8-2011

A Merging System for Integrated Person-Centric Information Systems

Swati Jain
Utah State University

Recommended Citation

Jain, Swati, "A Merging System for Integrated Person-Centric Information Systems" (2011). *All Graduate Plan B and other Reports*. 39.
https://digitalcommons.usu.edu/gradreports/39

A MERGING SYSTEM FOR INTEGRATED PERSON-CENTRIC INFORMATION SYSTEMS

by

Swati Jain

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Stephen W. Clyde
Major Professor

Curtis Dyreson
Committee Member

Nicholas Flann
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2011

ABSTRACT

A Merging System for Integrated Person-Centric Information Systems

by

Swati Jain, Master of Science

Utah State University, 2011

Major Professor: Dr. Stephen W. Clyde
Department: Computer Science

Large-scale integrated information systems correlate equivalent or related information from multiple data sources to provide a unified view of data. This report describes the design and implementation of a tool called *xMerger* that provides a unified view from multiple matching records, which could be multi-source duplicates and overlapping records. To achieve this *xMerger* provides a merging process that generates a complete and accurate merged record from conflicting and incomplete records. This report also discusses the challenges present in the process of merging and *xMerger*'s solutions. *xMerger*'s design and implementation was validated by adapting it to CHARM, a real world integrated system currently in use at the Utah Department of Health.

(65 pages)

ACKNOWLEDGMENTS

I am deeply indebted to my major advisor, Professor Stephen Clyde, for his invaluable advice, guidance, and patience. I would also like to thank the members of my committee, Professor Curtis Dyreson and Professor Nicholas Flann. Their suggestions, support, and hard work were invaluable and are deeply appreciated.

Many thanks to Myra Cook; her great work and help were indispensable for completing this report.

I would also like to thank my colleague, Lan Hu. Thanks for guiding me to become familiar with the project, as well as resolving many problems along the way.

Finally, I appreciate the continuous support and encouragement of my beloved family through the duration of my academic pursuits.

Swati Jain

CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

An integrated information system connects multiple, isolated information sources through an integration infrastructure [1] that provides end-users or external systems with a unified view of persons, places, or things represented in two or more of the data sources. For example, consider an integrated system of customer databases managed by three different subsidiaries of an insurance enterprise. One subsidiary sells home insurance, another handles car insurance, and the third deals with recreational-vehicle insurance. A customer, who we will call *Mary*, could have purchased insurance policies from each of the subsidiaries. An integrated system involving the three subsidiaries' systems could provide corporate users with a complete view of Mary's policies, including the "best version of truth"[1] for her demographics and contact information. In general, such integration can improve the amount and quality of data sharing among various sources and, thus, ultimately help users and organizations become more productive.

However, integrated systems suffer from three common problems that lower the quality of the integrated information and could eventually negate all of the system's potential benefits. First, the integrated system could fail to correlate records from two or more sources that represent the same person, place, or thing. Such errors result in *multi-source duplication*, which can dramatically reduce the system's potential value because end users will not see a correct and complete unified view of related data [3]. For example, if the integrated insurance system did

---

[1] The "best version of truth" is phrase that is sometimes used when referring to data records computed from multiple original records from disparate sources, such that the compute record contains the most accurate and consistent information possible [1].

not correlate Mary's car insurance with her motor-home insurance, it may mislead end users about Mary's complete profile. This multi-source duplication is primarily a record-matching problem that can be addressed by a variety of matching technologies [1].

A second problem stems from incorrectly correlating two more records that are actually not for the same person, place, or things. This problem, known as *overlaying duplication*, is also primarily a matching issue [1]. Overlaying duplication can cause serious confusion. Just imagine what might happen if Mary's information in the homeowner's insurance database was mistakenly correlated with some other person's information in the vehicle insurance database and that person had a history of missing premium payments and filing lots of claims. Section 2.2 provides additional background on multi-source and overlaying duplication, as well as other relevant challenges for integrated information systems.

Even if an integrated system minimizes the number of multi-source and overlaying duplicates, it can still suffer from a third problem that comes from incorrectly combining the data from matched records to form a unified view. In general, the unified view should represent the "best version of truth" based on all the original data. However, there can be various syntactic and semantic heterogeneities across the participating data sources that can make the process of formulating a unified view a huge challenge [15]. This process is called *merging, data integration,* or *resolution* [1, 32, and 34]. The main objective of this process is to provide a unified view that contains reliable, accurate and usable information, which requires awareness of and attention to the quality of the original data, variations in meaning of individual data fields over time and across the sources, the type and frequency of inaccurate or incomplete data [1]. Section 2.3 provides more detail on these issues.

A successful and effective merge process includes data analysis, data extraction, conflict resolution, data merging and its execution. This report introduces a general purpose customizable merging tool called *xMerger* that provides effective merging to create a unified view of data in an information system. *xMerger* generates a single merged record by combining multiple matching records in a person-centric information system, as well as eliminating original records if required. *xMerger* also provides un-merging of erroneously merged records to restore the original view of data. Section 2.3.4 discusses un-merging techniques in detail.

Chapter 3 discusses *xMerger's* goals and its applicability to different architectures for integration systems. It also discusses *xMerger's* relationship to issues of merging and unmerging. Comparisons between *xMerger* and other similar existing tools are also conducted and discussed.

xMerger supports data-source configuration, experimentation, and incremental refinement, which makes it adaptable and tunable to many different kinds of integrated information systems. To this end, *xMerger's* design includes a customizable structure that allows users to configure specific information associated with a system. This customized structure can then be adapted by other person-centric integrated information systems. *xMerger's* object-oriented architectural design is discussed in detail in Chapter 4. There, each *xMerger* component is shown not only to possess a strong encapsulation boundary, but also to provide meaningful and extensible abstractions through a well-formed interface. These design characteristics help the implementation to be readable, reusable and maintainable.

In sum, *xMerger's* strengths not only include its effective un/merging processes, but also its architectural design, which makes it adaptable and configurable to discrete and novel integrated system implementations.

The validation and evaluation of *xMerger* requires adapting it to a specific integrated system. For this purpose, we selected the Child Health Advanced Records Management (CHARM) [4] system, a child-centric integrated system, to be the testing environment. CHARM allows independent public health-care programs to exchange data about children who have received, are receiving, or need public health services. To fulfill its services, CHARM maintains a core relational database that contains a complete set of child-centric person profile records gathered from multiple data sources. This core database is both accessible and modifiable by *xMerger*.

Chapter 5 describes the implementation details of *xMerger* as well as its performance in the CHARM environment. Chapter 6 focuses on the testing of *xMerger*. Finally, Chapter 7 summarizes the contributions of the presented work and offers ideas for future work. Additional details about *xMerger* software and other work-products of this project are included in the Appendix. Specifically, the Appendix lists *xMerger*'s user-level goals and its functional requirements.

CHAPTER 2

DUPLICATION PROBLEMS AND
THE DEDUPLICATION PROCESS

To understand the full ramifications of merging data from multiple data sources, it is important to understand integrated information systems and their various architectures, duplication problems like multi-source duplication and overlapping duplication problem, and finally general deduplication process. Sections 2.1 – 2.3 provide this background. Section 2.4 then discusses general merging concepts and its challenge.

2.1 Integrated Information System and Their Architecture

An integrated information system connects multiple heterogeneous information sources through an integration infrastructure, and provides users and software applications with a unified view of a subject created from participating data sources [1]. Many business firms and public services already realize the benefit of the easy sharing of information an integrated system provides. Consider the following example published in July 2004 *Connecting for Health* [9]. Dr. J.T. Finnell was able to avert a dangerous medical error common to emergency departments across the country due to a connected information environment at the Wishard Memorial Hospital. A patient was complaining of crushing chest pain and admitted to the emergency room, but was not able to recount his medical history. Typically, a patient with symptoms suggesting a heart attack would have been given a blood thinner. Fortunately, attending physicians were able to access the patient's health records electronically from another institution, learning instantaneously that he had recently been treated for a head injury. Giving the patient a blood thinner would have put him at risk for bleeding in his brain and may well have caused serious injury. With the right information, doctors were able to prescribe the appropriate treatment. The patient's chest pain was relieved and turned out not to be a heart attack. Time, money, and possibly a patient's life were saved.

Through integration infrastructures, integrated systems realize information sharing among related yet independently operating participating systems. Depending on the system's requirements, there are almost as many possible architectural designs as individual integrated systems. This section briefly summarizes five prototypical architectures that represent many solutions in creating a person-centric integrated system. For more information on these and other architectures, see Chapter 2 of *The Unique Records Portfolio* [1].

2.1.1 Central Index

This architecture has a master person index system that acts as a central index and that contains demographic person information for persons known to the integrated system. A component called record locator services (RLS) helps participating programs find a specific person's record within the whole of the system. When finding person-specific information, an RLS can work in the following ways:

- The RLS tells the requesting program the location of the person record, and the requesting program goes there to find the person;
- The RLS initiates the communication between the requesting program and the source program, and the source program sends the person's record directly back to the requesting program;

- The RLS finds the person record from source programs and returns it to the requesting program.

2.1.2 Peer-to-Peer

This architecture does not require a central database. However, it either needs all the participating programs to know all the other participating programs or an intermediate communication component to know all the participating programs. In this architecture, a participating program locates information for a person by calling all the source programs with some partial demographic information, and the source programs reply directly to the requesting program.

2.1.3 Arm‘s Length Information Broker

This architecture maintains a central data repository with demographic person information for the persons known to the system. Different from a central index, and instead of an RLS, each participating program runs an agent to communicate with other participating programs and the central data repository.

2.1.4 Central Database

In this architecture, a central authority operates a central database for all the sharable information of a person from all the participating programs, including demographic information, service related information, and program data. This architecture requires periodic bi-directional synchronization on sharable information between participating programs and the central database.

2.1.5 Partitioned Central Database

Similar to the central database architecture, a partitioned central database also contains all the sharable information of a person. However, instead of having one central database, it maintains a collection of data vaults (data repositories). It allows one participating program to be associated with one data vault. In this case, the information synchronization only occurs between the participating program and its corresponding data vault.

2.1.6 Summary of the Architectures

No matter which architecture an integrated information system is built upon, its essential goal is to allow data sharing among multiple data sources. However, each of these participating data sources can be inherently different from each other in structure and semantics. This introduces potential problems to data integration in delivering a unified view of data. Additionally, non-technical factors also impact integrated information systems and their data integration process, including political issues, security and confidentiality concerns, organization boundaries, data management procedures and public perceptions, etc. From time to time, the changes in these areas may affect data sharing agreements, data confidential concerns, data management procedures, and even data structures, and data presentation (syntax and semantics). In turn, these factors may require changes to the integrated systems' design and operations, and thus further complicating the data integration process. Regardless of the architecture, all include components(s) that will merge records; as well, this component must be designed in such a way as to withstand these designs changes. Section 3.2 discusses how *xMerger* supports these architectures, and the following section discusses the duplication problems that are result of having heterogeneous data sources.

2.2 Source of Multiple Matches: The Duplication Problem

Merging is a process designed for integrated systems to unify multiple matching records, and is used once multiple matches have been identified in the system. These multiple matches are the result of duplication problems present in the integrated system. In general, duplication refers to having two or more records actually representing a single person, place, or thing in an integrated system. Duplication may be of the following types [1] (considering a person-centric integrated information system which provides a unified view of a person):

- *Same Source Duplication*

  This refers to two or more records for the same person from the same database. This happens when data comes into the same source through different methods. For example, if a database obtained a person's records through a user's self-registration as well as through a telephone interview, it is possible that even though the information came from the same person, since it is was given in two different settings, it is recorded twice in the database.

- *Multi-source Duplication*

  This refers to two or more records for the same person from different sources with similar types of data. This happens in an integrated information system when different organizations combine their data together. These organizations maintain their own databases but with the same structure and a well-unified data format. In such a scenario, a person can be a customer of more than one participating system, resulting in multiple entries in the integrated system.

- *Overlapping Duplication*

  This refers to two or more records for the same person from different sources with different types of data. For example, as shown in Table 2.2.1(a), person1 comes from source1, person2 comes from source2, and the source1 format does not have a middle name and gender. Further, source1 stores the birth date in a month/day/year format; source2 provides all person information, but it only stores the month/year for the birth date. Although these two records have different types of data, they are actually records for the same person.

The first type of duplication is called multiple matches in a single data source. The second and third types of duplication are considered as multi-source multiple matches. Merging these multiple matches is really necessary to provide a unified view of data; this is discussed in Section 2.4.

Overlaying records refers to cases wherein two or more records appear to be for the same person but are actually for different individuals. Table 2.2.2 shows an example of overlaying records. Here, person1 and person2 have similar data except that the SSN value of person1 is missing. However, they are not the same person as one might infer from the second record; they

Table 2.2.1. An Example of Overlapping Records.

	Source	First name	Last name	Middle name	Gender	Birth date	SSN
Person1	1	Anna	Joe			05/09/1988	123456789
Person2	2	Anna	Joe	M.	F	09/1988	123456789

Table 2.2.2. An Example of Overlaying Records.

	First name	Last name	Middle name	Gender	Birth date	SSN
Person1	Anna	Joe	M.	F	05/09/1988	
Person2	Anna	Joe	M.	F	05/09/1988	123456789

coincidently were born on the same day and happen to have the same name.

The records will be incorrectly matched as belonging to the same person, place, or thing, and so will be incorrectly merged. We note that overlaying records are not actually a duplication problem but a consequence of merged record false matches. However, *xMerger* provides a solution for this type of problem as well (see Section 2.5).

2.3. General Deduplication Process

Deduplication is the process of detecting and coalescing duplicate records in an integrated system. The deduplication process consists of matching, linking, and/or merging techniques [1]. Figure 1 depicts a general deduplication process that models what takes place in most integration systems. A *Matcher* is the component that performs matching, i.e., it tries to determine possible matches / multiple matches in an integrated system. A *Linker* is a mechanism that relates duplicate or overlapping records together without changing the original data.

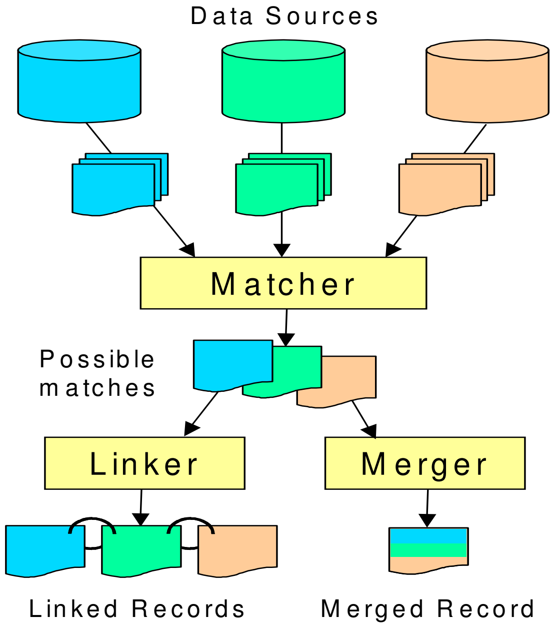

Figure2.3. Overall deduplication process.

Unlike a linker, A *Merger* combines multiple matched records into a single record through necessary modifications and eliminates originals if required. Thus, a merger is able to eliminate the duplication problems previously described. The next section focuses on the merging process and its challenges.

2.4. Merging

Merging consolidates two or more matching records into a single view to expose the most accurate or reliable data, i.e., the "best version of truth." To achieve this, it has to remove single-source or multi-source duplicates (data cleaning). However, single source and multi-source duplicate removal approaches differ from each other. These approaches are discussed below.

For single-source duplicates, the participating program can handle merging internally, without involving the integrated system. The integrated system, however, can assist if the participating program does not have the necessary tools. Also, while merging single-source duplicates, the database of this participating program can readily store the resulting records, and choose to delete or archive the original records [1] as required.

For multi-source duplicates, the integrated system needs to play a significant role in the merging process. It may be necessary for users to interactively direct the consolidation of data. Also, since the original records come from multiple sources, the integrated system cannot simply replace them with the merged records. Some options include [1]:

- Storing the merged record in an intermediate with links back to the original records.
- Sending the merged record back to its sources so they can integrate any new or changed data back into the original records.
- Not storing the merged records, but simply re-merging on-the-fly as needed.

Thus, merging is potentially a complicated process. Merging becomes still more complicated by the nature of data and the degree of heterogeneity among the data sources, which we discuss in the following subsection.

2.4.1. Challenges in Merging

Anomalies in the data, which are brought about by the “messiness” of real world data collection processes, create human concerns and technical problems during the merging process. Importantly, poor data quality results in loss of trust among users. Considering all these issues, effective merging requires awareness of as well as attention to data quality, data variability, data inconsistency, and potential to create duplicate records for an individual.

Dissimilarities in database schemes create yet another major concern. Even when merging two identically structured databases with controlled schemas and vocabularies, problems have been found that can lead to misinterpretations.

Another great challenge is controlling the effect of cumulative errors and inconsistencies. Over time, repetitive merging of the same records can lead to the loss of certain data, and the whole becomes less than the sum of its parts. The more data sources involved, the worse these effects become and the more difficult it is to anticipate the problems.

Table 2.3.2 lists and explains some of the common data problems that need to be considered when planning or implementing merging. The purpose of this table is to stimulate thoughts about the kinds of inconsistencies and variability that might exist in a data source, which makes merging inherently complex.

Table 2.3.2. Common Data Problems (Adapted from [1]).

Problem	Explanation and Considerations for Merging
Semantic heterogeneity	Seemingly similar but actually varied fields from different data sources may have slightly different connotations which, if unrecognized, could cause data inconsistencies or loss of credibility. For example, two data sources may both include a field for an address. For one source, these might be intended to hold a physical address that could be used to locate an individual or check residency. The other source might use its address field only for mailings and verifying identity. If values from these fields were merged, the resulting merge record could be slightly incorrect and of less use to both data sources. A merging process needs to be aware of these subtle semantic differences between seemingly similar fields.
Field-meaning shift	The meaning of a given field might drift over time, either intentionally or unintentionally. For example, the date field that originally meant the date of a point of service might gradually become the date of data entry for a point of service. A merging process needs to be aware of such shifts when they occur so it can properly interpret the semantics of a field value.
Incompatible code domain	Multi-source duplicates may have code fields representing the same information but have different domains (i.e., sets of possible codes). For example, consider two data sources of person records that include ethnicity codes. Not only are the codes likely to be different, but the two data sources might use a slightly different set and incompatible ethnicity categories. If code domains are truly incompatible, then a merging process may need to preserve the code value from all the records being merged, along with their sources so their meaning can be properly interpreted later on. Otherwise, if code domains are compatible, but different, a merge process simply needs to select a standard domain and map all others to it.
Impossible values	Over time, as programs and data structures change, some data fields may end up with values that are impossible or illogical (i.e., 99/99/99 for dates, -1 for birth weights, and numbers for names). Impossible and illogical values can find their way into the system through weak user interfaces, changes in database structures, programming errors, and field-meaning shift. A merging process needs to check for and know what to do about illogical or impossible values.

Meaningless values	Sometimes, a program requires information for a field, but the user does not know the information, or the information does not exist. To work around the requirement, the user enters a bogus or temporary value. For example, if a program requires a first and last name for child and a child does not have a first name yet, the user might simply enter "boy" or "girl." Such values are essentially meaningless with respect to the fields they are in. A merging process should try to recognize and properly handle meaningless values. Often, this means giving less precedence to meaningless values, so real values, if they exist, override them.
Extra information	Fields often include extra information that does not belong, changing the intended meaning of the field. For example, a name first may include a message about the person, such as "John Smith (deceased)." The "(deceased)" entry is extra information. A merging process should try to recognize common or expected forms for extra information and attempt to handle them. For example, "John Smith (deceased)" could result in the "John Smith" name being merged with other name data, and the "(deceased)" being merged with other possible death information.

Finally, an effective merging requires a process owner who is accountable for the quality of the results. Without a clear process owner, merging can lead to inconsistencies that, if unmanaged and unresolved, will decrease the accuracy in the data within the integrated system. The next subsection describes preparation activities for effective merging that address these challenges.

2.4.2. Preparing for Effective Merging

Preparation is the key in effectively overcoming the challenges of record merging and making sure that merging is correct and effective. Preparation activities can be rather involved and should include the following [1].

First, the data models for all participating programs are reviewed and analyzed in the areas of:

- Data syntax (structure, format, data types, hidden languages, etc.)
- Data semantics (meaning)
- Temporal effects, drift in meaning (content) of data fields
- The importance and relevance of the data fields that exist in the database

Next, a thorough analysis must be made of the potential problems with data from all of the participating data sources. An automated script containing a series of queries, computations, or other actions may be used to check for impossible, illogical, or meaningless values. For example, a script could check an individual's record to make sure that his death date, if there is one, is not earlier than his birth date. Another script could check to see if there are any names with meaningless values, such as "boy," "girl," or "unknown."

Finally, it is always important to understand the ramifications of both correctly merged as well as incorrectly merged data. For correctly merged data, the ramifications could involve increased concerns about confidentiality and more complex data stewardship relationships. For incorrectly merged data, the ramifications can be much further reaching, such as affecting the care and services to individuals that need them, and incorrect aggregated information resulting from simple calculations. The incorrectly merged data problem is discussed in the subsection below.

2.4.3. Unmerging

Because records can be merged due to erroneous matching, we need to consider the unmerging of records. Erroneous matching can happen because of weaknesses of the matching algorithms that thus provide mismatched records as matching records for merging. If the original records have simply been archived or deactivated in some way, unmerging can be relatively straightforward. As long as no changes have been made to the merged record, the system simply deletes the merged record and re-activates the old records. However, if changes have been made to the merged record, the system may have to allow a user to interactively undo the merging so that the desired changes can be made to the original record. For example, if person record A and person record B were merged into a new person record C, and then some additional person records were added to C, the unmerging process would have to allow someone to specify whether the subsequent person records belong to A or B.

If the merging process does not preserve the original records, there are the following options for unmerging:

- Keep track of the origin(s) of each piece of data so the original records can be reconstructed.
- Keep an audit trail of changes, including those caused by the merging process, so that the unmerging process can roll back those changes.
- Provide an interactive tool for users to manually direct the unmerging process.
- Keep all of the information as part of the merged record.

CHAPTER 3

WHAT DOES *xMERGER* DO?

This chapter explains the work of *xMerger* and its capabilities in an integrated environment in relation to the topics introduced in the previous chapter. Sections 3.1 – 3.3 describe the functionality of *xMerger* (also see the Appendix for *xMerger's* high level goals and functional requirements definition document). Section 3.1 discusses *xMerger's* **s**upport for multiple types of integration system architectures. Section 3.2 describes how *xMerger* resolves the different levels of integrated system heterogeneities. Section 3.3 explains features of xMerger's merging and un-merging operations. Finally, in Section 3.4, we discuss relevant work and major advantages of *xMerger*.

3.1. Supports for Multiple Types of Integration Architecture

xMerger can be adapted to any person-oriented integrated system utilizing one of the architectures mentioned in Section 2.1. *xMerger* must have the proper understanding of the structure and semantics of data sources to provide an effective record merging in the mentioned architectures. One exception is for peer-to-peer architecture because this does not support record merging across participating programs because record merging can take place within in a single data source and is therefore independent of the data integration [1]. In arms-length information broker architecture, *xMerger* can provide record merging at the server level for its own repository without immediately impacting the participating programs. Merges of the multiple matching records from a single data source can be propagated back to said data sources. All communications with other databases and the *xMerger* are through agents that have access to corresponding databases. In the case of a central database, *xMerger* can be configured on this main database which can have its own merged records yet can link back to the original data sources; in the case of multiple central databases, *xMerger* can be configured at any one of them, but synchronization among all databases will be difficult and need special concerns.

3.2. Dealing with Heterogeneities of Integrated Systems at Different Levels

Given the types of heterogeneities of integrated systems, one goal of *xMerger*'s design is to tolerate and resolve such heterogeneities as much as possible. At the system level and application level, the Sync Engine allows heterogeneities by using cross-platform techniques like J2SE and JDBC frameworks. This means as long as a database has a driver that conforms to the JDBC interface, it can be integrated with *xMerger* no matter on which kind of system it is running.

Schematic and semantic heterogeneities are far more difficult to tackle. We utilize the standard and widely used SQL to resolve most of the common heterogeneities in these areas. As *xMerger* provides merging at record level, so our concern is to resolve semantic heterogeneities. Specially, to handle this issue we implemented a supportive utility named "Virtual-database" to get metadata information of any database until it supports J2SE and JDBC frameworks. This is combined with the power of SQL which supports almost all types of databases. However, a good understanding and knowledge of the participating database's structure and data is needed to realize the complicated issues of semantic heterogeneity.

3.3. Support for Merging/Un-merging

One of the major features of *xMerger* is that it supports data merging to an individual record level, including field level merging, according to user choice. During the merging process, all records are first examined for reliability and accuracy and then merged, updated, deleted or inserted. Section 2.4.2 describes the mechanism to achieve an effective merging process by *xMerger*. Conflict-resolution policies in *xMerger* are left to the choice of the end user. To maintain accuracy and consistency among databases *xMerger* achieves merging synchronously, which is critical for most person-centric data sources in real world.

Similarly, *xMerger* also provides un-merging support for cases wherein previous merges were ordered and completed, but the merge data were found to contain errors. For example, matching algorithms provides mismatches as matching records and *xMerger* merged them to provide a single merged record. To resolve this situation, *xMerger* provides un-merge service to get original data back to maintain correctness. For this purpose, it maintains a history of every merging process in a persistent storage, which helps in the un-merging process to make decisions about what to un-merge to get the original data back. Section 2.4.3 provides background on the un-merge process.

3.4. Major Advantages of *xMerger* and Relevant Work

As we can see from the above sections, the power of *xMerger* comes from its design which makes it customizable to any intergrated environment, together with its effective merging/ unmerging mechanisms. Moreover, its design not only gives *xMerger* great flexibility but also enhances its compatibility with a wide range of database managers.

Currently, several data merging tools and products are available. Most of them give relatively good support for high-level schematic and some semantic heterogeneity. However, they tend to couple with one or another type of databases and application tightly, which means their work focuses on only few types of databases and sticks to only that application. *FamilyTree* is a merging tool that is basically applicable to an SQL-Server, and its automated merging eliminates multi-matched records at a very high level because there is no human interaction to avoid conflicts [23]. A lot of popular database vendors also provide database merging support for their own products. These include IBM’s DB2, MS-SQL Server, Oracle and MySQL, etc. Most of these tools tend to work well when databases are all running on the same type of DBMS, but they are able to resolve conflicts while merging only at a high level.

The design of *xMerger* takes all semantic and syntactic heterogeneities into consideration and provides a customization interface that can be applicable to any databases that are running on different platforms.

CHAPTER 4

ARCHITECTURE DESIGN

xMerger's design considers all common data problems that need to be considered for records merging, such understanding of the data structure, data quality, data variability, data inconsistency and other issues related with data like semantic heterogeneity, incompatible code domain, impossible values. This chapter describes architectural design of *xMerger*, and reveals its flexibility for being adapted by integrated person-centric information system. Specifically, Section 4.1 describes the design considerations for the *xMerger* and Section 4.2 discusses the overall architecture of the *xMerger*.

4.1. Design Considerations

To meet the requirements of general and flexible data merging in person-centric integrated information systems, *xMerger* must provide numerous design features. It should be able to: access and obtain participating database's data across network boundaries and run as a standalone application to provide its services. It also provides a platform that allows this tool to communicate with other databases to share information when required. Thus, each specific part of data contained in one portion of the system should be able to be shared in a manner independent from other parts of the system. Moreover, it should be able to access an integrated set of demographic data from database and refer it to a physical or virtual record of an individual within the integrated system.

To allow users to invoke its services with initial required information, it should provide a flexible interface that allows users to define how they want to process each type of request within their own local database.

4.2. Overall Structural Design

Based on considerations described in Section 4.1, we designed a new general purpose merging component, *xMerger*, for an integrated person-centric information system. Different from other general purpose deduplication components, the *xMerger* implements the effective merging mechanism. With a flexible and customizable structure, *xMerger* leaves the merging logic like database action (insert, update or delete) performed on a record to the domain experts of the integrated system that adapts *xMerger*.

4.2.1. ***xMerger*** **as a Client Server Application**

xMerger composed of a server and client, where the server is responsible for receiving requests from the client and for communication with other components across network boundaries. Figure 4.1 describes a high level communication between *xMerger* and its client, considering CHARM scenario.

To perform the merge of some number of health profiles, the Deferred Match Resolver (DMR) and Backend Matcher are considered as potential clients in CHARM. A DMR GUI retrieves the possible match information and builds all internal objects of *MergeRequest,* in other words, a request containing necessary information for merging, to capture the specifics of this user-defined merge, in case of merging. Finally, the DMR GUI sends this *MergeRequest* object to the merger, where the merge is actually performed. Backend Matcher is also responsible for the same process as mentioned above. Section 4.2.2 provides detail description of *Request* classes: *MergeRequest* and *UnMergeRequest*.

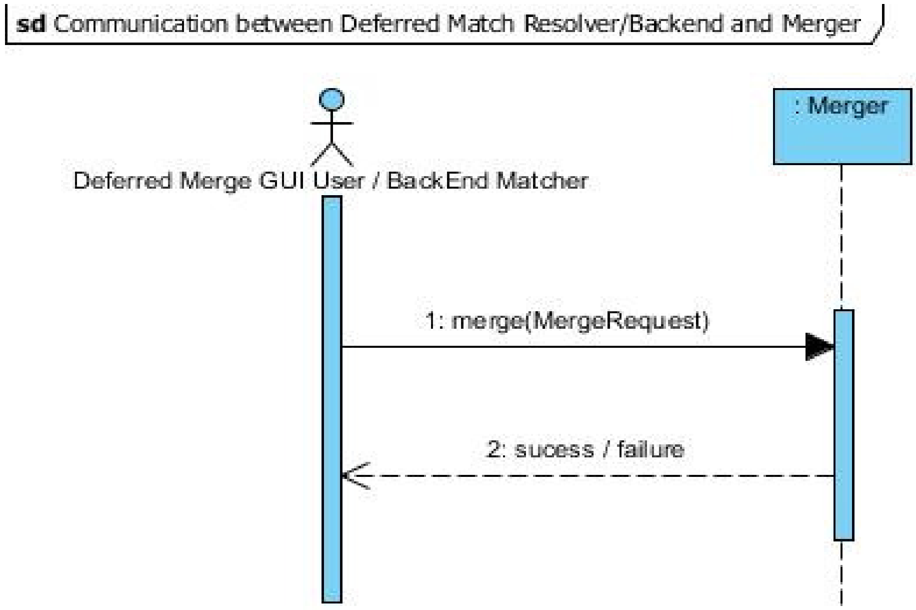
sd Communication between Deferred Match Resolver/Backend and Merger
: Merger
Deferred Merge GUI User / BackEnd Matcher
1: merge(MergeRequest)
2: sucess / failure

Figure 4.1.Communication between xMerger and its clients.

4.2.2. xMerger Overall Structure and It's Components

An initial challenge is to design a generalized request class for both client and server and behaves as a holder for key information for merging. We acknowledge this fact in our system by defining a *Request* class. The *Request* class represents the object(s) responsible for providing the necessary information to achieve merge/un-merge tasks.

To achieve these two separate tasks, the *Request* class is sub-classified into the *MergeRequest* and *UnMergeRequest* classes, as each requires different information to fulfill their goals. *MergeRequest* class objects participate in a merge request process; *UnMergeRequest* class objects participate in an unmerge request process. These two *Request* subclasses cover the required information needed to fulfill Merger's primary purpose – that merge/unmerge requests can be performed efficiently. The UML class diagram in Figure 4.2 represents *xMerger*'s *Request* class hierarchy.

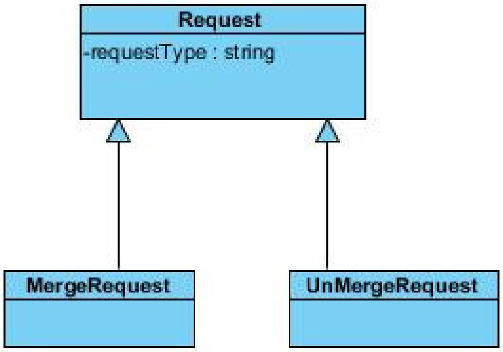

Figure 4.2. Request hierarchy.

After much analysis, we reached the conclusion that Request can exist either as the composite or independent type, depending on a database's structure and the relationships present among its entities. We noted that it would be easier to treat each and every request as an independent request. This finding was the motivation to introduce a variation of the Composite pattern inside the *MergeRequest* class design. The intent of this pattern is to "compose" merge requests into tree structures to represent part-whole hierarchies and still treat these requests in the same way as an individual instance of the *MergeRequest* class. For Merger, the heart of this pattern is the ability of a client to perform a merge operation using a *MergeRequest* class object without needing to know that there are many requests within it.

To keep this concern in mind we have created a specialized class of the *MergeRequest* class, called *LinkedMergeRequest*. This class object comes into existence when a client wants to create a composite request and so provides the attributes involved in one-to-many or many-to-many relationships present between two databases entities. Table 4.2.2 explains the requirements for these three different types of *LinkedMergeRequest*.

Table 4.2.2. The Three Types of the LinkedMergeRequest Class.

S.No.	Condition	Request Type
1.	One to many relationship between entities (From Parent to child)	*ParentToChildLinkedMergeRequest*
2.	One to many relationship between entities (From child to Parent)	*ChildToParentLinkedMergeRequest*
3.	Many to many relationship between two entities or with the same entity	*ManyToManyLinkedMergeRequest*

In the case of CHARM, we can create a representative composite *MergeRequest* class object that itself contains instances of other *MergeRequest* classes, as shown in Figure 4.3.

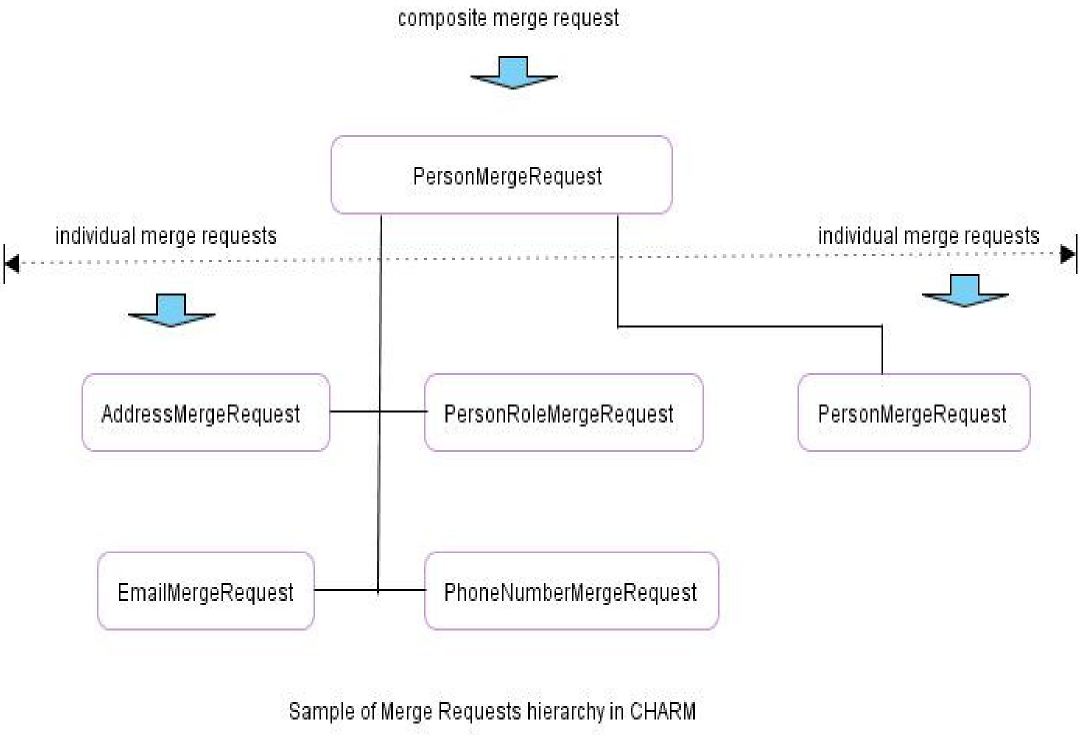

Figure 4.3. Example of MergeRequest in CHARM.

According to the *MergeRequest* class design, a client is able to create a *MergerRequest* object, as depicted in Figure 4.3. The top-level *MergeRequest* class instance named *PersonMergeRequest* is a composite request, and all other requests are part of this composite object. Note that *xMerger* treats this composite merge request the same as the other individual merge requests present inside it. Figures 4.4 show the detailed design of the *MergeRequest* class.

The *MergeRequest* class contains a list for the *SourceRecordID* class, a destination *RecordID* instance data member, and a list for the *FieldAction* class. Each instance of the *FieldAction* class within this list container is associated with the aforementioned destination *RecordID* data member. Each *RecordID* class object contains a unique record id and a timestamp. For this unique record id, the *RecordID* object also has its associated field name. Using this field and ID, the *Merger* extracts the record's profile information from the appropriate database. Accordingly, the *Merger* can extract source and destination record information through this *RecordID* object.

xMerger uses the timestamp details present in a *MergeRequest* class object's source and destination *RecordID* class objects for optimistic concurrency control. The timestamp denotes the last time the record whose id is held by this object was modified. By comparing timestamp values, we verify whether or not a record has actually been changed. *xMerger* aborts the merge process if any one of these timestamps for any *RecordID* differs in the value held by the matching database record. False positives are an unmatched timestamp for any *RecordID* object in a *MergeRequest* class object and its matching database record. These are not acceptable in Merger. The *xMerger's* process aborts without generating any warnings, but clearly indicates failure.

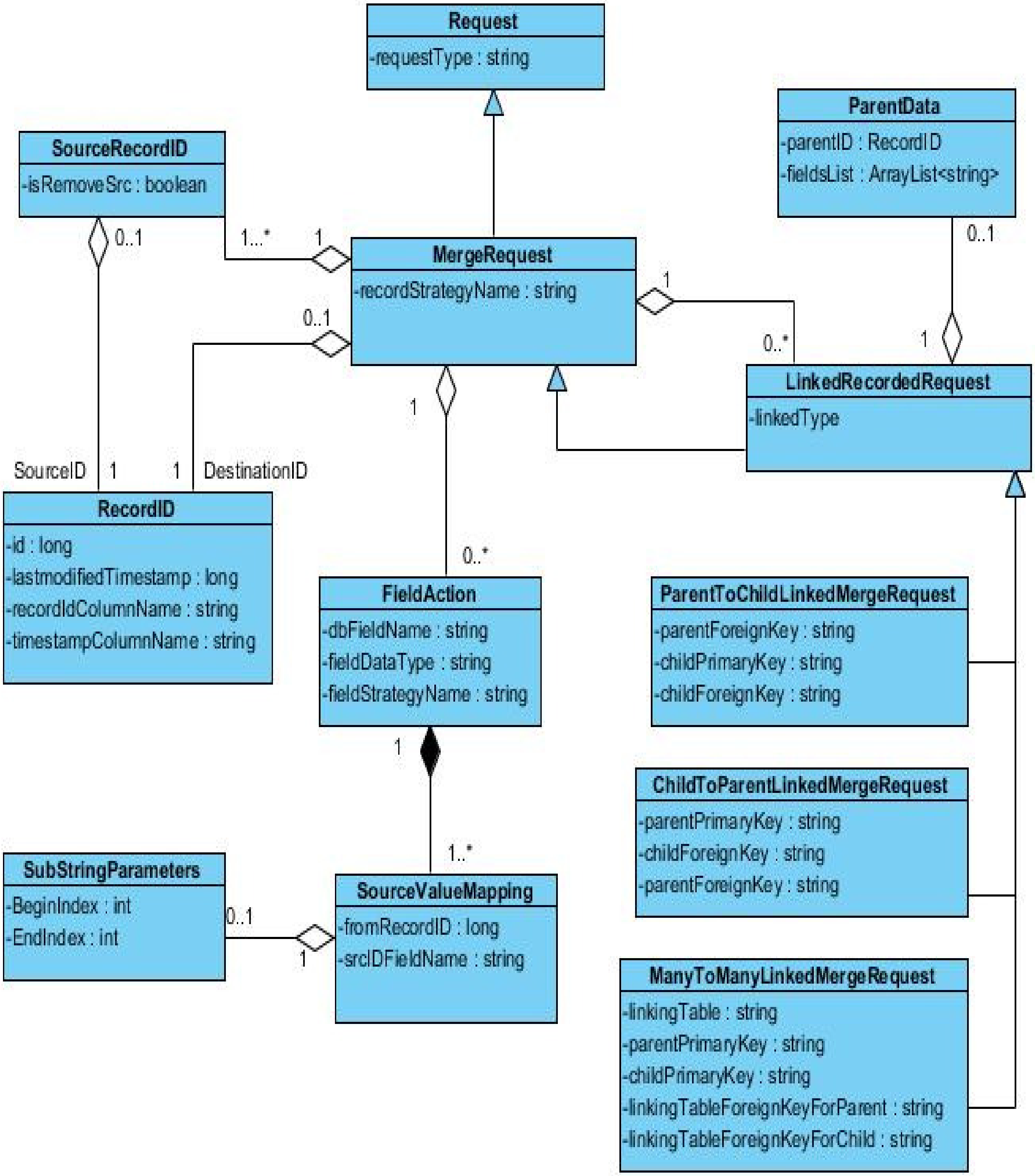
Request
-requestType : string
ParentData
-parentID : RecordID
-fieldsList : ArrayList<string>
SourceRecordID
-isRemoveSrc : boolean
0..1
1...*
1
MergeRequest
-recordStrategyName : string
1
0..1
0..*
1
0..1
LinkedRecordedRequest
-linkedType
1
SourceID
1
1
DestinationID
RecordID
-id : long
-lastmodifiedTimestamp : long
-recordIdColumnName : string
-timestampColumnName : string
0..*
FieldAction
-dbFieldName : string
-fieldDataType : string
-fieldStrategyName : string
ParentToChildLinkedMergeRequest
-parentForeignKey : string
-childPrimaryKey : string
-childForeignKey : string
1
ChildToParentLinkedMergeRequest
-parentPrimaryKey : string
-childForeignKey : string
-parentForeignKey : string
1..*
SubStringParameters
-BeginIndex : int
-EndIndex : int
0..1
1
SourceValueMapping
-fromRecordID : long
-srcIDFieldName : string
ManyToManyLinkedMergeRequest
-linkingTable : string
-parentPrimaryKey : string
-childPrimaryKey : string
-linkingTableForeignKeyForParent : string
-linkingTableForeignKeyForChild : string

Figure 4.4. MergeRequest class diagram.

MergerRequest also provides strategy details at the record and field levels for its destination *RecordID* instance data member. *MergeRequest's* data member *recordStrategyName* represents a simple action to be performed on the core database (e.g., insert an entity or edit an entity) for a destination record. *SourceRecordID*s can be left as they are or be removed from the database after the merging process, depending on the isRemove attribute's value as set by the client.

The *FieldAction* class specifies the simple action to be performed on a specific field of the destination record and should contain the data needed to perform that action. On the client side, any specifics regarding how an action/command is performed should be hidden; on the server side, the execution of an action should be simple and straightforward. To meet these requirements, we have created a strategy pattern design variation. The strategy pattern is used to maintain separation of functionality at the client end vs. the server end.

Figure 4.5 represents the design of the *UnMergeRequest* class which participates in the un-merge process of *xMerger* and corresponds in explanation to the *MergeRequest* class.

Unlike merge, in the case of un-merge *xMerger* generates multiple destination records from a single source record. This requirement of the *UnMergeRequest* class design differs from the *MergeRequest* design, specifically the *UnMergeRequest* class is composed of one *SourceRecordID* and multiple destination records represented by *DestinationRecordInfo* class objects. In the *MergeRequest* design description we have already discussed the design decisions of classes *SourceRecordID* and *RecordID*. Here, our focus is on the remaining classes that are part of *UnMergeRequest.*

Our requirement here is to create from a single source record a destination record that is a logical record composed of multiple fields. So, while creating a new destination record, either it can overwrite one of the source's fields value for its specific field or the user can specify its value in case of conflicts.

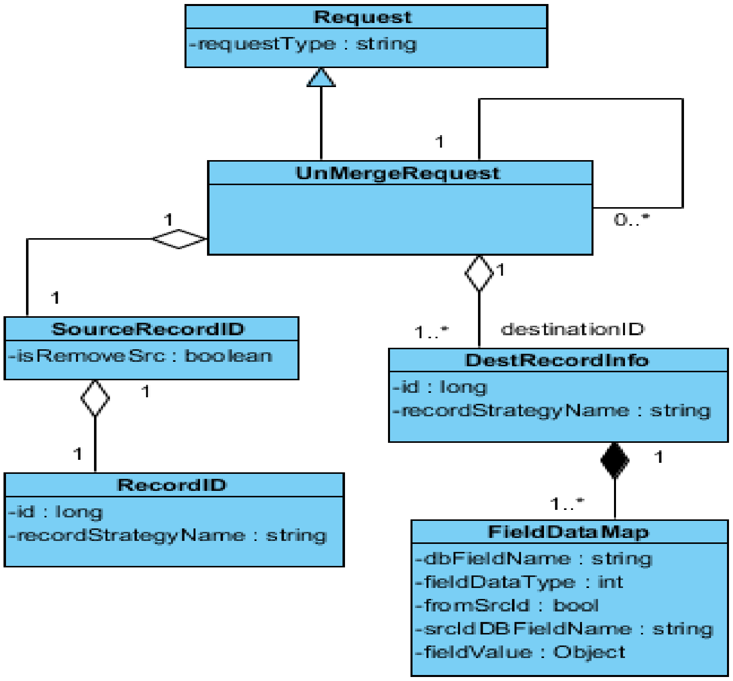

Figure 4.5. UnMergeRequest class diagram.

These conflicts can only occur when many merge operations have been been performed on a destination record already. In such a case, the user is responsible to mention its value explicitly to avoid conflicts. Considering this requirement, we designed *DestRecordInfo,* represented by a data member *id* (unique id) and *recordStrategyName,* which describes the action need to perform on this *DestinationRecordInfo* object. This *recordStrategyName* could be an insert or update action on a core database for this destination. Each field of a destination record is represented by a *FieldDataMap* object, specifically identified by the *dbFieldName* attribute of this object. *FieldDataMap* class data member *fromSrcID* state this field value status. If it is true, which means copy its value from given source record’s field represented by *srcIdDBFieldName,* otherwise user will have to provide its value by populating *fieldValue* attribute.

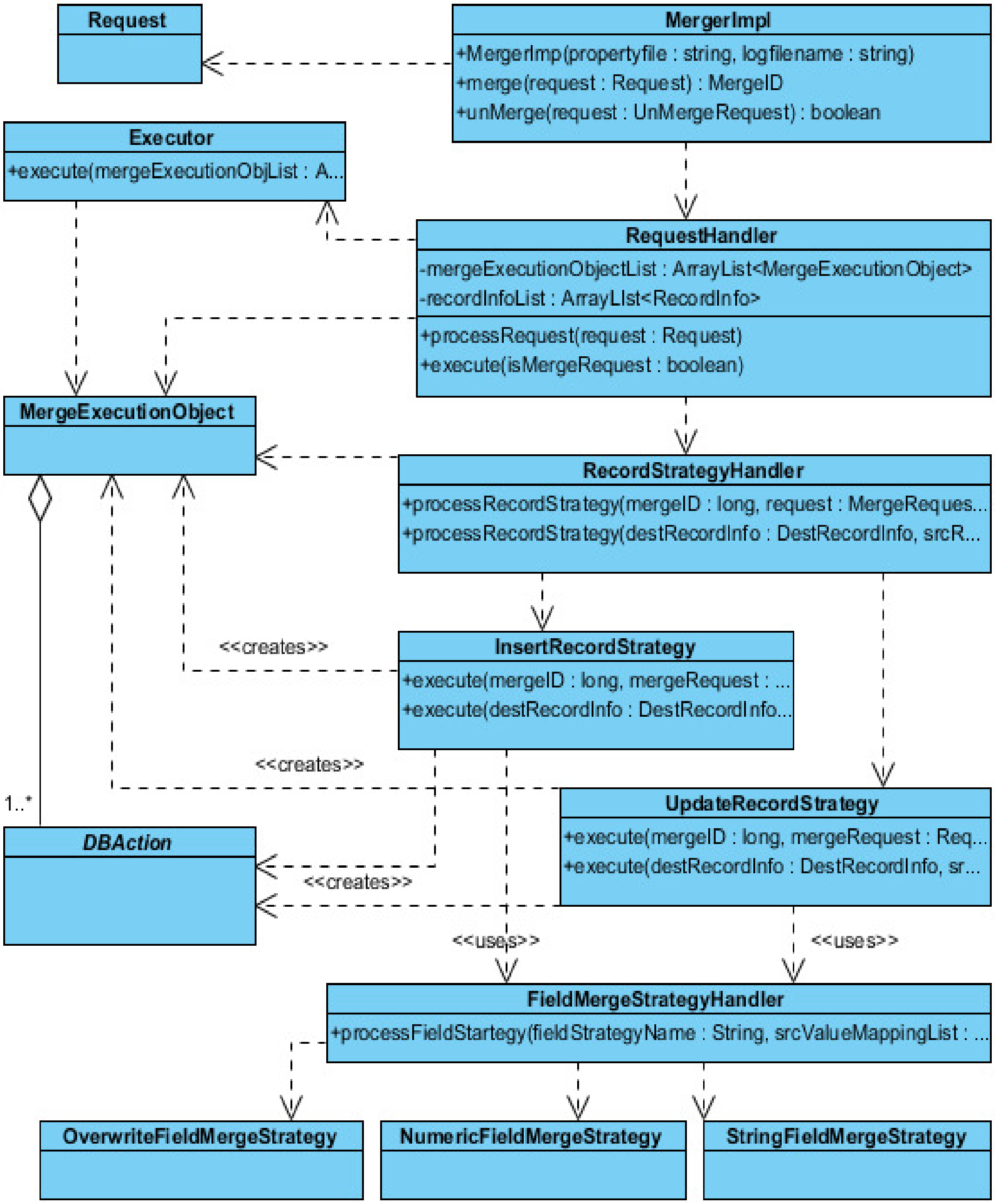
Request
MergerImpl
+MergerImp(propertyfile : string, logfilename : string)
+merge(request : Request) : MergeID
+unMerge(request : UnMergeRequest) : boolean
Executor
+execute(mergeExecutionObjList : A...
RequestHandler
-mergeExecutionObjectList : ArrayList<MergeExecutionObject>
-recordInfoList : ArrayLIst<RecordInfo>
+processRequest(request : Request)
+execute(isMergeRequest : boolean)
MergeExecutionObject
1
RecordStrategyHandler
+processRecordStrategy(mergeID : long, request : MergeReques...
+processRecordStrategy(destRecordInfo : DestRecordInfo, srcR...
<<creates>>
InsertRecordStrategy
+execute(mergeID : long, mergeRequest : ...
+execute(destRecordInfo : DestRecordInfo...
<<creates>>
1..*
UpdateRecordStrategy
+execute(mergeID : long, mergeRequest : Req...
+execute(destRecordInfo : DestRecordInfo, sr...
DBAction
<<creates>>
<<uses>>
<<uses>>
FieldMergeStrategyHandler
+processFieldStartegy(fieldStrategyName : String, srcValueMappingList : ...
OverwriteFieldMergeStrategy
NumericFieldMergeStrategy
StringFieldMergeStrategy

Figure. 4.6. Merger class diagrams.

Figure 4.6 shows a complete view of *xMerger*, including its components and their dependencies among each other. *MergerImpl* is a class that implements *xMerger's interface* and provides actual an implementation of merger and un-merge services of *xMerger.*

Once *xMerger* gets a request to process, *RequestHandler* becomes responsible to process this request and return the final result back to *MergerImpl.* Through *Request* classes design decisions, we are well aware that incoming requests contain strategy information at two separate levels, i.e., record and field. To keep these functionalities and responsibilities separate, we designed two separate handlers, namely:

- *RecordStrategyHandler* is responsible for handling record level strategies and depends on *RecordStrategy* classes to achieve its objective by invoking required by the *RecordStrategy class*, i.e., *UpdateRecordStrategy* or *InsertRecordStrategy* as per information present in the *Request* class. Figure 4.7 provides a detailed description of *RecordStrategy* classes.
- *FieldStrategyHandler* is responsible for handling field level strategies and depends on *FieldMergeStrategy* classes to achieve its objective by invoking a required method of *FieldMergeStatetgy* subclasses according to the strategy suggested in the *Request* class object sent by the client. Figure 4.8 provides a detailed description of *FieldMergeStrategy* classes and its design decisions.

All three mentioned handler classes are also responsible for holding intermediate objects generated while processing requests, and at the end, *RequestHandler* gathers intermediate objects all together and creates a *MergeExecution* object for final processing. The *MergeExecution* class object is composed of *DBAction* objects, which are responsible to generate SQL queries to execute them later at the end of request processing. Figure 4.9 illustrates *DBAction*.

RequestHandler also communicates with *Executor* once it prepares the complete *MergerExecution* object to execute. *Executor* executes this intermediate *MergeExecution* object on the database for final processing. If execution is successful, *Executor* returns a successful result; otherwise, it aborts the merge process and returns un-successful result.

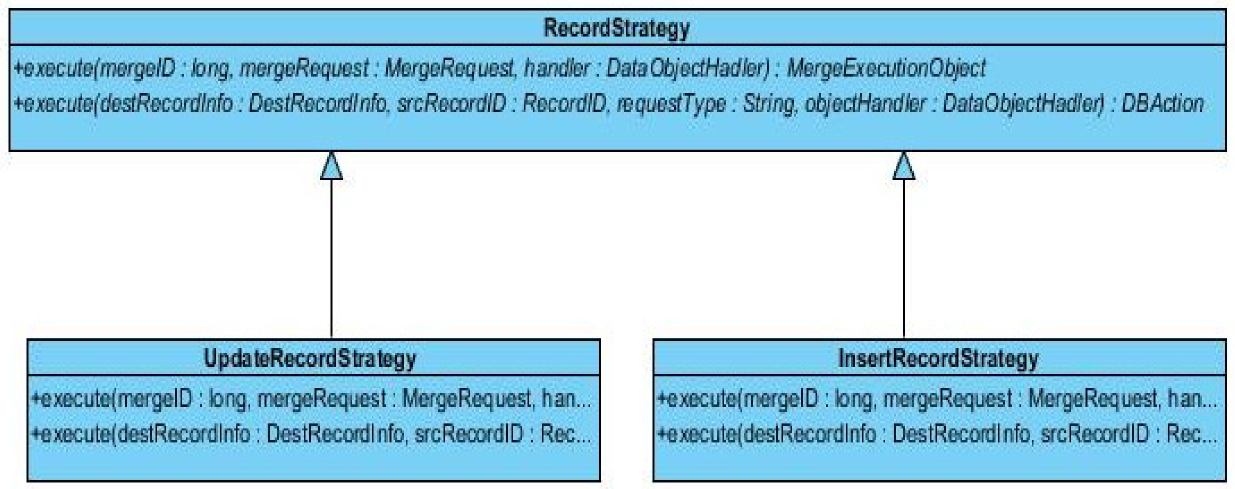
RecordStrategy
+execute(mergeID : long, mergeRequest : MergeRequest, handler : DataObjectHadler) : MergeExecutionObject
+execute(destRecordInfo : DestRecordInfo, srcRecordID : RecordID, requestType : String, objectHandler : DataObjectHadler) : DBAction
UpdateRecordStrategy
+execute(mergeID : long, mergeRequest : MergeRequest, han...
+execute(destRecordInfo : DestRecordInfo, srcRecordID : Rec...
InsertRecordStrategy
+execute(mergeID : long, mergeRequest : MergeRequest, han...
+execute(destRecordInfo : DestRecordInfo, srcRecordID : Rec...

Figure 4.7. RecordStrategy hierarchy.

Figure 4.7 presents a closer look at the record strategy classes involved in *xMerger* and its hierarchy. A record can be updated, inserted, and deleted in a database while performing un-merge/merge operations via *xMerger*. The *RecordStrategy* class is designed as an abstract class with two overloaded abstract *execute* methods in it. This design decision allows us to keep update and insert record strategies separate while still sharing common functionality. There are two subclasses of *Recordstrategy*:

- *UpdateRecordStrategy* is responsible to generate *UpdateDBAction* and *DeleteDBAction* objects for source and destination records if required. Further, to achieve this objective, this class also communicates with *FieldStrategyHandler* to generate intermediate multiple new field objects that help in creating mentioned *DBAction* objects.

- *InsertRecordStrategy* is responsible to generate *InsertDBAction and DeleteDBAction* objects for source and destination records if needed. Like *UpdateRecordStrategy*, this class communicates with *FieldStrategyHandler* to achieve its objective.

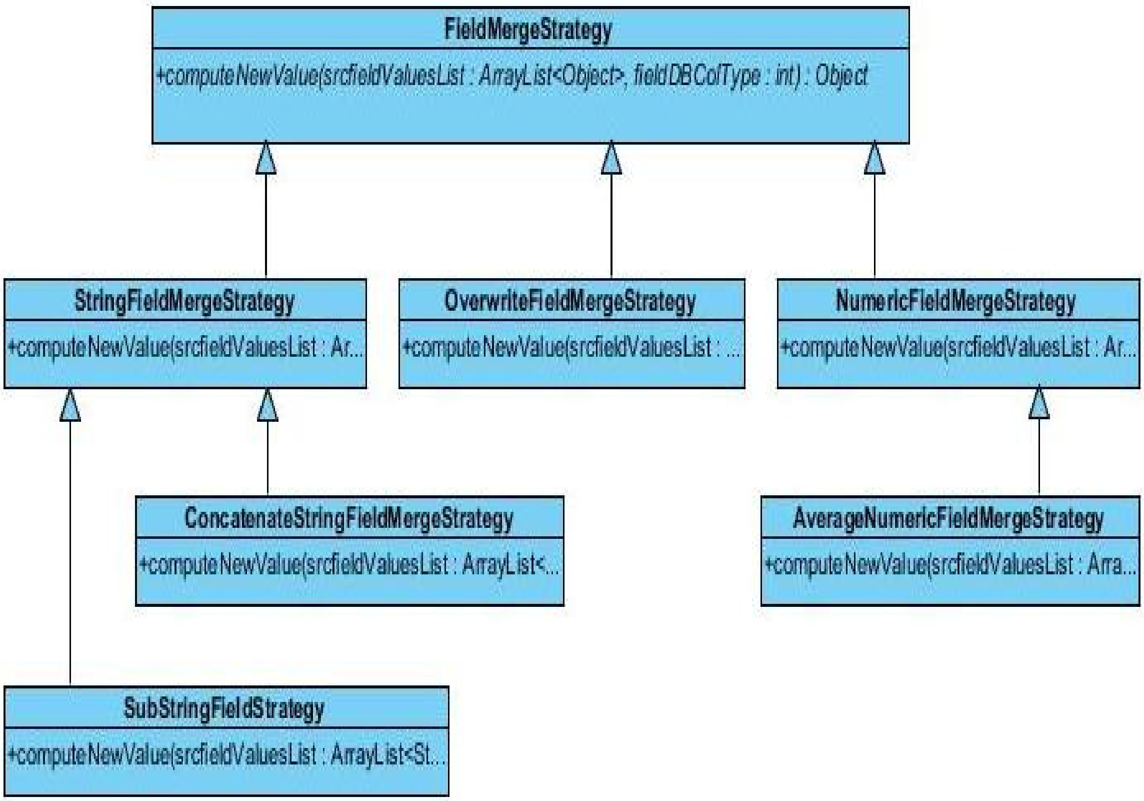
FieldMergeStrategy
+computeNewValue(srcfieldValuesList : ArrayList<Object>, fieldDBColType : int) : Object
StringFieldMergeStrategy
+computeNewValue(srcfieldValuesList : Ar...
OverwriteFieldMergeStrategy
+computeNewValue(srcfieldValuesList : ...
NumericFieldMergeStrategy
+computeNewValue(srcfieldValuesList : Ar...
ConcatenateStringFieldMergeStrategy
+computeNewValue(srcfieldValuesList : ArrayList<...
AverageNumericFieldMergeStrategy
+computeNewValue(srcfieldValuesList : Arra...
SubStringFieldStrategy
+computeNewValue(srcfieldValuesList : ArrayList<St...

Figure 4.8. FieldMergeStrategy hierarchy.

Figure 4.8 presents a closer look at field strategy classes involved in *xMerger*. *FieldStrategyHandler* is responsible for invoking the required *FieldMergeStrategy* for a specific field of a source or destination record. Similar to the *RecordStrategy* class, the *FieldMergeStrategy* class is also designed as an abstract class, which forces its subclasses to override a *computeNewValue* method, whose objective is to create a new value for a specific field as per a given field level strategy, present in *FieldAction* objects of a *Request* class. Each subclass of *FieldMergerStrategy* class has its own responsibility, and these subclasses are described below:

- *StringFieldMergeStrategy* is responsible to compute new values for fields having string data types. This class is again sub-classed in below the above mentioned classes, in order to share common functionality among string-type merge strategies and also provide flexible structure for future enhancement.
    - *SubStringFieldMergeStrategy* is responsible to compute a new string object by performing a sub-string operation on a source field value.
    - *ConactenateStringFIeldMergeStrategy* is responsible to compute a new string object by concatenating all source field values provided to its compute method. Currently, this class concatenates field values with a space and returns a single concatenated string.
- *OverwriteFieldMergeStrategy* is responsible to copy the source field value for a destination field value and return it. This strategy applies for any data type. This is the simplest strategy to deal with.
- *NumericFieldMergeStrategy* is responsible to compute a new value for a destination field whose sources field's data type is numeric. This design allows us to add new numeric strategies under this class in future. Currently, we only have one strategy under this class which is described below:
    - *AverageFieldMergeStartegy* is responsible to compute average value of all sources field's value, which will be a new value for a destination field.

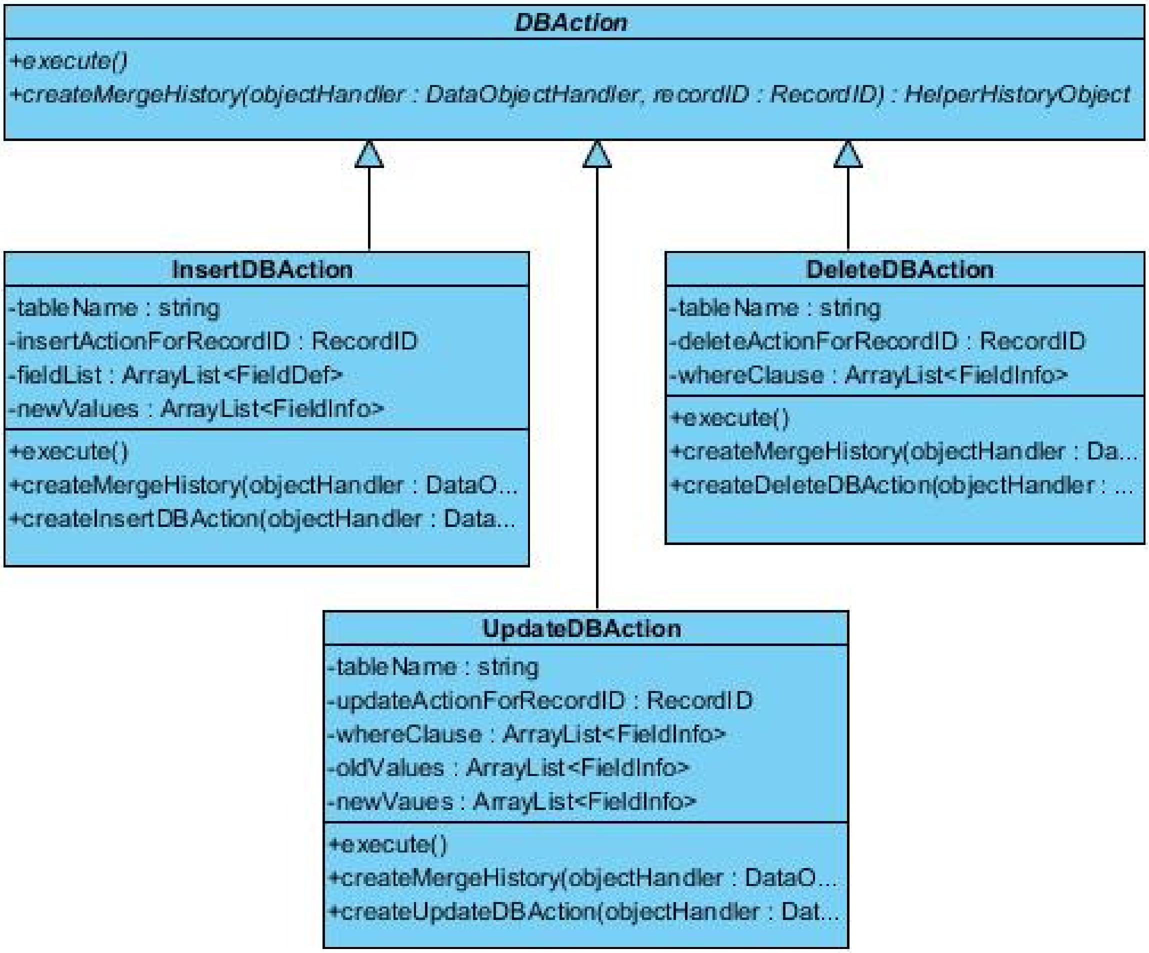
DBAction
+execute()
+createMergeHistory(objectHandler : DataObjectHandler, recordID : RecordID) : HelperHistoryObject
InsertDBAction
-tableName : string
-insertActionForRecordID : RecordID
-fieldList : ArrayList<FieldDef>
-newValues : ArrayList<FieldInfo>
+execute()
+createMergeHistory(objectHandler : DataO...
+createInsertDBAction(objectHandler : Data...
DeleteDBAction
-tableName : string
-deleteActionForRecordID : RecordID
-whereClause : ArrayList<FieldInfo>
+execute()
+createMergeHistory(objectHandler : Da...
+createDeleteDBAction(objectHandler : ...
UpdateDBAction
-tableName : string
-updateActionForRecordID : RecordID
-whereClause : ArrayList<FieldInfo>
-oldValues : ArrayList<FieldInfo>
-newVaues : ArrayList<FieldInfo>
+execute()
+createMergeHistory(objectHandler : DataO...
+createUpdateDBAction(objectHandler : Dat...

Figure 4.9 DBAction Classes.

Figure 4.9 presents a closer look at DBAction classes and their hierarchy. These classes are object form of SQL statements and responsible to perform appropriate actions on core the database, such as insert, update and delete. Similar to the above discussion, the *DBAction* class is also designed as an abstract class that provides its subclasses to have additional functionality apart from its own functionalities. The *execute()* method of each subclass is responsible to execute that action at the database level by creating an SQL statement from its own object. As shown in Figure 4.9, subclasses of *DBAction class* are as follows:

- *InsertDBAction* represents an object form of an insert SQL statement and on execute insert a record into a database. The *createMergerHistory()* method of this class generates a helper object whose action type is to delete.
- *DeleteDBAction* represents an object form of delete SQL statement and on execute delete a record from a database. The *createMergerHistory()* method of this class generates a helper object whose action type is to insert.
- *UpdateDBAction* represents an object form of an update SQL statement and on execute update a record in a database. The *createMergerHistory()* method of this class generates a helper object whose action type is to update.

4.2.3 Information Flow of *xMerger* (In the Case of Merging)

This section provides a detailed view of information flow among *xMerger*'s classes and how intermediate objects are created during processing a request for merging. Figure 4.10 provides a complete view of *xMerger* process through sequence diagrams.

Figure 4.11, detailed view of ref frame of Figure 4.10, illustrates the information flow among record strategy classes and creation of intermediate generated *DBAction* objects responsible for final execution; collectively create a list of action objects handle by *Executor*.

Executor is responsible for final execution of DBAction objects generated intermediately. Successful execution of these objects decides success of merge process and return unique merge id as a result, to identify this merging process. Any unsuccessful execution of these DBAction objects will generate the exception and halt the merging process.

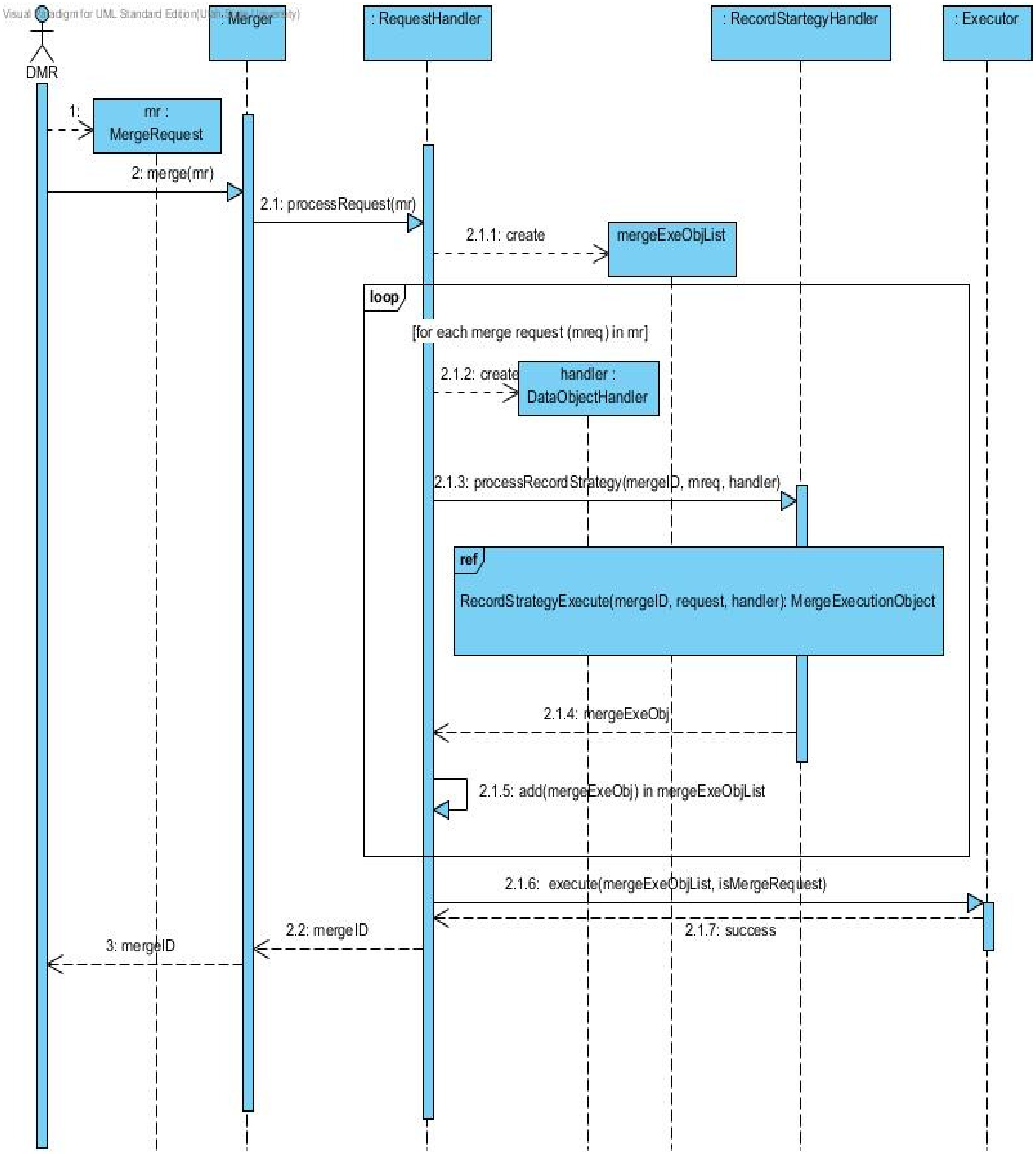
DMR
: Merger
: RequestHandler
: RecordStartegyHandler
: Executor
1:
mr :
MergeRequest
2: merge(mr)
2.1: processRequest(mr)
2.1.1: create
mergeExeObjList
loop
[for each merge request (mreq) in mr]
2.1.2: create
handler :
DataObjectHandler
2.1.3: processRecordStrategy(mergeID, mreq, handler)
ref
RecordStrategyExecute(mergeID, request, handler): MergeExecutionObject
2.1.4: mergeExeObj
2.1.5: add(mergeExeObj) in mergeExeObjList
2.1.6: execute(mergeExeObjList, isMergeRequest)
2.1.7: success
2.2: mergeID
3: mergeID

Figure 4.10. Overall information flow while merging.

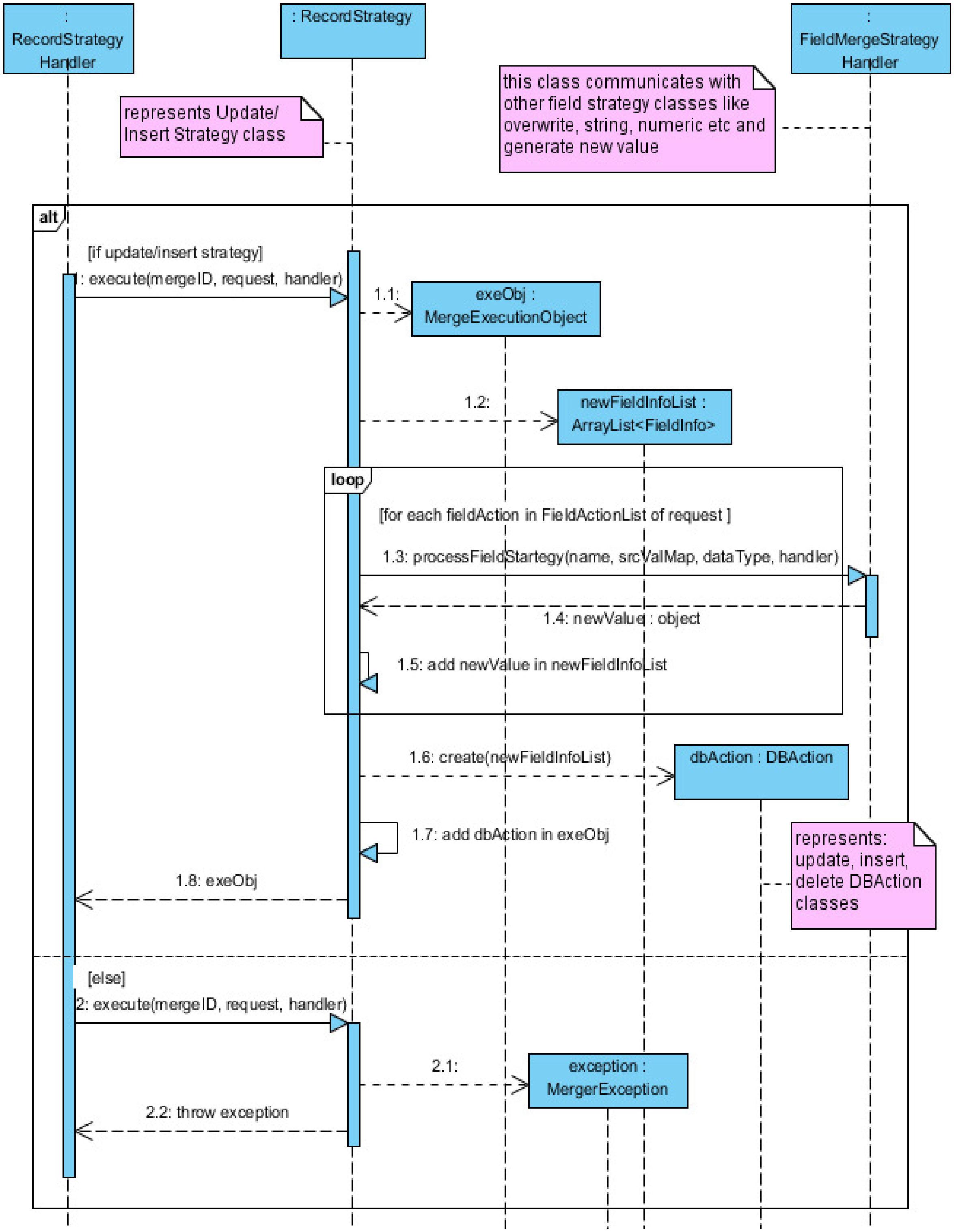
: RecordStrategy Handler
: RecordStrategy
: FieldMergeStrategy Handler
represents Update/ Insert Strategy class
this class communicates with other field strategy classes like overwrite, string, numeric etc and generate new value
alt
[if update/insert strategy]
1: execute(mergeID, request, handler)
1.1:
exeObj : MergeExecutionObject
1.2:
newFieldInfoList : ArrayList<FieldInfo>
loop
[for each fieldAction in FieldActionList of request]
1.3: processFieldStartegy(name, srcValMap, dataType, handler)
1.4: newValue : object
1.5: add newValue in newFieldInfoList
1.6: create(newFieldInfoList)
dbAction : DBAction
1.7: add dbAction in exeObj
represents: update, insert, delete DBAction classes
1.8: exeObj
[else]
2: execute(mergeID, request, handler)
2.1:
exception : MergerException
2.2: throw exception

Figure 4.11. Overall information flow in record strategies.

4.2.4 *xMerger* at Package Level

Now considering internal details, *xMerger* is composed of several packages that organize the elements of *xMerger* into related groups to minimize dependencies among them. Figure 4.12 describes the overall package structure in this system:

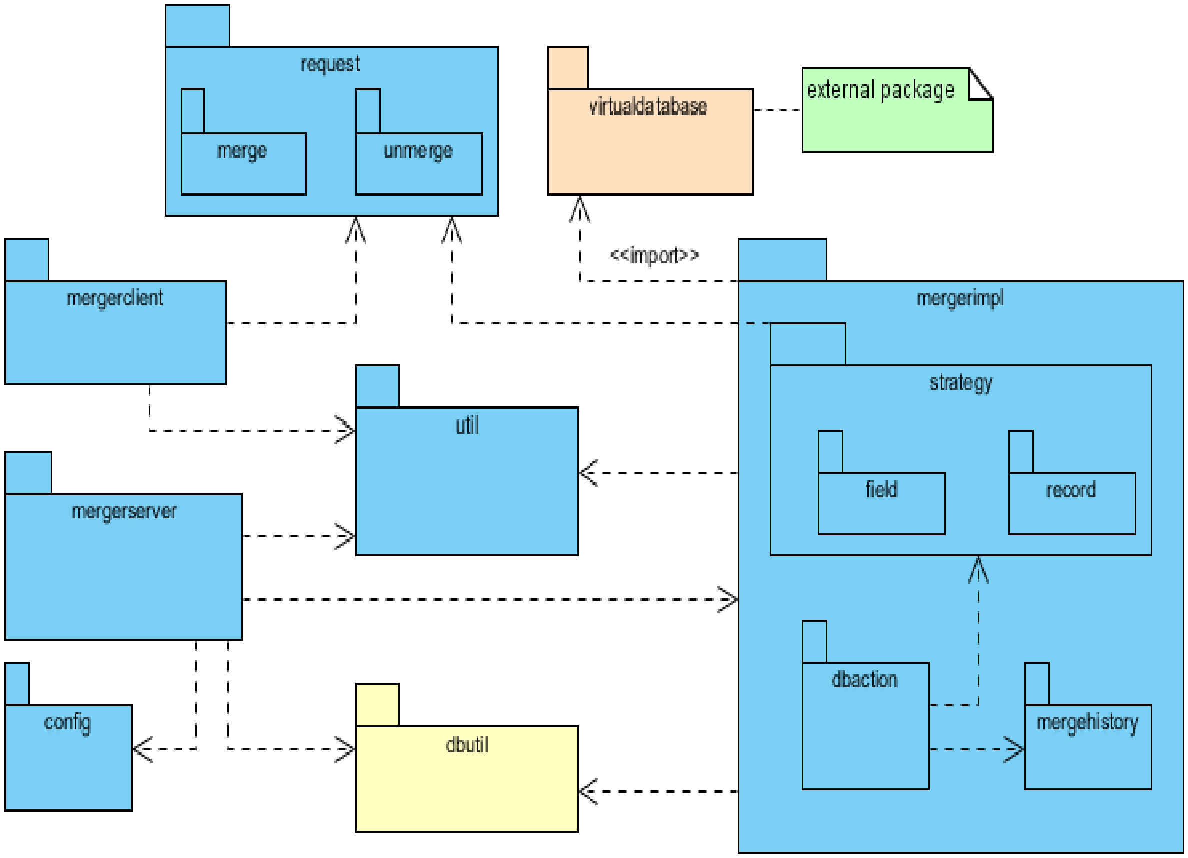

Figure 4.12.Package level diagram of xMerger and its dependencies.

CHAPTER 5

IMPLEMENTATION DETAILS

5.1 Introduction

xMerger as a client- server application that can run on different platforms with different databases to provide un/merging services. *xMerger* needs database-metadata information to support different databases. To achieve this, we designed and implemented an independent utility named Virtual-database that provides the complete metadata information of a database. This chapter focuses on the technology used and implementation issues faced while implementing *xMerger*.

5.2 Technology

xMerger resorts to Java's platform independent feature and its wide support for various relational databases with JDBC to provide platform independency together with support to different databases, as discussed in Section 3.2. This means any database that has a JDBC driver and runs on a system that has a JVM can be integrated with *xMerger*. Currently, *xMerger* only runs on SQL embedded databases. All the queries in the system are executed through Java JDBC. So, for different database servers, users can apply a different JDBC driver to use *xMerger* without changing the code for queries, except for the configuration constants.

5.3 Virtual-Database

Virtual-database provides an object that represents an object-relational mapping. This technique converts data between incompatible type systems in relational databases and object-oriented programming languages. This creates, in effect, a “virtual object database” that can be used form within the programming languages.

Virtual-database is able to represent the relational model into an object model. This is done through the mapping among attributes of the relational model with the attributes of the object model. A table maps to the class and all columns, where each column again maps to another class, which in turn maps to one of the properties in the class corresponding to table. It also provides the facility to hold the value of each and every column of a table as an object of another class.

Virtual-database can be imported as a jar into any system and provides the entire database tables’ details, such as, all tables present in database, all columns present in each table, and relationships among tables in the form of imported and exported keys list. It is intelligent enough to distinguish between one-to-one and one-to-many relationships.

This utility is also implemented in Java to maintain its independency among platforms and to support *xMerger*.

5.4 Logging

For debugging and maintenance purposes, *xMerger* allows flexible logging mechanisms by using log4j framework. The advanced features of log4j allow the user to freely choose what to log, how to log, and where to log.

Our designed debug log logs all the details of a merging/ unmerging process. Users can turn this log on or off through the starting server scripts.

5.5 Implementation Details and Challenges

As mentioned before, *xMerger* runs as a stand-alone server for easy adaptation. To hide the network handling from the users, we implemented a general purpose merger client that builds up the network connection and serializes the request.

A JDBC connection and RMI are used by distributed components to communicate with each other. A client sends messages to the server through RMI. RMI ensures delivery of messages and has a built-in buffering mechanism to handle concurrent calls.

For an easy start *xMerger* server, we created a command file (.cmd) for starting the service in Windows, also a batch file (.sh) for starting the server in Unix. The command file or batch file allows users to configure the JDBC driver, configuration files, and log files.

The design of *xMerger* involves dozens of classes and configuration artifacts. To verify successful implementation of these classes, we have done extensive testing from the bottom to top. Unit testing is indispensable for each of the basic classes to make sure they function properly. A system test is conducted with dummy data and databases. A regression test is conducted from time to time when a new component is integrated into the system.

As a final overall test, we needed an integrated system to test the adaptation of *xMerger* and evaluate it. Chapter 6 presents the details of this testing.

5.6 Discussion and Future Improvements

5.6.1 Design Quality

The strength of *xMerger* mainly comes from it flexible design. The following sections discuss the design decisions that contribute to *xMerger's* great flexibility and how the flexible design can be applied to various systems.

xMerger's design provides enough flexibility to handle its own responsibility, such as changing the logic of handlers and adding new record, field strategies and database actions, etc. The composite pattern present in the Request design lets client treat individual objects and compositions uniformly. The key concept is that we can manipulate a single instance of the object just as we manipulate a group of them, and this makes the code less complex and therefore, less error prone. The variation of strategy patterns included provide different strategies at both the record level and the field level encapsulate each one as object and make them interchangeable. This makes design more flexible in terms of adding and removing strategies.

xMerger interacts with users at run time, so it requires finishing the merging process as fast as possible. The composite design for the requests fits this requirement by executing the leave level requests first, and then composite requests one by one. With these features, we expect the merging request in *xMerger* to finish in less than one second in a database. However, the speed of handling one merging also depends on the speed of the SQL queries, the network connection, and communication with other database to update id mapping.

5.6.2 Adaptation for Various Integrated Systems

xMerger is a general purpose merging system for integrated information systems. It is applicable for systems in fields like medical care, business, customer service, etc. The following are some basic requirement for adapting the system:

- The system supports Java 1.4.
- The system uses a relational database system.
- The system has a database that has an appropriate JDBC driver.

5.6.3 Future Enhancement

- The current version of *xMerger* relies on the RMI internal multi-threading functionality to handle multiple requests. Moreover, connection pool size may create a long wait for requests to get a free connection object for its execution, hence, cause a later request to time out because of wait. To eliminate this dependency and fix this problem, *xMerger* could have its own multithreading system to handle requests.

- The configuration file provides flexibility for an initial setup process like database url, username, password, and number of connections and other database information like sequence ids associated with tables. In future versions, we could have a configuration tool that visualizes the XML configuration file and allows users to modify the configuration file through a simple GUI interface. This tool will simplify maintenance of a system that changes frequently.
- The merging process generates merger history records to correct erroneous merged records in future. The user himself is responsible to get a detailed view of merge history data, but in future versions we could have a tool that can provide the flat detailed view of merge history objects present in persistent storage through a simple GUI interface. This would help users to create unmerge requests in a simple way.

CHAPTER 6

SOFTWARE TESTING

6.1 Introduction

Software testing is essential for ensuring the quality and usability of a product. To test performance and proper functioning of *xMerger*, unit testing integration testing and user acceptance testing is used. The next section discusses software testing in the context of the *xMerger*.

6.2 Unit Testing

Unit testing is a testing technique by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. In procedural programming, a unit may be an individual function or procedure. This method ensures that the code meets its design objectives and behaves as intended [21].

In *xMerger*, unit testing was done on almost all classes present in the system, that are responsible any processing. These classes include handlers, record strategies, field strategies, and database action classes as discussed in Chapter 4. Unit testing was done using logic-based testing and input validation, for which real data and random data were used. These test cases were designed such that they met functional requirement specifications of *xMerger.* This is discussed in more detail in the Appendix.

6.3 Integration Testing

Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before system testing. Integration testing takes as its input modules that have been unit tested, groups them into larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing [22].

In the integration testing of *xMerger*, test cases focused on the flow of data/information/control from one component to the other. This testing followed a bottom testing approach in the following order:

1) Testing of connection pool and other *dbutil* package classes.
2) Testing of *DbAction* classes like *UpdateDBAction*, *InsertDBAction* and *DeleteDBAction*.
3) Testing of field strategies and database actions.
4) Testing of record strategies, field strategies, and db actions.
5) Testing of handlers like record strategy and field strategy handler.
6) Testing of handlers, record strategies, field strategies and db actions.
7) Testing of request handle and executor with all strategies and actions.
8) Testing of merge history classes with step 7 to verify results.
9) Testing of merger implementer which include step 8, too.

Test data consist of different type of merge and unmerge requests like composite and independent requests. We covered all data-types such as float, date, and string.

6.4 User Acceptance Testing

User acceptance testing (UAT) is generally done before the delivery of a product. This project performed UAT with real data to check whether the end goals were achieved or not.

xMerger is integrated as a independent module into Child Health Advanced Records Management (CHARM), an integrated child-centric information system to achieve this testing. Moreover, CHARM is a perfect environment for system integration and adaptation testing of *xMerger* because it represents a variety of integrated information systems that can adapt *xMerger*. CHARM manages a central database, core data that stores demographic person information loaded from several participating programs, and *xMerger* accessed this central database for its testing.

CHAPTER 7

SUMMARY

As a general-purpose merger, *xMerger* is adaptable to any integrated person-centric information system that can be accessed by Java code and support an SQL query. *xMerger* works as a standalone server waiting for merging requests, processes these requests once it gets them, and generate a unified view of an individual.

xMerger relies on Java and its JDBC technology to resolve system level heterogeneities. Schematic and semantic heterogeneities and other data quality problems are resolved by allowing the client to provide strategies and specific details.

For verification of the effectiveness of the *xMerger* design, we adapted it and applied it to the CHARM systems. With domain knowledge of CHARM's participating programs together with an understanding of CHARM's central core database, we were able to configure *xMerger* efficiently.

In the application of *xMerge* to CHARM, we noted *xMerger* could be improved in the following ways:

- A GUI may be designed to facilitate the user to access and modify all the server side configuration files. Going through configuration files can be painful for some of the less technical users. A built in GUI would be a great help to this category of user. However, portability of the program may be limited because of limited GUI support on some client systems.
- Also from the design's perspective, we could add some new record and field strategies to

provide more flexibility to the system, although *xMerger*'s current design provides some facility to add new strategies without any modifications.

- There is a possibility to add new configuration details associated with different databases to generalize this system for all databases. This enhancement would make xMerger easier to be adapted by various types of integrated information systems with different databases.
- A GUI may be designed to provide users with a detailed view of the persistent merge history objects generated in a merging process.
- Another possible improvement for *xMerger* would be to make the integration of third party components easier and pluggable. This would require some adaptation and refactoring of the existing component.

Finally, *xMerger* provides an effective merging process to provide a unified view of data in an integrated information system. Adopting or developing *xMerger* requires awareness of the underlying issues, the key design choices, and the consequences of those choices. This report provides a starting point to achieve awareness and a foundation upon which to enhance these features in the future.

REFERENCES

[1] Clyde, S.W. *The Unique Records Portfolio*. Public Health Information Institute, 2006

[2] Clyde, S.W. *Overview for Public Health Leaders.* Public Health Information Institute, 2006

[3] Markle Foundation Working Group on Accurately Linking Information for Health Care Quality and Safety. Linking Health Care Information: Proposed Methods for Improving Care and Protecting Privacy. Markle Foundation, 2005.

[4] Clyde, S. Executive Summary: Child-Health Advanced Record Management Integration Infrastructure, CHARM Project. Utah Department of Health, Jan 2002.

[5] Wang, R.Y. and Strong, D.M. Beyond accuracy: What data quality means to data consumers. J. Management Information Systems, 12, 4 (1996), 5-33

[6] Levitin, A.V. and Redman, T.C. Data as a resource: properties, implications and prescriptions. Sloan Management Review, 40, 1 (1998) 89-101.

[7] Leitheiser, R.L. Data quality in health care data warehouse environments. In Proceedings of the 34th Hawaii International Conference on System Sciences, IEEE, 2001.

[8] Strong, D.M., Lee, Y.W., and Wang, R.Y. Data quality in context. Comms. ACM, 40, 5 (May 1997), 103-110.

[9] Van den Broeck, J., Argeseanu Cunningham, S., Eeckels, R., and Herbst, K. Data cleaning: Detecting, diagnosing, and editing data abnormalities. PLoS Med 2, 10 (2005): e267. doi:10.1371/journal.pmed.0020267.

[10] Gibbs, M.R., Shanks, G., and Lederman, R. Data quality, database fragmentation and information privacy. Surveillance and Society 3 1 (2002), 45-58

[11] http://www.connectingforhealth.org/commonframework/docs/T5_Background_Issues_Data.pdf Page 6. (July 2010)

[12] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. Journal of Data Semantics 4 (2005),146-171.

[13] Rahm, E. and Do, H. H. Data cleaning: Problems and current approaches. *IEEE Bulletin of the Technical Committee on Data Engineering 23*, 4 (2000), 3-13.

[14] Sheth, A. P. Changing focus on interoperability in information systems: From system, syntax, structure to semantics. In *Interoperating Geographic Information Systems*, M. F. Goodchild, M. J. Egenhofer, R. Fegeas, and C. A. Kottman Eds., Kluwer, Academic Publishers. 1999, 1-27.

[15] Dixon, M., Kohoutková, J., and Cook, S.,Jeffery, K., and Read, B. Managing heterogeneity in inter-operating medical information systems. In *10th ERCIM Database Research Group Workshop on Heterogeneous Information Management,* 1996.

[16] Kim, W. and Seo, J. Classifying schematic and data heterogeneity in multidatabase systems. *Computer 24* (1991), 12-18.

[17] Sheth, A. P. and Kashyap, V. So far (schematically) yet so near (semantically). In *Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems*, 1993.

[18] Sheth, A.P. and Larson, J.A. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys 22* , 3 (1990),183-236.

[19] George, D. Understanding structural and semantic heterogeneity in the context of database schema integration. *Journal of the Dept. of Computing, UCLan,* (May 2005), 29-44.

[20] http://en.wikipedia.org/wiki/Composite_pattern

[21] http://en.wikipedia.org/wiki/Strategy_pattern

[22] http://en.wikipedia.org/wiki/Use_case. (July 2010 Month year)

[23] http://www.bredemeyer.com/pdf_files/functreq.pdf. (July 2010)

[24] http://en.wikipedia.org/wiki/Object-oriented_analysis_and_design. (July 2010)

[25] Bass, L., Clements, P., and Kazman, R. (2003). Software Architecture in Practice, 2nd Ed. Addison-Wesley.

[26] http://www.python.org/~jeremy/pubs/thesis/node7.html

[27] http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2232154/pdf/procamiasymp00005-1067.pdf

[28] http://www.cs.washington.edu/education/courses/cse590q/04au/papers/Hernandez95.pdf

[29] http://en.wikipedia.org/wiki/Object-relational_mapping. (June 2010)

[30] http://en.wikipedia.org/wiki/Unit_testing. (July 2010)

[31] http://en.wikipedia.org/wiki/Integration_testing. (July 2010)

[32] Lenzerini, M. Data integration: A theoretical perspective. In *Symposium on Principles of Database Systems*, 2002.

[33] Sattler, K.-U., Conrad, S., and Saake,G. Adding conflict resolution features to a query language for database federations. In *Proceedings of 3rd International Workshop on Engineering Federated Information Systems*, 2000.

[34] http://www.legacyfamilytree.com/tipsNGSMayJun99.asp

APPENDIX

Appendix

FUNCTIONAL REQUIREMENTS AND USER GOALS

High-level Goals of *xMerger*

Below are the primary goals for *xMerger:*

- To create a unified view of data from multiple matching records, which could be single source duplicates, multi-source duplicates and overlapping records;
- To create complete and accurate merged record from conflicting and incomplete source records.
- To keep a record of every merge process for further verification and evaluation.
- To allow un-merging of erroneous merged records using merge history records.
- To enhance future matching by adding value to the merged records.

Functional Requirements

1. *Specify merging criteria and required parameters*

   Its purpose is to provide user-specified criteria like:

   1.1 Data sources information to extract data from it.

   1.2 Records- and fields-level merging strategies.

   1.3 Other configuration parameters using properties file.

2. It should provide merging and un-merging.
3. It should provide support for various types of integrated system architecture.
4. It should provide support to multiple platforms in an integrated system.

User Goals

A use case defines the interactions between external actors and the system under consideration to accomplish a goal. An actor specifies a role played by a person or system while interacting with the system [22]. There is one primary actor for *xMerger*, namely the *Database Owner*.

The *Database owner* (DB owner) is a person or other electronic system, sub-classified as *DB User* and *Interactive System*, respectively. A *DB User* is a person responsible for a particular database. Therefore, a *DB User* wants to maintain data quality by executing various cleaning activities on their owned database(s) and hence responsible to invoke *xMerger*'s services.

An *Interactive System* has the same role, but instead of a human, it is another electronic system that needs to periodically execute cleaning activities on database(s) for which it has rights and responsibilities. In the case of CHARM, *Deferred Match Resolver* is responsible for communicating with Merger to initiate merging activities.

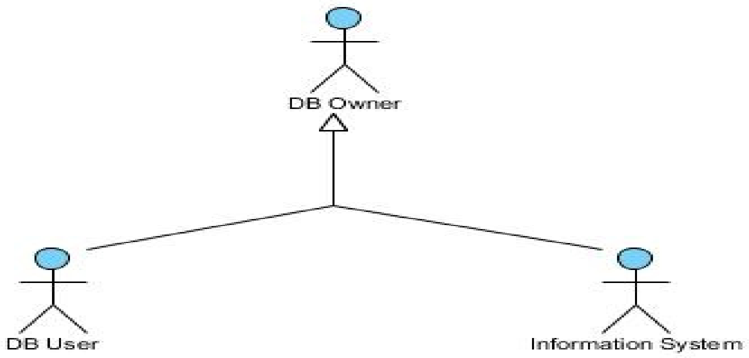

Figure A-1. Actors involved in the system.

Figure A-2 describes the goals of a *DB User* or an *Information System*. The user defines the merging requests once he finds any duplicate record in the system, that is needed to invoke the merge service provided by *xMerger* and to provide all the necessary information to merge two or more records into a single record. *DB Owner* can then create independent as well as composite merge requests if the records are interrelated. Similarly, *DB Owner* can define independent and composite unmerge requests for any erroneous merged records present in the system to provide required information to initiate un-merge service provided by *xMerger.*

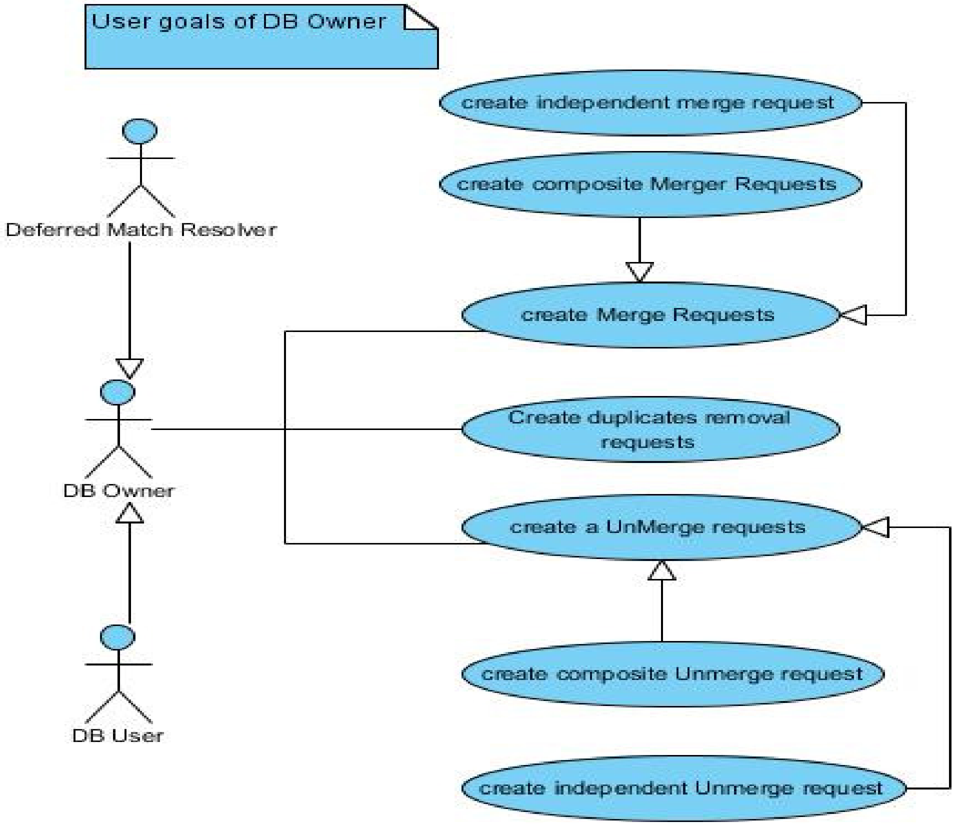
User goals of DB Owner
create independent merge request
create composite Merger Requests
Deferred Match Resolver
create Merge Requests
Create duplicates removal requests
DB Owner
create a UnMerge requests
create composite Unmerge request
DB User
create independent Unmerge request

Figure A-2. Goals associated with DB users and information systems.