A Prototype Automated Resolution Service For Public-Health Master Person Index

Sarvesh Jain

Follow this and additional works at: https://digitalcommons.usu.edu/gradreports

Part of the Computer Sciences Commons

Recommended Citation

Jain, Sarvesh, "A Prototype Automated Resolution Service For Public-Health Master Person Index" (2012). All Graduate Plan B and other Reports. 239. https://digitalcommons.usu.edu/gradreports/239

This Report is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Plan B and other Reports by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.
A PROTOTYPE AUTOMATED RESOLUTION SERVICE FOR PUBLIC-HEALTH

MASTER PERSON INDEX

by

Sarvesh Jain

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

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Stephen W. Clyde
Major Professor

Dr. Curtis Dyreson
Committee Member

Dr. Dan Watson
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah
2012
ABSTRACT

A Prototype Automated Resolution Service for a Public-health Master Person Index

by

Sarvesh Jain, Master of Science

Utah State University, 2012

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

A Master Person Index (MPI) is a software system that links together the records about individuals from a diverse set of data sources and that allows data consumers to retrieve correlated data for a given person. The Utah Department of Health recently contracted with Utah State University to build an MPI that links data from four initial data sources, and with enough extensibility that it will eventually handle dozens. This MPI, called the Utah Public-health Master Person Index or phMPI, was to include a special software component that automatically computed the “best version of truth” about each individual using correlated data from the various sources. This report documents the design, implementation, and testing of a prototype for that component. Because the phMPI’s design uses a service-oriented architecture, this component is a stand-alone software service that could be adapted and integrated into other MPI’s.
ACKNOWLEDGMENTS

I would like to thank Dr. Stephen Clyde for providing his valuable guidance and support throughout my graduate career. He taught me good software practice and design, which helped me to build strong programming concepts.

I would also like to thank my committee members, Dr. Curtis Dyreson and Dr. Dan Watson, for their support and valuable suggestions.

Finally, I am grateful to my family for their continuous love, support, and encouragement through all my academic pursuits.

Sarvesh Jain
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Data Chunks and their Attributes</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Metadata Table</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>An Example of the GOLDEN Record</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Data Quality Dimensions</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>A Table showing Duplicate Records</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Same-source duplicate records</td>
<td>14</td>
</tr>
<tr>
<td>2.7</td>
<td>Multi-source duplicate records</td>
<td>15</td>
</tr>
<tr>
<td>2.8</td>
<td>Overlapping records</td>
<td>15</td>
</tr>
<tr>
<td>2.9</td>
<td>Overlaying records</td>
<td>15</td>
</tr>
<tr>
<td>2.10</td>
<td>A table explaining wrongly merged record</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Timeliness Functions</td>
<td>41</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

2.1. An Overview of phMPI system .................................................................6
3.1. Use Case Diagram from Workflow Manager ..........................................21
3.2. Class Structure of the ARS ....................................................................25
3.3. Specializations of the DataChunkList class .............................................26
3.4. Specialization of DataChunk class ............................................................27
4.1. Package level diagram for the ARS ...........................................................31
4.2. A detailed package level diagram for ARS ..............................................32
4.3. Sequence diagram for empty DataChunkList ............................................34
4.4. Sequence diagram for one DataChunk in DataChunkList .......................35
4.5. Sequence Diagram for Calculation of Overall Quality Factor ...................36
4.6. Sequence diagram for no merging ............................................................37
4.7. Sequence diagram for merging between two or more records ..................38
4.8. Color schemes for inter-field dependency ...............................................42
4.9. Inter-field Dependencies in the Address class ...........................................43
4.10 Inter-field dependencies in the AdoptionEvent class ...............................43
4.11 Inter-field dependencies in the BirthEvent class .......................................44
4.12 Inter-field dependencies in the DeathEvent class .....................................45
4.13 Inter-field dependencies in the Email class ..............................................46
4.14 Inter-field dependencies in the Gender class ............................................46
4.15 Inter-field dependencies in the MaritalEvent class ....................................46
4.16 Inter-field dependencies in the Note class .................................................47
4.17 Inter-field dependencies in the PersonIdentifier class .............................47
4.18 Inter-field dependencies in the PersonName class .....................................48
4.19. Inter-field dependencies in the PersonRace class ..................................................48
4.20. Inter-field dependencies in the PersonRelation class .............................................49
4.21. Inter-field dependencies in the Phone class ..........................................................49
CHAPTER 1
INTRODUCTION

An integrated information system improves data sharing and provides better services to the users by connecting multiple isolated data sources to provide correlated sets of data to external systems or end users [1]. To achieve this, an integrated information system uses a software infrastructure that includes data converters, matchers, and mergers. Converters transform data from the various sources to a canonical form so those data can be safely and efficiently integrated. Matchers try to determine if a record from one source represents same entity (e.g., a person, place, or thing) as records from other sources. Mergers combine multiple records about the same entity to form a more complete or more accurate representation of that entity.

A *Master Person Index* (MPI) is a person-centric integrated information system that links person records across multiple data sources to provide unified views of data. An MPI can save time, money, and resources in many different application domains, but are considered particularly important in the healthcare domain, where inaccurate patient information can lead to medical errors, fraud, and medical/insurance identity thefts. Integrating patient’s information from an MPI can provide better patient care, more accurate and complete information, and can possibly save a patient’s life.

As mentioned above, mergers combine multiple records for the same real-world entity. These records are sometimes referred to as *overlapping duplicates* because some of the information that they contain may be the same, but other pieces of information may be unique [1]. Most MPI mergers combine overlapping duplicates by including all known values for particular field or set of related fields. For example, if there are five
different first names for a specific person, then the merger simply includes them all in the combined data. This approach to merging is referred to as *stacking* [1]. Other MPI mergers, however, try to determine which of those names is the most accurate or the *best version of truth*, instead of leaving that decision process up to the users of the MPI. This later approach, referred to here as *resolution*, is particularly valuable when the users of the MPI are other electronic systems and when those systems can’t handle multiple values for pieces of information like the first name.

The Utah Department of Health has contracted with Utah State University to build an MPI, called Utah’s Public-health Master Patient Index or *phMPI*, that must include a merger that uses resolution to determine the most current and accurate view of a person’s demographic data. This report describes a prototype for a re-usable software service that addresses this requirement for the phMPI and sets the stage for future research in efficient and effective resolution algorithms and flexible resolution services. We call this software component the *Automatic Resolution Service* (ARS), because it tries to resolve differences between overlapping duplicates, even if those records are inconsistent or incomplete.

Chapter 2 provides background information about integrated systems, the phMPI, and issues important to understand the challenges of person-centric data resolution. Chapter 3 then continues with a system analysis specific to the prototype ARS described in this report. The system analysis led to a service-oriented design that could be extended or adapted to other MPIs in future. Chapter 4 describes this design in the context of the phMPI and Chapter 5 highlights the interesting parts of the implementation. It also explains how the implementation uses an object gateway [24], based on the Vitruvian
DBObjects, as an interface to the phMPI database. The ARS was thoroughly tested in the context of the phMPI, using unit, integration, and system testing methods (see Chapter 6). Finally, Chapter 7 provides a summary of the project and discusses future enhancements for making the ARS more robust and flexible.
CHAPTER 2
BACKGROUND

To understand the complexities of the resolution problem and the contributions of the ARS, it is important to understand a few fundamental concepts of integrated information systems, and more specifically MPIs. Section 2.1 gives an overview of integrated information systems and Section 2.2 provides some background on Utah’s phMPI. Section 2.3 summarizes ten data quality dimensions that are commonly considered for information systems, in general. Of these, five are particularly relevant to the resolution problem in an MPI. Section 2.4 discusses these five problems in more detail and describes their relevancy to the resolution problem. Section 2.5 describes a few preparatory steps for data resolution and summarizes several approaches. Section 2.6 provides background information about inter-field dependency that can be used to apply one of the several approaches for data resolution.

2.1 Integrated Information Systems

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]. The integrated information systems can be useful in many arenas. Consider the following example in the health care arena, published in the July 2004 Connecting for Health. “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 was admitted to the emergency room, but was not able to recount his medical history.
Typically, a patient with symptoms suggesting a heart attack is given a blood thinner. However, the 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” [7].

Through integration infrastructures, integrated systems realize information sharing among related yet independently operating information systems. Depending on the system’s requirements, there are almost as many possible architectural designs for infrastructures as individual integrated systems. Further details on integrated information systems and stereotypical infrastructures are available in Chapter 2 of “The Unique Records Portfolio” [1].

2.2 Overview of Utah’s Public Health Master Person Index (phMPI)

As mentioned earlier, an MPI is an integrated system. It is often also a distributed system that gathers data from multiple heterogeneous information systems running on their own platforms. Utah’s Public-health MPI (phMPI) is such a system, with potential dozens of heterogeneous data sources, each hosted on its own machine. The prototype of the ARS will work with data from four sources:

- Vital Statistic’s Birth Master Record (VS-BM),
- Vital Statistic’s Death Record, also called EDEN,
- Utah Statewide Immunization Information System (USIIS), and
• Child-Health Advanced Record Management System (CHARM).

Figure 2.1 shows the phMPI system that consists of several loosely coupled layers or subsystems. Each subsystem provides a cohesive set of well-defined services to the other subsystems to external information systems (e.g., the participating data sources), or to end-users. The system relies on a common data model and messaging between the layers to provide interoperability between the layers.

The phMPI receives external service requests from data sources through the Communication Services, which simply place the requests in a request queue. The
Workflow Manager retrieves a request to process from the request queue and executes a well-defined set of services based on the type of request. The Workflow Manager and its interaction with other services are discussed in next section.

In the phMPI system, all the information for a person is stored in loosely coupled, cohesive units, called data chunks. Subsection 2.2.2 describes the structure of the data chunks in a phMPI system and Subsection 2.2.3 introduces the concept of GOLDEN data chunks.

2.2.1 Workflow Manager

The Workflow Manager coordinates the execution of data standardization, translation, matching, linking, resolution, publication, and consent management services [4]. When Workflow Manager gets a request from the request queue, it first chooses an appropriate workflow to guide the processing of that request based on the request type (e.g., insert person), the structure of the request’s payload (e.g. XML), and the data source (e.g. USIIS). In general, a workflow is a set of actions that instruct the Workflow Manager what services need to be called, the order in which they are called, and how to handle various conditions that could arise during the processing.

A workflow that executes the services that add, modify, or delete a person’s data in the phMPI will also include a call to the Automated Resolution Service (ARS), which re-computes the best version of truth for the affected person.

2.2.2 Data chunk

In general, a data chunk is a cohesive set of information about a person. For example, an Address data chunk could contain a person’s street address, city, county, state, zip, and country. Table 2.1 describes the 13 different types of data chunk in the
The phMPI system in terms of the data fields that they can contain. The phMPI can store multiple data chunks of the same type for a single a person.

Table 2.1. Data Chunks and their Attributes.

<table>
<thead>
<tr>
<th>Data-chunk Type</th>
<th>Possible Data-fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Address type, street line 1, street line 2, city, state or province, country, postal code, start date from when the address was known to be valid, end date, is the address can be released? (Y or N), is the address is CASS certified? (Y or N), longitude, and latitude</td>
</tr>
<tr>
<td>Adoption Event</td>
<td>Adoption Type, Adoption Date, relinquished by mother</td>
</tr>
<tr>
<td>Birth Event</td>
<td>Birth facility info, Date of birth (year, month and day), birth time, city of birth, county, state or province, country, single or multiple birth, birth order(if multiple birth), birth weight in ounces, birth weight in grams</td>
</tr>
<tr>
<td>Death Event</td>
<td>Death facility info, date of death(year, month, and day), county, state or province, country</td>
</tr>
<tr>
<td>Email</td>
<td>Email type, email id, start date from when the email id was known to be valid, end date after which the email id is not valid</td>
</tr>
<tr>
<td>Gender</td>
<td>Gender of the person</td>
</tr>
<tr>
<td>Marital Event</td>
<td>Marital status of the person, start date from when the current status was known to be valid, end date after which the current status was changed</td>
</tr>
<tr>
<td>Notes</td>
<td>Note text, date of creation, author of creation</td>
</tr>
<tr>
<td>Person Identifier</td>
<td>Type of person identifier, identifier value</td>
</tr>
<tr>
<td>Person Name</td>
<td>Type of name(Legal, general, alias), first name, middle name, last name, salutation, suffix, maiden name</td>
</tr>
<tr>
<td>Person Race</td>
<td>Person’s race</td>
</tr>
<tr>
<td>Person Relation</td>
<td>related person information, relation type, from when the relation was known to be valid, end date after which the relation became invalid</td>
</tr>
<tr>
<td>Phone</td>
<td>Type of phone number, phone number, from when the number was known to be valid, end date after which the number became invalid.</td>
</tr>
</tbody>
</table>

The phMPI stores each type of data chunk in a separate table with an id that uniquely identifies each data chunks, a person id that links a data chunk to the person that it is for, and a link to metadata about the data chunk. Table 2.2 lists the data elements that
can be in the metadata for a data chunk. The metadata is of particular importance to data resolution because it provides the basis for making decisions about which data is the most current and accurate.

### Table 2.2. Data-chunk Metadata.

<table>
<thead>
<tr>
<th>Data Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>original data source, or “GOLDEN”, or “PREVIOUSGOLDEN”</td>
</tr>
<tr>
<td>sourceTimestamp</td>
<td>Timestamp stored in original data source</td>
</tr>
<tr>
<td>createdOnDate</td>
<td>Date when data was entered into phMPI</td>
</tr>
<tr>
<td>lastUpdateTimestamp</td>
<td>Date when data was last updated by phMPI</td>
</tr>
<tr>
<td>accuracyRating</td>
<td>A rating providing by the data source that estimates the initial accuracy of data on a scale from 0 to 1, with 1 meaning full accuracy.</td>
</tr>
<tr>
<td>dataAgingType</td>
<td>Aging function for data that can be used to compute the estimated accuracy of the data over time</td>
</tr>
<tr>
<td>dataExpirationDate</td>
<td>A date that represent when the accuracy of the data is no longer trusted. It can be used by the aging function to compute a pro-rated accuracy based on the current date.</td>
</tr>
<tr>
<td>expirationRate</td>
<td>A factor that can be used by the aging function, in conjunction with dataExpirationDate</td>
</tr>
<tr>
<td>verifiedOnDate</td>
<td>Data Verification Date</td>
</tr>
<tr>
<td>verifiedBy</td>
<td>The end-user who verified the information in the data chunk</td>
</tr>
<tr>
<td>verificationLevel</td>
<td>A measure that indicates the level or type of verification that took place. The higher the number, the stronger the verification. The highest level is represents a in-person verification where legal identification and official documents reviewed.</td>
</tr>
<tr>
<td>expirationRate</td>
<td>The rate at which relevancy of the data decays</td>
</tr>
<tr>
<td>linkToPersonOn</td>
<td>Date when the data chunk was linked to person</td>
</tr>
</tbody>
</table>

#### 2.2.3 GOLDEN data chunk

A GOLDEN data chunk is a unique data-chunk record of a specific type, like Person Name, that represents the best version of truth for a person and that data-chunk type. For example, consider the person-name data chunks shown in Table 2.3. There is one name for person P1 from VS-BM and one name for the same person from CHARM.
The two names differ the first name, middle name, and salutation. If the data chunk from VS-BM is believed to be more accuracy, but incomplete with respect to the salutation, then a resolution algorithm could compute a GOLDEN data chunk by taking the first, middle, and last name from the first data chunk and the salutation from the second.

Table 2.3. An Example of the GOLDEN Data Chunk.

<table>
<thead>
<tr>
<th>Person</th>
<th>Source</th>
<th>First Name</th>
<th>Middle Name</th>
<th>Last Name</th>
<th>Salutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>VS-BM</td>
<td>Joseph</td>
<td>M</td>
<td>Jones</td>
<td>-</td>
</tr>
<tr>
<td>P1</td>
<td>CHARM</td>
<td>Joe</td>
<td></td>
<td>Jones</td>
<td>Dr.</td>
</tr>
<tr>
<td>P1</td>
<td>GOLDEN</td>
<td>Joseph</td>
<td>M</td>
<td>Jones</td>
<td>Dr.</td>
</tr>
</tbody>
</table>

In phMPI, original data chunks from the data sources are not updated or deleted, so the matching and resolution functions can have access to the full historical record of a person’s information. However, the phMPI keeps only one GOLDEN data chunk of each type at any time. When it computes a new GOLDEN data chunk of a certain type, it relabels the previous GOLDEN data chunk of that type as PREVIOUSGOLDEN. Existing GOLDEN and PREVIOUSGOLDEN data chunks are not taken into consideration during resolution.

2.3 Data Quality

The quality of integrated data helps an integrated information system to fulfill its goal of providing meaningful information. The phrase “data quality” does not have a well-defined meaning by itself; rather it depends on several issues. So, the issues and
their impact in the quality of data should to be well defined [3]. Following are some of the important issues that can be considered to determining quality of data.

**Table 2.4. Issues that can impact Data Quality (Adapted from [5])**

<table>
<thead>
<tr>
<th>Issues</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessibility</td>
<td>The extent to which data is available, or easily and quickly retrievable.</td>
</tr>
<tr>
<td>Accuracy</td>
<td>The extent to which data is correct, reliable, and free of error.</td>
</tr>
<tr>
<td>Believability</td>
<td>The extent to which data is trustable, i.e., it can be regarded as true and credible.</td>
</tr>
<tr>
<td>Completeness</td>
<td>The extent to which data is not missing and is of sufficient breadth and depth for the task at hand.</td>
</tr>
<tr>
<td>Interpretability</td>
<td>The extent to which data is in appropriate languages, symbols, and units, and the definitions are clear.</td>
</tr>
<tr>
<td>Non-overlapping</td>
<td>The extent to which same data is not repeated.</td>
</tr>
<tr>
<td>Relevancy</td>
<td>The extent to which data is applicable and helpful for the task at hand.</td>
</tr>
<tr>
<td>Timeliness</td>
<td>The extent to which data is sufficiently up-to-date for the task in hand.</td>
</tr>
<tr>
<td>Understandability</td>
<td>The extent to which data is easily comprehended.</td>
</tr>
<tr>
<td>Validation</td>
<td>The extent to which data is verified</td>
</tr>
</tbody>
</table>

Out of the ten issues mentioned in Table 2.4, five are particularly relevant to resolution problem. These issues are: accuracy, believability, completeness, non-redundancy, timeliness, and validation. Accuracy is the extent to which a data is correct, reliable, and free of error. Believability is the extent to which data is regarded as true and credible. Among other factors, it may reflect an individual’s assessment of the credibility of the data source, comparison to a commonly accepted standard, and previous
experience. Timeliness shows how up-to-date the data is with respect to the task it’s used for. It can be measured as maximum of 0 and one minus the ration of currency to volatility. Here, currency is defined as the age plus the delivery time minus the input time. Volatility refers to the length of time data remains valid; delivery time refers to when data is delivered to the user; input time refers to when data is received by the system; and age refers to the age of the data when first received by the system.

Completeness can be measured at different levels: schema completeness, data completeness, and population completeness [17]. At the data level, completeness can be defined as a function of the missing values in a column of a table. It can be measured by taking the ratio of complete items to the total number of items. Validation means the extent to which the data is verified. The verification can be done by a human user by cross-checking the values with pre-verified data. Non-overlapping measures the extent to which the data is not repeated in two different entities.

2.4 Issues associated with the problem of resolution

2.4.1 Poor Data Quality Issues

The quality of integrated information is considered very important for an MPI to provide better services. If quality of data is not achieved, information is not useful [3]. However, inconsistent, duplicate, and incomplete data can impact the quality of data by creating redundancy and confusion.

Inconsistent data refers to the data that conflicting field values, often caused by semantic heterogeneity in data sources, impossible or meaningless values, or obsolete data. An example, consider a newborn with data from two sources. One source says that a baby’s birth weight was 2140 grams and the other says it was 75 ounces. However, 75
ounces is only 2126 grams. The inconsistency could have come from the fact that the second system didn’t allow the user to enter fraction of ounces and that the baby’s actually birth weight in ounces was 75.5 grams.

A record or set of records that provides less than what would typically be expected is considered *incomplete data*. Incomplete data can be caused by poor validation during data entry, the lack of information at data-entry time, or As an example of the later, consider a user interface that does not facilitate to record birth time for a newborn’s birth record. In such a case, even if the birth time is available at the time of data entry, it cannot be recorded in the system and will eventually create an incomplete birth record.

The social and economic impact of poor data quality costs billions of dollars [20]. As reported by Wang and Strong, a big New York bank found that the data in its credit risk management database were only 60% complete. In a survey of 500 medium-size corporations (annual sales more than $20 million), it was found that more than 60% of them have problems with data quality [28]. In healthcare, data quality can be a matter of life and death. Inaccurate patient demographic information can lead to medical errors, fraud, and medical identify thefts. In 1999, the Institute of Medicine shocked the public with a report that 98,000 people die every year from medical errors, and some of the errors are a result of bad or missing information about drugs, orders, and treatments [6].

### 2.4.2 Duplication Issues

In general, all MPIs deal with many multiple records that represent the same individual. These records may come from same or multiple sources. This creates the
problem of duplication. Duplication refers to having two or more records actually representing a single person in an integrated information system. For example, in Table 2.5, a person P1 has two different name records.

Table 2.5. A Table showing Duplicate Records.

<table>
<thead>
<tr>
<th>Person</th>
<th>First Name</th>
<th>Middle Name</th>
<th>Last Name</th>
<th>Salutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>J</td>
<td>-</td>
<td>Jones</td>
<td>Dr.</td>
</tr>
<tr>
<td>P1</td>
<td>Joe</td>
<td>J</td>
<td>Jones</td>
<td>Dr.</td>
</tr>
</tbody>
</table>

Duplication can be further classified as same source duplicates, multiple source duplicates, overlaying duplicates, and overlapping duplicates.

- **Same-source duplicates:** Same-source duplication refers to when two or more records for a person come from the same source. For example, in Table 2.6, source S1 contains duplicate record for person P1.

Table 2.6. Same-source duplicate records

<table>
<thead>
<tr>
<th>Person</th>
<th>Source</th>
<th>First Name</th>
<th>Middle Name</th>
<th>Last Name</th>
<th>Salutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>S1</td>
<td>Daniel</td>
<td>J</td>
<td>Jones</td>
<td>Dr.</td>
</tr>
<tr>
<td>P1</td>
<td>S1</td>
<td>Daniel</td>
<td>J</td>
<td>Jones</td>
<td>Dr.</td>
</tr>
</tbody>
</table>

- **Multi-source duplicates:** Multi-source duplication refers to the case when two or more records for a person contain similar data, but they come from different sources. For example, in
• Table 2.7, two different sources S1 and S2 contain duplicate records for person P1.

<table>
<thead>
<tr>
<th>Person</th>
<th>Source</th>
<th>First Name</th>
<th>Middle Name</th>
<th>Last Name</th>
<th>Salutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>S1</td>
<td>Daniel</td>
<td>J</td>
<td>Jones</td>
<td>Dr.</td>
</tr>
<tr>
<td>P1</td>
<td>S2</td>
<td>Daniel</td>
<td>J</td>
<td>Jones</td>
<td>Dr.</td>
</tr>
</tbody>
</table>

Table 2.7. Multi-source duplicate records

• **Overlapping Records**: Overlapping Records refers to when two or more records with different data come from different sources for the same person. For example, in Table 2.8, two different sources S1 and S2 contain overlapping duplicate records for person P1.

<table>
<thead>
<tr>
<th>Person</th>
<th>Source</th>
<th>First Name</th>
<th>Middle Name</th>
<th>Last Name</th>
<th>Salutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>S1</td>
<td>Dan</td>
<td>-</td>
<td>Jones</td>
<td>Dr.</td>
</tr>
<tr>
<td>P1</td>
<td>S2</td>
<td>Don</td>
<td>J</td>
<td>Jones</td>
<td>Dr.</td>
</tr>
</tbody>
</table>

Table 2.8. Overlapping records

• **Overlaying Records**: Overlaying records refers to when two or more records appear to contain information about the same person, but are actually for different individuals. For example, in Table 2.8, if salutation is not considered, the two records look like duplicates, but they actually represent two different people P1 and P2.

<table>
<thead>
<tr>
<th>Person</th>
<th>Source</th>
<th>First Name</th>
<th>Middle Name</th>
<th>Last Name</th>
<th>Salutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>S1</td>
<td>Dan</td>
<td>-</td>
<td>Jones</td>
<td>Dr.</td>
</tr>
<tr>
<td>P1</td>
<td>S2</td>
<td>Don</td>
<td>J</td>
<td>Jones</td>
<td>Dr.</td>
</tr>
</tbody>
</table>

Table 2.9. Overlaying records
It is also important to understand that duplication and poor data quality can coexist at the same time and can go hand in hand to worsen the situation. Duplicate data can reduce the quality and integrity of data by creating conflicts among different version of data. On the other hand, inconsistent or incomplete data can impact one’s ability to detect and resolve duplication. These problems are sometimes solved by merging the records. The merging process is explained in next section.

2.5 Merging (Adapted from [1])

The process of merging, or data resolution [4], consolidates two or more records into a single view to expose the most accurate and reliable data. To achieve this, single-source or multi-source duplicates have to be removed. However, single source and multi-source duplicate removal approaches differ from each other. These approaches are briefly discussed in the following paragraphs.

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.

2.5.1 Merging Techniques (Adapted from [1])

Merging techniques can be used by a merging process to meet the challenges discussed in Section 2.5.3. Some of the merging techniques are as follows:

- Source Tagging: A merger can tag the original source of value with the merged record. This tagging can enhance the reliability and accuracy of matching, merging and unmerging operations.

- Stacking: A merger can keep the stack of all the values rather than combining them into a specific field. The stacking can improve the matching operations significantly.

- Merge Rules: Some mergers combine data from multiple records by using a set of customizable rules. Basic merge rules include:
- Precedence Rules: Precedence orders are assigned to multiple data sources for a specific piece of data.
- Data-combination and conversion rules: These rules specify how the merge should combine or convert the data.
- Redirection rules: These rules specify how and when data might get redirected to alternate fields.
- Information-stacking rules: These rules specify when and how to do value stacking.

2.6 Inter-field Dependency (Adopted from [2])

The process of merging looks simple, but it is fairly complicated. A wrong resolve can decrease the data quality and data integrity of the record. For example, in Table 2.10. A table explaining wrongly merged record, the record R1+R2 is a result of merging records R1 and R2, but since record R1 is missing State information, we cannot assume that the address in record R1 is in Iowa State. It might be the case that record R1 contains address information about the city of Logan in Utah.

Table 2.10. A table explaining wrongly merged record

<table>
<thead>
<tr>
<th>Record</th>
<th>Street1</th>
<th>Street 2</th>
<th>City</th>
<th>State</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>600 N 600 E</td>
<td>Apt. #5</td>
<td>Logan</td>
<td>-</td>
<td>USA</td>
</tr>
<tr>
<td>R2</td>
<td>600 N 600 E</td>
<td>-</td>
<td>Logan</td>
<td>IA</td>
<td>USA</td>
</tr>
<tr>
<td>R1+R2</td>
<td>600 N 600 E</td>
<td>Apt. #5</td>
<td>Logan</td>
<td>IA</td>
<td>USA</td>
</tr>
</tbody>
</table>

In the ARS, the above mentioned problem was be solved by defining a constraint called inter-field dependency [2]. Inter-field dependency can be classified as:
• **One-way data dependency:** In table T, column A determines the value for another column B or A#B. For example, consider an address table with complete address information. In this table, zip code can determine the city, but a city can have many zip codes.

• **Bi-directional data dependency:** For table T, columns A and B are dependent on each other, or A#B and B#A. For example, consider a state mapping table that maps state codes to state names. In this table, both columns determine values for each other. “UT” determines “UTAH” and vice versa.

• **Aggregate data dependency:** A dependency exists among a group of columns say C1, C2, C3 ….. Cn. for Table T such that individually they have bits of incomplete information, but as a unit they provide a complete description. For example, address table can be composed from five columns that have aggregate data dependency among them, i.e., \{street_address1, street_address2, state1, zip1, city1\}. Individual fields contain pieces of address information about a person and are partially dependent on one another.
CHAPTER 3
SYSTEM ANALYSIS

In software engineering, system analysis is the study of a system under consideration. The purpose of system analysis is to understand and document the essential characteristics of the system being studied [12]. The Unified Modeling Language (UML) is a general-purpose object-oriented modeling language and is considered an industry standard for capturing the result of a system analysis. It includes sub-languages for modeling groups of users and their goals, object structures, interactions, and behaviors [1]. This makes it a good choice for the analysis of the ARS.

The use-case diagrams in Section 3.1 provide a high-level overview of users and their goals. The functional requirements listed in Section 3.2 expand on these goals with a more detailed description, including constraints of the system's features and behavior. Non-functional requirements listed in Section 3.3 specify the criteria that can be used to judge the operation of the system [21]. Finally, the class diagrams in Section 3.4 describe the key objects in the system and their relationships to each other from an analysis perspective.

3.1 User Goals

The user goals are captured by use-case diagrams. A use-case diagram defines the associations between actors and the use cases of a system under consideration to accomplish a goal. An actor specifies a role played by a person or system while interacting with the system [14]. The Workflow Manager is responsible for the execution of the ARS, as discussed in Section 2.2.1, so the primary actor for the ARS is the
Workflow Manager. The next subsection captures the user goals of the Workflow Manager.

### 3.1.1 User Goals for the Workflow Manager

Figure 3.1 describes the primary goals of the Workflow Manager in context of the ARS. The primary goal of the Workflow Manager is to execute the ARS. This goal includes two different sub goals. The ARS should be able to create the *GOLDEN* record. To create this record, the ARS should be able to calculate the overall data quality factor, sort list of data chunks on the basis of that factor, merge the best data chunk with the second best record, and create the Metadata information for the *GOLDEN* record.

Further, if there is already a *GOLDEN* data chunk, the ARS should be able to change the current *GOLDEN* record to a *PREVIOUSGOLDEN* record. This sub goal is particularly important to make sure that there is one and only one *GOLDEN* data chunk at any time.

**Figure 3.1 Use case diagram showing goals of Workflow Manager**

Further, if there is already a *GOLDEN* data chunk, the ARS should be able to change the current *GOLDEN* record to a *PREVIOUSGOLDEN* record. This sub goal is particularly important to make sure that there is one and only one *GOLDEN* data chunk at any time.
3.2 Functional Requirements

Functional requirements describe the services provided by a system. Functional requirements can be high level or detailed, expressing inputs, outputs, exceptions, and so on [15]. The functional requirements for the ARS are as follows:

3.2.1 The ARS subsystem should compute a GOLDEN master record (composite record) from all the linked records for a person in the MPI.

- The ARS should compute the GOLDEN record for all the data chunks in the phMPI system.

- If a particular data chunk has subtypes, the ARS should create a GOLDEN record for each subtype of that data chunk. For example, a person can have 2 GOLDEN addresses (1 GOLDEN address of type Home and 1 GOLDEN address of type Office).

3.2.2 The GOLDEN record should be a unique record.

- At any time, there should only be a single GOLDEN record for a data chunk.

- The GOLDEN record may contain information from different data chunks of same type. For example A GOLDEN birth-event record may contain a birth year from one source and birth facility from another source.

- After the computation of the GOLDEN record, it should be stored in the MPI repository. The Metadata record associated with it should also be stored with the data source tagged as “GOLDEN.”
3.2.3 The ARS should store a series of rules for creating the GOLDEN data chunks.

- The rules should take into account the reliability of the data from the data source, e.g. whether a death date came from a death certificate and, thus, take precedence over a death date listed on the hospital record.
- The rules should take into account the accuracy of the data.
- The rules should take into account the timeliness of the data. For example, the data that is more recent should be given more precedence than the old data.

3.2.4 The ARS should be called as a part of an MPI request.

3.2.5 The ARS shall merge the records or choose the best available record depending on the inter-field dependencies.

- The ARS should identify the inter-field dependencies for a data chunk before merging two records. For example, in address data chunk, addressLine1 can be dependent on addressLine2 and vice-versa.
- The ARS should contain a set of predefined rules for inter-field dependency. For example, the rules define which fields in a particular data chunk are interdependent.
- The ARS should choose the best available record if there is only one record.
- The ARS should choose the best available record, if all the data fields are either non-empty or independent.
- The ARS should choose the best available record, if for all empty fields in a data chunk, one or more dependent fields of an empty field is non-empty.
The ARS should merge the best and the second best record if all the empty data fields are either independent or their dependent fields are also empty.

3.3 Non-functional Requirements

The ARS is one of many existing components of the phMPI. Consequently, it needs to coordinate its services with the other components. For this reason, the ARS must comply with the development process and standards that are used in the phMPI.

Following are the non-functional requirements for the ARS:

3.3.1 Operating System

The phMPI system is compatible with the Microsoft Windows operating systems. So, the ARS should be compatible with the Microsoft Windows operating systems.

3.3.2 Languages and Platform

- The programming language should be C# (C-Sharp).
- The system should use Vitruvian for distribution.
- The system should use MS SQL database.
- The system should use Vitruvian framework to support logging.

3.3.3 Quality Control

Thorough unit testing and integration testing are essential to ensure performance and quality of the ARS.

3.3.4 Documentation

Design documents and a report about the system should be created to assist users in understanding the design and functionalities supported by the ARS.
3.4 Object-oriented Structural Analysis

The requirements help in understanding the system and the key functionalities offered by the system. They also provide a sound basis for defining the structure of the key classes associated with the system. This section explores data structures involved in meeting those requirements. UML class diagrams describe the structure of a system by showing and describing classes of objects, their names, their attributes, operations (or methods), as well as the relationships among the classes [22].

Figure 3.1 illustrates the primary class diagram for the ARS. Additional information about each of the classes in this diagram can be found in the following subsections.

Figure 3.2. Class Structure of the ARS.

3.4.1 Resolver

The Resolver class is responsible for providing the best version of truth by creating the GOLDEN record. It is also responsible for communicating with the Workflow Manager. To create the GOLDEN record, Resolver gets a list of data chunks
(sorted in ascending order of overall data quality) from the DataChunkList class, and merges the best record with the second best record.

### 3.4.2 DataChunkList

As its name indicates, the DataChunkList class is a list of the DataChunk objects (explained in Section 3.4.3), for example AddressList class contains a list of Address objects. It is responsible for sorting the list of data chunks based on their *Overall Quality*. Figure 3.2 depicts the specialization of DataChunkList class. The data chunk list classes are auto-generated using Vitruvian DBObjects (Section 5.2). The DataChunkList class is specialized all the classes that represent list of data chunks. It establishes the foundational functionality and a common interface, which can be inherited or modified by the specialized classes.

![Diagram of DataChunkList class specialization](image)

**Figure 3.3. Specializations of the DataChunkList class.**
3.4.3 DataChunk

The DataChunk class is specialized by all the data chunks mentioned in Table 2.1 of Chapter 2. The data chunk classes are auto-generated using Vitruvian DBObjects (Section 5.2). Each DataChunk object is associated with a Metadata object. The DataChunk class is also responsible for calculating the overall quality factor using of Metadata information and the data quality metrics. Similar to the DataChunkList class, the DataChunk class is specialized by all the data chunks.

![Figure 3.4. Specialization of DataChunk class.](image-url)
3.4.4 **Metadata**

The Metadata class is an auto-generated class using Vitruvian DBObjects that contains all the information mentioned in Table 2.2 in Section 2.2.2. The information in the Metadata class can be used to compute the data quality of a data chunk.

3.4.5 **DataField**

The DataField class contains information about a specific data field for an individual data chunk. Each DataField object can be associated with many Dependency objects (explained in section 3.4.6).

3.4.6 **Dependency**

This class contains information about the DataField objects that are dependent on a specific DataField object.
CHAPTER 4
ARCHITECTURAL DESIGN

4.1 Introduction

The software architecture of a program or computing system is the structure of the system that comprises of the software components, the externally visible properties of those components, and the relationships between them. Documenting software architecture facilitates communication between stakeholders, documents early decisions about high-level design, and allows for the reuse of design components and patterns between projects [16].

System design is the process of defining the system’s architecture with components, modules, interfaces, and data structures that satisfy the specified requirements [10]. In design, the result of the analysis is expanded into a technical solution. The ARS is designed using the Unified Modeling Language (UML) as it is convenient to expand the analysis into a design using UML.

The features and responsibilities of the ARS are explained in detail in Section 4.2. Section 4.3 illustrates the dependencies between different packages. Object interactions in different scenarios are explained in Section 4.4. The design decisions taken while creating the Overall Quality Factor are documented in Section 4.5. The design of the inter-field dependencies (Section 2.6) is described in Section 4.6.

4.2 Automated Resolution Service

The Automated Resolution Service (ARS) automates the computation of the GOLDEN record from all the linked records for a person in the phMPI system. This GOLDEN record is computed from data chunks of the same type.
The ARS deals with inconsistent and incomplete data by calculating the Overall Quality Factor (Section 4.4), and then selects individual or aggregated fields based on that factor. The Overall Quality Factor ensures that the ARS creates the most complete, accurate, reliable, latest, and verified records. The Overall Quality Factor is calculated using the data quality dimensions mentioned in Error! Reference source not found. Error! Reference source not found. of Chapter 2.

The ARS looks for missing data fields in the best record and merges the best record with other records to fill out the missing data fields and create a more complete and concrete record. While merging, the ARS maintains the data integrity of the GOLDEN data chunks by specifying data combination and conversion rules (mentioned in Section 2.5.1) The ARS creates constraints, called Inter-field Dependencies that prohibit the merging of inter-related data.

The ARS deals with duplicate records by making sure that at any time there is one and only one GOLDEN record and there are no duplicate GOLDEN records for a particular data chunk. If there is already a GOLDEN record, the ARS change its tag to PREVIOUSGOLDEN. So, there can be more than one PREVIOUSGOLDEN record for a particular data chunk.

The ARS doesn’t delete the original records. It uses stacking and source tagging techniques (Section 2.5.1) so the original records are easily retrievable. The ARS tags the source of the GOLDEN record as GOLDEN. The ARS also keeps the previous GOLDEN records in the phMPI repository.

The ARS handles the case where a person can have two or more record of data chunk with different sub-types. There can be some cases where a person can have more
than one phone number, address, email, or person identifiers of different types, for example, a person can have a home phone, a cellphone, and an office phone. In such a case the ARS maintains all subtypes of information by creating a *GOLDEN* record for each subtype of information.

4.3 Package-level Dependencies

Packages are software components that perform a particular thing. The software can be composed of several packages to improve maintainability [26]. The ARS is divided into different packages, which make the maintenance and testing easier. Having different packages in the ARS allowed to have low coupling among different classes and methods. The classes related to each other or used together are kept in the same package to achieve high cohesion among the classes.
Figure 4.1. Package level diagram for the ARS.

Package diagrams depict the dependencies between the packages of the ARS. In UML, package diagrams visualize packages and depict the dependencies between them [18]. The main packages for the ARS are illustrated in Figure 4.1. Figure 4.2 illustrates a detailed package diagram with all the attributes and operations of individual classes. In the phMPI system, all the subsystems act as a service by inheriting from an IService interface of Vitruvian.Services package. The Vitruvian.Services package is part of the Vitruvian service-oriented framework. For the ARS to act as a service, it is also required to inherit from the IService interface.
Figure 4.2. A detailed package level diagram for ARS

The Resolver class is responsible for providing the main functionalities of the ARS. The Resolver class in the CoreMPI.Core package (Application Layer) implements the Service interface in the Vitruvian.Services package.

The CoreMPI.DataLayer package (Persistent Layer) contains all the auto-generated classes from the Vitruvian DBObjects. This package contains all lists of data chunk classes mentioned in Section 3.4.2 of Chapter 3. These classes are generalized by the DataChunkList class. Similarly, all of data chunks are generalized by the DataChunk
class. Every DataChunk object is associated with a Metadata object. The Metadata class contains vital information required to calculate the *Overall Quality Factor*.

The DataField class contains all the information about a specific field in a DataChunk class. The Dependency class is used by the ARS to determine the dependencies of an object of the DataField class. These classes are mainly used for the purpose of merging records.

### 4.4 Object Interactions

Interaction is a kind of action that occurs as two or more objects have an effect upon one another. In UML, the interaction between different objects can be explained using the sequence diagram that shows how objects communicate with each other in different scenarios in terms of a sequence of messages. It also indicates the life spans of objects relative to those messages [17]. In the ARS, the interaction between different object is explained using the sequence diagrams.

As mentioned in Chapter 1, the ARS creates a *GOLDEN* data chunk by merging the two best records. This merging depends on the number of data chunks and their data quality. It also depends on the missing fields and their dependencies in a data chunk. Based on these factors, the interaction between different objects is illustrated by five key scenarios:

#### 4.4.1 The DataChunkList object does not contain any DataChunk objects

As illustrated in Figure 4.3, the Resolver object $r$ gets all the list of data chunks ($dckList$ object) from object $p$ of the Person class. For every object $dckList$ of the DataChunk class, the resolver invokes the SortListByQuality method of the $dckList$
object, which returns an empty list. Since the list is empty, there are no records to resolve, and so the object $r$ skips all the steps and moves to the next $dckList$ object.

Figure 4.3. Sequence diagram for empty DataChunkList

4.4.2 The DataChunkList contains only one DataChunk object

As illustrated in Figure 4.4, the Resolver object $r$ gets the complete list of data chunks ($dckList$ object) from object $p$ of the Person class. For every object $dckList$, object $r$ invokes the SortListByQuality method. If there is only one DataChunk object ($dck$), this method returns the $dckList$ object as it is. The Resolver object $r$ directly declares the $dck$ as the GOLDEN DataChunk object.
4.4.3 Calculation of Overall Quality Factor

In this scenario (Figure 4.5), first, the Resolver object \( r \) calls the SortListByQuality method of the DataChunkList object \( dckList \). The SortListByQuality method calls the getOverallQuality method for all the DataChunk objects \( (dck) \) in \( dckList \) object. Each \( dck \) object collects information from the Metadata object \( m \) to compute the Overall Quality Factor. The SortListByQuality method then calls the Compare method to compare the data chunks based on their Overall Quality Factor and sort the list in ascending order of Overall Quality Factor. Finally, the SortListByQuality method returns the sorted list to \( r \).
4.4.4 No merging when DataChunkList contains two or more DataChunk objects

This scenario is illustrated in Figure 4.6. This scenario can be possible in two cases:

- When the best DataChunk object do not have any empty fields.
- When the best DataChunk object has some empty fields missing but their dependent fields are available.

The initial steps of this scenario are same as in Section 4.4.3. Once, the Resolver object r gets a sorted list it looks for missing fields in the best data chunk (last element in sorted list). Since, in this scenario the best record contains all the fields, or if any of the Dependency object d of an empty DataField object emptyField is non-empty, then there is no merging between the records. The GOLDEN record will be a replica of the best record.
4.4.5 Merging of records when DataChunkList has two or more DataChunk objects

In this scenario (Figure 4.7), the initial steps are similar to Scenarios 4.4.3 and 4.4.4, but the best record contains some empty data fields. In that case, the Resolver object \( r \) first looks for the missing DataField objects \( emptyField \), and for each \( emptyField \), object \( r \) looks for its Dependency objects \( d \) for the best data chunk. If all the \( d \) objects for an \( emptyField \) object are also empty, then object \( r \) calls the getNextBestDataChunk method that returns the next best data chunk. Then, \( r \) looks for all the empty fields (of best data chunk) in the next best data chunk. If next best data chunk contains any extra information, then it merges the best and next best data chunk and creates the \textit{GOLDEN} DataChunk object.
4.5 Overall Quality Factor

To create the most accurate and reliable data, it is important to measure the quality of data. For this purpose, the ARS uses the data quality dimensions mentioned in Section 2.2.3 and devises a formula to calculate the overall quality of a data chunk. The formula is as follows:

\[
\text{Overall\_Quality} = \left(\frac{\text{Accuracy} + \text{Verification}}{2}\right) \times \left(\frac{\text{Believability\_Weight} \times \text{Believability} + \text{Total\_Completeness} + \text{Timeliness\_Weight} \times \text{Timeliness}}{3}\right)
\]

In this formula, Accuracy and Verification are considered more important and are kept separate from the other factors. The weights Believability\_Weight and
Timeliness Weight control the impact of the Believability and the Timeliness attributes, respectively, in the Overall Quality. These values can be set by the user and can vary for different data chunks. The data quality parameters in the formula are as follows.

4.5.1 Believability

In the phMPI system’s database, a precedence level is recorded for the entire data source with respect to a data chunk. For example, if a data source S keeps track of births, the birth event data from that source will be more believable than any other source. Using the precedence values, the believability factor is calculated within the range of 0 to 1. The precedence values range from 1 to n (n is the total number of data sources), where 1 is given to the data source with highest precedence and n to the data source with lowest precedence for a particular data chunk. It is calculated by the formula:

$$Believability = \frac{1}{Precedence}$$

4.5.2 Accuracy

The accuracy of a data chunk comes directly from the data sources. The Metadata table contains an accuracyRating column, which stores a rating as a decimal value between 0 and 1. A value of 1 means the data is believed to be completely accurate and a 0 means that the data source has no confidence in the accuracy of the value. Values in between 0 and 1 represent some intermediate level of confidence in the value’s accuracy.

4.5.3 Total Completeness

Total Completeness of a data chunk depends on the number of data fields available in it and the Metadata object associated with it. Total Completeness ranges
from 0 to 1, where 0 means all the fields are empty and 1 means all the fields are non-empty. It is calculated by a simple formula:

\[
\text{Completeness} = \frac{\text{Number of non-empty fields}}{\text{Total number of fields}}
\]

4.5.4 Verification

Verification is calculated using verified dates of data chunks. Verification ranges from 0 to 1. It is calculated using the following formula:

\[
\text{Verification} = \frac{(Sd - Od)}{(Rd - Od)}
\]

- **Sd:** Subject Date (verifiedOnDate of current data chunk)
- **Od:** Oldest known Date (oldest verifiedOnDate for the same person, same data chunk type)
- **Rd:** Most recent known Date – newest verifiedOnDate for the same person, same data chunk type

4.5.5 Timeliness

To calculate the timeliness of a data chunk, different dataAgingType functions are created. These functions are executed based on the dataAgingType value (available in Metadata object).

All Timeliness Functions take the following parameters:

- **Sd:** Subject Date (lastUpdateTimestamp of current data chunk)
- **Cd:** Current Date (System’s current date)
**Od:** Oldest known Date (oldest lastUpdateTime for the same person, same data chunk type)

**Rd:** Most recent known Date – newest lastUpdateTime for the same person, same data chunk type

**Ed:** Expiration Date - if available, a date on which the data expires

Table 4.1 illustrates the different formulas for calculating Timeliness. These formulas are used depending on the dataAgingType Values (available in the Metadata object).

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Value</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant function</td>
<td>0/null</td>
<td>( f(Sd, Cd, Od, Rd, Ed, Er) = 1 )</td>
</tr>
<tr>
<td>Linear-based on oldest and cents dates</td>
<td>1</td>
<td>( f(Sd, Cd, Od, Rd, Ed, Er) = (Sd - Od) / (Rd - Od) )</td>
</tr>
<tr>
<td>Linear-based on oldest and current date</td>
<td>2</td>
<td>( f(Sd, Cd, Od, Rd, Ed, Er) = (Sd - Od) / (Cd - Od) )</td>
</tr>
<tr>
<td>Step function based on expiration date</td>
<td>3</td>
<td>( f(Sd, Cd, Od, Rd, Ed, Er) = \begin{cases} 1 &amp; \text{if } Ed!=null &amp;&amp; Sd&lt;=Ed \text{ then } 1 \ 0 &amp; \text{else} \end{cases} )</td>
</tr>
<tr>
<td>Linear decay after expiration</td>
<td>4</td>
<td>( f(Sd, Cd, Od, Rd, Ed, Er) = \begin{cases} 1 &amp; \text{if } Ed!=null &amp;&amp; Sd&lt;=Ed \text{ then } 1 - \text{Min}(1, (Cd-Sd) * Er) \ 0 &amp; \text{else} \end{cases} )</td>
</tr>
<tr>
<td>Inverse square decay after expiration</td>
<td>5</td>
<td>( f(Sd, Cd, Od, Rd, Ed, Er) = \begin{cases} 1 &amp; \text{if } Ed!=null &amp;&amp; Sd&lt;=Ed \text{ then } 1 - \text{Min}(1, (Cd-Sd) * Er^1/2) \ 0 &amp; \text{else} \end{cases} )</td>
</tr>
<tr>
<td>Gradual decay around expiration date (Cubed function)</td>
<td>6</td>
<td>( f(Sd, Cd, Od, Rd, Ed, Er) = \begin{cases} 1 &amp; \text{if } Ed!=null &amp;&amp; Sd&lt;=Ed \text{ then } 1 - \text{Min}(1, (Cd-Sd) * Er^3) \ 0 &amp; \text{else} \end{cases} )</td>
</tr>
</tbody>
</table>

### 4.6 Inter-field Dependencies

As explained in Section 2.6, Inter-field Dependencies are similar to the functional dependencies of a relational database management system. They act as a constraint and define the way in which a data field functionally determines another data field [5]. In the ARS, aggregate data dependencies are defined as a group of data fields. For example,
birth hour and birth minute are related to each other and should be grouped together. So, if the best record has birth hour, but doesn't have birth minute, it shouldn't merge the birth minute field.

In all the tables, the id field is the primary key so it cannot be changed. Similarly, mpiid and metaid fields are foreign keys from Person table and Metadata table respectively, so they are also unchangeable fields. Different colors are used show unchangeable, independent and dependent fields. Figure 4.8 illustrates the color schemes for inter-field dependency. The inter-field dependencies for all the data chunks are explained below.

![Color schemes for inter-field dependency](image)

**Figure 4.8. Color schemes for inter-field dependency**

### 4.6.1 Inter-field dependency in the Address Class

The inter-field dependencies in the Address class are shown in Figure 4.9. The DataField object addressype defines the type of address (Home, Office, etc.). Since we are computing the *GOLDEN* data chunk for each type of address, this object is also irreplaceable. The DataField objects streetline1, streetline2, city, stateorprovince, country, postalcode, donotreleaseflag, casscertified, longitude, and latitude are dependent on each other. It is possible that the best available address is significantly different from other records. Merging the best record with other records will create false records with lower data integrity.
4.6.2 Inter-field dependency in the AdoptionEvent Class

Figure 4.10 shows the inter-field dependencies in the AdoptionEvent Class. In this class we have kept adoptiontype, adoptiondate, and motherrelinquished DataField objects together, which are dependent on each other.

4.6.3 Inter-field dependency in the BirthEvent Class

Figure 4.11 shows the inter-field dependencies in the BirthEvent Class. In this class, the DataField facilityid is a reference to the Facility class. Since it’s not related to any other field in the class, we have kept it independent. The DataField objects birthyear, birthmonth, and birthday fields are all kept together as they represent birthdate. Similarly,
birthhour and birthminute represent time of birth, so they are kept together.

<table>
<thead>
<tr>
<th>BirthEvent Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
</tr>
<tr>
<td>Mpiid</td>
</tr>
<tr>
<td>Metaid</td>
</tr>
<tr>
<td>Facilityid</td>
</tr>
<tr>
<td>birthyear, birthmonth, birthday</td>
</tr>
<tr>
<td>birthhour, birthminute</td>
</tr>
<tr>
<td>city, county, state, country</td>
</tr>
<tr>
<td>multiplebirth, birthorder</td>
</tr>
<tr>
<td>birthweightounces, birthweightgrams</td>
</tr>
</tbody>
</table>

**Figure 4.11. Inter-field dependencies in the BirthEvent class**

The DataField objects city, county, stateorprovince, and country represents birthplace, so they are kept together. The DataField objects multiple birth and birthorder are interrelated, since birthorder is important only if there is a multiple birth. The birthweightgrams and birthweightounces fields represent birthweight in different units, so they are dependent.

4.6.4 *Inter-field dependency in the DeathEvent Class*

Figure 4.12 shows the inter-field dependencies in the DeathEvent Class. The dependencies are pretty similar to BirthEvent class. The DataField object facilityid is a reference of Facility class and is independent. The DataField objects deathyear, deathmonth and deathday are all kept together. The DataField objects city, county, stateorprovince and, country all together provide information about the death place of that person, so they are kept together.
4.6.5 Inter-field dependency in the Email Class

In the Email class (Figure 4.13) the DataField object addresstype defines the type of address (Home, Office, etc.). Since we are computing the GOLDEN data chunk for each type of address, this field is also irreplaceable. The DataField object email has been kept completely independent, since there is no other DataField object that is functionally dependent on it. Start date and end date are dependent on each other, so we have created dependency for them.

4.6.6 Inter-field dependency in the Gender Class

In the Gender class, Figure 4.14, there are no dependencies. The DataField object gender is independent.
4.6.7  *Inter-field dependency in the MaritalEvent Class*

In the MaritalEvent class (Figure 4.15), the DataField object maritalstatus has been kept completely independent, since no other DataField object is functionally dependent on it. Start date and end date are dependent on each other, so we have created dependency for them.

```
MaritalEvent Class
| id     | mpiid | metaid | maritalstatus | startdate, enddate |
```

*Figure 4.15 Inter-field dependencies in the MaritalEvent class*

4.6.8  *Inter-field dependency in the Note Class*

In the Note class (Figure 4.16), the DataField objects note, createdon (date), and createdby are related to each other, so there is a dependency between them.

```
Note Class
| id     | mpiid | metaid | note, createdon, createdby |
```

*Figure 4.16. Inter-field dependencies in the Note class*
4.6.9  Inter-field dependency in the PersonIdentifier Class

In the PersonIdentifier class (Figure 4.17), the DataField object idtype is a subtype field. Since we are creating GOLDEN records for subtypes also, it cannot be changed. The DataField object idvalue has been kept completely independent, since there is no other DataField object that is functionally dependent on it.

<table>
<thead>
<tr>
<th>PersonIdentifier Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>mpiid</td>
</tr>
<tr>
<td>metaid</td>
</tr>
<tr>
<td>idtype</td>
</tr>
<tr>
<td>idvalue</td>
</tr>
</tbody>
</table>

Figure 4.17. Inter-field dependencies in the PersonIdentifier class

4.6.10  Inter-field dependency in the PersonName Class

The PersonName class (Figure 4.18), is most interesting class of all. In this class, the DataField object nametype is a subtype field (Legal, General, or Alias name type). GOLDEN records are created for subtypes also so this field cannot be changed. Since there is a very little chance that a person has two completely distinct names of the same subtype, an interesting decision to keep rest of the DataField objects independent. This makes the merging of PersonName object quite flexible.

<table>
<thead>
<tr>
<th>PersonName class</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>mpiid</td>
</tr>
<tr>
<td>metaid</td>
</tr>
<tr>
<td>nametype</td>
</tr>
<tr>
<td>firstname</td>
</tr>
<tr>
<td>middlename</td>
</tr>
<tr>
<td>lastname</td>
</tr>
<tr>
<td>salutation</td>
</tr>
<tr>
<td>suffix</td>
</tr>
<tr>
<td>maidenname</td>
</tr>
</tbody>
</table>

Figure 4.18. Inter-field dependencies in the PersonName class
4.6.11 Inter-field dependency in the PersonRace Class

In the PersonRace class, Figure 4.19, there are no dependencies. The DataField object racecode is independent.

<table>
<thead>
<tr>
<th>PersonRace class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
</tr>
<tr>
<td>Mpiid</td>
</tr>
<tr>
<td>Metaid</td>
</tr>
<tr>
<td>racecode</td>
</tr>
</tbody>
</table>

Figure 4.19. Inter-field dependencies in the PersonRace class

4.6.12 Inter-field dependency in the PersonRelation Class

In the PersonRelation class (Figure 4.20) the DataField objects relatedmpiid and rolecode are related to each other, so they are kept together. Start date and end date are depends on each other, so we have created dependency for them.

<table>
<thead>
<tr>
<th>PersonRelation Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
</tr>
<tr>
<td>Mpiid</td>
</tr>
<tr>
<td>Metaid</td>
</tr>
<tr>
<td>relatedmpiid, rolecode</td>
</tr>
<tr>
<td>startdate, enddate</td>
</tr>
</tbody>
</table>

Figure 4.20. Inter-field dependencies in the PersonRelation class

4.6.13 Inter-field dependency in the Phone Class

In the Phone class (Figure 4.21) the DataField object phonetype is a subtype field. Since, we are creating GOLDEN records for subtypes also, this field cannot be changed.

The DataField object phone has been kept completely independent, since there is no other
DataField object that is functionally dependent on it. Start date and end date are depends on each other, so we have created dependency for them.

<table>
<thead>
<tr>
<th>Phone class</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>mpiid</td>
</tr>
<tr>
<td>metuid</td>
</tr>
<tr>
<td>phonetype</td>
</tr>
<tr>
<td>phone</td>
</tr>
<tr>
<td>startdate, enndate</td>
</tr>
</tbody>
</table>

**Figure 4.21. Inter-field dependencies in the Phone class**

The design decisions taken in this chapter were implemented in by writing the programming code in C# language. The next chapter documents the implementation of the ARS.
CHAPTER 5
IMPLEMENTATION DETAILS

The ARS System is implemented in C# language on .NET Framework 4.0, and the database is managed by Microsoft SQL Server 2008. Vitruvian DBObjects is used as an Object-relational mapping (ORM) tool. Other implementation details are explained in Section 5.2. Important methods are described in Section 5.3.

5.1 Introduction to Vitruvian DBObjects

One of the problems encountered when mapping an object-oriented language, such as Java or C++, to a declarative language, like SQL, is impedance mismatch. Impedance mismatch is caused by the fact that one object in the application can contain data from multiple tables and multiple rows within a table [25].

There are several techniques for overcoming impedance mismatch. Typically, the developer writes classes for each of the tables or for each of the required objects. Doing so involves writing hundreds, possibly thousands of lines of code. This process is error-prone and, therefore, it requires writing a lot of test cases, and then there is an added problem of maintaining the classes and their test cases.

Using object-relational mapping (ORM) is a more streamlined approach to overcoming impedance mismatch. ORM is a programming technique for converting 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 from within the programming language [12].

We used Vitruvian DBObjects for an ORM. The use of DBObjects minimizes, and in some cases eliminates, the need to access the database directly. The database is
represented and maintained by DBOjects. Data transfers to and from the user interface are handled by DBOjects. The DBOjects also take care of reading and updating the database. Important features of DBOjects include the following:

1. Automatically generate classes for tables and views in the database.
2. Avoid or minimize writing SQL for create, read, update, and delete (CRUD) operations.
3. Navigate between related objects.
4. Lazy loading of objects.
5. Specify filters and sort order for loading DBOject lists from the database.

Vitruvian provides a wizard for generating DBOject classes and the DBList classes for tables and/or views. Properties are generated in the classes for each of the columns in the corresponding tables/views. The relationships between the tables are captured as properties in either or both the related classes. One-to-one relationships are represented as DBOjects while the one-to-many relationships are represented as DBLists. The wizard allows us to choose the relationships to be represented in the generated classes. The user can customize the names of classes, their properties, and relationships.

Vitruvian provides the following methods for using the DBOjects:

1. Load(): Load the data into the DBOject or DBList. Data can be filtered before loading.
2. Reload(): Load the new set of data from database.
3. Save(): Save the DBOject to the database.
4. Delete(): Delete the DBObject from the database.
5. ResetValues(): Reset the values (i.e., all properties) of a DBObject.
6. RelationalSave(): Save the DBObject and the children tables of the current DBObjects.
7. RelationalDelete(): Delete the DBObject and the children tables of the current DBObjects.

5.2 Implementation Details and Challenges

In ARS, the Resolver class is the entry point and is responsible for communicating with all the objects in the ARS. All the methods in the ARS are generic methods. By using the generic methods we ensure that even if the data chunks are added or removed in future, the ARS doesn’t require major changes in the code.

For the purpose of merging the data chunks, it is not possible to directly copy a certain data field from one data chunk to another, since we wrote generic methods. So reflection is used to solve this problem. Reflection is the ability of a computer program to examine (see type introspection) and modify the structure and behavior (specifically the values, meta-data, properties and functions) of an object at runtime [22]. Using reflection it is easier to find a specific field with its name and change its value for a specific object.

The template-method pattern was followed for implementation of all the data chunk and data chunk list classes. A template method defines the program skeleton of an algorithm. One or more of the algorithm steps can be overridden by subclasses to allow differing behaviors while ensuring that the overarching algorithm is still followed [23].
Another issue was to define the settings information including the weights for the Overall Quality Factor and the inter-field dependencies. Hard coding the values was an option but it is not considered a good software engineering practice, so a separate XML configuration file was created to save the settings since XML files are fast and efficient to read and easy to edit. A sample example of the XML file is as follows:

```xml
<object type="DataLayer.Data-chunk, DataLayer, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
  <property name="ReliabilityWeight" value="0.7" />
  <property name="CompletenessWeight" value="0.8" />
  <property name="TimelinessWeight" value="0.9" />
  <property name="MetaCompletenessWeight" value="0.5" />
  <field name="Dependencies" type="System.Collections.Generic.Dictionary`2[[System.String, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089], [System.Collections.Generic.Dictionary`2[[System.String, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089], <entry>
    <entry-key encoding="W3C" value="DataLayer%2EAddress" />
    <entry-value type="System.Collections.Generic.Dictionary`2[[System.String, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089], [System.Collections.Generic.List`1[[DataLayer.DataField+Dependency, DataLayer, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null]], mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]], mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]">
      <item>
        <property name="FieldName" encoding="W3C" value="streetline1" />
      </item>
      <item>
        <property name="FieldName" encoding="W3C" value="streetline2" />
      </item>
      <item>
        <property name="FieldName" encoding="W3C" value="city" />
      </item>
      <item>
        <property name="FieldName" encoding="W3C" value="stateorprovince" />
      </item>
      <item>
        <property name="FieldName" encoding="W3C" value="country" />
      </item>
      <item>
        <property name="FieldName" encoding="W3C" value="postalcode" />
      </item>
      <item>
        <property name="FieldName" encoding="W3C" value="donotreleaseflag" />
      </item>
    </entry>
  </field>
</object>
```
CHAPTER 6
SOFTWARE TESTING

6.1 Introduction

Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation. Software testing also provides an objective, independent view of the software to allow the system users to appreciate and understand the risks of software implementation [17]. Software testing is also stated as the process of validating and verifying that a software program/application/product:

- Meets the business and technical requirements that guides its design and development.
- Works as expected.

Software testing is involved in every stage of software development life cycle. Unit testing of all the modules is done at the lowest level, followed by integrating the modules and performing Integration testing. Code reviews can be done at all levels to keep the code consistent at those levels. Finally, System testing is performed to affirm the end-to-end quality of the entire system.

The ARS is tested by following the bottom-up approach. Unit testing (Section 6.2) was performed over all the modules of the code. Section 6.3 explains the importance of code reviews in testing and maintaining the code. Finally, Section 6.4 explains the successful integration of the ARS into phMPI system and the testing techniques used to perform integration testing on the ARS.
6.2 Unit Testing

Unit testing focuses a verification effort on the smallest unit of software the software component or module [7]. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of every module. Different test cases were designed for different data types. These test cases usually involves testing of implemented methods with various types of inputs that user can provide. Most common input values are:

- For integers and floats: Positive, Negative, Zero.
- For Objects: Null and Not Null.
- For string: Null, empty, incorrect and correct.
- For List: Null, Not Null, empty, not empty.

6.3 Code Reviews

Code reviews help a developer to communicate his or her ideas and provides a feedback for coding style to the developer. Since the ARS is a part of the phMPI project, the coding style should be consistent to be integrated in to the phMPI system. To ensure this, the developer and manager reviewed code iteratively, and required changes were completed by the developer. These changes improved the overall quality of code and the efficiency of the system.

6.4 System Integration Testing

Integration testing verifies the system when it is integrated with other system. Different modules in the ARS were integrated and tested to work as an independent service. To achieve this, an automated testing environment was created using Vitruvian
Testing. This program reloads the test data and calls the ARS. Then the results are tested against known results to verify them.

6.5 System Testing

After integration testing, the ARS was integrated into the *phMPI* system and the whole system was tested thoroughly. During the system testing of the *phMPI* system, there were some conflicts between the ARS and other services in the *phMPI* system. Those conflicts were successfully removed and the whole system was again tested. The results of the test were satisfactory and the system testing was a success.
CHAPTER 7

SUMMARY

The ARS is a prototype general-purpose resolver that computes \textit{the best version of truth}, based on a quality factor computed from the data’s

- believability, as defined source data-source precedence rules for each data type);
- completeness, determined by the ratio of non-empty fields over total fields;
- accuracy, as rated by the data source;
- timeliness, determined by data and time when the data was last updated
- verification, as recorded by the data source

It solves the problems caused by duplicate, inconsistent, and incomplete data by creating \textit{GOLDEN} records – one for each different type of data chunk. It maintains data-integrity by adhering to inter-field dependencies that requires certain pieces of data to be in the \textit{GOLDEN} records either together or not at all.

Although the ARS is specific to \textit{phMPI} project, it can be adapted for any person-centric information system. The design of the ARS allows making changes in one module without affecting other modules. Generic programming and the design patterns allows the code to be reused, high cohesion, and low coupling among the classes.

The current version of the ARS is a prototype that can be improved in number of ways. For example, the current formula for computing the Overall Quality Factor was developed only from insight to the phMPI data. Ideally, this formula should be based on empirical studies that compare data qualities across a number of person-centric information systems. Also, the prototype computes timeliness based solely on a last-
updated date. The means for computing timeliness could vary by data-chunk type and data source. For example, birth-event information from Vital Statistics may not be ever changed after its initial creation, yet it could and should be considered to be the most reliable data about a person birth event.
REFERENCES


[4] Requirements Definition of Public Health MPI (phMPI) project.


[28] Wang, Richard Y; Strong, Diane M, What data quality means to data consumers