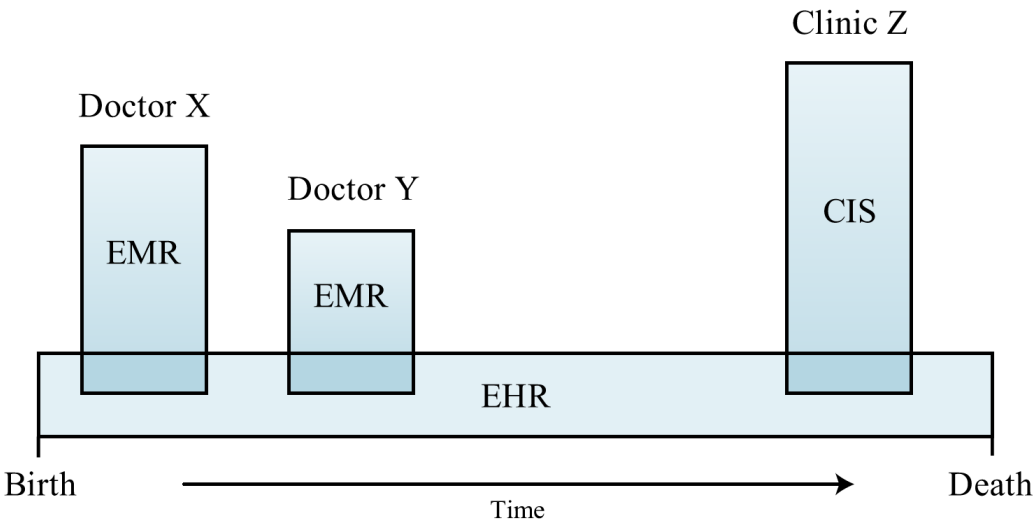


Figure 2.2: EMR, EHR, CIS in sample relation.

2.2 Status Quo

The domain of electronic medical records is still pretty young and because of this standards, one of the most important aspects of any such domain, are still underdeveloped. There are only a few accepted and implemented standards. Also internationally and nationally acknowledged institutions, managing and asserting such standards are sparse. There are, however, some standards which are already well established, like HL7 (Health Level 7)[1], a group of standards concerning the exchange of data between medical institutions and their information systems. However, there are also competing standards such as IHE (Integrating the Healthcare Enterprise)[2], a European initiative with similar goals. There are numerous other examples of conflicting standards.

[1] http://www.hl7.org
[2] http://www.ihe.net

However, these issues are being worked on and awareness for the issues is growing. Solutions are actively worked on internationally and nationally. In Switzerland, as noted above, the government started its "eHealth" initiative of which one major subproject is to find and specify standards.

The current lack of standards, many times also the non-compliance to available standards and the resulting lack of interoperability between existing software systems, are some of the reasons for the slow Adoption of EMR and other health information technology.

In spite of studies showing revenue gains after implementation of EMR, the healthcare industry spends much less of its budget on IT than other information intensive industries. Aside from the immaturity of standards there are several industry specific issues responsible for this.

One reason is the difficulty to incorporate older records of patient data into the new systems. The process is time consuming and expensive and should be done following precise standards to ensure that all information concerning the patient is available in the new system. Information on a patient may exist in any number of formats, sizes, media types and qualities, which further complicates accurate conversion. If a practice has used paper-based patient management for many years or even decades, the costs of a switch to EMR are very high.

Another major issue is privacy. Individual electronic records have to be managed confidentially and secured from unauthorized access. Because of crosslinking of digital media over networks such as the Internet, multiple access points are present, facilitating interception of patient data. Skepticism about the security of electronic patient records is high, on both the patients and the medical institutions side.

Another problem is preservation of patient data and the mandatory ability of patients to be able to have access to all their medical data at all time. Data has to be kept for long periods of time without data degradation. This too creates new challenges for electronic systems and their acceptance.

2.3 Elexis

Elexis was first conceived in 2006 by Gerry Weirich[3], a general practitioner and internist. Before he decided to start from scratch with an open source Eclipse-based project, he was involved in several other small EMR projects. Elexis was first developed by Weirich alone and was deployed in his own practice. Requirements and feature requests were created out of his own needs. Soon other people started to get aware of Elexis and more physicians became interested in the project.

One aspect which sets Elexis apart from other EMR systems is that it is mainly developed by actual practitioners, not by software engineers. Also requirements, bug fixing request etc. come directly from facilities where it is in productional use. This gives Elexis credibility and closeness with and to its users.

Elexis got noticed in medical journalism and was sponsored to be presented at some conferences. Medical equipment manufacturers started sponsoring development of equipment specific data importers.

Today Elexis is deployed in around 50 practices [Weir09b]. It offers medical record functionality, inventory management, billing functionality and debtor control. There are many plug-ins available which offer functionalities ranging from laboratory equipment data import to several other import and export possibilities, from financial extensions to general EMR extensions. It provides interfaces to numerous medical portals and supports standards like HL7, SGAM eXchange, and XML Invoice 4.0.

From January 2008 to January 2009 Elexis was at the center of argoLEAD[4]: a project which was in its core a field test with numerous involved physicians to answer questions like: how high are the costs of a switch from paper-based patient management to an EMR, where are the main difficulties, how are the changes managed by facility employees and how practical is Elexis as a solution.

The project was a success and was evaluated. As a result Argomed[5], an organisation of medical practitioners with around 1000 members, decided to contribute to a company that wants to assist with the distribution of Elexis and provide support to the current user base.

[3] http://www.weirich.ch
[4] http://www.rgw.ch/elexis/dox/argoLEAD.pdf
[5] http://www.argomed.ch

Chapter 3

Design and Evolution

We want to create a piece of software that will have an active userbase after releasing it. Based on the points mentioned in the previous chapter, EMR systems will most likely become more and more popular which is why we decided to create an application that is part of such an EMR system. The following sections describe our initial idea for this project and its evolution, as well as some important design desicions.

3.1 Building A Prototype

Before we started working on Archie, in order to improve our knowledge of the fairly extensive Eclipse RCP framework, we decided to create a simple prototype of a basic Eclipse application — *Sanclipse*. The functionality of Sanclipse as seen in Figure 3.1 is very limited. Its purpose is to simulate a rudimentary EMR system using the Eclipse RCP framework.

The main window is divided into two parts — the left hand side containing a list of fictive users or patients in the system, the right hand side displaying details of selected users such as both names and the gender. A user of the system can add new patients or remove existing ones from the list by using actions residing in the toolbar on the top.

Although the functions of Sanclipse are rather scarce, we learned about important methods and core concepts of the Eclipse API. Not only through the entire prototyping process, but also by developing Sanclipse according to the tutorial of creating a chat application called *Hyperbola* as described throughout the book [McAf07]. This process was crucial for making some of

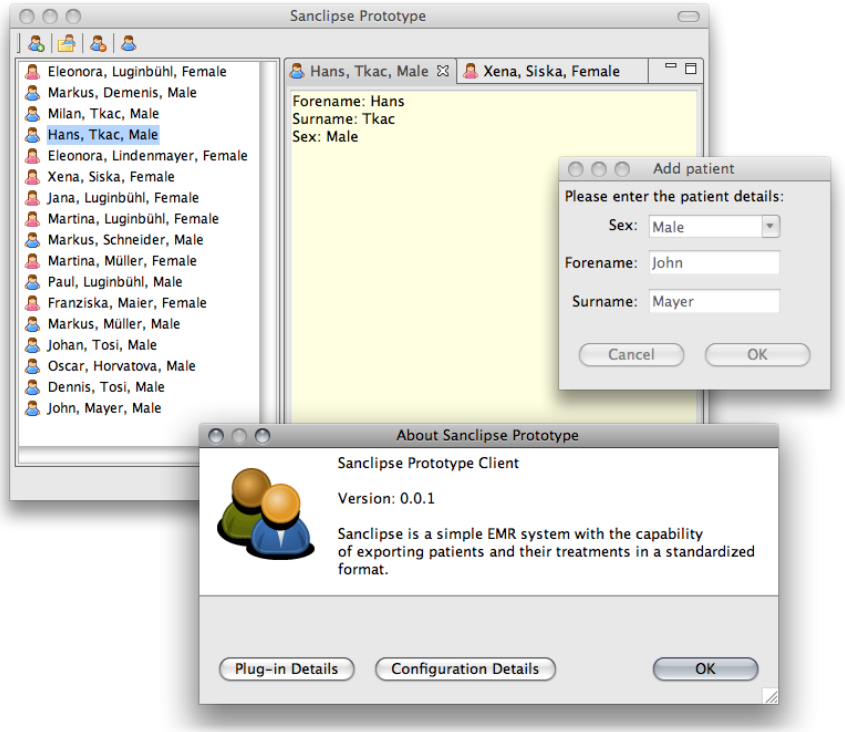


Figure 3.1: Sanclipse Prototype Overview

the architectural decisions described in the following sections and developing Archie.

3.2 Ideas for Architecture

The Sanclipse prototype described in the previous section is a stand-alone Eclipse RCP application. It is packaged and deployed on its own thus not depending on any other Eclipse program or framework. Because Elexis itself already is a stand-alone Eclipse RCP application, we hope to find the appropriate method of packaging and deploying Archie as part of Elexis. This is one important criteria, another one is that users should be able to easily install or uninstall Archie with all its functionality.

Fortunately, every Eclipse application is modular, composed out of small functional parts into one application. Everything in Eclipse is a plug-in [Gall02]. The term *plug-in* in Eclipse refers to the unit of modularity. Except

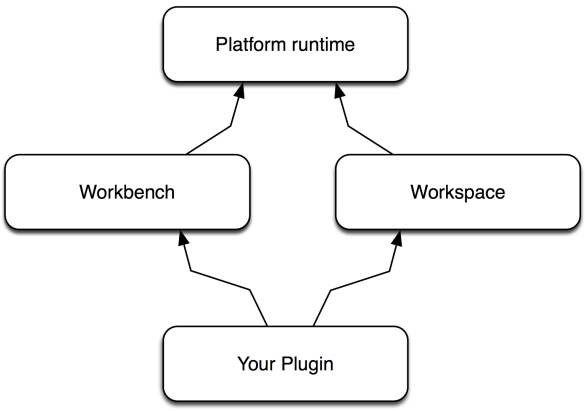


Figure 3.2: Eclipse Plug-in Architecture

for two indispensable core plug-ins, the *Workspace* and the *Workbench*, an Eclipse based application can be completely stripped down and built up from the ground by using custom plug-ins that extend the core functionality.

Plug-ins in Eclipse provide extension points that can be used by other plug-ins to add or change functionality of the providing plug-ins. The *Workspace* plug-in allows other plug-ins to extend the Eclipse user interface, to contribute menu items or entire menus, and to create new views and add additional buttons to the toolbars. The Eclipse framework of course provides plug-ins that already extend the core but can also further be extended by other plug-ins. An illustration showing the Eclipse plug-in architecture is depicted in Figure 3.2.

Elexis is an Eclipse RCP based application, which means it extends Eclipse by implementing the extension points defined in the framework, but also offers custom defined extension and provides interfaces for other plug-ins to extend Elexis itself.

Figure 3.3 illustrates the architectural situation with regard to Elexis and the Eclipse framework. Elexis consists of a main plug-in encapsulating core functionality that enables it to run. The main plug-in also defines the extension points for other plug-ins, contains definitions about how Elexis is to be deployed on different operating system and holds all the image resources used throughout the UI. Moreover, the main plug-in provides *ant* build scripts for automated building as well as *Latex* and *JavaDoc* documentation of Elexis for both users and developers.

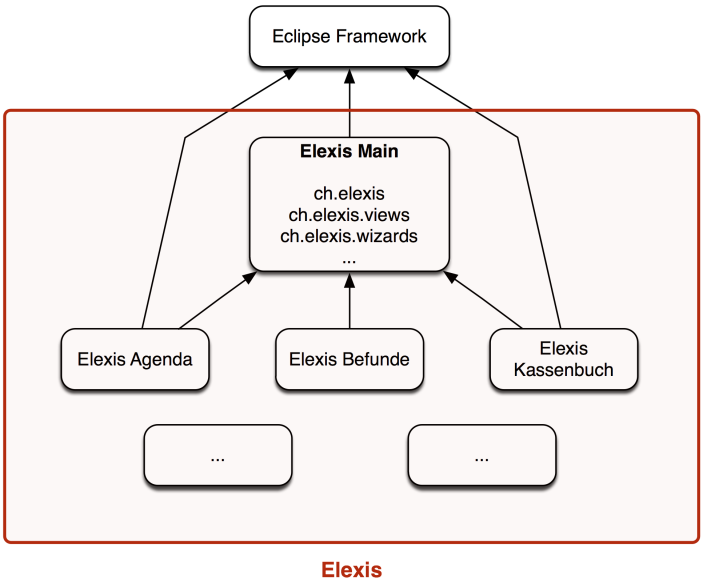


Figure 3.3: Elexis Architecture

The extending plug-ins are dependent on the main plug-in and cannot be run without it, only extending or adding new functions to the main plug-in.

Based on the architecture of Eclipse and Elexis, one of our main goals, providing an application that is easily deployable into existing Elexis installations, can be easily accomplished by creating a plug-in of our own.

3.3 Architectural and Design Decisions

We decided that Archie is an Eclipse plug-in and provides some sort of an extension point, allowing other developers to easily hook into Archie and provide statistical data about Elexis. The next step is to define an application design that allows us to meet the goals set in Section 1.2:

1. Create a framework for developers allowing them to quickly and easily assemble their own statistics.
2. Give users the possibility to quickly generate and export textual and graphical reports about any data in Elexis.
3. Create accompanying statistics about important Elexis data.

First, we define an abstract class AbstractDataSource that provides a first entry-point for other plug-ins to extend and write their own statistics. This AbstractDataSource extends the Elexis class BackgroundJob which is a sub-class of Eclipse API class Job.

In Eclipse *Jobs* are tasks that can be scheduled and executed. During their execution the API provides methods for monitoring, locking resources, or cancelling currently running jobs. Eclipse Jobs also provide user feedback during execution [Vale04]. These properties are perfectly suited for our data providers. Data providers are gathering statistical data about Elexis. During this process a visual feedback about the current progress is shown to the user. This is the the same concept Elexis already uses to retrieve data from the database – most parts of Elexis itself where data retrieval takes place, particularly if the retrieval may take some time, use a BackgroundJob and therefore the Eclipse Job API.

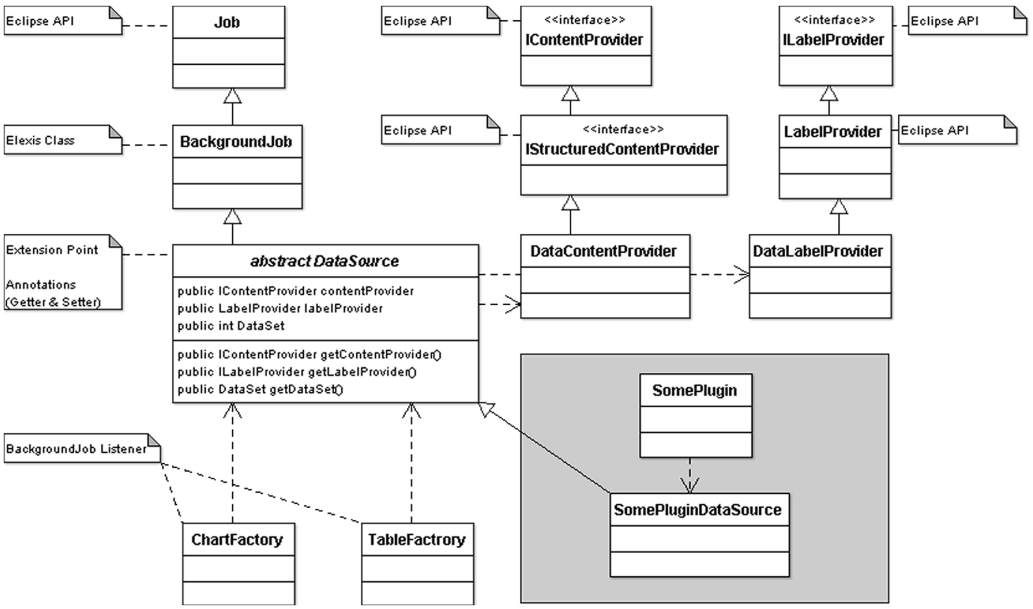


Figure 3.4: First, Conceptual UML Draft

Furthermore, we need two controllers for creating tables and charts. Tables are used in the user interface (UI) to display the gathered data to the user. Charts are used to build visualizations of data already displayed in the UI. For this, we define two controller classes — a TableFactory for generating tables and a ChartFactory for creating pie, line, or bar charts based on the data in AbstractDataSource.

Tables in Eclipse can be wrapped in a TableViewer object. Content of a table in Eclipse can only be built using strings, whereas viewers can have a content and label provider. These providers define whether and how elements in table rows are being displayed. Tables wrapped in a TableViewer can thus be populated with any type of objects the content provider of a viewer understands.

In order to allow our TableFactory to generate results and create tables from any kind of data, an AbstractDataSource has a DataContentProvider and a DataLabelProvider associated with it. Both providers are subclasses of provider objects from the Eclipse API. This also allows different data sources to declare their own providers and control how the data is being displayed to the user. Figure 3.4 shows these architectural and design decisions in more detail.

In addition to these model based decisions, we decided to create a dual-pane user interface for Archie. This means to have a settings pane on the right hand side where users can chose what statistics they want to create and adjust their parameters and show the results from these statistics in a pane on the left side.

During the development of Archie the architectural and design decisions described in previous section were constantly refined. The main idea of having some sort of central data provider, modelled as an Eclipse *Job*, remains. Also the integration into the Eclipse API via *Content-* and *LabelProvider* is still the same in the final version. Some implementation details are missing in the initial design in Figure 3.4, especially questions such as *"How exactly do the data providers store raw data from the database for further processing?"*, *"How does do the chart and table factories work?"*, and *"How is the UI to model interaction handled?"*. The following section described how the initial design evolved during the implementation process.

3.4 Evolution

At first we create a *DataSet* class which holds any data in a matrix-like form and simplifies conversion to textual, table-, and chart based output. As we progress, the fact that the dataset can contain any data leads to problems in the UI when trying to sort results tables. Because of this, we continue refactoring Datasets by adding another constraint. Datasets should only be able to contain objects that implement the *Comparable* interface. This means

that for some classes that do not implement said interface, developers have to create a wrapper class. We did this for example for the Elexis class *Person* with the *PersonWrapper* class in the samples project, to enable alphabetical ordering. See Section 4.2 for more details on the final implementation of *DataSet*.

One of the bigger refinements is the introduction of actions as user input and controlling mechanism. Read more about our final implementation in Section 4.3. Following this introduction the concepts of *Table- and ChartFactory* lost their initial meaning of controlling entities. The concepts of these two classes are converted to less dominant roles, namely as data creation factories described in Section 4.5.

Most refactoring efforts and changes happen in the UI part though. The reason for this is probably that we didn't have a fleshed out concept for the UI and made it up as the project progressed. One of the most important aspects is the introduction of annotated DataProvider methods. Using *Java* annotations we are able to programm the UI generically as described in Section 4.2.2. This means that we have an UI that sets itself up with the right input fields, just by looking at how the corresponding methods are annotated.

Another important aspect in the UI is the creation of generic input field classes for specific input types, which are able to validate themselves and provide general user assistance as described in Section 4.8. Another aspect of the UI we had not thought about in detail was how the charts were being created from statistic results. First we wanted to create chart generation somehow generically. However, we noticed very quickly that this went beyond the scope of our project and decided to create a chart creation wizard. With such a wizard a user can choose what parts of the results he wants to use to generate a chart. Moreover, the chart wizard determines the type and look of the generated chart. This is explained in more detail in Section 3.5 with implementation details in Section 4.6.

Another thing that changed from our initial design was that AbstractDataProvider inherits directly from the abstract class Job of the Eclipse API instead of the Elexis BackgroundJob class. In Eclipse 3.3 new Job API classes were introducted which maked the Elexis wrapper class BackgroundJob obsolete.

3.5 Visualization and Export

At the beginning of the project we decided that we wanted to provide some sort of graphical output for statistic results. Since creating such functionality from scratch was not in the scope of our project we searched for open source solutions and found JFreeChart[1]. JFreeChart is a chart building library licenced under the GNU Lesser General Public Licence (LGPL)[2].

We quickly discovered that the best way to use JFreeChart within our framework was, as previously mentioned, to provide some sort of wizard where a user can specify what exactly he wants to plot, and how. To create an completely generic solution where charts are generated automatically based on the result sets is, although interesting to accomplish, beyond the scope of our project. This is why we decided to limit chart drawing functionalities to three basic but commonly used chart types: Pie, Bar and Line Charts.

A user should nevertheless be able to create more elaborate charts with statistical results — thus why we decided that our framework also needed to have export functionalities, namely to Comma Separated Value (CSV) files, as they can be read by most spreadsheet software solutions such as OpenOffice Calc[3] or Microsoft Excel[4]. With such software users can refine the presentation of statistical results and further processing the data.

[1] http://www.jfree.org/jfreechart/
[2] http://www.gnu.org/licenses/lgpl.html
[3] http://www.openoffice.org/product/calc.html
[4] http://office.microsoft.com/excel

Chapter 4

Implementation

4.1 Architecture

4.1.1 Structure of an Eclipse Plug-in

As described in Section 3.2 we decided to create Archie as an extension of Eclipse and Elexis. A plug-in, the basic building block of Eclipse, contains a collection of files and a manifest file MANIFEST.MF describing the relation of the plug-in to other programs, packages, and libraries. The collection of files can include program source code, read-only content such as images, translation message files used for internationalization, documentation or others.

In addition to the manifest file, plug-ins also have a plugin.xml file. In former Eclipse versions, particularly 3.0 and older, the plugin.xml used to contain execution-related information but since Eclipse version 3.1, this has been moved to the MANIFEST.MF file. The plugin.xml however still contains important information, namely the definitions of extension points the plug-in either uses to extend other parts of an application, and/or extension points it defines for other plug-ins to extend [McAf07].

4.1.2 Archie Plug-in Definition

Archie consists of two parts — a main plug-in and a plug-in fragment. A plug-in fragment is considered as an optional part of a plug-in. The main difference between a plug-in and a plug-in fragment is that a fragment needs

to define a Fragment−Host in their manifest file thus belongs to a plug-in without which it cannot exist. Moreover, the extension point definitions are stored in a fragment.xml file instead of plugin.xml[Arth08].

Archie defines the following plug-in structure in Eclipse:

ch.unibe.iam.scg.archie. This is the main plug-in, containing the entire object model, views and controllers, definitions and implementations of Eclipse extension points, and all read-only data such as images and documentation.

ch.unibe.iam.scg.archie.samples. The *samples* plug-in fragment contains sample statistics implementations to show how the main plug-in and its framework can be used.

4.1.3 Extension Point Definition

The core of the entire Archie framework structure is the extension point defined in the main plug-in. This extension point allows other, 3rd party plug-ins or plug-in fragments to interact with Archie. Without other plug-ins implementing this extension point, Archie only has a very limited use.

Listing 4.1: Extension Point Definition in plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension-point
      id="ch.unibe.iam.scg.archie.dataprovider"
      name="Archie Data Provider"
      schema="schema/ch.unibe.iam.scg.archie.dataprovider.exsd"
   />
</plugin>
```

An extension point in Eclipse is a definition of a plug-in interface and describes the configuration point using an XML schema. Plug-ins can publish these extension points and thus specify what an implementation needs to fulfill in order to be usable by the plug-in.

Archie only defines one extension point. The definition contains an unique identifier ch.unibe.iam.scg.archie.dataprovider, a human readable name, and a reference to a extension point definition schema as shown in Listing 4.1. As

indicated by its name, this extension point allows other plug-ins to contribute their (statistical) data to Archie.

4.1.4 Initial Extension Schema

The other part needed for an extension point in Eclipse to work, is an extension schema as referenced by the extension point definition in plugin.xml from the previous Section 4.1.3. Basically, an extension schema contains an XML formed definition of elements and attributes that extensions to that extension point (implementors) must declare. Furthermore, the extension schema can contain implementation examples, API descriptions, and additional documentation or licensing information.

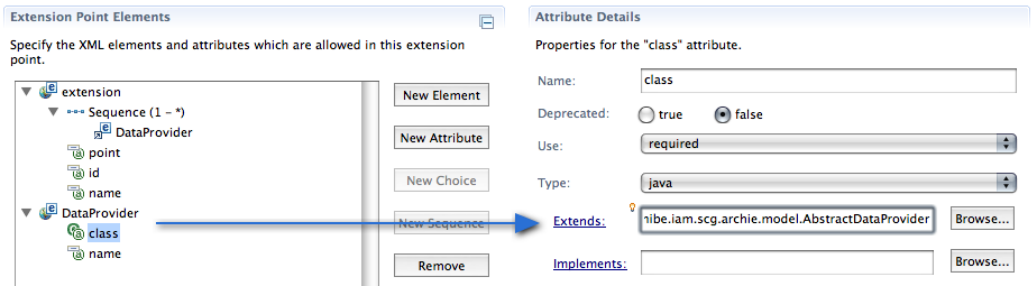


Figure 4.1: Extension Schema Definition Screenshot

Initially, our extension schema defines that an extension point can contain one or more elements of the type DataProvider whereas each DataProvider element needs to have a name and a class specified. Furthermore, the class definition in an element provides a constraint that asserts classes used in those definitions are subclasses of AbstractDataProvider as illustrated in Figure 4.1. Although the extension point schema definitions are written in XML, the use of the graphical editor is encouraged [Laff06] that is why the file syntax won't be explained further.

4.1.5 Improving Usability

The initial extension schema implementation does not allow for grouping of data provider names. The name of each provider is listed in the user interface in a combo box. Our samples plug-in consists of seven different plug-ins, each with a different name. Towards the end of the project, Gerry Weirich, lead

developer of Elexis, contributed his own set of data providers in a custom plug-in for Archie. The list of data providers as presented to the user got confusing and messy which eventually lead to a lower usability.

Although we had an autocompletion mechanism defined and in place to facilitate the searching of available providers in the list, we decided to add the possibility to group certain data providers that belong to the same plug-in or to the same family of tasks. The simplest and in terms of usability yet effective solution is to prefix every data provider name in the list with a defined term.

There are two possibilities for the definition of that prefix. One of which is by using a term based on the fairly cryptic ID of each plug-in contributing to the data provider extension point, such as ch.unibe.iam.scg.archie.samples. The first part of the plug-in ID would be obsolete in the UI, so we need to manipulate the IDs to only retrieve the latest portion of it in order to get a nice human readable prefix, such as *Samples*. The other variant implemented in the end is to give developers the possibility to define an optional prefix in a handy way by using the same extension schema as for their data providers.

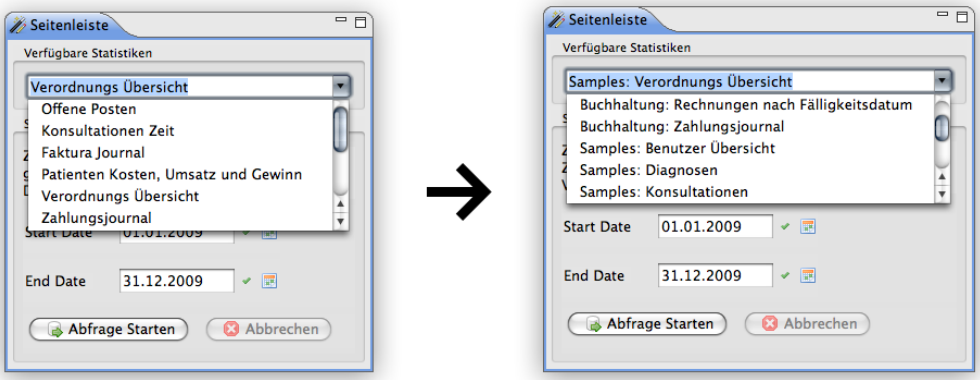


Figure 4.2: List of Providers Before and After the Extension Point Changes

We extend the initial extension schema by adding a new, optional element called category. A category consists two attributes: a unique identifier id and a name. Every ch.unibe.iam.scg.archie.dataprovider extension point implementation can now define one or more categories and assign them to the DataProvider elements in the definition. The name of a category is then rendered as a prefix to each data provider in the list. As an example shown

in Figure 4.2: all our data providers in the samples plug-in are in the *Samples* category so that they show up as *Samples: Some Provider Name* in the list. Grouping of other core Eclipse plug-ins such as views or perspectives is defined equally.

Another advantage of this approach is that defined categories are available across plug-ins allowing developers to create provider "families" and group together providers across different plug-ins (only if plug-ins that need to share categories are both included in Eclipse run configurations or application distributions).

4.1.6 Extension Point Implementation

Any plug-in can now implement the defined extension point. We do this in our Eclipse fragment ch.unibe.iam.scg.archie.samples for every class that provides statistical data to the main plug-in.

Listing 4.2: User Overview Extension Point Implementation

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<fragment>
  <extension point="ch.unibe.iam.scg.archie.dataprovider">
    <DataProvider
        class="ch.unibe.iam.scg.archie.samples.UserOverview"
        name="System User Overview">
    </DataProvider>
    <category
          id="ch.unibe.iam.scg.archie.dataprovider.ProviderCategory"
          name="Samples"
    </category>
</fragment>
```

A simple example of the user statistics is shown in Listing 4.2, where UserOverview needs to be a subclass of AbstractDataProvider described in the following sections. The definition can optionally include a category element as described in Section 4.1.5.

4.2 Data Providers and Datasets

The heart of Archie is the abstract class AbstractDataProvider. Plug-ins that hook into Archie have to extend this class in order to provide the data they collect. Furthermore an AbstractDataProvider extends the Eclipse API class Job, contains a Dataset object as well as label- and content-providers for Eclipse viewers whose details are described in Section 4.2.1

Name	Consultations	Money
John Mayer	142	2'192
Hans Friedrich	291	1'239
Perry White	91	99'221
Barbara Grail	130	129

Figure 4.3: An Example Dataset

Datasets are tables modeled as Java objects. A dataset consists of a list of headings, the top most row of a table, and a content being the rest of the rows in that table. In addition to this, a dataset implements the Iterable< Comparable<?>[]> and Cloneable interfaces which, on the one hand allows us to retrieve an Iterator for a datasets content, on the other hand to create duplicates of datasets for further processing. The headings are modeled as a list of strings (where each element is a heading of one column in the table). The content is a list of Comparable<?>[] arrays. Datasets provide methods for accessing and updating headings, entire rows and/or individual cells. An example of a dataset is depicted in Figure 4.3

The dataset class was called AbstractDataSource in our initial design described in Section 3.3. We renamed it during the project. Moreover, when we first modeled the datasets, their content was composed out of Object arrays. However we later discovered that this was not ideal when used in combination with sorting Eclipse tables that were composed out of datasets — this is why we decided that every object in a dataset needs to implement Comparable interface.

Listing 4.3: Abstract Data Provider Methods

```
public abstract class AbstractDataProvider extends Job {

    /**
     * Returns the description for this data provider.
     * @return Returns the description for this data provider.
     */
    public abstract String getDescription();

    /**
     * Creates headings for each column in the dataset object of this provider.
     * @return A list of strings (List<String>) containing the headings.
     */
    protected abstract List<String> createHeadings();

    /**
     * This method should do all the work necessary to populate the dataset's
     * content. It's called in the job's execute method after some
     * initializations have been done.
     *
     * @return The status of the current job.
     * @see org.eclipse.core.runtime.IStatus
     */
    protected abstract IStatus createContent(IProgressMonitor monitor);
}
```

In addition to access methods for providers and dataset, the AbstractDataProvider class contracts its subclasses to implement the abstractly defined methods shown in Listing 4.3.

Because a data provider extends an Eclipse job, it can be scheduled for execution. Upon the start of a provider its inherited run(IProgressMonitor monitor) method then populates the datasets headings with the result from the createHeadings() method and triggers the createContent(IProgressMonitor monitor) method where subclasses need to initialize the datasets content and return a status of the execution signalizing its outcome. The monitor passed to the creation method allows for propagation of the progress of a running job to the UI. However, it's the responsibility of each subclass to implement the monitoring of a data provider properly.

An UML overview of the discussed classes can be seen in Figure 4.4.

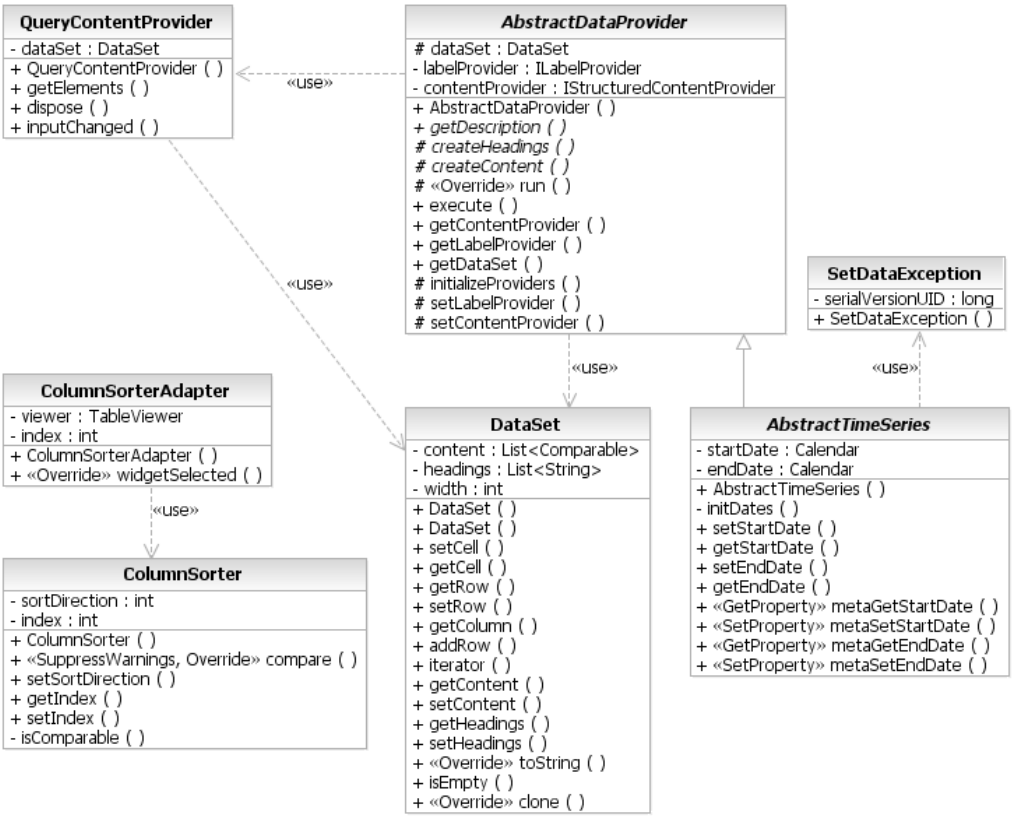


Figure 4.4: AbstractDataProvider class and surroundings

4.2.1 Content and Label Providers

Our initial definition of the entire application design already contained label and content providers as described in Section 3.3, what merely has changed during the development process are the names of those providers — namely from DataContentProvider to QueryContentProvider for content providers and from DataLabelProvider to QueryLabelProvider for label providers.

These providers are the connection between the data in the model and its representation in the user interface as they control how the data from a data provider's DataSet is being rendered in the table in the result view.

Content providers need to implement the IContentProvider interface from the Eclipse API, whereas label providers need to implement one of the appropriate interfaces from the Eclipse API, such as ILabelProvider, ITableLabelProvider , or CellLabelProvider. In our implementation, the default content provider in our data providers is a IStructuredContentProvider, whereas the label provider is a simple implementation of an ITableLabelProvider.

Our content provider merely returns the stored dataset's content as an array in the interface method getElements(Object inputElement), ignoring the *inputElement* passed to this function. The label provider uses the interface method getColumnText(Object element, int columnIndex) to return the corresponding cell in the dataset.

In contrast to the content provider which ignores the *inputElement* parameter, the label provider takes the *element* parameter as a row in the dataset and returns the string representation of the element at the given *columnIndex*. Label providers can also provide images for their elements. Our default label provider implementation does not deliver any images for dataset elements, this is why the method getColumnImage returns null.

Subclasses of AbstractDataProvider can easily override these default provider implementations by setting their own content and label providers and thus controlling how the data they return will look like when presented to the user.

4.2.2 Parameterization through Annotations

An important question that came up during the refinement of our initial design was how parameters for data providers whose implementation we not necessarily know as Archie can and should be extended by anyone, could

be set in our framework. We solved this particular problem by annotating *getter* and *setter* methods for data provider parameters with custom annotations.

Particularly, Archie provides two kinds of *method annotations* to use in data providers: GetProperty for annotating getter methods and SetProperty for annotating setters. Each get- and set-property pair is associated by a *name*, and they both have an *index* variable that can be set. *Index* controls the order in which the getters and setters are called by the controller or UI.

Listing 4.4: GetProperty Annotation Definition

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface GetProperty {

    /**
     * Property name. There has to be a setter method annotated with a
     * SetProperty method with the same value to have any effect on the data
     * provider. This name will also be the title of the corresponding widget
     * shown in the UI.
     */
    public String name();

    /**
     * A brief description about the getter method.
     * This is used for toolips of the UI elements.
     */
    public String description() default "";

    /**
     * Property index. Defines the order in which this property is
     * displayed and processed.
     */
    public int index() default -1;

    /** Widget type. What kind of widget? */
    public WidgetTypes widgetType() default WidgetTypes.TEXT;

    /**
     * A regular expression pattern string to be performed as validations on the
     * given method.
     */
    public String validationRegex() default "";
```

```
        /**
         * A validation error message that will be displayed upon unsuccessful
         * validation of the input for the method containing this annotation.
         */
        public String validationMessage() default "";
}
```

In addition to the *index* and *name* variables, the GetProperty annotation is composed by couple of options. Each of these options is described in more detail in Listing 4.4.

Not only can the annotations for getter methods contain a *description* used in the UI to provide a more detailed feedback about a provider parameter, but getter annotations also take care of a proper validation expression and message as well as the widget type with which they should be presented to the user.

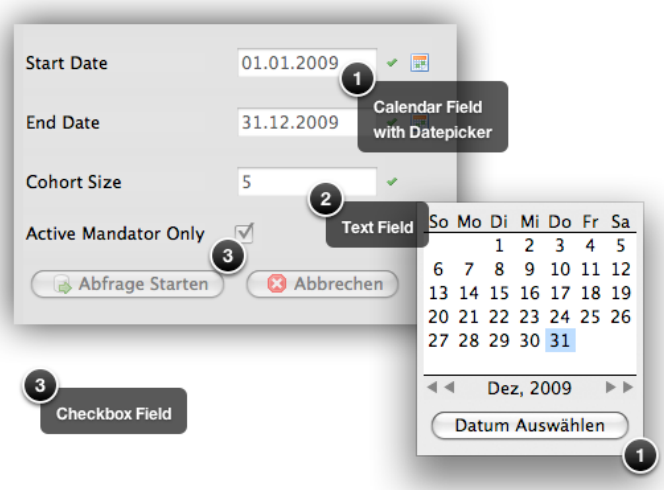


Figure 4.5: Three different UI widgets.

Archie provides four different widget types that can be used together with annotations: a simple text widget, a numeric widget, a date widget, and a checkbox. A screenshot showing all widgets is depicted on Figure 4.5. Each of these UI widgets provides default validation and quick fix methods. A date widget for example is rendered together with a date picker pop-up for users to be able to chose a date quickly and easily. Moreover, additional val-

idations using regular expression patterns[1] can be set by each data provider implementation using annotations.

4.2.3 Provider Implementations

Implementing a custom data provider is a two step process. First the implementing plug-in needs to hook into the data provider extension point as described in Section 4.1.3. As stated in the extension point definition, a plug-in must have at least one, but can have as many as wanted, implementation of the AbstractDataProvider class.

The second step consists of defining an implementation of a data provider. This is where the plug-ins do the statistical computations and wrap them in the appropriate model classes in order for Archie to use it. A more detailed description on how to implement your own data providers can be found in Section C.1.

4.3 Controllers

Most of the controller part of Archie is implemented by using *actions*. The term action in Eclipse refers to a visible element that allows users to initiate a unit of work [McAf07]. Given this definition, actions in Eclipse are tightly integrated into the UI.

Eclipse defines several different extension points for several kinds of actions — meaning that actions can be contributed to the UI either by definition, using extension points, but also programmatically using code. In our case the actions always belongs to a view, a part of the user interface, thus are being defined programmatically.

Archie defines a couple of actions, some of them for starting a new statistical query, opening the chart wizard, starting the chart creation in the dashboard or exporting results of a data provider. An overview of these classes can be seen in Figure 4.6. We describe them further in the following sections.

[1] http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html

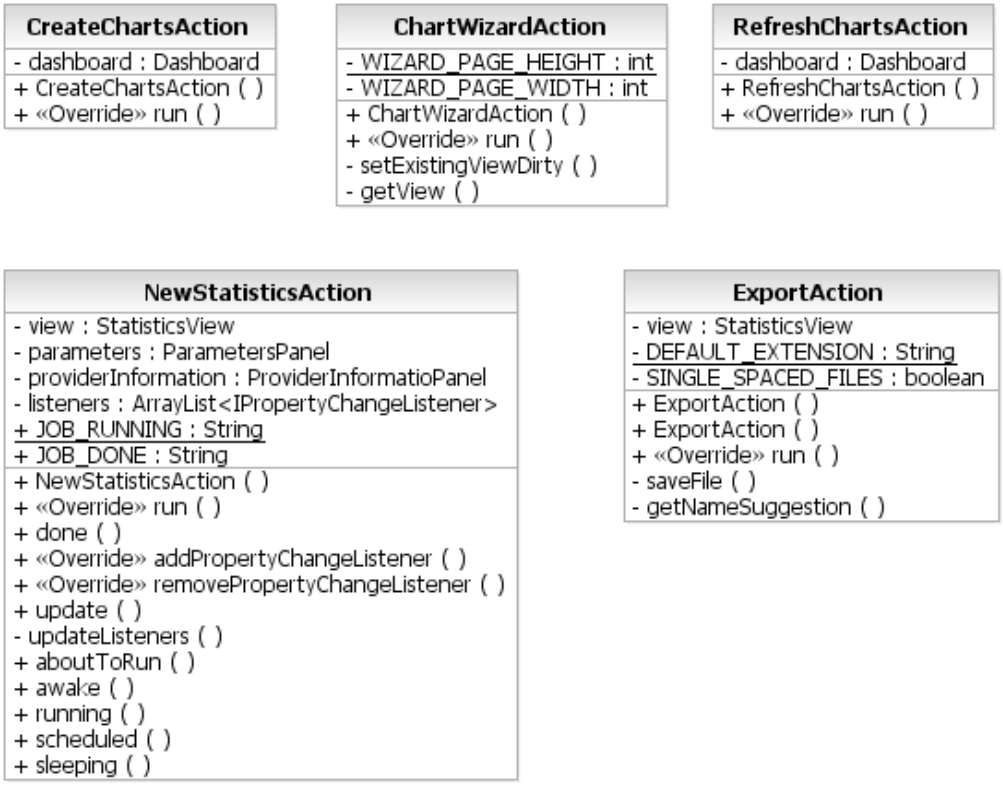


Figure 4.6: UML overview of the action classes

4.3.1 "New Statistics" Action

This action is the main controller, defined in the class NewStatisticsAction, which is why it is probably the most important one. It not only controls *when* a provider is scheduled and tells it to start gathering the data, but also acts as a mediator between the result view (main view) and the sidebar view. The sidebar view contains custom widgets necessary for setting parameters for a currently selected provider. Widgets are described in Section 4.8 in more detail.

Each action object needs to extend the Eclipse API class Action. The heart of each action is the run() method. This method is called when the user triggers the action from the UI — in our case by clicking the *Start Query* button in the sidebar. The run method in NewStatisticsAction performs a series of actions related to the currently selected data provider. First, it takes care of checking the validity of provider parameters and sets them on the provider if the parameters are valid. Then the action registers itself as a job listener on the data provider and updates and populates the main view with information about the currently selected provider and its parameters. Finally the "new statistics" action tells the data provider to run.

In this action, we use the *Observer Pattern* as well as the IJobChangeListener interface of the Eclipse API. On the one hand, the action observes the ProviderManager which propagates an update event every time the currently managed provider changes. By default the "new statistics" action is disabled. As soon as the provider manager has set a valid provider, the action enables itself and users can start statistical queries. Additionally if at any time during the runtime of the application the provider changes and/or is not valid anymore, the action disables itself.

Thanks to the IJobChangeListener interface, the action can react on different states of a job — which in our case is the data provider. We namely use the done(final IJobChangeEvent event) method which is called after a job has terminated (successfully or unsuccessfully). The action populates the result view with results from the data provider. This can either be a table containing the data from a dataset of a provider or, an appropriate message for the user, if the query did not return any results.

The sequence diagram shown in Figure 4.7 demonstrates how the action interacts with the data provider as well as important controller objects and result panel [Bell04].

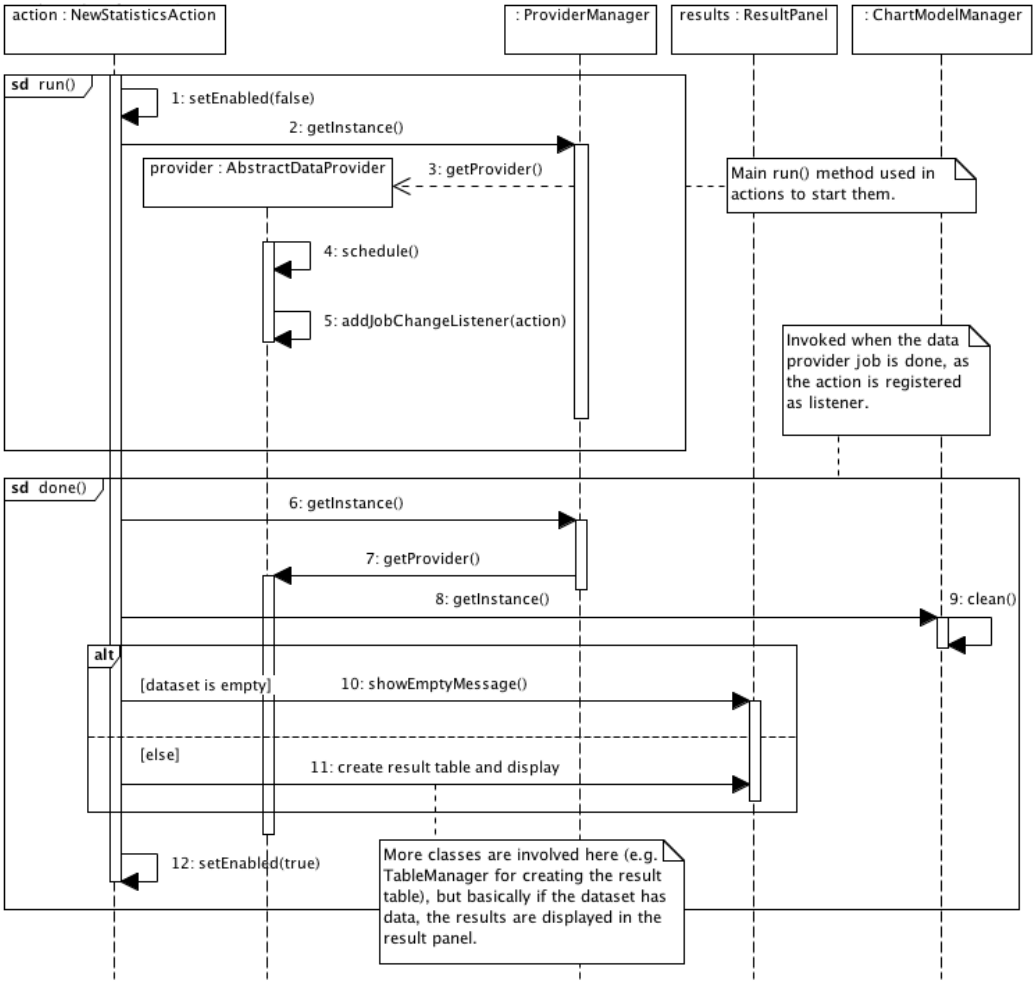


Figure 4.7: Data Provider – Controller Interaction

4.3.2 Additional Actions

As previously mentioned in Section 4.3, there are a few additional actions we use in Archie other than the "new statistics" action. The following list shows and describes these actions used as controllers in Archie.

ChartWizardAction. This action is defined in the main view where the results of a statistics are shown. It opens a new chart creation wizard where users can build graphical charts based on the data in the results table.

ExportAction. This action triggers an export dialog where the contents of the currently shown results table can be exported into a Comma Separated Values (CSV) file.

CreateChartsAction. This action is used in the dashboard view to start the dashboard chart generation. We implemented this action so that the chart creation could be triggered by the user instead of being started automatically as soon as the view is displayed. This is particularly important for users with slower computers where the creation of these charts can take a while.

RefreshChartsAction. This action refreshes the already created charts which is mostly needed when a user has been working some time since he generated the dashboard charts and needs to refresh them with updated data.

4.4 Managing Classes

During the application runtime, there are situations when we want to have only one instance of a certain object in order to either reuse it or its attributes, or to be able to access it from different parts of the program while ensuring that the instance we access is the right one for all accessors. This is why we implemented three *manager* classes. An UML representation of all manager classes can be seen in Figure 4.8.

Every manager class is an implementation of the *Singleton Pattern* and stores exactly one instance of a specific type of object. By providing accessor methods for the stored instance, it can be accessed by different classes (mostly controllers) at different times while ensuring that the instance of the object

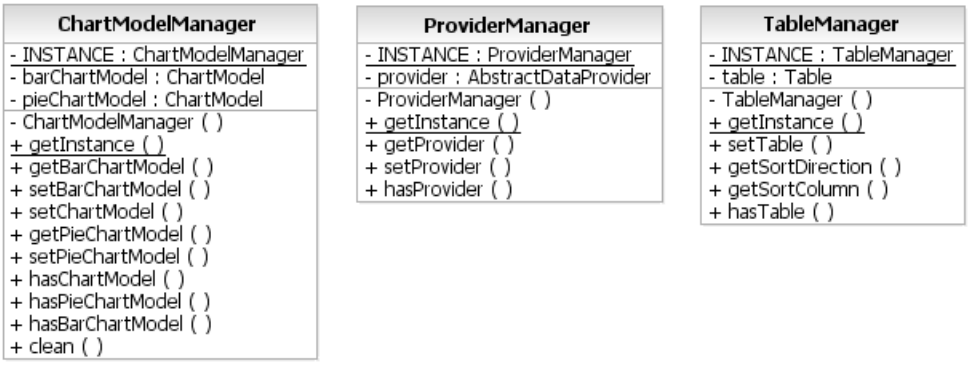


Figure 4.8: UML overview of the manager classes

remains the same. We differentiate between the following three manager objects in Archie:

ProviderManager. This class manages the currently active data provider instance. When a user selects and runs a particular statistic from the list, the data provider for that statistic is set in the ProviderManager. It is reused in some controller classes such as in NewStatisticsAction , in ProviderInformationPanel, a user interface class, or in the chart generation wizard.

TableManager. This is a manager class for the Eclipse SWT table displayed as a result in the main view. We use it to store the current table instance in order to be able to reuse its sorting attributes in the chart creation wizard.

ChartModelManager. This class manages the last used chart model. We implemented it in order to properly encapsulate and reuse the model and its attributes when a new chart wizard is started. This way a user does not have to (but of course can) enter all chart parameters again when creating new charts, especially when only a few parameters need to be changed.

4.5 Factories

There are two different situations where we have to create certain types of objects based on a specific object or its properties. Particularly in Archie,

we create either tables or charts. In order to group these repetitive tasks we created two factory classes as can be seen in Figure 4.9.

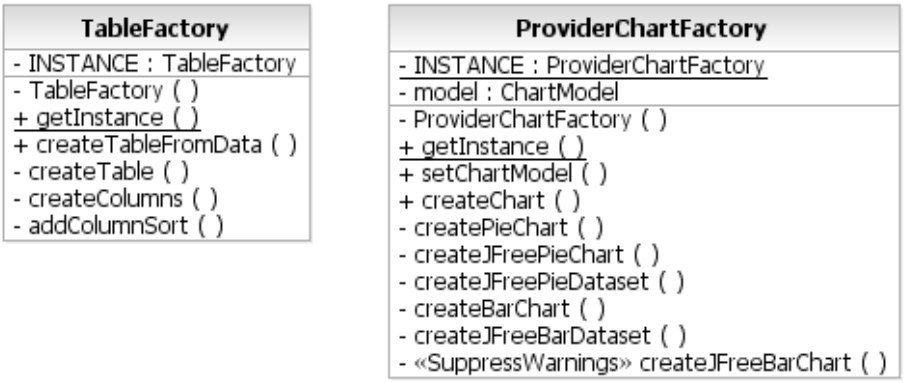


Figure 4.9: UML overview of the factory classes

TableFactory. Creates an Eclipse JFace TableViewer containing a Table object based on the data from a given dataset. The factory method createTableFromData(Composite parent, DataSet dataset, ILabelProvider labelProvider, IContentProvider contentProvider) also takes two providers (label and content) as parameters, as well as the Composite object the table gets displayed in.

Context menus and other enhancements can only be added to viewer- (contain tables) rather than table-objects, which is why we decided to return the TableViewer rather than the Table object during the development of the project as opposed to our initial implementation.

ProviderChartFactory. This class creates ChartComposites based on a given ChartModel object as described in Section 4.6.3. This also means taking care of mapping the definition of our dataset into a particular definition of a JFreeChart dataset as described in Section 4.6.

Both factories are implemented as *Singleton* although they could also be implemented as classes containing only static methods thus not have an instance.

4.6 Charts

We want to give users a means to evaluate the result of a data provider not only by consulting the result table in the UI but also by providing a convenient means to generate graphical charts. In order to achieve this in a general way, one that is in line with displaying data not known upfront, we created a *chart model* and a *chart wizard*.

The data displayed in the results table can be exported into a graphical chart by using the chart wizard in the result view. There are three chart types that can be created using this wizard — *pie*-, *bar*- and *line*-charts.

All of our charts are rendered using the free and feature-rich Java chart library *JFreeChart* described in Section 3.5. JFreeChart provides a well documented API and supports a wide range of chart types out of which we only use the three types mentioned before.

A chart object in JFreeChart is based on a dataset (this is a different dataset object than ours, but the term *dataset* in JFreeChart refers to something very similar — a collection of data, keys and values). This chart object can be used to create a ChartComposite, which is a subclass of an ordinary SWT Composite and thus can be used and displayed in other composites and containers in the UI.

For each chart type, we need to create a factory method which maps the values in our dataset into an particular dataset definition in JFreeChart in order to be used to create a chart.

4.6.1 Pie Charts

Pie charts in JFreeChart are based on datasets that implement the PieDataset interface. JFreeChart provides a default implementation of this interface in a class called DefaultPieDataset. This dataset is a mapping of *keys* to *values*. As each row in the JFreeChart DefaultPieDataset is a row in our dataset definition, we only had to create the mapping between these two datasets by defining the *keys* and *values*, which in our case corresponds to picking one column from our dataset for each — a keys column and a values column.

4.6.2 Bar and Line Charts

The implementation of bar and line charts in JFreeChart is slightly different to that of pie charts as the underlying dataset needs to implement a different interface — namely the CategoryDataset interface. A default implementation of this interface is provided in JFreeChart by the class DefaultCategoryDataset . The difference between a bar and a pie chart lies in the possibility to plot more than one category (column) in a bar chart.

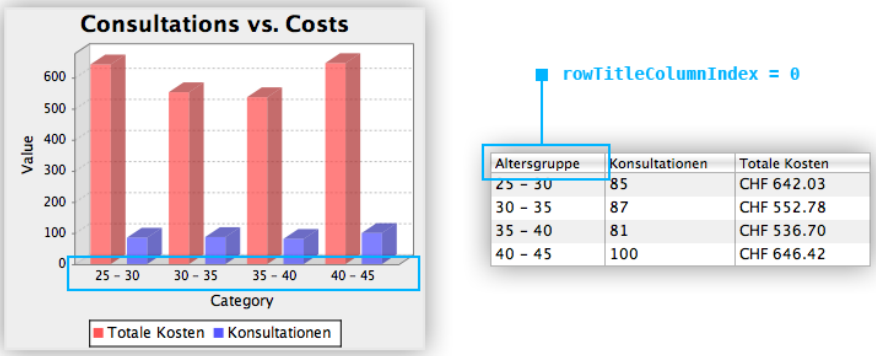


Figure 4.10: Dataset Mapping in Bar Charts

We created the mapping by defining a variable rowTitleColumnIndex that holds the index of the column in our dataset which provides the title for each row and each category (column) in the category dataset, as well as, similarly to the pie chart mapping, an index of all rows and columns (categories) that should be plotted. The row titles based on the rowTitleColumnIndex variable are the labels on the x-axis in a bar chart. Line charts in JFreeChart are based on the same dataset as bar charts, which is why the same definitions and mappings apply for line charts. An example of the dataset mapping in bar charts is shown in Figure 4.10.

4.6.3 The Chart Model

The underlying object in a chart wizard is the chart model. The chart model is a simple class that contains definitions and settings about how a chart based on this model should be rendered, what kind of chart it is and what data from a dataset the chart contains.

Listing 4.5: Chart Model Object Variables

```
public class ChartModel {

    /**
     * Constant for pie chart types, 1.
     */
    public static final int CHART_PIE = 1;

    /**
     * Constant for bar chart types, 2. Bar charts can also be handled as line
     * charts as they both are created from a category dataset. There's a switch
     * in the bar chart type that can be activated for line charts.
     */
    public static final int CHART_BAR = 2;

    /**
     * This switch can be activated for bar charts which makes them render as
     * line charts.
     */
    private boolean isLineChart;

    private String chartName;
    private DataSet dataSet;

    private int[] rows;
    private int[] columns;

    private int keysIndex;
    private int valuesIndex;
    private int categoryColumnIndex; // used in bar & line charts
    private int chartType;

    private boolean isThreeDimensional;
}
```

Listing 4.5 shows the object variables in a chart model. These variables are set in the chart wizard steps, the chart model method isValid() returns whether a chart model is valid and can be used for chart creation. Each chart model has a chartType which is either ChartModel.CHART_PIE or ChartModel .CHART_BAR. Although there are three types of charts, we only define two chart types and add the variable isLineChart for that case, as a line chart is a special case of a bar chart both depending on the same dataset type (JFreeChart dataset as described in Section 4.6.2).

In addition to this, a chart model also specifies which columns and rows from a dataset, also stored in the model, will be used for categories, keys and/or values in the chart's dataset. JFreeChart also provides methods for creating three dimensional charts. However, the underlying datasets are independent of this setting which is why we only specify a boolean isThreeDimensional for switching to a 3D chart.

4.7 Helper Classes

Several places in our program source code make use of small and almost identical pieces of code. Because of this we define certain utility or helper classes. These classes help us to keep the code clean and structured and allow us to reuse common, shared functionalities and methods in different places. All helper classes contain only static methods. We also expose the helper classes in Archie to other plug-ins by defining entries for each helper class in the plugin.xml file.

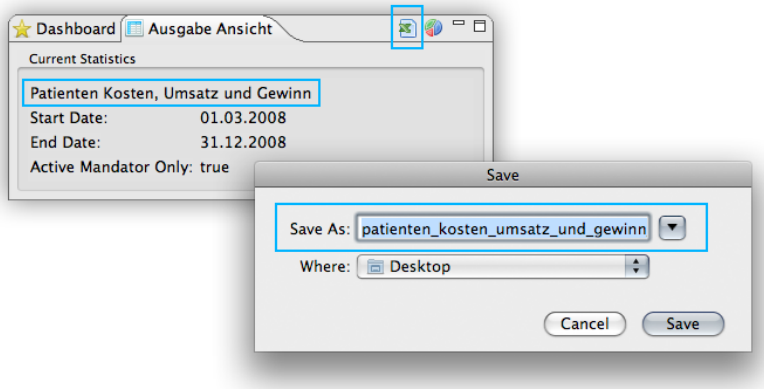


Figure 4.11: Valid Filename Proposal During CSV Export

One example application of such a helper class can be found in the CSV export functionality. In the popup dialog presented to the user where he can type in the filename used to save the data, a filename proposal is already present. An illustration of this is shown in Figure 4.11. This filename is a cleaned-up string composed by the name of the data provider and the current date, containing only valid filename characters. Moreover, all whitespace characters are replaced by underscores. This string is generated by the

StringHelper class. In addition to this StringHelper class, the following list shows all other helper classes available in Archie.

ArrayUtils. Provides convenience methods for printing out as well as checking the existence of certain elements in arrays.

DatabaseHelper. This class is used in the dashboard overview composite in the UI and provides methods for easy access to certain data in the database.

DatasetHelper. Provides methods to sort datasets, to type check dataset columns, and to retrieve column indexes based on the column title.

ProviderHelper. This class offers methods to retrieve all annotated methods (and thus parameters) for a data provider as well as to retrieve and set values for these methods. Furthermore, it takes care of sorting and storing these methods in a map for further processing.

SWTUtils. Offers convenience methods for user interface classes to easily create commonly used Java Standard Widget Toolkit (SWT) elements such as labels and separators.

4.8 User Interface

The entire user interface is built on top of the Standard Widget Toolkit (SWT) used throughout the Eclipse API. Although most of the components used are standard SWT containers and widgets, in order to provide a nice user feedback upon parameter setting in the parameters panel, we created custom *widgets* that handle the user input. An example can be seen in Figure 4.12.

The superclass for every such widget is the AbstractWidget class, an abstract class extending an SWT composite which makes it usable in other SWT components throughout the UI. Each AbstractWidget consists of a *label* and a *control* (controls such as simple text fields, combo boxes or similar are parts of SWT). Concrete implementations of abstract widgets provide methods for setting and retrieving the value of a widget as well as (in certain implementing classes) validation and quick fix methods.

These widget types as well as the validation parameters can be set in the data providers (implementing plug-ins) by using annotations and regex patterns, Annotations are described in Section 4.2.2, regex patterns in Section C.2.

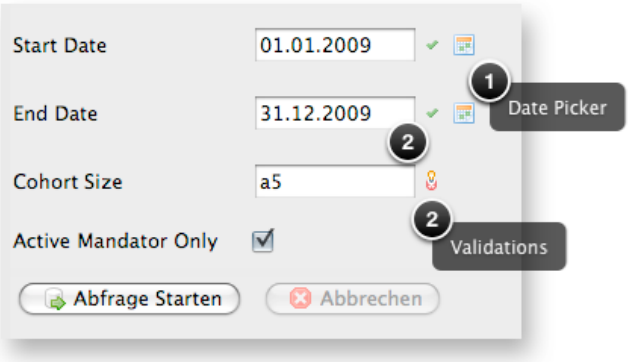


Figure 4.12: Widgets in the UI

This gives each plug-in total control over how the corresponding UI widgets to set parameters get rendered and validated.

We provide implementations for most common widget types. A list of available widgets is shown below, a more detailed view of the entire class hierarchy is depicted in Figure 4.13.

TextWidget. Is an implementation of an abstract widget containing a simple text input field. In contrast to the abstract widget, a TextWidget contains a SmartField object, a wrapper class for a widget's control that provides label decorations for signalising the validity of a widget's value to the user.

NumericTextWidget. Is a subclass of the TextWidget providing an implementation of text widgets which may only contain numeric characters. A numeric text widget also has a custom implementation of the SmartField class, a SmartNumericField, in order to provide proper validation methods for numbers. This class also takes care of parsing the content of the text widget in order to return an int value.

DateTextWidget. A widget based on the TextWidget which additionally provides a date picker popup displaying a calendar to the user in order to simplify the input of dates.

CheckboxWidget. A simple checkbox widget implementation. The control in this widget is a SWT Button, the value returned is either true or false, depending on the selected state of the button.

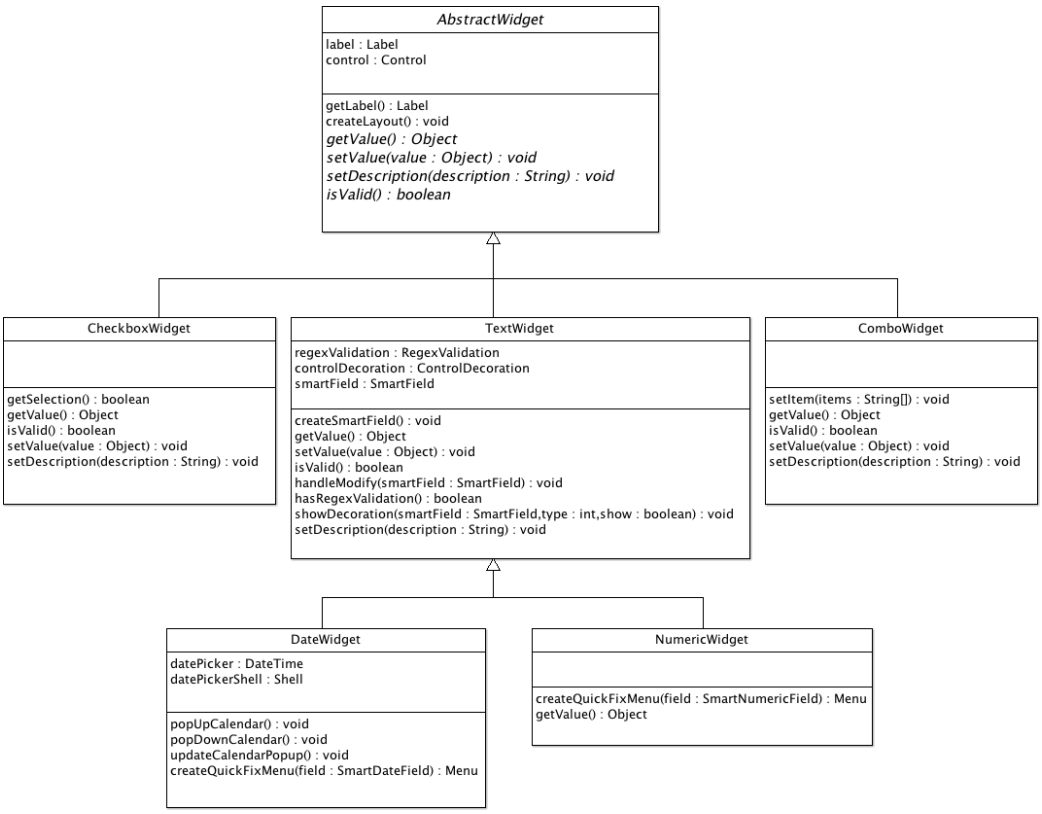


Figure 4.13: Class hierarchy diagramm of UI Widgets.

4.9 Elexis User Interface Contribution

As previously stated one of our goals is to create a modern, good-looking and functional user interface. Elexis itself, in our opinion, needs some overhauls in this area. This is why we also contributed to Elexis itself and its user interface. Figure 4.14 shows an example screen of Elexis in version 1.3 — prior to our contribution. There are many different styles of icons used throughout the application. There is an additional inconsistency in the way forms look. Some form elements have bad looking additional backgrounds. Frames and borders are used in strange ways, etc.

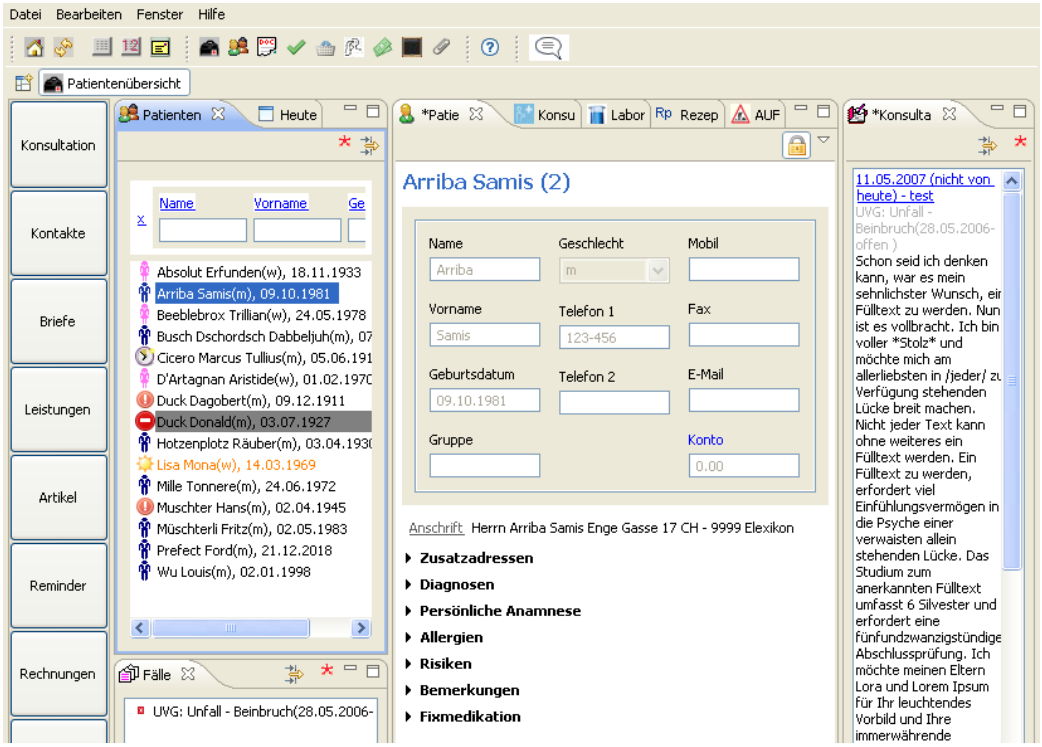


Figure 4.14: Elexis 1.3 User Interface (German)

We think with some small but important changes the look-and-feel of Elexis can be optimized. Figure 4.15 shows some of the changes we applied: We provided consistent looking icons, changed background colors, and adjusted the way forms are displayed (an example of this is the patient list on the left).

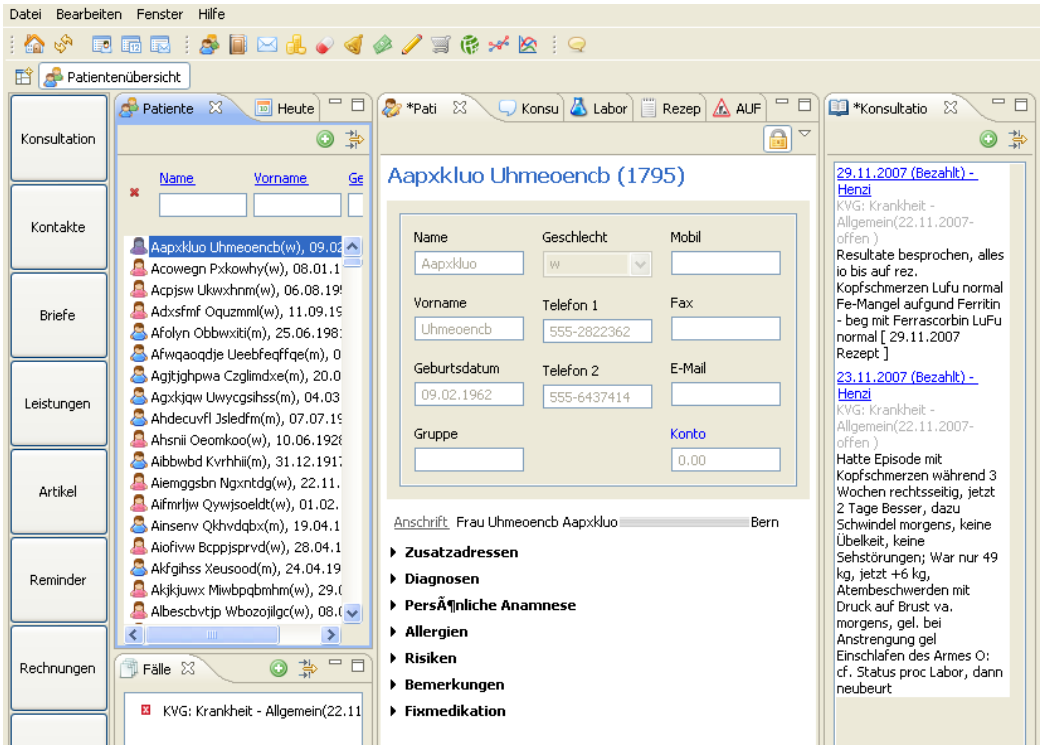


Figure 4.15: Elexis 1.4 User Interface (German)

4.10 Dashboard Charts

In addition to the previously mentioned widgets for user input, other user interface components with a custom implementation in Archie are the dashboard charts.

Dashboard charts are customized SWT composites that consist of two important parts — a ChartComposite being a JFreeChart class containing the actual chart that is displayed to the user, and a AbstractDatasetCreator that is responsible for creating the dataset for each chart in our chart composite. Again it is important to note that the dataset created by this creator is not the same dataset object as we use in our providers, but that JFreeChart merely uses the same naming for their chart object's data.

The AbstractDatasetCreator object is very similar to our AbstractDataProvider. Every dataset creator is a job so that the creation of each dataset can be monitored outside of the creator — more specifically by the AbstractChartComposite class being the job listener. As soon as the dataset creator's job is done, it

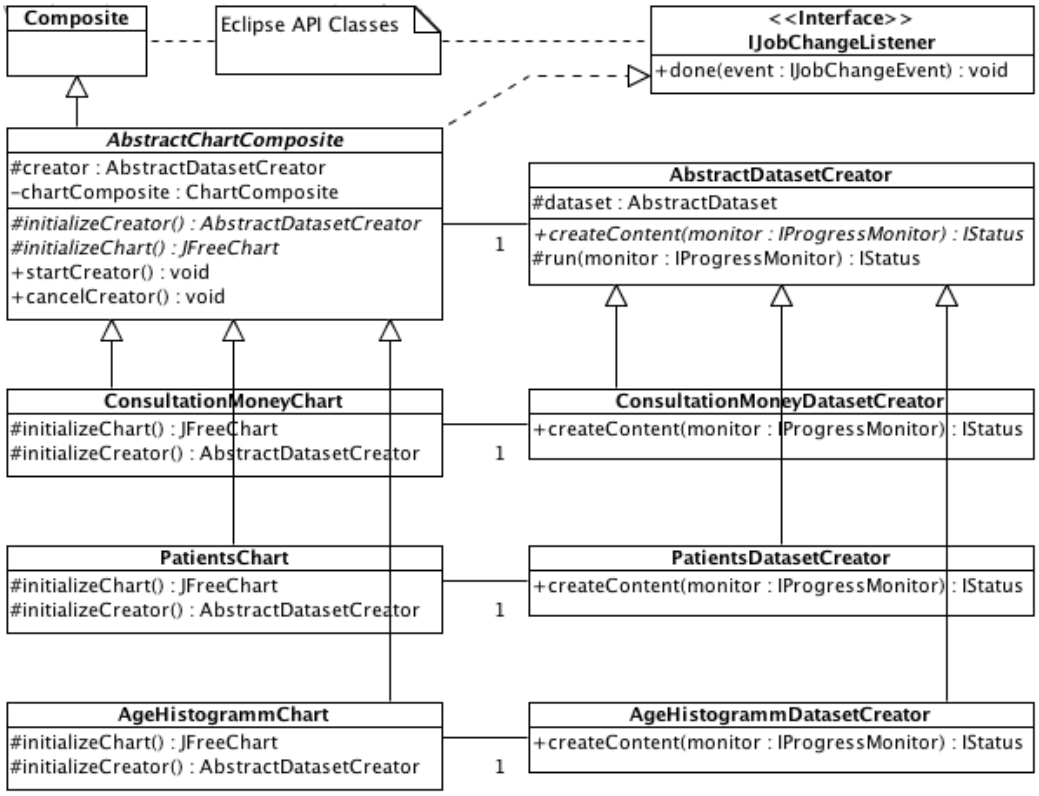


Figure 4.16: Dashboard Charts Class Diagram

removes the initially displayed *working* message and shows the created chart based on the creator's dataset.

Every implementing class of a chart composite needs to provide its own particular dataset creator based on the type of the chart the composite should display. A detailed illustration about the class dashboard charts class hierarchy is depicted in Figure 4.16.

The big advantage of this approach is not only the ability to monitor the creation of the charts along with their datasets, which depending on the size of the dataset can be time consuming, but also the ability to make use of the Eclipse Jobs API and control the state of every chart creation. Most important for us was to provide the ability to start a chart creation on demand. Running the chart creators contained in the four dashboard charts on slower computers takes a lot of time. Additionally these creators block the entire UI when Archie launches. Implementing the chart creators as Eclipse jobs allows us to start the chart generation on demand as well

as cancel already running chart creators. This also solves the UI blocking problem.

4.11 Limitations

Although we were able to realize most of the features we or the Elexis users wanted to have, there are a few limitations to Archie. These limitations can be divided into two groups — *internal* and *external*. Internal limitations result from our implementation approach, external limitations are caused either by the implementation of Elexis or the Eclipse API. Some of these limitations are described in the following sections.

4.11.1 Internal

Limited Widget Types The list of available widget types is modelled as an *enum*. Enums in Java are a special type (instead of being a class) that represent a fixed set of constants. This means the number of widget types in Archie is limited to the types described in Section 4.8. Although a plug-in for Archie theoretically could implement its own widgets by extending the AbstractWidget class, enums cannot be subclassed so a plug-in cannot add its own type to the list of available widget types.

Totals in Results Table The result table in Archie is a visual representation of a dataset. A dataset is modelled as a table as described in Section 4.2 and shown in Figure 4.3. Despite that there are several situations where a user is interested in the totals of columns, currently there is no option to display the totals of the columns or rows of such a dataset in the results table. A feature request based on this limitation is also listed among the list of issues on the Google Code project website[2].

Interaction with Results Table The results table in Archie shows different types of objects. The dataset displayed in the table can hold simple types such as *numbers* or *strings*, but also more complex and Elexis-specific types like *persons*, *invoices*, or similar. Currently there is no built-in mechanism in Archie to allow contributing plug-ins to show context-specific information

[2] http://code.google.com/p/archie/issues/list

based on an object type in the results table cell. When a user for example double-clicks a cell showing the value of an invoice in Elexis, a contributing plug-in might want to react to this event and show the details of the invoice in a separate window. This limitation is also noted in the form of a feature request on the Google Code project website.

4.11.2 External

Internationalization Archie can be completely translated into different languages, as well as almost every part of each Eclipse plug-in— but unfortunately not every part. There are limitations of the Eclipse internationalization mechanism, namely when it comes to translating VM retained annotations, that are being read at run-time. In Archie this is the case for parameters of a data provider.

Listing 4.6: Example Annotation

```
/**
 * @param currentMandatorOnly
 */
@SetProperty(name = "Active Mandator Only")
public void setCurrentMandatorOnly(final boolean currentMandatorOnly) {
        this.currentMandatorOnly = currentMandatorOnly;
}
```

Each parameter of these annotations, such as for example name in Listing 4.6 must have a value. Translations in Eclipse work by defining constants in a Messages.java file and associating these constants with their language specific values in a separate text file such as messages.properties or in a dedicated language file such as messages_de.properties for German.

The values of these constants of translatable strings are associated by Eclipse at runtime, depending on the language set in the running application. Annotation parameter values need either to be concrete values or constants with defined values at compile time, which is why we cannot use translations in annotation parameters.

Access Control Lists Archie uses the access control list (ACL) extension point definition ch.elexis.ACLContribution provided by Elexis for restricting access to certain users to the entire Archie application. This is particularly

important in medical practices where not only the physician might have access to Elexis. As Archie provides access to sensitive statistics about patients, access to this data needs to be restricted in certain cases.

Unfortunately, the entire access control protection only works when Elexis is started with the de_CH language parameter and thus localized in German. This problem is known to the Elexis core developers but at the point of this writing, there is no working solution, which is why the ACL restrictions do not work in Elexis language setting other than German.

This is a limitation of the currently stable Elexis 1.4 release and has been fixed in the latest 2.0 beta version.

Chapter 5

Team Organization

Because of the uncommon situation of doing this project as well as writing this report in a team, we needed to divide the tasks in each phase of the project. The phases of the project included creation of the initial project presentation, implementation of the prototype, making architectural desicions for the programming part, programming Archie along with the sample statistics plug-in, and at last, writing this thesis.

Throughout the project, we tried to work in each phase in pair when it was efficient and divide the tasks when it was not. We made the initial project decisions and discussed the entire architectural process together. During the programming part, although working side-by-side in the same room each on his own computer, we had to separate the coding areas. This way not both of us were working on the same task simultaneously and thus could advance faster, while still having the possibility to discuss problem areas as well as possible solutions together. This process proved to be very effective.

Before we started working on the report, we defined the initial table of contents together based on which we assigned specific chapters to each of us. We wrote the chapters individually and apart from each other before we corrected and combined them as a pair to one entire document in an iterative process. Dennis wrote initial versions of Chapter 2, Appendix B, Appendix C, and Chapter 6. Peter wrote initial versions of Chapter 1, Chapter 3, Chapter 4, and Chapter 5. We did not assign the chapters according to a specific scheme.

For the task management process, we made heavy use of *activeCollab*[1], a

[1] http://www.activecollab.com/

project management and collaboration tool, as well as *Subversion*[2] for versioning source code and text files. Near the end of the project, we defined a Google Code project website[3] for Archie in order to maintain official releases and downloads as well as provide a public interface for reporting issues and enhancement requests.

[2] http://subversion.tigris.org/
[3] http://code.google.com/p/archie/

Chapter 6

Requirements Engineering and Validation

At the beginning of our project we visited various doctors and practices. They showed us what system they had in place to manage their patients, what their workflow was and what the advantages and disadvantages of these were. We asked what they would expect from an EMR system and how they would go on about migrating. This way we got an insight into the difficulties of migrating paper based systems, but we also saw that human factors are actually more important. Radically changing workflow for example can be very hard, especially for doctors and employees of long existing practices. With this and other input that was given we got a feeling for what was important and what challenges, but also chances, lie in the domain of EMR.

Later, the idea of Archie was already born, we visited a practice where Elexis was used. We talked to the doctors using the system and got more information on it in general and insight into how it changed workflow and practice management. We also met Gerry Weirich, the main developer of Elexis, and talked with him about our ideas.

The next step was to contact more developers of Elexis and its user base, which, interestingly, are mostly the same people: the main developers all use Elexis in their own practices. We registered ourselves on the Elexis mailing list and on the forum, where we proposed our idea for Archie. We got various feedback, e.g. for what kind of statistics they would wish the most for. From these ideas and wishes we started to draw out what capabilities our statistic framework should have. Slowly but surely the requirements for our projects became clearer.

After the initial exchange of ideas and requirement proposals we started to write Archie. From time to time we presented our advancements and got feedback on them. This was also a good way to see if we were still on track with the project. The feedback we got was mostly very good and we were glad to find that the Elexis community is, albeit a small one, very active and constructive.

At the last stages of development we started working closer with Gerry Weirich, which was very fruitful. He started to use and test Elexis which gave us more important feedback. Also we were able to ask questions about the Elexis code base directly to its creator and could thus improve and optimize Archie to work better and closer with Elexis.

Subsequently the Archie framework was used by Gerry to create an accounting tool [Weir09a]. It contains multiple analyses like an open bills statistic, a payment journal statistic etc. We were glad that our framework was already being used as a base for other projects and that we were told that it had been pretty easy to do so, which was one of our goals with Archie.

Reception of the framework has been very good up until writing of this paper. We feel that there is great interest from users and developers alike. The plugin gets many feature requests and it will become part of the next major Elexis release.

Chapter 7

Conclusions

In Chapter 1 we stated the goals of this project. Then we showed how to the data provider can be used for retrieving data about Elexis and how the data provider stores this data in a dataset modelled as a table for further processing. We have shown how other Eclipse plug-ins can hook into the Archie framework in order to provide their own statistics. Moreover, Archie provides a chart wizard that allows users to visualize statistical, textual data by using pie, bar, or line charts, as well as export this data in a standardized format for further processing.

Archie is written as an Eclipse plugin which allows the users to easily deploy as well as maintain it. Furthermore, we worked closely together with an active Elexis userbase as well as its main developer to ensure that the final product will be used on a day-to-day base by Elexis users.

7.1 Lessons Learned

We learned that contributing to an already existing project can often be difficult and time consuming. Not only do you have to familiarize yourself with the available program code, but with the programming style of the project owners as well. The latter can have a great impact on how fast this familiarization process progresses.

Moreover, we learned that Eclipse is a great Java framework. It provides a very extensive set of API classes and methods that allow you to develop complex applications. On one hand, this great amount of already available building blocks can save you a lot of time. However, on the other hand it is

this very amount of blocks that makes it difficult to use. The documentation on some parts of Eclipse is very scarce, often your best source is a good web search engine or the Eclipse newsgroups.

Another lesson was that working in a team often requires adapting the team organization to the current task. As an example, for us when writing code it was most efficient to do it in pair as described in Chapter 5, albeit when writing the thesis it was best to work individually on separate chapters or sections.

7.2 Future Work

In this section we outline future work and possible additional functionalities of Archie. Some of the following paragraphs are related to the limitations described in Section 4.11.

Flexible Widget Types Archie currently supports only a fixed set of UI widgets used for user input. The widget types could be implemented as a class instead of an *enum*. With this an implementing plug-in can extend the available widgets by adding its own widget types such as radio, drop-down, or other custom widgets. Another approach would be to define a new extension point in the Archie plug-in definition file and let implementing classes use this hook to provide custom widget types.

Totals in Result Table For several statistics users might be interested in a computed total of each column. This could be implemented by extending the dataset model and adding a *special row*. Each cell of this additional row contains the totalised value of all corresponding cells in a numeric column (column that contains numeric values only as totalising make most sense for numeric values). Such a row needs to be excluded from the sorting of a dataset which is why it is marked as *special* in the dataset model. Figure 7.1 shows how a possible solution for showing totals in the results table could look like.

User Interaction in Result Table The misssing user interaction could be implemented by allowing plug-ins to provide (custom) selection listeners. Currently, every plug-in (data provider) can define a custom content and

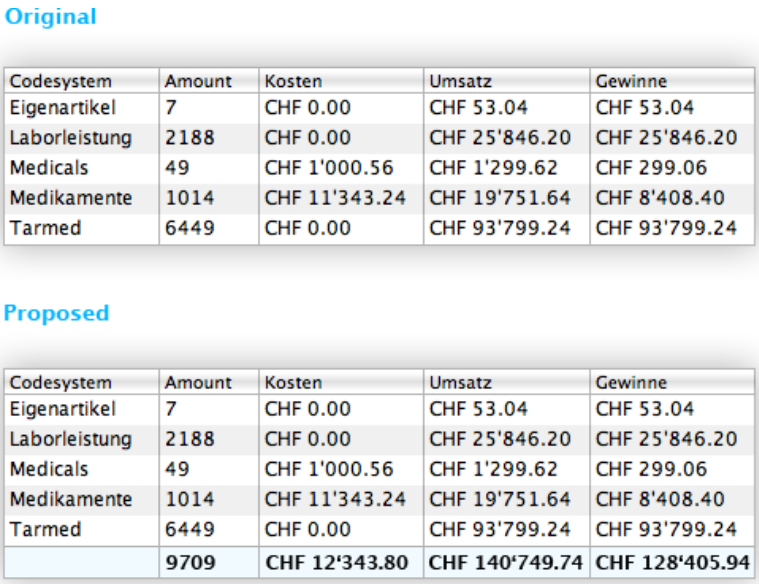


Figure 7.1: Possible solution for showing totals in the results table.

label provider. These providers are respected when the results table is rendered in the UI. Using the same system, a data provider could define selection listeners that would react to double-blick events on a table row or cell accordingly.

Adding More Statistics The *samples* part of Archie contains seven different implementations of the framework, providing seven different statistics. Although statistics developed for Archie by other parties [Weir09a] already exist, additional statistics implementations could be provided in the future.

Appendix A

Licensing of Archie

From the very beginning our stated goal has been to distribute Archie as part of the official Elexis release. To accomplish this goal we release Archie under the Eclipse Public License (EPL), the same license that Elexis uses. 1.0[1]. The EPL has lower restrictions than other licences such as the GNU General Public License (GPL)[2]. For instance, every software building upon or using software licenced under the GPL has to be released with the same licence while this is not necessary under EPL.

Archie will become part of the Elexis core with the next major release.

[1] http://www.eclipse.org/legal/epl-v10.html
[2] http://www.gnu.org/licenses/gpl-3.0.txt

Appendix B

User Manual

B.1 Quick Start

To install Elexis and Archie follow these steps:

1. Download an Elexis 1.4 installation package for your platform from http://www.elexis.ch/jp/index.php?option=content&task=view&id=57 and install it on your computer.
2. Download Archie from archie.designchuchi.ch or directly from http://code.google.com/p/archie/downloads/list
3. Unpack the contents of the archive file into the plug-ins directory inside your Elexis installation directory.

   Note: The archive should contain two .jar files, they should end up in the plug-ins directory along with the files already present.
4. Download the demo database from http://www.elexis.ch/files/demoDB.zip and extract it to your Elexis installation folder.

   Note: the current demo database only contains entries for one month, namely August 2008. Statistics queries which use date ranges outside of this month return no results.
5. Start Elexis with the −nl de_CH parameter. On Mac OS X systems issue the following command in a terminal window <path to your elexis folder>/elexis.app/Contents/MacOS/elexis −nl de_CH. On other operating systems add the −nl de_CH parameter to the elexis executable when running it. Login with user name *test* and password *test*.

Note: In order to be able to set access privileges for Archie in Elexis, either the system locale has to be set to *de_CH* or Elexis needs to be started with the −nl de_CH parameter. This is a known limitation of the Elexis 1.4 version as described in Section 4.11. This bug will be fixed in Elexis version 2.0.

6. To use Archie you have to give the *test* user admin privileges. To do so open the preferences, select "groups and rights" and "access control", click on "Archie access", select the *test* user, and click on "apply".

B.2 Usage

To use Archie, open up the Archie perspective by either clicking on the button in the main button tool bar or by clicking on *Window* → *Open Perspective* → *Other* → *Archie Perspective* in the menu bar.

In the perspective you see two views: the Dashboard and the Sidebar.

B.2.1 Dashboard View

The Dashboard gives a quick overview of your Elexis system. It shows some key data like number of patients and number of consultations. With a click on the "Create Charts" action button on the upper right side of the view, the dashboard draws four charts:

Number Of Consultations. Shows number of consultations during the last six months.

Consultation Money. Shows how much profit, income and costs we had for consultations during the last six months.

Costs of Consultations. Shows costs of all consultations in a histogram, grouped by age-group and gender.

Age Histogram. Shows all patients in the system grouped by age-group and gender.

The size of the age-groups of the last two charts can be altered in the Archie preferences. To open them click on *File* → *Settings* → *Archie* .

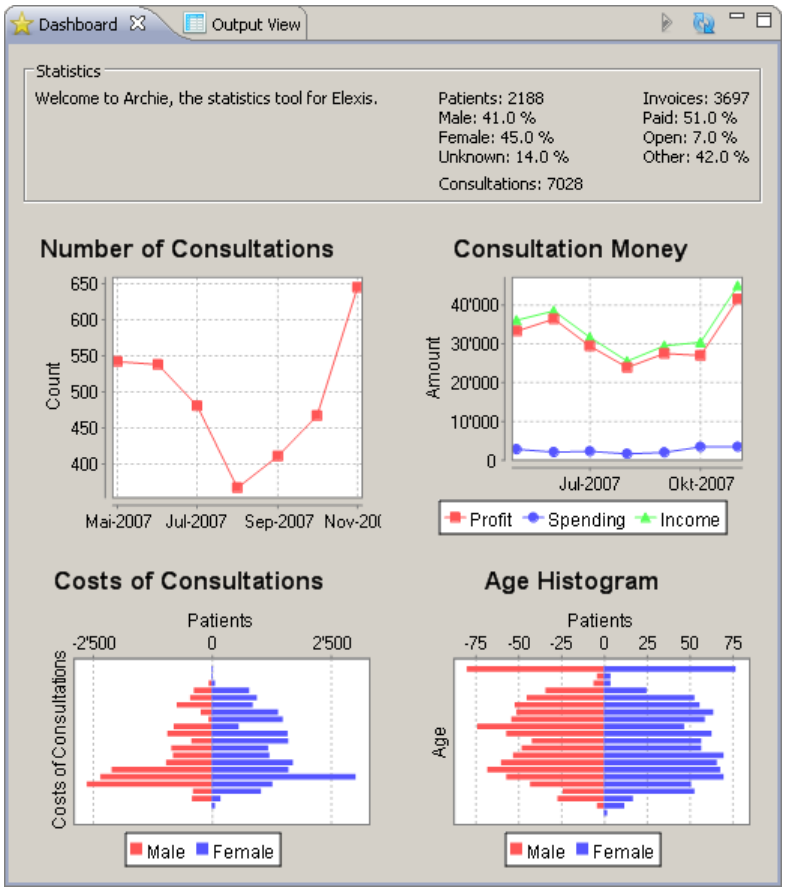


Figure B.1: Dashboard View

B.2.2 The Sidebar View

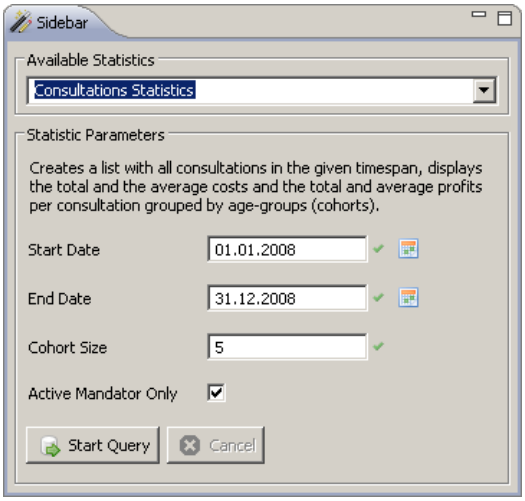


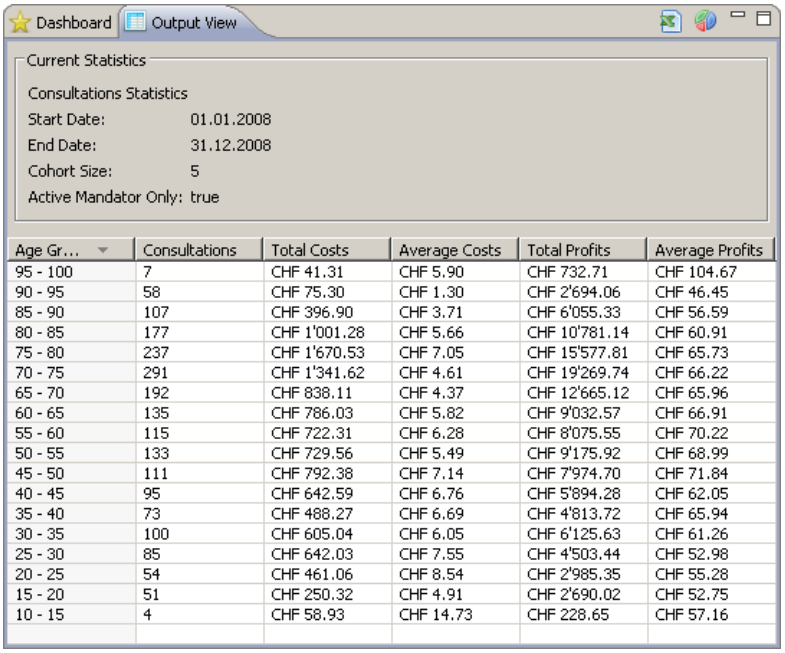
Figure B.2: The Sidebar View

In the *Sidebar View* you can find all the available statistics in your Elexis system in a drop down list. If there are too many statistics to scroll through you can also type in the name of a statistic and an auto-completion helps you find it. If you chose a statistic, additional options and settings are shown in the sidebar. When you have made your choices, click on "Start Query" and a background job will be started to assemble the data.

B.2.3 The Output View

When a statistics creation job is finished, the results will be shown in the Output View. Here you can sort the results by clicking on any table heading. If you want to export the data to a Comma Separated Values (CSV) file, you can do so by clicking on the *"Export Action"* button in the upper right corner of the view. After exporting the data, you can then open it for further processing in Microsoft Excel, Open Office Calc, or any other software which supports CSV files.

If you want to draw the results as a graphical chart click on the *"Chart Wizard"* action button, on the upper right corner of the Output View.



Age Gr...	Consultations	Total Costs	Average Costs	Total Profits	Average Profits
95 - 100	7	CHF 41.31	CHF 5.90	CHF 732.71	CHF 104.67
90 - 95	58	CHF 75.30	CHF 1.30	CHF 2'694.06	CHF 46.45
85 - 90	107	CHF 396.90	CHF 3.71	CHF 6'055.33	CHF 56.59
80 - 85	177	CHF 1'001.28	CHF 5.66	CHF 10'781.14	CHF 60.91
75 - 80	237	CHF 1'670.53	CHF 7.05	CHF 15'577.81	CHF 65.73
70 - 75	291	CHF 1'341.62	CHF 4.61	CHF 19'269.74	CHF 66.22
65 - 70	192	CHF 838.11	CHF 4.37	CHF 12'665.12	CHF 65.96
60 - 65	135	CHF 786.03	CHF 5.82	CHF 9'032.57	CHF 66.91
55 - 60	115	CHF 722.31	CHF 6.28	CHF 8'075.55	CHF 70.22
50 - 55	133	CHF 729.56	CHF 5.49	CHF 9'175.92	CHF 68.99
45 - 50	111	CHF 792.38	CHF 7.14	CHF 7'974.70	CHF 71.84
40 - 45	95	CHF 642.59	CHF 6.76	CHF 5'894.28	CHF 62.05
35 - 40	73	CHF 488.27	CHF 6.69	CHF 4'813.72	CHF 65.94
30 - 35	100	CHF 605.04	CHF 6.05	CHF 6'125.63	CHF 61.26
25 - 30	85	CHF 642.03	CHF 7.55	CHF 4'503.44	CHF 52.98
20 - 25	54	CHF 461.06	CHF 8.54	CHF 2'985.35	CHF 55.28
15 - 20	51	CHF 250.32	CHF 4.91	CHF 2'690.02	CHF 52.75
10 - 15	4	CHF 58.93	CHF 14.73	CHF 228.65	CHF 57.16

Figure B.3: The Output View

The Chart Wizard The Chart Wizard allows you to create charts from any result table. To explain the Chart Wizard we show an example creation of a chart from a result set. When you click on the action button, you are presented two options: Creating a pie chart or creating a bar chart (which can also create line charts) as seen in Figure B.4.

The next dialog asks you to specify what exactly you want to plot and how, as can be seen in Figure B.5.

When all parameters are set you can choose which rows of the dataset you want to use in your chart. For bar charts for example you might want to sort the rows and only take the top ten to plot. In our example we choose all rows, as there are only five anyway. See Figure B.6.

When we click on *Finished* our chart will be created – as the example chart in Figure B.7 shows.

You can further investigate the chart by zooming in or out on certain parts. When you move your mouse around, tool tips get displayed, explaining what you see. You are also able to change certain aspects of the chart (such as colors, captions etc.) by right-clicking on it. You can also save the image to your hard drive by clicking on *"Save As..."*.

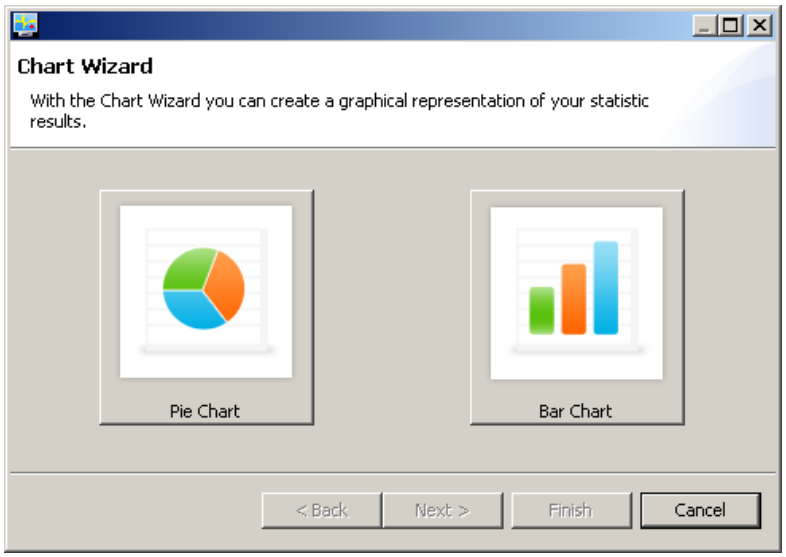


Figure B.4: Chart Wizard: Choose Chart Type

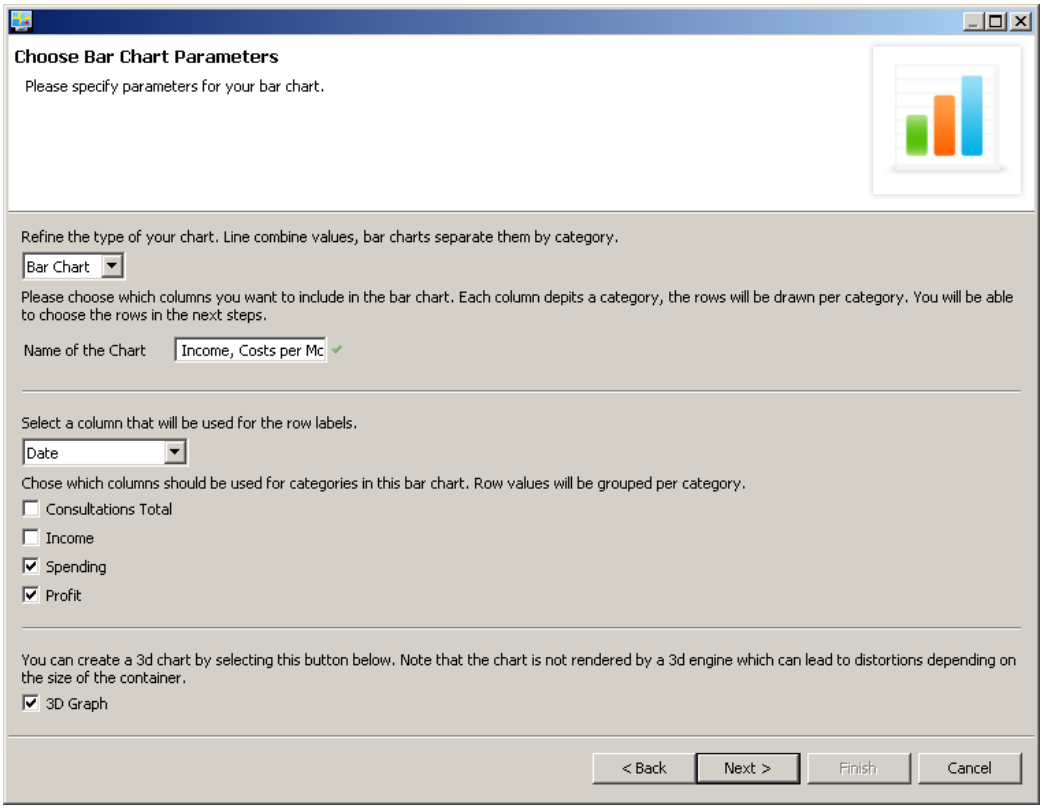


Figure B.5: Chart Wizard: Bar Chart Page

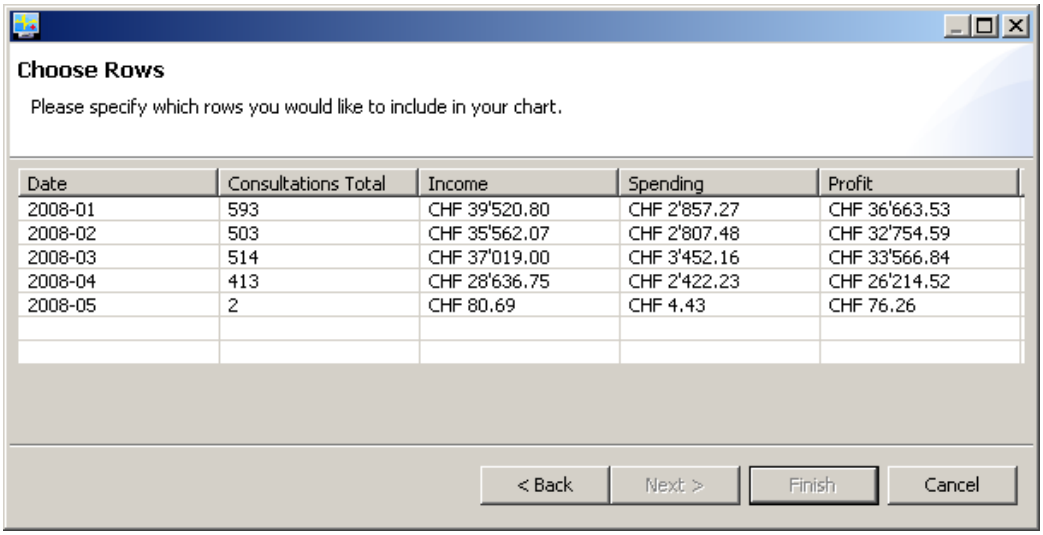


Figure B.6: Chart Wizard: Choose Rows

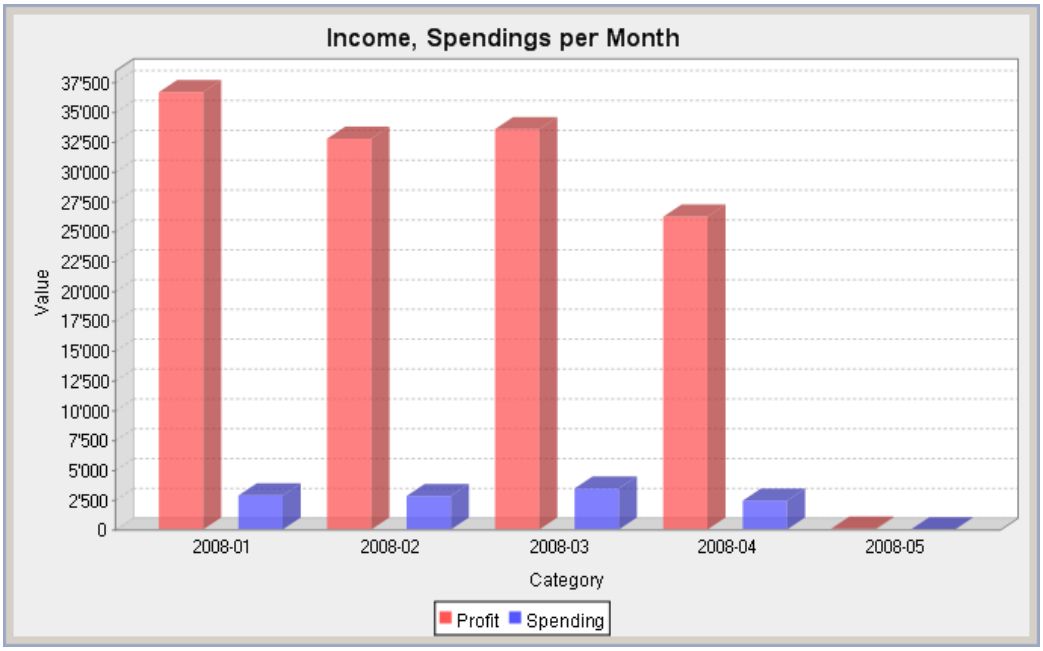


Figure B.7: Example chart created with Chart Wizard

Appendix C

Developer Manual

Writing a statistic which uses the Archie framework is pretty easy. There are two main steps involved: Extending the AbstractDataProvider class and Registering the statistic with the Archie Extension Point. We will look into these steps in detail in the next sections.

For examples on how to write statistics please consult the source code of the Archie samples project. You can find information on how to check out the source files via SVN from the Archie project website at http://code.google.com/p/archie/. If you want to get information on how to develop for Elexis in general, please consult the developer section on the Elexis website at http://www.elexis.ch.

This manual assumes that you have basic knowledge in Java, Eclipse, and Elexis development.

C.1 Extending AbstractDataProvider

The first step is to create a class in your project which extends

ch.unibe.iam.scg.archie.model.AbstractDataProvider.

If your statistics needs to constrain itself to a timespan, you can also directly extend

ch.unibe.iam.scg.archie.model.AbstractTimeSeries

which extends AbstractDataProvider and is already set up for the task of handling time spans. It has instance variables startDate and endDate which

you can use. How to use these two variables in your code is up to you, AbstractTimeSeries just sets up the UI in the right way and provides the two dates the user entered.

In any case you will have to override the following methods:

Listing C.1: Methods to override

```
/**
 * Return an appropriate description
 */
public String getDescription()

/**
 * Create dataset headings in this method.
 */
protected List<String> createHeadings()

/**
 * Compose the contents of a dataset here
 */
protected IStatus createContent(IProgressMonitor monitor)
```

C.1.1 Constructor

In the constructor you set the name of your statistics job and you can carry out any additional initialization you might need.

C.1.2 getDescription

This method returns a short description of your statistic. This description gets displayed in the Archie sidebar when your statistic is selected. It should convey what your statistic is about in general and what kind of results it will return.

C.1.3 createHeadings

This method returns a list with strings. The list will be used to populate the table headings in the result view. For example:

Listing C.2: createHeadings() Example

```
/**
 * @see ch.unibe.iam.scg.archie.model.AbstractDataProvider#createHeadings()
 */
@Override
protected List<String> createHeadings() {
    final ArrayList<String> headings = new ArrayList<String>(2);
    headings.add("Heading 1");
    headings.add("Heading 2");
    return headings;
}
```

Pay attention to the number of headings you return, it has to correspond to the number of columns you create in createContent()

C.1.4 createContent

This is the method that does the actual work of the statistic. Let us look at an example:

Listing C.3: createContent() Example

```
/** {@inheritDoc} */
@Override
public IStatus createContent(IProgressMonitor monitor) {

    // Initialize result list
    final List<Comparable<?>[]> result = new ArrayList<Comparable<?>[]>();

    // Set job size and begin task
    int size = allPatients.size();
    monitor.beginTask("Working on patients\ldots", size);

    for (Patient patient : allPatients) {
        // Check for Cancellation.
        if(monitor.isCanceled()) {
            return Status.CANCEL_STATUS;
        }

        // Do actual work...
        // Fill result array
```

```
            monitor.worked(1);
        }

        // set content
        this.dataSet.setContent(result);

        // job finished successfully
        monitor.done();
        return Status.OK_STATUS;
    }
```

Listing C.3 shows a skeleton of an actual createContent method. The first thing you might notice is that the method does not return the actual data, but the status of the job. If the job is finished the method should return an OK status. It is also important to check for cancel request inside loops, if the method doesn't check for that, the cancel button in the sidebar will not work. You should also set the size of your job, and work with the monitor, so users get feedback on how your job is progressing. When the job is finished, be sure to set the content of the dataset with this.dataSet.setContent(result), else it will not work.

The most important part is that you put the right data into the dataset. As you can see in the example you should fill a list which contains arrays of objects that implement the comparable interface. If the objects you want to put into the dataset don't implement said interface, you might have to write a wrapper class for them. The reason is that we have to make sure that the results are sortable in the result view.

The arrays in the list will end up as rows in the result table, they have a determined size which has to be the same as the headings size. The list holds all these rows and is of variable size.

C.2 Adding Additional Parameters

To present the user with additional parameters that you want to use in the creation of your statistical data you can write getter and setter methods for these parameters and annotate them in the right way. What follows is an example of such a method pair which is responsible of providing the user with a checkbox in the UI asking whether he wants to create the statistic only for the active mandator (involving only his consultations) or for all mandators

(involving all consultations in the system). Providing users with such an option has been established as good practice during development.

Listing C.4: Annotated method pair for additional statistic parameters

```
/**
 * @return True if statistic should be created for current mandator only, false else.
 */
@GetProperty(
    name        =  "Active Mandator Only",
    index       =  1,
    widgetType  =  WidgetTypes.BUTTON_CHECKBOX,
    description  =  "Compute statistics only for the current mandator.
                    If unchecked, the statistic will be computed for all mandators."
)
public boolean getCurrentMandatorOnly() {
    return this.currentMandatorOnly;
}

/**
 * @param currentMandatorOnly
 */
@SetProperty(name = "Active Mandator Only")
public void setCurrentMandatorOnly(final boolean currentMandatorOnly) {
    this.currentMandatorOnly = currentMandatorOnly;
}
```

You are now able to use this.currentMandatorOnly in your code, the UI has been set up automatically. A good practice is also to initialize the member variable with a value. If you default it to true for example, the checkbox is selected in the UI from the beginning, otherwise it is not. Now let us take a closer look at the annotations.

The *name* is an identifier for your method pairs, it is important that they are the same in both methods. It also gets used as label displayed in front of the control in the UI.

The *index* parameter controls in which order your properties will be displayed in the UI as well as processed when they are set. If you have several properties they get sorted in descending order based on the index.

With *widgetType* you can specify what kind of widget will be displayed in the UI. For available widget types please consult: ch.unibe.iam.scg.archie.ui.WidgetTypes. The most important are:

TEXT. A vanilla text widget which can validate itself.

TEXT_NUMERIC. A text widget only for digits.

TEXT_DATE. A date widget which has a inline datepicker for easy date setting.

BUTTON_CHECKBOX. A simple checkbox, is either selected or not.

TEXT_NUMERIC makes sure that only numbers are entered, TEXT_DATE does the same for dates. If you want different, more specific validations for text widgets, you can also specify a regular expression for it.

With validationRegex you can specify a regular expression pattern. If the pattern will match the contents of a text widget a green tick will be shown – symbolising that the widget has been filled in correctly. Else a warning or error symbol will be shown. Here is an example:

validationRegex = *"^([1−9]){1}\\d{0,2}"*.

If you provide a custom regex pattern, you also have to provide a description of it using validationMessage. This message will be displayed if the widget does not validate. Here is an example, corresponding to the one above:

validationMessage = *"This field has to consist of at least one, at most three numbers ."*)

C.3 Registering with the Archie Extension Point

After you have written your statistic you have to register it with the Archie Extension Point. To do this you have to edit the plugin.xml or fragment.xml file of the Elexis plug-in your statistic class resides in. Just add and edit the following lines:

Listing C.5: Registering a statistic with the Archie Extension Point

```
<extension point="ch.unibe.iam.scg.archie.dataprovider">
  <DataProvider
    category="name.of.your.statistics.category"
    class="name.of.your.statistics.class"
    name="Name of your Statistic">
  </DataProvider>
  <category
    id="name.of.your.statistics.category"
```

```
    name="someCategory">
  </category>
</extension>
```

The category is used to group similar statistics together. For example all statistics dealing with some monetary aspect could be grouped in one category.

After you recompile and restart Elexis, your newly created statistic should now be in the drop down list of available statistics in the Archie perspective, prefixed with the name of the category they are in.

C.4 Where to Get Additional Help

You can find help for Elexis development in general at the Elexis forum at http://www.elexis-forum.ch. For help with Archie development you can go to the official project website at http://archie.designchuchi.ch or the already mentioned Google code website at http://code.google.com/p/archie.

List of Figures

2.1 Schematic concept of a sample EMR. 9
2.2 EMR, EHR, CIS in sample relation. 10

3.1 Sanclipse Prototype Overview 14
3.2 Eclipse Plug-in Architecture 15
3.3 Elexis Architecture 16
3.4 First, Conceptual UML Draft 17

4.1 Extension Schema Definition Screenshot 23
4.2 List of Providers Before and After the Extension Point Changes 24
4.3 An Example Dataset 26
4.4 AbstractDataProvider class and surroundings 28
4.5 Three different UI widgets. 31
4.6 UML overview of the action classes 33
4.7 Data Provider – Controller Interaction 35
4.8 UML overview of the manager classes 37
4.9 UML overview of the factory classes 38
4.10 Dataset Mapping in Bar Charts 40
4.11 Valid Filename Proposal During CSV Export 42
4.12 Widgets in the UI 44
4.13 Class hierarchy diagramm of UI Widgets. 45
4.14 Elexis 1.3 User Interface (German) 46
4.15 Elexis 1.4 User Interface (German) 47
4.16 Dashboard Charts Class Diagram 48

7.1 Possible solution for showing totals in the results table. 58

B.1 Dashboard View 62
B.2 The Sidebar View 63
B.3 The Output View 64
B.4 Chart Wizard: Choose Chart Type 65
B.5 Chart Wizard: Bar Chart Page 66
B.6 Chart Wizard: Choose Rows 66
B.7 Example chart created with Chart Wizard 67

Listings

4.1 Extension Point Definition in plugin.xml 22
4.2 User Overview Extension Point Implementation 25
4.3 Abstract Data Provider Methods 27
4.4 GetProperty Annotation Definition 30
4.5 Chart Model Object Variables 41
4.6 Example Annotation 50
C.1 Methods to override 69
C.2 createHeadings() Example 70
C.3 createContent() Example 70
C.4 Annotated method pair for additional statistic parameters . . 72
C.5 Registering a statistic with the Archie Extension Point 73

Bibliography

[Arth08] John Arthorne, Chris Laffra, and Gary Johnston. "What is a plug-in fragment?". http://wiki.eclipse.org/FAQ_What_is_a_plug-in_fragment%3F, February 2008.

[Bell04] Donald Bell. "UML's Sequence Diagram". http://www.ibm.com/developerworks/rational/library/3101.html, February 2004.

[Bhen06] Heinz Bhend. "TEXTDOKUMENT der CD-ROM "Die elektronische Krankengeschichte in der Arztpraxis"". DOCUMENT to CD-ROM, October 2006.

[Coch89] A L Cochrane and Max Blythe. *One man's medicine*. Memoir Club, 1989.

[Gall02] David Gallardo. "Developing Eclipse plug-ins". http://www.ibm.com/developerworks/opensource/library/os-ecplug/, December 2002.

[Gesu07] Bundesamt für Gesundheit. "Strategie "eHealth" Schweiz". http://www.ehealth.admin.ch/, June 2007.

[Laff06] Chris Laffra and Nick Veys. "What is an extension point schema?". http://wiki.eclipse.org/FAQ_What_is_an_extension_point_schema%3F, June 2006.

[McAf07] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Richt Client Platform*. Addison Wesley, 2007.

[Vale04] Michael Valenta. "On the Job: The Eclipse 3.0 Jobs API". http://www.eclipse.org/articles/Article-Concurrency/jobs-api.html, September 2004.

[Weir09a] Gerry Weirich. "elexis-buchhaltung-basis". http://www.elexis.ch/jp/content/view/236/75/, January 2009.

[Weir09b] Gerry Weirich. "Verbreitung von Elexis". http://www.elexis.ch/jp/content/view/245/87/, February 2009.