Mid Sweden University
The Department of Information Technology and Media (ITM)
Author: Xuchen Lu
E-mail address: luxuchen88@hotmail.com
Study programme: Computer Engineering MA, Final Project, 30
Examiner: Professor Tingting Zhang, Tingting.Zhang@miun.se
Tutors: Professor Tingting Zhang, Tingting.Zhang@miun.se
Scope: 17,155 words inclusive of appendices
Date: 2013-01-06

M.Sc. Thesis
within Computer Engineering C, course, 30 points

HETEROGENEOUS DATA SOURCE MIDDLEWARE

Xuchen Lu
Abstract

As the complexity of data sources increases, it becomes a significant challenge to develop and maintain applications which are required to interact with heterogeneous data sources. It becomes even more difficult when the intention of the application is to frequently modify the data source. Application developers must look into thousands lines of source code to locate the corresponding part and to modify them one by one. The diversity of data source properties and data schemas makes it complicated for the application to adapt to a new data source.

In this thesis, a middleware solution is proposed to address this problem. Both permanent and real-time data should pass through the middleware during communication between the data source and the application, or among data sources.

The middleware creates a data source service wrapper for each data source and uses an abstract class to shield different wrappers. A global-local mapping mechanism is used to eliminate the data schema incompatibility. In addition, xml technology is used to accomplish data transmission, which eases the interface design and ensures platform independence. At the same time, a Data Source Wizard has been developed to assist the maintainer of the middleware with items for middleware extensions. The Data Source Wizard makes it more convenient and standardized to add a new data source.

The experience of integrating the middleware into this Android E-Health application has proved the flexibility and extensibility of the middleware. Because the middleware does not contain any specific business logic, it can be generally applied to other applications with different purposes.

Keywords: Middleware, Heterogeneous Data Source, XML, Abstraction, Modularization.
Acknowledgements

First of all, I would like to thank Professor Tingting Zhang. As my supervisor in Mid Sweden University, she provided me with the opportunity to take a part in this meaningful project. In addition, she provided me with a great deal of useful guidance and suggestions about my research work and thesis writing.

Secondly, I want to express heartfelt thanks to Professor Jiajin Le. Professor Le is my supervisor in Donghua University. He has always been encouraging and taking care of me both in study and life.

Then, I would like to thank my teacher Sebastian Bader of the scientific writing and presentation lecture. He informed me about the skills regarding how to write a good scientific paper.

I would particularly like to thank my classmates who gave me useful advice and helped me with the project. Thanks to Jun Tang who helped me to accommodate myself to the new circumstances and to become familiar with the whole project. Thanks to Hongling Tang who implemented the executable version of Android E-Health Application. Thanks to Wenli Cheng who put forward basic idea of middleware design.

Finally, I offer my deepest gratitude to my parents. Thanks for their constant support and unchanging love.
Table of Contents

Abstract ............................................................................................................. ii
Acknowledgements ....................................................................................... iii
Table of Contents ........................................................................................... iv

1 Introduction ............................................................................................ 1
  1.1 Background and problem motivation ...................................... 1
  1.2 Overall aim ................................................................................... 3
  1.3 Scope ............................................................................................. 4
  1.4 Concrete goals .............................................................................. 4
  1.5 Outline .......................................................................................... 5

2 Related work .......................................................................................... 6

3 Methodology and methods ..................................................................... 10
  3.1 Modularization .......................................................................... 10
  3.2 Middleware Solution ................................................................ 11
  3.3 OOP ............................................................................................. 12
  3.4 Abstraction ................................................................................. 13
  3.5 Generalization ............................................................................ 15

4 Design and implementation .................................................................... 16
  4.1 Android E-Health application ................................................. 16
     4.1.1 Main Functions ............................................................... 16
     4.1.2 Layered structure ........................................................... 18
  4.2 Heterogeneous Data Source Middleware .................................. 20
     4.2.1 Architecture ..................................................................... 21
     4.2.2 Data source service wrapper ........................................ 22
     4.2.3 Data transmission ........................................................... 25
     4.2.4 Data source localization ................................................. 27
     4.2.5 XML Handler .................................................................. 30
     4.2.6 Integration ....................................................................... 32
     4.2.7 Extending ......................................................................... 34
  4.3 Data Source Wizard .................................................................... 35
     4.3.1 Overview ......................................................................... 35
     4.3.2 Basic Info setting............................................................... 38
     4.3.3 Configuration Info setting ............................................. 39
     4.3.4 Mapping info setting ...................................................... 41
     4.3.5 Data source service wrapper generation ..................... 44
5 Evaluation ........................................................................................................... 46
5.1 Generality ........................................................................................................ 46
5.2 Easy-to-use ....................................................................................................... 47
5.2.1 Middleware manager side ........................................................................ 47
5.2.2 Application developer side .................................................................... 47
5.3 Flexibility .......................................................................................................... 51
5.4 SQL Injection ................................................................................................ 52
5.5 Efficiency .......................................................................................................... 54

6 Conclusions ....................................................................................................... 56

References .............................................................................................................. 58

Appendix A: Documentation of own developed program code........ 61
Code of AbstractDSService .................................................................................. 61
Code of XMLHandler ........................................................................................... 61
Code Example of business logic layer ................................................................. 64
1 **Introduction**

With the advent of the Information Age, information has become extremely important. Information can assist people to make the correct decisions in business, political and social activities.

Data should be processed in order to turn it into information, so that it can reveal useful knowledge for people. Data can be obtained through different channels and can be stored in different forms. Different data sources have their own properties and follow different prototypes or standards. Not only the data organization structures but, also, the operation behaviors may differ significantly from each other. Strictly speaking, every data source is unique. Thus the problem is to make use of and to manage different data sources efficiently. The situation will be more severe for data-centered applications which must interact with heterogeneous data sources and which have a high probability of requiring frequent changes. Ill-designed data source management mechanisms will increase the system’s complexity in relation to both development and maintenance.

At the present time, there is a lack of a standardized data source management interface which can satisfy the demands associated with extensibility and heterogeneity. The motivation behind this research work is a desire to change this situation.

1.1 **Background and problem motivation**

Our research group has been working on an E-Health project. The project is senior-oriented. It is used to provide home healthcare. It can make elderly persons have much better information regarding their state of health without the need for frequent hospital visits. At the same time, it can assist them to adopt a healthier living style. The daily data regarding health situation can provide a reference for a doctor to make a diagnosis without the need for a home visit.

The E-Health system is built on an Android platform and can be applied to any smart phone which can support the Android system. The application interacts with a local database for user info management. User information includes, but is not limited to, personal information, system
custom settings, memorandum content and so on. The application should also have the ability to receive real-time data from sensors in order to obtain environment information or human physiological indicator data. It must also be able to connect to the internet and transfer data to a user’s or a doctor’s personal computer for further processing and data analysis.

To consider the potential demands, the E-Health system may extend its functions and become involved with more data sources in the future. Data source heterogeneity makes it a difficult task in relation to the system development and maintenance. The diversity of sensors makes it even worse, as a unified standard for sensors has not yet been achieved. Sensors of different sorts or different vendors may have distinct manufacturer specifications and follow different communication protocols, which makes it complicated for the E-Health application to adapt to a new sensor. When a data source changes, the data schema definition and data operation implementation details of the application must also make corresponding changes. The data source change occurs when a database breaks down, a base station collapses, a device is updated, a new function needs to be extended or merely if the user preference changes.

To address the data source heterogeneity problem, a middleware has been designed in order to take over data source management tasks. The middleware can reduce the complexity of the system by separating the problem. With the middleware, the data source would become transparent to application developers. They are then able to directly call for services from the middleware to accomplish data source interaction issues without being concerned with the concrete implementation of data source operations. As the middleware provided a standardized and unified data source interface for application developers, it is named heterogeneous data source middleware. The heterogeneous data source middleware is supposed to largely reduce the workload regarding system development and maintenance in relation to its flexibility and extensibility.

The heterogeneous data source middleware has been used so as to simplify the data source problem which emerged during the Android E-Health system development, but it is not only apply to the E-Health system. The middleware can be applied to any other applications which require data source interaction, for it does not contain any specific
business logic. The middleware is focused on providing services related with and only with data sources. As a result, the middleware has a high generality and reusability.

1.2 Overall aim

This research work is mainly aimed at solving the data source heterogeneity problem faced by data-centered application. A middleware solution is proposed to address this problem. The middleware should achieve the following goals:

(1) Be able to provide a generic and unified interface for data source interaction. The interface is supposed to act as a standardized language, but is much simpler and easier.

(2) Be able to separate data interaction from business logic and promote system modularization.

(3) Be able to push the application developer to adopt a better system framework.

(4) Be able to improve the extensibility of the application and thus ease the system maintenance.

(5) Be able to reduce the system complexity and programming workload.

(6) The middleware itself should be extensible and easy-to-use.

For the third goal, to encourage the developer to adopt a better system framework, it is said that a good framework should meet the following design principles:

“(1) Abstraction, Encapsulation, Information Hiding;
(2) Divide-And-Conquer, modularization;
(3) Coupling and Cohesion;
(4) Separation of Policy and Implementation;
(5) Separation of Interface and Implementation;
(6) Single Point of Reference.” [1]

The middleware is supposed to make the system framework easier to realize these principles.
1.3 Scope

This research work concentrates on the middleware architecture and interface design. It is similar to building a generic and easy-to-use framework rather than implementing an application. In this phase, the middleware will provide data source services required in the Android E-Health application and only include the related data sources. Each available data source will be tested by integrating the middleware into the E-Health application.

The middleware will place a special emphasis on extensibility. Not all data source categories will be added to the middleware and be tested. The final outcome of the research work will not cover all kinds of data source services and operations. But it is supposed to be extensible and be compatible with any potential data source. As long as the interface remains the same, an application developer will not be required to modify any source code when the middleware is updated.

1.4 Concrete goals

This research work is supposed to accomplish the following tasks:

(1) Conduct research about middleware, especially data source middleware, sensor network middleware and mobile application middleware. Summarize ideas and technologies with regards to previous research. Analyze the advantages and disadvantages for each solution. Determine what has been done and what is required to be done. The survey can assist in relation to making fewer detours and to perform less repeated work.

(2) Design the middleware interface. This not only indicates a declaration of service interfaces, but also involves designing a data transmission mechanism between middleware and application.

(3) Design the middleware architecture. It includes tasks such as abstraction, layer division and declaring relationships between each layer. At the same time, it must also provide a mechanism to handle data schema diversity.

(4) Implement the middleware and integrate it into the Android E-Health application. This ensures that the interface is being tested for a real scenario.
(5) Make an evaluation of the middleware. The evaluation can be mainly divided into two aspects. One is to evaluate the workload of the application development and maintenance using the middleware. The other is to analyze the influence regarding the system response time compared with the former application version.

1.5 Outline
The structure of this thesis is described below:

Chapter 1 introduces the background and motivation of the research work and represented both high-level and low-level problem statements.

Chapter 2 introduces several related middleware products and technologies.

Chapter 3 introduces the chosen methodology and methods to accomplish the research work. The described methodology and methods include modularization, middleware solution, OOP, abstraction and generalization.

Chapter 4 first briefly introduces an overview of our Android E-Health System. Following this, it introduces the design and implementation of the heterogeneous data source middleware in detail. To ease extension issues, the use of the Data Source Wizard was represented.

Chapter 5 provides a detailed evaluation of the middleware in relation to generality, easy-to-use, flexibility, SQL Injection and efficiency issues.

Chapter 6 summarizes the whole research work and points out future directions.
2 Related work

The Biomedical and Multimedia Information Technology (BMIT) Group in Sydney NSW University has cooperated with the Centre for Multimedia Signal Processing (CMSP) in Hong Kong Polytechnic University and proposed an intelligent middleware in 2005 [2]. This middleware is used for the dynamic integration of heterogeneous health care applications. It is composed of two layers, the data sources integration layer and the standard-based applications integration layer. The data sources integration layer integrates heterogeneous data source schemas into data source ontology and produces global API metadata, which deals with the data schema diversity issues. The standard-based applications integration layer takes advantage of the unified data model provided by the data sources integration layer in order to accomplish the communication and interaction between different health care applications. The middleware can simplify and standardize the integration and extension procedures of heterogeneous health care applications. However, this middleware is specific for health care application usage and is unable to be used for other purposes.

Karl Aberer, Manfred Hauswirth, et al have worked out a middleware named Global Sensor Networks (GSN) [3][4], which can provide efficient and flexible deployment and interconnection of sensor networks. This research work is part of “The Swiss NCCR-MICS initiative: Data Management for a Smart Earth” project. The project was motivated by the Swiss National Competence Center for Research in Information and Communication Systems and has been sponsored by the Swiss National Science Foundation. The GSN middleware is aimed at solving the software and hardware heterogeneity in sensor network technology. It proposed a new concept, namely a virtual sensor abstraction, which enables the extensibility and flexibility of the middleware. At the same time, it allows the user to declaratively specify XML-based deployment descriptors and makes it possible to integrate sensor network data through plain SQL queries over local and remote sensor data sources [3]. This mechanism can support dynamic adaption of system configuration during an operation. It can also support the flexible integration of sensor networks and the flexible discovery of sensor data. With GSN middleware, the workload of sensor network deployment and application
development will be greatly reduced. However, this middleware is specific to a sensor network and would prove difficult to extend to other data sources. Furthermore, it is more similar to an application than a general middleware.

Orange labs and LIG laboratory in France worked out a service oriented middleware, SStreaMWare [5]. SStreaMWare is designed to accomplish heterogeneous sensor data management. It created a declarative query language which can receive data streams generated by heterogeneous sensors. At the same time, it uses a mediator-adaptor approach to cope with data heterogeneity. It makes use of the natural hierarchy of geographic locations and, the middleware adopts a service-oriented framework to operate on a dynamic set of distributed sensors according to the query evaluation distribution change at different levels. SStreaMWare can use its generic query service to hide sensor software heterogeneity, but it is specific for sensors and is unable to handle other data source categories.

Takeshi Mishima and Hiroshi Nakamura in Tokyo University proposed an eager database replication middleware Pangea [6], which is able to guarantee snapshot isolation without any modification of database servers. They put forward a new correctness criterion called global snapshot isolation (GSI) to guarantee SI to clients and maintain consistency between database servers. They also chose to use the “first updater wins” rule to ensure the consistency between duplications. Using this first-updater-wins rule, it delegates a designated “leader” replica to detect conflicting write operations so as to regulate the order of conflicting write operations. This regulation can avoid deadlock. The middleware Pangea guarantees the GSI with key control and it can execute non-conflicting write operations concurrently using tuple-level locking. They have discussed the consistency issues under two scenarios, read-only and update. However, this work focuses on different duplications of a same database, while the work in this thesis is required to deal with different data sources which may have different properties, follow different standards and be organized in different data schema.

In 2010, M. Brito, L. Vale, P. Carvalho and J. Henriques put forward a sensor middleware which is used to integrate heterogeneous medical devices [7]. They indicated that the middleware solution has been developed to solve two increasingly relevant problems in modern health systems:
"i) to aggregate a number of heterogeneous, off-the-shelf, devices from which clinical measurements can be acquired and ii) to provide access and integration with an 802.15.4 network of wearable sensors" [7]

The sensor middleware is service oriented and is mainly used for data collection and processing. It separates sensor communication logic from the application so as to break down the complexity of the whole system. However, this middleware is designed specifically for medical devices and does not offer a good generality.

Christian Seeger, Alejandro Buchmann, Kristof Van Laerhoven have looked into three practical issues for wireless sensor networks in the wild after middleware deployment [8]. They proposed an event-driven middleware for body sensor networks, which is designed to seamlessly handle sensor configuration modifications. At the same time, they put forward three questions when deploying the middleware for a body sensor network, using commercial biosensors. The questions are listed as below:

“(1) How can the architecture cope with different levels of data fidelity and propagate those levels to the applications?
(2) What is the optimal way to handle temporary disconnections from sensors?
(3) How should the middleware implement sensor-specific peculiarities?” [8]

The middleware is hosted by a smart phone and allows day-long user monitoring. For the three questions, they have introduced some solutions based on the middleware and healthcare application scenario. To deal with data fidelity issues, the middleware can provide two different types of sensor readings for the same sensor, namely, raw sensor events and weighed sensor events. Weighed sensor events not only contain the original sensor data but also an additional fidelity level, which allows the application to discard inaccurate sensor readings. To deal with temporary disconnections, the middleware adopts a buffering mechanism. It collects and buffers old sensor readings. When the network connection is re-established, it will deliver them as events to the application. To deal with sensor-specific peculiarities, the middleware proposed a new concept, namely, Sensor Module. A Sensor Module is an abstraction of an individual sensor and acts in a similar manner to a driver for each individual sensor. Developers can use a Sensor Module
to translate raw sensor readings to sensor events. Additionally, with the abstraction mechanism, the application developer does not require to make any change to the application when changing a sensor. It only needs to change the corresponding sensor module. This research work provided good advices and guidance for sensor network application developments, but it does not build a unified and standardized interface for heterogeneous data source.

As listed above, this chapter has introduced some middleware productions and technologies. Though middleware solution is not new in relation to a sensor network or for data source management, as far as is known by the author, there has not been a generic middleware which is able to provide a standardized interface for all data sources. As data source numbers grow and heterogeneous data source management becomes more and more complex, there is an urgent demand for a generic standard or framework to manage heterogeneous data sources when developing data-centered systems. The aim of this work is to meet this require. This thesis will introduce a heterogeneous data source middleware which is designed to allow an application developer to use a unified interface to interact with any data source as is required.
3 **Methodology and methods**

Using appropriate methodologies and choosing corresponding methods is the first step and the key to project development. This chapter will introduce the methodology and concrete methods used to accomplish this research work.

To break down the complex problem of E-Healthcare Application, modularization will be applied to the whole system. To solve data source diversity and extensible issues, a middleware solution is proposed for heterogeneous data source management. A heterogeneous data source middleware will be designed and implemented in order to provide data source services for an application. At the same time, the middleware solution further emphasizes system modularization. As modularization might be defined as the natural predecessor of object oriented programming framework, it is natural to adopt object oriented thinking and utilize OOP so as to accomplish the middleware. At the same time, as the middleware should provide different kinds of data source services, abstraction should be processed to ensure both extensibility and maintainability. Meanwhile generalization is applied for concrete data source service implementation.

3.1 **Modularization**

Modularization is a software design technique. It refers to a logical partitioning of the software system, which may be based on data links, related functions, implementation considerations or other criteria [9].

Modular programming is also called "top-down design" or "stepwise refinement". [10] It emphasizes separating the functionality of a program into independent, interchangeable modules. In this way, each module contains everything necessary to execute only one aspect of the desired functionality.

"Conceptually, modules represent a separation of concerns, and improve maintainability by enforcing logical boundaries between components. "[10]

As a result, it can simplify complex issues. This also facilitates the "breaking down" of projects into several smaller projects, which utilize
"divide and conquer" thinking. The advantage of modularization is obvious. As stated in [11]:

"Several programmers can work on individual programs at the same time, thus, making development of program faster. The code base is easier to debug, update and modify. It leads to a structured approach as a complex problem can be broken into simpler tasks. Theoretically, a modularized software project will be more easily assembled by large teams, since no team member is required to know about the system as a whole. They are thus able to focus merely on the assigned smaller task. “[11]

To realize modularization, this E-Health project has been divided into different layers, with each layer being for different modules. Most importantly, the middleware solution for heterogeneous data source is also deals with modularization considerations. Details of the layered structure of the E-Health system will be discussed in the next chapter.

3.2 Middleware Solution

The earliest approach of software development merely involves programming and writing code. The biggest drawback is that the majority part of the codes is unable to be reused. After this, the component-based software development method was proposed. It packages common programming features, and provides standardized and unified interfaces which can be invoked by other programs. With the component-based software development approach, it is possible to achieve system modularization. To put this development approach into practical use, middleware is the key enabling technology. Middleware is a kind of computer software that connects software components and applications.

"Middleware is an independent program of software or service. It not only achieves interconnection, but also realizes interoperability between different components in the application. It consists of a set of enabling services that allow other program processes to interact with.”[12]

Nowadays, middleware is widely used in a variety of large-scale projects. It provides services, supports application functions, separates concerns, and integrates components. Both the "polymorphism" of an application and the complexity of system software must rely on the various types of middleware, which play a role in the logistics and mediation.
The advantage of using middleware is obvious. The program interface of the middleware creates a relatively stable high-level application environment. Regardless of how the bottom computer hardware and system software changes, the application software requires almost no modification, as far as the middleware updated while the middleware external interface definition unchanged. This property can assist enterprises to reduce the significant investment based on the application software development and maintenance.

It is obvious from the above that the middleware solution is a good choice to achieve the goal in this case. The main task for the author in relation to this research is in designing and implementing a heterogeneous data source middleware which can provide different kinds of data source services. The middleware is supposed to assist other team members and, the application developers, in order to build an elegant framework which is loose coupling and high cohesion. It is supposed to make the whole E-Health system easier to develop and maintain. In addition, this middleware can also be applied to other applications based on its generality and reusability.

### 3.3 OOP

Object Oriented Programming is a programming framework [13]. A basic principle of OOP is that a computer program consists of several units/objects. Each unit/object can play a role by itself in a subroutine. OOP is able to achieve three main objectives of software engineering, which are reusability, flexibility and extensibility. In order to implement the overall computing, each object can receive information, process data and send information to other objects.

In OOP, an object (usually instances of a class) is the foundation of modularization and structuralization. An object is a unit which consists of data and methods together with their interactions in the running program. OOP includes features such as data abstraction, encapsulation, polymorphism and inheritance, which distinguish it from traditional process-oriented programming. Data abstraction indicates that the program has the ability to ignore certain aspects of processed information, namely, the ability of focusing on the main aspects of the information.
Encapsulation ensures that one object will not change the other objects’ internal state in unpredictable ways. Each type of object provides an interface to contact with other objects, and sets the rule about how the interface is to be invoked. Polymorphism is the ability to create a variable, a function, or an object that has more than one form.[13]

“The purpose of polymorphism is to implement a style of programming called message-passing in the literature, in which objects of various types define a common interface of operations for users.”[14]

Inheritance allows for the establishment of a subtype from an existing object and is a way to reuse the source code of existing objects. “It unifies and enhances polymorphism and encapsulation.”[15]

OOP makes programming closer to the actual world. It gives every object attributes and methods and, as a result, it makes programming more humane. To adopt an object oriented programming framework, the implementation of heterogeneous data source middleware uses Java as the programming language.

“Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers ‘write once, run anywhere’, meaning that code that runs on one platform does not need to be recompiled to run on another.”[16]

Java can formally support the module concept as packages are considered to be modules in the JLS. In addition, Java has the property of platform independence. Java classes can be easily exported as jar package, which can be imported into projects programmed by other languages, such as C, C#. As a result, the middleware can also be applied to other applications with slight changes.

### 3.4 Abstraction

Abstraction is a common methodology used in many fields. Generally, it is a process involving a mental distraction of properties and relations of phenomena. At the same time, researchers mark out the properties of interest, which are first of all, essential and general properties of a phenomenon [17]. The most important question of abstraction is clarification, i.e. what properties are essential and what properties are lateral. This question is decided first of all in each research work. It depends on
the nature of the subject under study, as well as the specific objectives of the study [17].

As described in last section, data abstraction is one of the remarkable properties in OOP. Java uses abstract class and java interfaces in order to realize the abstraction of objects.

An abstract class [18] is often used to characterize the abstract concepts when analyzing or designing the problem area. It abstracts a series of specific concepts. Though these concepts may appear to be different, they are essentially the same. An abstract class is used for type hidden. It is possible to construct an abstract description of a fixed set of behaviors, which may have all kinds of possible performance. This abstract description is an abstract class, and all the possible performance can be viewed as subclasses of the abstract class. There is an obvious advantage to limiting the application to only operate with the abstract class. The application depends on a fixed abstraction, so it cannot be modified. While on the other hand, the behavioral function of the application can be extended by implementing specific subclasses of the abstract class.

As the heterogeneous data source middleware need to support different kinds of data sources and may be extended frequently, an abstract class is designed to hide the differences between each data source. Detail will be introduced in the next chapter.

An interface in Java is a series of function declarations and a collection of certain behavior properties. An interface only contains the declaration of the function without any implementation. As a result, these functions can be implemented in different classes, and can have different performances. The difference between the abstract class and interface is that the abstract class can also contain its own implemented functions. In addition, a class can only inherit from one parent class, while it can inherit from multiply interfaces. The use of an interface can provide the project with both diversity of classes and flexibility of structural combinations. Furthermore, when new functions are required to be added to a certain feature, the developer only needs to modify the interface.

Data source can be classified according to practical usage, inherent properties, access methods or other ways. The data sources available in middleware can be divided into several categories, such as local database, remote database, active sensor, passive sensor and so on. Data
sources of the same category may have several common attributes and behaviors. It is obviously a good choice to create an interface for each data source category. In relation to java interfaces, they can provide a well-formed framework for the middleware. At the same time, it can make the middleware easier to maintain. The details will be introduced in the next chapter.

3.5 Generalization

In OOP, generalization is the relationship between the general description and the specific description of object. “Specific description is based on general description and extends general description.”[19] Generalization will be realized when implementing specific data sources. Each data source class will be derived from the abstract data source class and will implement certain interfaces of the corresponding data source category.

There are two important OOP properties related to generalization.

One is inheritance, which enables the addition of a self-defined description under the premise of sharing an ancestor part. The shared part of the description is supposed to be declared only once and be shared by many classes instead of being redeclared in each class. This sharing mechanism reduces the size of the model. More importantly, it reduces the modification workload and unexpected inconsistent definitions caused by a model update.

The other is polymorphism. Whenever the ancestor is declared, an instance of posterity can be used. A base class can have many subclasses. Each subclass can implement different behaviors for the same operation. A variable is declared as the base class, then an instance of any derived class can be used, each one having its own unique behavior. This rule is particularly useful. A new class can be added without changing the existing polymorphic.

The detailed implementation of subclass will be introduced in the next chapter.
4 Design and implementation

This chapter will provide a brief look into the Android E-Health application and its layered structure, in which the middleware serves as the data access layer. Following this, the design and implementation of the heterogeneous data source middleware will be described in detail. Finally, a GUI called Data Source Wizard will be represented, which is used to ease the middleware extension work.

4.1 Android E-Health application

To reduce medical equipment expenditure and to simultaneously arise the health level, e-healthcare has already become a hot topic in both the health service and computer science fields. Our team has been working on a senior-oriented E-Health application [20] for a considerable time. It is aimed at providing home healthcare and self-monitoring for senior citizens.

As the mobile phone penetration rate in European countries is rather high, this E-Health application was designed as a mobile application. The application is supposed to be applied to smart phones, so that it will be easy to use and convenient to carry about. The Android platform was chosen to develop the application based on its high popularity and open-source property. Android [21] is an open-source operating system for portable devices. Additionally, it can be supported by various third-party plug-ins and apps. This will ensure the practicability of this product and ease any development efforts. As a result, it can be set up in any smart phones which are available with Android. By using the E-Health system, senior citizens can adopt a good living style and maintain a good health without the necessity of frequent hospital visits.

4.1.1 Main Functions

As shown in Figure 1, the E-Health application is composed of four main parts, the user settings module, the record management module, the alarm management module, and the acoustic control module. This section will introduce each module of the application.
The user settings module is related to the user information and custom settings. It ensures user privacy by means of registration and login mechanisms. The user is required to create a unique account and login using his/her own account before starting the E-Health service, which makes sure that only the user can gain access to his/her own data. The module also provides the function of environment settings to enhance the user experience. Available setting factors include a brightness setting, background setting, font size setting, etc. The custom setting information is supposed to be saved in the local phone memory to enable quick access, while the basic user information is supposed to be saved in database located in some kind of base station in order to enhance security.

The record management module is used to receive and store human physiological indicator data, which is collected by wireless sensors, into the database. The human physiological indicator data can be blood pressure, heart rate, blood glucose, and so on. During this phase, these data are supposed to be entered manually because of a lack of devices. At the same time, the record management module is responsible for showing the requested data. In order to display the daily health status vividly, the data should be able to be shown in both list and histogram formats. The human physiological indicator data should be stored in the phone memory to ensure quick access and also because it must be able to be transmitted to more powerful a computer to perform further analysis.
The alarm management module is mainly focused on the daily schedule alarms setting in order to assist the aged to achieve a more regular and healthy lifestyle. At the same time, the user can also set a note in order to assist in memorizing important issues. The note is to be saved in the local phone memory.

The acoustic control module is used to provide user-friendly functions. As many of the aged have bad eyesight, the mobile phone screen may be too small for them to switch and choose the required function. However, with voice recognition, the aged user can merely speak out the function name, for example, “setting”, “alarm”, etc. and the application would switch to the demanded GUI automatically. To implement this part, the voice recognition application provided by Google has been used.

4.1.2 Layered structure

The former version of this application was focused on the availability of each function. However, this version placed all the source files together in one package and all the source code for a certain function in one single Java source file. The mix of presentation and business logic makes the source code difficult to read. As a result, it causes any application modification and maintenance to be a complex task.

As introduced in section 4.1.1, the application must be able to interact with different kinds of data sources, such as local databases, remote databases, real time sensors and so on. The data source diversity causes the application implementation and maintenance to become very complicated and somewhat difficult to accomplish. To deal with the potential problem of data source alteration, the data source management part is further separated from the application to a middleware, which is called heterogeneous data source middleware. The middleware provides a unified interface and shields the diversity of data sources from the consumers. The middleware consumer can be a specific application or another data source. In this case, the middleware is integrated into the application in order to support data source services.

When integrating the middleware into the application, another part was conducted so as to make the system structure more organized and easy to manage. Hierarchical design is adopted to divide the whole E-Health system into several layers. The business logic part was separated from the presentation part. As a result, the system becomes a three-layered
architecture, which includes a presentation layer, business logic layer and data access layer, as shown in Figure 2. The presentation layer concentrates on data representation. It arranges the layout of the graphic components and decides how to present the data which has been received from the lower layers. The presentation layer does not need to be involved with regards to how the system works. It is only required to obtain information from the business logic layer and present them in a proper and elegant manner. The business logic layer concentrates on the business logic processing. It obtains data from the data access layer and implements the core processing algorithms. The data access layer concentrates on the basic operation of data source. It deals with how to gain access to the data source, how to obtain the desired data from the data source, how to save new data into the data source, how to delete designated data from the data source and how to update designated data in the data source. In this system, the heterogeneous data source middleware plays the role of the data access layer. With the layered structure, an appropriate developer can be assigned to be responsible for one specific layer. As a result, a developer can become more focused on one particular area and thus concentrate on that part without having to deal with other matters.
Figure 2: Relationship of each part in the Android E-Health System.

To make the source code clearer and easier to maintain, a different layer has been located in a different package, as shown in Table 1. The source files of the presentation layer are included in package \texttt{com.hyl.alarm}. This part is almost the same as for the former version with the business logic part being excluded. The business logic layer source files are included in the package \texttt{com.bll.*}. There are two main business logic objects, namely, user and record. The user business logic is used to accomplish the user settings module and the record business logic is used to accomplish the records manage module. The main business logic functions for each object are shown in Table 2. The part of data access layer is included in package \texttt{com.middleware.*}, which contains the whole middleware.

Table 1: Package structure of E-Health application.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presentation Layer</td>
<td>\texttt{com.hyl.alarm}</td>
</tr>
<tr>
<td>Business Logic Layer</td>
<td>\texttt{com.bll.*}</td>
</tr>
<tr>
<td>Data Access Layer</td>
<td>\texttt{com.middleware.*}</td>
</tr>
</tbody>
</table>

Table 2: Business Logic Layer structure.

<table>
<thead>
<tr>
<th>Business Logic Object</th>
<th>Package</th>
<th>Business Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>\texttt{com.bll.users}</td>
<td>UserLogin, UserRegister, UserBean</td>
</tr>
<tr>
<td>Record</td>
<td>\texttt{com.bll.records}</td>
<td>AddRecord, DeleteRecord, UpdateRecord, ShowRecords, RecordBean</td>
</tr>
</tbody>
</table>

As this is the core of this paper, the detailed design and implementation details of the middleware will be introduced in the next section.

### 4.2 Heterogeneous Data Source Middleware

The middleware is composed of multiple data source service wrappers. Each data source service wrapper represents one particular data source and is responsible for implementing basic data source operations and
providing relevant service to the consumers. To cover up the difference between each data source, the middleware adopted an abstraction mechanism and provided a unified interface. As a result, generality and extensibility has been achieved. To relieve the limitation of parameter type and number, XML technology was used to transmit data between the middleware and the application. At the same time, a global-local data schema mapping mechanism is used to eliminate data schema diversity.

This section will introduce details of the heterogeneous data source middleware.

**4.2.1 Architecture**

The heterogeneous data source middleware is responsible for providing the application with all the services related to data operations, for instance, to select the required data from the data source. The application should guarantee that it is able to implement all the data interaction functions by using the services offered by the middleware. At the same time, the middleware must also interact with a lower layer, namely, the data source layer. It should be able to gain access to the described data source and obtain the required data through its native interface. The basic construction of the middleware is shown in Figure 3.
Figure 3: Heterogeneous data source middleware architecture.

In relation to the demand regarding data sharing, data replication, etc, data sources can also communicate with each other through the middleware as shown in Figure 3. For instance, if the data source of the user information had to be changed to another database, and the administrator wanted to retain the existing user information, he/she would have to make a copy of the old one to the new one. This data migration job can be performed using the middleware by implementing the corresponding interfaces.

The main component in the middleware is the data source service wrapper. The middleware contains various kinds of data source service wrappers. As shown in Figure 3, a data source can be a local database, a remote database in the base station or in a cloud platform, or even a real time sensor. The implementation details relating to the accessing and managing of a particular data source is wrapped into a specific data source service wrapper. With the data source service wrappers, the application layer can make use of the service provided by the middleware without the need to know how it works.

4.2.2 Data source service wrapper

The Data Source Service Wrapper is the basic and most important unit in the middleware. As its name indicates, the data source service wrapper wraps data source details and provides services for middleware consumers. Each data source service wrapper stands for one specific data source. Only the interface is exposed to the consumers and the implementation details are hidden. The wrapper shows what they are able to do, what service they are able to provide, but not how they should do it.

As described in section 3.4, an abstraction mechanism is used to accomplish generality and extensibility of the middleware. An abstract class AbstractDSService(Abstract Data Source Service) has been declared to standardize interfaces for different data source service wrappers and hide the data source diversity from middleware consumers.

As shown in Figure 4, AbstractDSService provides four interfaces. The setXMLHandler() interface is used to set a XMLHandler for the data source service wrapper. Details about XMLHandler will be introduced in latter sections. The setDataObjectName() interface is used to inform
the data source service wrapper which data object to work on. A data object stands for a particular table in the case of a database. The `connect()` interface is used to connect to the required data source. The connection behavior can differ from one to another. The interface will shield the differences from the middleware consumers. The `chooseOption()` interface is used to run a desired data source operation, which can be “add”, “delete”, “update” or “select”. The behaviors of each operation can also be different in different data sources and the differences are also hidden by the interface.

The abstract class only provides standardized and unified interfaces. It does not accomplish concrete data source behavior implementations. Different behavior of the data source connection and the basic data source operation is implemented in the subclasses of AbstractDSService. Each subclass stands for a specific concrete data source service wrapper. It is responsible for the implementation of the data source connection and the data source operation details. The subclass will handle data source specific properties and can have its own behaviors. As can be seen from Figure 4, both SQLiteService and MySQLService are able to save data to the data source, while the TemperatureSensorService cannot.

Figure 4: Functions in AbstractDSService and its subclasses (the parameters are omitted for the space).
To hide data source diversity, middleware consumers are only able to create an AbstractDSService variable. They can use the constructor of the abstract class to obtain a specific data source service wrapper instance and assign the instance to the AbstractDSService variable. However, they will not know of the existence of any concrete subclasses, nor have the ability to create variables for them. As a result, as long as the abstract class remains unchanged, the application developer will not have to make any changes to their work when the middleware is updated. This ensures the generality and extensibility of the middleware.

As mentioned in section 3.4, some specific Java interfaces have been declared in order to classify the data source categories. A Java interface only contains the declaration of the function without any implementation. It defines the common features for each data source category. With a Java interface, the middleware extension procedure will become more standardized. It will be much clearer regarding what functions require to be implemented when adding a new data source service wrapper, implementing the corresponding Java interface. Furthermore, when new functions are required to be added to a particular data source category, the middleware maintainer only needs to modify the interface.

Figure 5 shows the relationship between each Java class and Java interface in the middleware. The classes between the abstract class layer and the interface layer are subclasses of the abstract class, AbstractDSService. They inherit the functions declared in the abstract class, but they are not responsible for detailed data source operation work. These classes are used to accomplish auxiliary work in order to get data source parameters and instructions ready. They are responsible for, to name a few, parsing the parameters in the Temp.xml, exchanging the results from the data source into the required format, obtaining the configuration information from the DataSource.xml to prepare for a data source connection, etc. XML files will be introduced at a later stage. They call functions declared in the Java interfaces to deal with custom requirements. The functions are implemented in the classes of a lower layer. These implementation classes are responsible for the detailed accomplishment of data source interaction.
As the management of a temperature sensor, such as obtaining real time data from the sensor, has not yet been implemented, the application does not provide the environment monitor function in the current phase. However, as can be seen from Figure 5, the interface for the sensor data source and the temperature sensor wrapper has been designed in advance. As a result, the interface is only required to be implemented when the temperature monitoring service is required in the application.

4.2.3 Data transmission

The abstract class has defined interfaces for the middleware. However, it does not have a defined data transmission mechanism between the data source service wrapper and the middleware consumer. XML was chosen to serve as the medium of data transmission for its platform independence. A temporary xml file called Temp.xml is created to achieve this goal. Temp.xml is used for two situations. One is to deliver consumer parameters to the middleware and the other is to deliver query results to the middleware consumer.

In the first case, in which the xml file serves as the input, the role of Temp.xml file in the whole system is shown in Figure 6. The xml file is composed of key-value pairs. The key is recorded in the tag and presents the parameter name. The value is recorded as the content of the tag, and presents the parameter value. One function may contain more than one parameter, so the xml file can also record multiply key-value pairs, which means it can contain different tags. With key-value pairs, the middleware can obtain the data source service input. However, in the case of “select”, the middleware must, additionally know the output
parameter information. Thus, the middleware consumer must also provide output parameter information to the end of input info. In Temp.xml, to record output parameter info, the tag will be “output” plus a suffix which indicates the output order. The output parameter name will also be recorded as the content of the corresponding tag. The Temp.xml structure will be in the pattern of repeated “<key>value</key>” plus repeated “<outputindex>outputKey</outputindex>”. The “<outputindex>outputKey</outputindex>” part is only required when “select” service revoked. Figure 7 shows one example of Temp.xml, which indicates the obtaining of the userId of the one with the username “lu@test.se” and password “111111”.

```
<?xml version="1.0" encoding="UTF-8"?>
  <table>
    <column>
      <username>lu@test.se</username>
      <password>111111</password>
      <output1>userId</output1>
    </column>
  </table>
```

Figure 7: Example of Temp.xml as input.

In the second case, in which the xml file is served as output, the role of Temp.xml file in the whole system is shown in Figure 8. Data adding, deleting and updating services only need to feedback a BOOL value to inform the middleware consumer whether or not the operation has been successfully completed. However, for the “select” operation, the middleware must perform an additional task, to transfer the query results. The query results are recorded in Temp.xml. The root node of Temp.xml is “table”, and it can contain multiply child nodes called “column”. Each node “column” and its content stands for one record of the query result. Each child node of the node “column” stands for one attribute of the
recode. The value of the node is the column value. Figure 9 shows an example of Temp.xml, which is the result of selecting all the human physiological indicator records.

![Diagram of Query result transmission]

Figure 8: Query result transmission.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<table>
  <column>
    <record_id>1</record_id>
    <testname>Blood Pressure</testname>
    <testvalue>88</testvalue>
    <testdate>2012-5-21 18:26</testdate>
    <testnote>test</testnote>
  </column>
  <column>
    <record_id>2</record_id>
    <testname>Heart Rate</testname>
    <testvalue>86</testvalue>
    <testdate>2012-5-23 13:50</testdate>
    <testnote>test</testnote>
  </column>
  <column>
    <record_id>3</record_id>
    <testname>Blood Pressure</testname>
    <testvalue>56</testvalue>
    <testdate>2012-5-23 14:58</testdate>
    <testnote>t</testnote>
  </column>
</table>
```

Figure 9: Example of Temp.xml as output.

### 4.2.4 Data source localization

As described above, the middleware is supposed to become connected to the desired data source using the connect() function. However different data sources require different configuration information to make the connection. Even for a certain data source, there is a high possibility that the basic configuration information will change over time, e.g. the DBA may change the password of a particular user in a remote database to enhance data security, and the URL of the Web Service may become unavailable for a variety of reasons and thus require changing. To support rapid deployment, the configuration properties of data sources
should be provided in a declarative deployment descriptor, which can make the configuration information be directly perceived.

At the same time, as different data sources of the same object not only can have different content, but also different data schemas, this will raise the problem of data format incompatibility between the application and the middleware, and also among different data sources. To eliminate data schema differences, a data mapping mechanism is applied for each particular data source and application. As the amount of data sources grows rapidly and each data schema may change over time, it will be complicated and time consuming to make a one-to-one mapping. To solve the problem, a unified virtual global schema is asked for in order to shield the heterogeneity of different data sources, while retaining each data source's native schemas locally. The global data schema should be defined by the middleware user. It is often related to the business logic of the particular application. It will also be the middleware developer’s duty to build the global data schema.

In this middleware, XML was chosen to be used for the configuration and data mapping. XML is a markup language used to mark the electronic file and make it structural. It can be used to tag data and define data types. XML provides a unified approach to describing and exchanging structured data independently from the application or vendor. It has the flexibility for representing multifarious heterogeneous data. The simplification of XML makes it easy to read and write data in any application. Therefore, XML is the best choice to serve as the declarative deployment descriptor of the data source configuration information and global-local data schema mapping information.

An XML file named DataSource.xml will be created to record the configuration information for each available data source in the middleware. In DataSource.xml, the element <datasource> is the basic element. Each element <datasource> contains configuration information for a particular data source. Every element <datasource> has an attribute named “id”. The attribute “id” records the unique identification of the data source, which is presented as the DataSourceID in the application. Each child element of the element <datasource> stands for one parameter of the data source configuration information, which can differ from data source to data source. Figure 10 shows an example of a DataSource.xml. It can be seem from the example that each data source can have its own tags for specific configuration information.
To shield the data schema diversity, an XML file must be created separately for each data source in order to support global-local data format transformation of a particular data object. A naming rule has been made in this case in order to distinguish each mapping file. The stem of the file name is “DataMap.xml”. The name of the data object and the name of the data source will be the prefix of the data mapping file name. Thus the whole file name will appear as [DataObjectName] + [DataSourceName] + “DataMap.xml”. As a result, the file name itself contains the data source and data object information. The data source service wrapper can obtain mapping information from the correct xml file according to the data object and data source. The format of the mapping file is one tableName-localTableName pair and a repeated globalColumnName-localColumnName pair with local column type information. Figure 11 shows an example of a mapping file. The content of the element <tableName> is the local data object name. For each global column name element, the content is the local column name. At the same time, a tag with the global column name as a prefix and “Type” as the stem will be created to record the local column type info.
Figure 11: Example of data schema mapping XML file

Figure 12 shows the location of the DataSource.xml and mapping files in the whole system. A data source service wrapper can obtain the required configuration information from DataSource.xml by means of the DataSourceID, which is the unique identifier for each data source. At the same time, with the DataSourceID and pre-specified data object, a data source service wrapper can obtain mapping information from a corresponding mapping xml file in the mapping file pool. In this way, when the data source configuration information changes, the middleware will not be required to make any changes to its source code and need only to modify the corresponding value in the DataSource.xml. The incompatibility of data schema in different data sources has also been solved by the mapping files. Obviously, the using of a DataSource.xml and data schema mapping file makes a significant contribution towards achieving middleware generality.

![Fig12](image.png)

Figure 12: Data source localization module.

### 4.2.5 XML Handler

To reduce the workload of the application development with the middleware, the middleware provides an XMLHandler module to manage the xml creating and parsing tasks. The XMLHandler declared a series of interfaces for middleware users. However, the amount and type of parameters the user want to write or read will differ from one to another. It will thus prove difficult to unify the interface. The factory model does not work as there are too many different possible and potential demands. To solve this problem, a data structure named XMLData has been designed. The data structure is shown in Figure 13. XMLData
contains two fields, `columnName` and `valueList`. The `columnName` field is used to record the parameter names, which will be turned into tags in the xml file. The `valueList` field is used to record the parameter values, which will be turned into the element’s content in the xml file. Each string cell in the `columnName` field list is mapped with one `XMLDataFormat` cell in the `valueList` field list. `XMLDataFormat` is a string list, which indicates that one column can contain multiple values. This property is useful when dealing with query results, while one input parameter will, often, only occupy one cell in the `XMLDataFormat` list.

![XMLData Structure](image)

**Figure 13: XMLData Structure.**

XMLHandler can make use of a `XMLData` variable to simplify and unify the interfaces. An `XMLData` variable must be constructed for the parameters or query results which need to be written into the Temp.xml. The main functions of the XMLHandler class are shown in Table 3. The `readXMLData()` function and the `writeXMLData()` function are used in data transmission between the middleware and the application. The `readXMLData()` function reads data from the Temp.xml into a `XMLData` variable. The `writeXMLData()` function writes a specific `XMLData` variable into the Temp.xml. The `parseDataSourceXML()` function is used to obtain the data source configuration information from the `DataSource.xml`. It will return the desired configuration information in Properties format according to the input `sourceID`. The `getMappingProps()` function is used to obtain the predefined global-local data schema mapping information from a particular mapping xml
file. It takes the mapping xml file name as its input and will return the desired mapping information in Properties format.

Table 3: XMLHandler functions.

<table>
<thead>
<tr>
<th>target</th>
<th>function</th>
<th>input type</th>
<th>output type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temp.xml</td>
<td>readXMLData()</td>
<td>XMLData</td>
<td></td>
</tr>
<tr>
<td></td>
<td>writeXMLData()</td>
<td>XMLData</td>
<td></td>
</tr>
<tr>
<td>DataSource.xml</td>
<td>parseDataSourceXML()</td>
<td>String</td>
<td>Properties</td>
</tr>
<tr>
<td>Mapping files</td>
<td>getMappingProps()</td>
<td>String</td>
<td>Properties</td>
</tr>
</tbody>
</table>

To be noticed, the DataSource.xml and data mapping xml files should be permanently stored and are required to be modified and managed manually according to particular demands. As a result, there are no interfaces for the DataSource.xml and the mapping file creating and writing. Unlike the case for the DataSource.xml and data mapping xml files, the Temp.xml however, must be deleted immediately after it has been parsed. This is because the Temp.xml is merely used for data transmission. As long as the parameters or query results have been received by the middleware or the application, its job is completed and there is no longer any necessity for it to be retained. Thus, at the end of the readXMLData() function, file deleting will proceed.

4.2.6 Integration

As mentioned previously, in order to shield the properties of each data source and ensure extensibility of the middleware, the application programmer is only able to operate with the AbstractDSService variable. An application programmer will never have the knowledge concerning about details of the data source being used. Figure 14 shows the interaction pattern between the application and the middleware. The application must supply a DataSourceID so as to obtain a specific data source service wrapper instance and assign this instance to an AbstractDSService variable. With the AbstractDSService variable, the application can obtain the demanded service by setting a specific OptionID and a particular data object name. The OptionID can be “add”, “delete”, “update” and “select”. The data object name is the global one defined by the application developer. The middleware will perform the required operation on the certain data object according to the OptionID and the data object name. All the function parameters and data query results will be transferred by xml files as previously discussed.
At the present stage, after integrating the middleware into the E-Health application, each data source related function should be assigned with a particular data source in advance in a declarative descriptor, which can be a XML file or a properties file. In this case, the xml file “res/values/strings.xml” was chosen to store the data source responsibility configuration information. The strings.xml is generally used as a string resource definition file in the Android applications. Figure 15 shows the relationship between each part of our Android E-Health System after it has been integrated with the middleware.

To make the interaction procedure more clear, an example is now provided regarding the use of the middleware to implement the login...
function of the E-Health application. Figure 16 presents a fragment of
the code implementing the login function in the business logic layer of
the application. To make the code easier to read, parameter wrapping
part is replaced by the XMLHandler.createXML() and the query result
parsing part is replaced by an XMLHandler.parseXML(). Meanwhile,
the configuration information in res/values/strings.xml should be
"<string name= "loginService">sqlite</string>".

```java
XMLHandler.createXML("username", username);
AbstractDSService dsService =
    DataSources.chooseDataSourceService( getString( R.string.loginService ) );
    dsService.setDataObjectName("Users");
    dsService.connect();
    dsService.chooseOption( "select" );
    String correctPsw = XMLHandler.parseXML();
    if( correctPsw == NULL )
        msg = “username doesn’t exist!”;
    else if( correctPsw.equals(password))
        msg = “login successful!”;
    else
        msg = “wrong password!”;
```

Figure 16: Login application code example.

### 4.2.7 Extending

It can be seen from the login example in section 4.2.6 that when the
application developer wants to change the data source for a given func-
tion, he/she need only to modify the corresponding value in the declara-
tive descriptor, which is used to assign a specific data source to each
function. The application developer does not require to perform any
modification of the source code. As long as the required data source has
already been included in the middleware, the application developer is
able to continue using the current middleware version. Otherwise, the
middleware must be updated by the middleware maintainer.

To extend the middleware to provide a new demanded data source
service, the middleware maintainer should implement a new wrapper
for the data source. The data source service wrapper should inherit the
AbstractDSService class and implement the corresponding Java inter-
faces. At the same time, new data source connection information should be
appended to the DataSource.xml. The global-local data schema mapping files must also be created for the required data objects.

Although the extending work follows a given pattern, it may cause problems. The middleware maintainer may forget to add the necessary data source configuration information into the DataSource.xml or may create a data source service wrapper in an incorrect manner which breaks the abstraction rule. To make the extending work more simplified and standardized, a small application called the Data Source Wizard was designed in order to guide the middleware maintainer regarding the addition of a new data source. Details will be introduced in the next section.

4.3 Data Source Wizard

In order to simplify and standardize the extending work when adding a new data source to the middleware, the intention is to create a GUI which serves as a project or framework Custom Wizard in some integrated development environments, such as eclipse and visual studio. This is called the Data Source Wizard as it is used in assisting to maintain the heterogeneous data source middleware. It is supposed to reduce the workload regarding the middleware maintenance.

4.3.1 Overview

As previously described, when the middleware maintainer wanted to add a new data source to the middleware, he/she should implement a new data source service wrapper and create data mapping files for a particular data object. The data source service wrapper will be a subclass of the AbstractDSService. The data mapping file will contain the mapping information from the pre-defined global data schema to the local data schema. After this, the wrapper and mapping file should be added into the middleware. In addition, the data source connection information should be appended to the DataSource.xml.

To make the procedure of adding data source easier and ensure that the middleware maintainer will not omit any of the components, the wizard should provide the functions described below:

(1) guide the developer to fill in the demanded configuration information of the data source (to append to DataSource.xml);
(2) guide the developer to fill in the information required to create data schema mapping file;

(3) automatically create the basic framework of data source wrapper and generate general methods which must be implemented in certain classes.

To sum up, the Data Source Wizard should assist in building the components list in Table 4. The DataSource.xml records the configuration information of all data sources available in the middleware. The mapping xml file records the mapping information of the global data schema and the local data schema. The service class is used to accomplish some auxiliary functions, such as parsing parameters, packaging query results, useful variable initialization and so on. The implementation class is used to accomplish the basic operations of the data source.

Table 4: The list of files which Data Source Wizard should create.

<table>
<thead>
<tr>
<th>Specific Files</th>
<th>Default Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSource.xml</td>
<td>[basepath]\MiddlewareXML</td>
<td>Data Source Configuration Info (database connection info...)</td>
</tr>
</tbody>
</table>
| [DataObjectName][DataSourceName]DataMap.xml | [basepath]\MiddlewareXML                         | global data schema--
l>local data schema prefix: DataObjectName + DataSourceName |
| [DataSourceName]Service.java | [basepath]\src\com\middleware\DataSourceService | provide interface to App developer; deal with XML file, variable initialization... |
| [DataSourceName]Impl.java | [basepath]\src\com\middleware\DataSourceServiceImpl | Implementation of data source operations                                    |

From all of the files, the mapping xml file, the service class and the implementation class should be newly created. However, if the DataSource.xml of the middleware already exists, it need only append the data source configuration information into that file, otherwise create an xml file. A data source can have multiple configuration attributes (key-value pairs), and can have any amount of mapping files, but only one service class and one implementation class.
The Data Source Wizard GUI consists of two frames and one dialogue box. Figure 17 shows the architecture of the Data Source Wizard. As described in Table 5, the DataSourceWizard frame is the main graphic interface. It allows the user to input basic information of the new data source. It allows the user to call the DataSourceConfigSetting dialogue box so as to edit data source configuration parameter name and value. In addition, it allows the user to call the DataMappingSetting frame in order to edit the global-local data schema mapping information. The Data schema mapping file will be created immediately after the DataMappingSetting frame has been invoked to add a new mapping file and it has been saved. [DataSourceName]Service.java and [DataSourceName]Impl.java will be created after the “Finish” button in the DataSourceWizard frame has been clicked. The wizard will generate the basic methods and source code in the two classes. At the same time, the DataSource.xml will be created, if it does not already exist, and the data source configuration information will be appended to the DataSource.xml. If the user clicked the “Cancel” button in the DataSourceWizard frame, all the created mapping files would be deleted to ensure that the data source addition procedure has been completely cancelled.

![Data Source Wizard architecture](image)

Figure 17: Data Source Wizard architecture.

Table 5: Data Source Wizard components.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
</table>
| DataSourceWizard             | three parts:  
Basic Info Setting  
Data Source Configuration Info Setting  
Data Mapping Info Setting  |
| DataSourceConfigSetting      | allow user to input parameter-value pairs for data source configuration info (parameter |
DataMappingSetting allow user to choose a data object, then input the local data schema info

4.3.2 Basic Info setting

The basic information required to be input in the Data Source Wizard main frame is listed in Table 6. The Data Source Name is the name of the new data source and is defined by the user and is unable to be null. The base path is the base path of the middleware project. It has a default value and this usually does not need changing and it is also unable to be null.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Source Name</td>
<td>name of data source</td>
<td>can’t be null</td>
</tr>
<tr>
<td>Base Path</td>
<td>the base path of the middleware project</td>
<td>can’t be null</td>
</tr>
</tbody>
</table>

Table 6: Basic Info items.

Figure 18 shows the layout of the DataSourceWizard frame. The frame is divided into three areas, Basic Setting area, Data Source Configuration Information area and Data Mapping Information area. For basic information setting, only the first area is concerned. It allows the user to input the data source name. Before performing any further settings, the “Confirm” button should be clicked in order to confirm the data source name. This because the mapping file name contains the data source name and this name has to be set down before the file is created. After the “Confirm” button has been clicked it can never again be changed and thus, this must be carefully thought through before the decision to click is taken. In relation to the base path information, a default value has been set for it. If the requirement is to change the base path, then the “Browse” button must be clicked to open a file dialogue and then the required path must be picked.
4.3.3 Configuration Info setting

The format of the configuration information is listed in Table 7. One item of configuration information is composed of the parameter name and the parameter value. A parameter can be driver name, username, password for databases and can also be the URL for web resources.

Table 7: Configuration Info items.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Parameter, Value&gt;</td>
<td>used to get access to data source (drivername, username, password, url...)</td>
<td>key-value pairs; can be added, edited, deleted</td>
</tr>
</tbody>
</table>

The second area in the DataSourceWizard frame controls the configuration information setting. Clicking the “Add” button in the Data Source Configuration Information area, causes the Configuration Setting dialogue box to appear. The first picture of Figure 19 shows the layout of the Configuration Setting dialogue box. The information should be filled in ensuring that the parameter name is not empty. The “Save” button is clicked, which causes a return to the main frame and the new parameter-value pair will be represented in the data source configuration information table, as shown in the second picture of Figure 19. As can be seen from Figure 20, it is possibly to choose one row in the table and by clicking the “Edit” button it is then possible to modify the setting. One
row in the table can also be chosen and the “Delete” button can be clicked in order to delete it, as shown in Figure 21.

Figure 19: Add configuration info.

Figure 20: Edit configuration info.
4.3.4 Mapping info setting

Table 8 lists the items necessary to provide the building of the mapping information. The Data Object Name is the global name of the data object which is used in the application business logic. The Local Table Name is the local name of the data object in the new data source. For the database, it is the table name. One local column name is matched with one global column name and the corresponding local column type must be specified at the same time.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Object Name</td>
<td>global name of the data object</td>
<td>DataObjects.xml</td>
</tr>
<tr>
<td>Local Table Name</td>
<td>local name of the data object (table name in database)</td>
<td>can’t be null</td>
</tr>
<tr>
<td>&lt;Global Column Name, Local Column Name, Local Column Type&gt;</td>
<td>global-local mapping info</td>
<td>global column name info stored in [GlobalDataObjectName].xml</td>
</tr>
</tbody>
</table>

The third area in the DataSourceWizard frame controls the mapping information setting. If the “Add” button in the Data Mapping Info area is clicked then the Mapping Setting frame will appear. As previously mentioned, it is necessary to ensure that the data source name has firstly been confirmed, because this is required in order to create the mapping file. The Mapping Setting frame layout is shown in Figure 22. It is possible to choose a data object and the global column information will be shown in the table below. The data object information has to be provided by application developer in advance.
The developer should define the global schema for each data object used in the application. The names of the available data objects are required to be added into the specific xml file, DataObjects.xml. Additionally, global schema information regarding each data object is recorded in the \[DataObjectName].xml. These configuration files are stored in the [DataSourceWizard base path]\GlobalDataSchema. As described in Table 9, the Data Source Wizard can call the function getDataObjectNameList() to obtain a global data object name list from the DataObjects.xml. In addition, with the function getGlobalColumnNameList(), the Data Source Wizard is able to obtain the global data schema for a particular data object from the [objectName].xml.

As shown in Figure 23, it is necessary to fill in the local column name which is matched with the global column and then select the data type for it. After clicking the “Save” button, a mapping XML file will be created and the file name will be shown in the mapping file list in the Data Mapping Information area of main window. The xml file is created by invoking the function createMappingFile(), which is listed in Table 9. At the same time, as shown in Figure 24, it is possible to choose one row in the mapping file list and click “Edit” button to modify the setting. It is also possible to choose one row and then click the “Delete” button to delete the corresponding mapping file, as shown in Figure 25.
Figure 23: Add mapping info.

Figure 24: Edit mapping info.
4.3.5 Data source service wrapper generation

After the basic information, configuration information and mapping information have all been set, it is then possible to click the “Finish” button in the DataSourceWizard frame. Then, the files of the data source wrapper will be generated and the data source configuration information will be recorded, as shown in Figure 26. The Java files of the data source wrapper are generated according to the templates which are stored in the [DataSourceWizard base path]\DataSourceWizardTemplate. The Data Source Wizard will also utilize the DataSourceWrapperBuilder which is listed in Table 10 to create the Java files. Otherwise, if the “Cancel” button has been clicked, then the corresponding mapping files created during the data mapping information setting will be deleted.

Figure 25: Delete mapping info.

Figure 26: Data Source Service Wrapper generation.

Table 9: XML Handle functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDataObjectNameList()</td>
<td>read from DataObjects.xml to get global data object name list (developer can choose one of the items in the list and set mapping info for this certain data object)</td>
<td>List&lt;String&gt;</td>
</tr>
</tbody>
</table>
### Table 10: Assist classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants</td>
<td>set all the default file paths and file names</td>
</tr>
<tr>
<td>DataSourceWrapperBuilder</td>
<td>create java classes according to Templates in “DataSourceWizardTemplate” folder</td>
</tr>
</tbody>
</table>
5 Evaluation

This chapter will analyses the heterogeneous data source middleware in relation to generality, easy-to-use, flexibility, SQL Injection and efficiency aspects.

5.1 Generality

Generality in software engineering often refers to whether the software can achieve multi-purpose usage or whether it can run on cross-platforms. For middleware, generality is especially important, as reusability should be middleware’s main property.

The abstract class provided a unified interface and each subclass can have its own property and behavior. As a result, the middleware maintainer can extend any data source service to the middleware. A local database service with SQLite has already been implemented and tested and a remote database service with MySQL. In addition, interfaces have been designed for active and passive sensors and the middleware is also able to become supportable for the sensor data source as long as the subclass implementation relating to the detailed sensor data operation is accomplished. Of course, it also works for any other data sources. As can be seen from the designing part, the middleware architecture is very flexible and extensible. Theoretically speaking, it can be extended to support any data sources.

Although the middleware is motivated by the Android E-Health application, it is purely focused on providing data source service and has nothing to do with business logic. Thus it can be applied to any application so as to deal with heterogeneous data source issues. Of course, an application which demands a single and fixed data source can also use this middleware. For instance, it can provide a data source service for E-commerce applications and it can also provide a data source service for biology and scientific research systems. All in all, no matter what the application is developed for, it should be able to use the middleware to achieve a data source interaction service.

As previously described, with the platform independence property of the Java language, the middleware can run under any operation systems.
At the same time, in addition to DLL (Dynamic Link Library), the Java project can also be packaged and exported as a JAR (Java Archive). JAR is a platform-independent file format and can be used to publish and use a class library. It usually serves as the unit to build and extend an application. With all the Java properties, the middleware can be integrated into applications developed with any programming language.

5.2 Easy-to-use

Easy-to-use feature is considered in two aspects. One is if it is easy-to-use for the middleware manager and the other is if it is easy-to-use for the application developer, who utilizes the middleware to build the application.

5.2.1 Middleware manager side

As described in section 4.3, the Data Source Wizard assists the middleware manager to add a new data source in an easy, quick and standardized way. The middleware developer need only fill in the corresponding information in the GUI. It is very easy to add, edit and delete a demanded item for the configuration information and mapping information. The only necessity is to click a button and fill in the text field without writing an xml manually. At the same time, the basic framework and functions of the data source wrapper are generated automatically. The middleware manager is only required to complete each function according to the new data source properties. With the Data Source Wizard, the workload in relation to adding a new data source has been significantly reduced.

5.2.2 Application developer side

Before the decision was taken to use a middleware in order to accomplish the operation and management of data sources for this E-Health application, a single data source, SQLite, was used to implement the application for the user and record information management. This section will compare the workload of the application developer before and after using the implementation of the middleware for the user and record information management part. To make it comparable, the data sources are set to “sqlite” for related functions in the strings.xml for the new middleware version.
The number of lines which the application developer requires to program indicates the development workload to a certain degree. Observe the SQLite version and the middleware version, count the source code lines of data source part for each function. A comparison is then be made and the results are shown in Figure 27. It can be seen that there is only a minor difference in the number of lines between the two versions. As mentioned previously, in the SQLite version, the data operation details are mixed with the data presentation. It is able to obtain the parameters directly from the GUI components input and immediately use them to perform adding, deleting, updating or selecting. While using the middleware, the parameters are required to be wrapped into a specific data format (XMLData) and written to a temporary XML file for data transmission between the application and middleware. This extra phase will occupy a considerable portion of the entire lines of coding. However, this part has a regular, repeated pattern, so it is very simple and easy for programming.

![Figure 27: Comparison of application development workload with and without the middleware.](image)

For further observation, detailed statistics are gathered for each function development in the middleware version. In the middleware version of the E-Health application, it is the business logic layer which is concerned with the data sources. As shown in Figure 28, each function in the business logic layer is basically divided into four parts: the parameters wrap part, data source service part, XML output part and business logic part.
Figure 28: Comparison of application development workload in different parts of business logic layer integrated with the middleware.

In the parameters wrap part, a temporary XML file is created for data transmission between the application and the middleware. The application developer must create a XMLData variable for the parameters and call for the writeXML() function with the XMLData variable as the input parameter. The parameters should be wrapped into the XMLData format one by one with both name and value. As a result, it will take many lines when the number of parameters is large. However, as mentioned previously, the developer can follow a regular and repeated pattern for this part and it will thus involve minimal effort.

In the data source service part, the interaction with the data source will be conducted. The data source service part generally contains five steps:

1. Call chooseDataSourceService() to get a specific data source service instance, and assign this instance to the AbstractDataSourceService variable;

2. Call setXMLHandler() to set xml handler instance for the data source service instance;

3. Call setDataObjectName() to set the data object which need to be operated;

4. Call connect() to gain access to the data source;

5. Call chooseOption() for the desired basic operation to be performed.
The code is simple and standardized, with one line for one step.

In the XML output part, the result of the “select” operation will be parsed from a specific temporary XML file into an XMLData variable. The result may be used for data presentation, private information confirmation and so on.

In the business logic part, the relevant action will be performed according to the result from the data source operation.

From Figure 28, it can be noted that the UserRegister function requires much more source code lines when compared to other functions. The reason for this is that the middleware can only provide four basic data source operations (add, select, update and delete). It must invoke a different service for each operation separately when dealing with complex business logic. Then returning to Figure 27, it can be seen that the ShowRecords function requires significantly more lines using SQLite. This is because there are two frames required to show the records in the list, and a similar function was coded twice in the SQLite version. Additionally, in the case of using the single data source SQLite, there are another 58 lines of code to implement the derived classes of the SQLiteOpenHelper. The SQLiteOpenHelper is an abstract class, used to take care of database creation and version management.

At the same time, the effort to locate the correct portion of the whole system to start the relevant code can also show the workload of the application development. The data source part of both the middleware version and the SQLite version is merely part of the entire E-Health project, and the majority is mixed with other function codes, which is particularly true in the SQLite version. As a result, it is possible to directly use tools such as the SourceMonitor to perform the static code analysis. It is necessary to discover and count the lines manually. To count the lines of the data source interaction part in the application, takes the middleware version less than five minutes, while in the SQLite version this takes at least fifteen minutes. For the middleware version, the data source interaction part is in the business logic layer and in a separated package. It appears clear and well-organized. However, the SQLite version appears muddled and disordered with all the source code mixed together. It is also very difficult to locate the correct part for a specific function accurately and rapidly.
As SQLite might be the most general data source for Android application, interaction with other data source services without the use of the middleware is supposed to require more lines of code and make the system development even more complex. The middleware version always has less code complexity no matter what kind of data source is demanded.

From the above, it can be seen that it is possible to significantly reduce the workload of application development using the data source middleware. The reason for this is that the data source middleware offers regular and standardized programming steps. The data source middleware can make the application more organized. It means that a more standardized and modularized framework will be used which will assist in forming a more elegant programming style. This can make the programming procedure more logical and smooth. With a separated and independent data source module, it becomes easier to build and control the system framework.

5.3 Flexibility

Normally, software flexibility indicates the ability for the software to change over time. It refers to the degree of adaptability and extensibility of the software in relation to a given change.[22] Thus, high flexibility often means more generic and configurable properties for the system. At the same time, high flexibility could eventually lead to high maintainability.

As previously described, the abstraction design ensures good extensibility for the middleware. Abstraction makes the middleware architecture flexible and changeable. The abstract class unifies the service interface of the middleware. This makes the application depend on a fixed abstraction when using the middleware. As a result, data source changing will not affect the business logic of the application. On the other hand, a new data source service and its particular behavior can be extended by implementing a specific subclass of the abstract class. As each subclass for each specific data source service is independent of any other, the middleware can be modified and updated by implementing, adding or replacing a particular subclass without compiling the entire middleware.

At the same time, the usage of a declarative descriptor file can also increase the flexibility. In this middleware, the XML serves as the de-
It is possible to divide a system into two parts. One is mutable, while the other is relatively stable. The mutable part includes the configuration information, data schema and so on. Both the name and value of these variables may change frequently according to the preference of the user. The stable part refers to the source code of the data source operation implementation and the basic assistance methods.

Once implemented and the test passed, this part can only be extended but not modified. The heterogeneous data source middleware separates the mutable part from the more stable ones and organizes the mutable part into an XML. With this mechanism, the maintainer of the middleware can easily change variables in the structured declarative descriptor file without disturbing any stable part.

In relation to the flexibility of using the middleware, this is mainly concerned with the workload involved in changing a data source for the application developer. As discussed in an earlier chapter, when the data source for a particular function in the application requires to be changed, the application programmer need only to modify the related value in the declarative descriptor. This process is extremely simple and easy. However, without the middleware, the application programmer has to investigate thousands of lines of source code in order to locate all the data source interaction parts, and modify them one by one. This process is not only time consuming and human intensive, but also has a significant chance of making negligence and failing to tune the program comprehensively. At the same time, the whole system should be tested and debugged in relation to the new data source part. However, with the middleware, the application developer does not have to worry about the data source operation implementation. As the middleware guarantees the correctness of the data source service, using the middleware to build the application will significantly reduce the risk of potential errors after the data source has been changed. This also implies a good maintainability for an application using the middleware.

### 5.4 SQL Injection

As the heterogeneous data source middleware is data-centered and message-oriented, the most important security problem is data security. First of all, the middleware should eliminate a SQL injection attack [23], as it is involved in using SQL to retrieve data from database.
SQL Injection is one of database attack threatens. It is mostly known as an attack vector for websites, but can be used to attack any type of SQL database. SQL Injection uses some external interface of the database to insert user data into the actual database manipulation language, so as to achieve the purpose of database invasion and even operating system invasion. The main form of SQL Injection is to insert code directly into the user input parameters, which will be part of the SQL commands and will be executed. It can also be the indirect injection of malicious code into a string which will be stored in a table or stored as metadata. After the string is connected to a dynamic SQL command, the malicious code will be executed.

Essentially, SQL Injection uses SQL syntax as a tool and is focused on the vulnerability of application development programming. If the attacker can manipulate the data and insert some SQL statements into the application, then SQL Injection occurs. In fact, SQL Injection is a vulnerability which exists in the common multi-connected applications. Attackers often attach extra SQL statement elements to the end of the predefined SQL statement. By this method, the attacker can deceive the database server to perform any unauthorized query.

For this particular data source middleware, the parameter transmission mechanism can naturally reduce the SQL injection risk to a minimum. It uses XML to implement the data transmission between the application and the middleware. Each parameter should be well structured into the XMLData format and then written into a temporary XML file. This mechanism forces the user, who is generally the application programmer, to specify the number and type of input parameters and output results for a given function. Additionally, the structured data format makes only one value match with one parameter name, which is determined by the application programmer for each specific function. This ensures that the SQL generation procedure of the data source service to be significantly more standardized and parameterized, thus making it more difficult for the attacker to make use of SQL syntax to insert ill-willed SQL statements. At the same time, the data source service in the middleware will check every input which is read from the XML. It will detect sensitive characters such as a single quote mark, double ‘-’ and so on. On discovering any sensitive character which may lead to an SQL injection, the data source service will perform the filtering and replace it with a space.
5.5 Efficiency

Based all the abstraction mechanism and data transmission mechanism, there must be testing in relation to how the middleware affects the system efficiency when dealing with any data source interaction. In relation to efficiency measuring, the main concern is with regards to the response time.

To make it easier to analyses, this experiment uses the same settings as section 5.2.2. It compares the response time of the SQLite version and the middleware version for each function related to the user or record operations. Repeated experiments are conducted and abnormal values are removed and the result is shown in Figure 29.

![Figure 29: Comparison of each function response time with and without the middleware.](image)

Figure 29 shows that compared with the SQLite version, some delay exists in the middleware version. As declared in section 5.2.2, the middleware part is basically divided into four parts: parameters wrap part, data source service part, XML output part and business logic part. Compared with the SQLite version, the middleware version is required to perform extra parameter wrapping and query result parsing work, which is used to achieve generality and extensibility of the middleware. It is believed that the delay is caused by the data transmission between the application and the middleware using XML. Based on the six functions, the delay in UserLogin, UserRegister and ShowRecord appears to be larger than the others. These three functions all invoke “select” operation service. Obviously, the larger delay is caused by the extra XML writing and parsing to transfer the query results. Meanwhile, the
UserRegister must invoke the “select” operation twice, once to check if the username exists and once to obtain the newly generated userId for login. This thus causes the largest delay.

Although the middleware version takes more time to respond, the delay for each function is normally around 10 milliseconds and the largest is less than 80 milliseconds. This is acceptable compared with the entire response time and in relation to the generality achieved, is worthwhile.

A further experiment is taken for the “select” operation and response times are observed with different query result numbers both under the SQLite version and the middleware version. Repeated experiments are conducted and abnormal values removed, with the result being shown in Figure 30, it can be seen that the response time of the middleware version rises marginally faster than the SQLite version when the number of the selected records number increases. This might be caused by the fact that obtaining a query result from the XML takes more time with the increase in the number of results. It is related to the XML parsing cost and the XMLData construction cost. Although this has not yet caused any serious problems, it might take a significantly longer response time when the data scale increases substantially. In the next stage, the data transmission mechanism should be improved so as to cut down the transmission cost of the data source service output.

![Figure 30: Comparison of ShowRecords response time with different number of selected records.](image-url)
6 Conclusions

Due to the diversity of the data sources, the management of multiply data sources has become a significant challenge for project developing and maintaining. To address heterogeneous data source problems, a heterogeneous data source middleware has been designed and implemented and integrated it into the Android E-Health application as data access layer.

The middleware is composed of various data source service wrappers. Each data source service wrapper stands for one particular data source and is responsible for the implementation of the data source operation details. To ensure the generality and extensibility of the middleware, the middleware made an abstraction of all the data source service wrappers and declared an abstract class AbstractDSService to provide unified interfaces. Every data source service wrapper is derived from the abstract class. To eliminate data schema incompatibility between the different data sources, the middleware adopted a mapping mechanism which uses an xml file to describe the global-local data schema mapping information. At the same time, to ease the interface design and provide platform independence, data transmission is also achieved by means of the xml. For extension issues, the Data Source Wizard is provided in order to simplify and standardize the procedure in relation to the addition of new data sources.

The experience of utilizing the middleware to build the data access layer of this Android E-Health Application has proved the extensibility and flexibility of the middleware. It can significantly reduce the application programming workload and allow the application developer to modify a data source for a given business logic function without changing any source code. Although the motivation behind this middleware was in relation to our E-Health project, it can be generally applied to any other systems with different purposes.

However, the evaluation of the middleware has uncovered some problems. Although the extra work of XML creating and parsing for data transmission has a regular, repeated pattern, this part has proved to occupy a large portion of the source code. It also caused a response time delay, which will increase as the amount of data increases. In the next
stage, the data transmission mechanism will require to be improved or re-designed so as to simplify the application programming and to improve the middleware efficiency. At the same time, further efforts are required to be made on security issues. In the future, data encryption and user privilege (or authority mechanism) are to be considered to ensure system security when using the middleware.
References


Appendix A: Documentation of own developed program code

Code of AbstractDSService

AbstractDSService.java:

```java
package com.middleware.DataSourceService;

import android.database.SQLException;
import com.middleware.XML.XMLHandler;

public abstract class AbstractDSService {
    private XMLHandler xmlHandler;
    private String sourceID;
    private String dataObjectName;

    public abstract boolean chooseOption(String optionID);

    public XMLHandler getXmlHandler() {
        return xmlHandler;
    }

    public void setXmlHandler(XMLHandler xmlHandler) {
        this.xmlHandler = xmlHandler;
    }

    public String getSourceID() {
        return sourceID;
    }

    public void setSourceID(String sourceID) {
        this.sourceID = sourceID;
    }

    public String getDataObjectName() {
        return dataObjectName;
    }

    public void setDataObjectName(String dataObjectName) {
        this.dataObjectName = dataObjectName;
    }

    public abstract void connect() throws ClassNotFoundException, SQLException;
}
```

Code of XMLHandler

XMLHandler.java (portion):

```java
package com.middleware.XML;

import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.FileInputStream;
```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;
import java.util.InvalidPropertiesFormatException;
import org.dom4j.io.OutputFormat;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.xmlpull.v1.XmlPullParser;
import org.xmlpull.v1.XmlPullParserException;
import android.content.Context;
import android.util.Xml;

public class XMLHandler {
    final private String tempFile = "temp.xml";
    private String dataSourceFile;
    private Context context;
    public Properties parseDataSourceXML(String sourceID)
        throws Exception {
        Properties props = new Properties();
        boolean flag = false;
        InputStream inputStream = context.openFileInput(dataSourceFile);
        XmlPullParser parser = Xml.newPullParser();
        parser.setInput(inputStream, "UTF-8");
        int eventType = parser.getEventType();
        while (eventType != XmlPullParser.END_DOCUMENT) {
            switch (eventType) {
                case XmlPullParser.START_TAG:
                    String key = parser.getName();
                    if (key.equals("datasource"))
                        flag = true;
                    else
                        break;
            }
        }
        while (eventType != XmlPullParser.END_DOCUMENT) {
            switch (eventType) {
                case XmlPullParser.START_TAG:
                    String key = parser.getName();
                    if (key.equals("datasource"))
                        flag = true;
                    else
                        break;
            }
        }
        return props;
    }
}
```java
{ String value = parser.nextText();
    props.setProperty(key, value);
}
break;
}
if(flag)
    break;
eventType = parser.next();}
}
break;
}
if(flag) break;
eventType = parser.next();
}
inputStream.close();
    return props;
}

public Properties getMappingProps(String mapFile) throws InvalidPropertiesFormatException, IOException {
    ReadXML xmlHandler = new ReadXML();
    String filename = context.getFilesDir().toString() + "/" + mapFile;
    Properties props = null;
    try {
        props = xmlHandler.readMappingXML(filename);
    } catch (XmlPullParserException e) {
        e.printStackTrace();
    }
    return props;
}

public int writeXMLData(XMLData xmlData) {
    String filename = context.getFilesDir().toString() + "/" + tempFile;
    WriteXML xmlHandler = new WriteXML();
    return xmlHandler.writeXMLFile(xmlData, filename);
}

public XMLData readXMLData() {
```
String filename = context.getFilesDir().toString() + "/" + tempFile;
ReadXML xmlHandler = new ReadXML();
XMLData xmlData = null;
try {
    xmlData = xmlHandler.readXMLFile(filename);
} catch (Exception e) {
    e.printStackTrace();
}
return xmlData;

public Context getContext() {
    return context;
}

public void setContext(Context context) {
    this.context = context;
}

public String getDataSourceFile() {
    return dataSourceFile;
}

public void setDataSourceFile(String dataSourceFile) {
    this.dataSourceFile = dataSourceFile;
}

}  

Code Example of business logic layer  
UserLogin.java: 

package com.bll.users;
import android.content.Context;
import android.database.SQLException;
import android.widget.Toast;
import com.hyl.alarm.R;
import com.middleware.DataSource.DataSource;
import com.middleware.XML.XMLData;
import com.middleware.XML.XMLDataFormat;
import com.middleware.XML.XMLHandler;
public class UserLogin {

    public int login(User user, Context context){
        boolean flag = false;
        int userid = 0;
        XMLData xmlData = new XMLData();
        XMLDataFormat parameters = new XMLDataFormat();
        xmlData.getColumnName().add("username");
        xmlData.getColumnName().add("password");
        xmlData.getColumnName().add("output1");
parameters.getValues().add(user.getUsername());
parameters.getValues().add(user.getPassword());
parameters.getValues().add("userId");
xmData.getValueList().add(parameters);
XMLHandler xmlHandler = new XMLHandler();
xmlHandler.setContext(context);

xmlHandler.setDataSourceFile(context.getString(R.string.dataSourceFile));
xmlHandler.writeXMLData(xmData);
AbstractDSService service = 
DataSource.chooseDataSourceService(context.getString(R.string.loginDataSource));
service.setXmlHandler(xmlHandler);
ser-
vise.setDataObjectName(context.getString(R.string.userDataObeject));
try {
  service.connect();
} catch (SQLException e) {
  e.printStackTrace();
} catch (ClassNotFoundException e) {
  e.printStackTrace();
}
flag = service.chooseOption("select");
if( !flag ){
Toast.makeText(context, "UserName or Password is not correct!",
Toast.LENGTH_SHORT).show();
} else {
  try {
    xmlData = null;
    xmlData = xmlHandler.readXMLData();
  } catch (Exception e) {
    e.printStackTrace();
  }
  if(xmlData != null &&
xmlData.getColumnName().size() != 0)
    userid = Integer.parseInt(xmlData.getValueList().get(0).getValues().get(0));
  else
    Toast.makeText(context,
"UserName or Password is not correct!",
Toast.LENGTH_SHORT).show();
}
return userid;