

Chapter 2

Architecture

Although the various data exchange standards address the communication between different information systems, certain assumptions must be made concerning the capability and architecture of the connected systems in order to ensure interoperability. These assumptions are discussed in this chapter.

2.1 Interface

An interface is a mechanism (like a socket or port) by which data are imported and exported (i.e., exchanged) between two applications. An interface normally only refers to either a sending or receiving application, but in order to exchange data both end points are needed. (There must be some “thing” that provides the data, and another “thing” that consumes the data.) Analysis to distinguish between the different types of interfaces is not necessary here. An integration consists of an interface for a sending application endpoint (that assembles and prepares the data for export), an interface for a receiving application endpoint (that processes the data for import in a suitable manner), and a method for moving data between the endpoints (the transport). The content that is exchanged can vary significantly and is defined by (communication) protocols such as HL7 v2.x¹ or EDIFACT. Interfaces can support exchange of simple objects (e.g., an HL7 v2.x ADT message) or a set of transactions associated with an integration profile (a complex use case). For the former, the interface is dedicated to a single interaction (e.g., an HL7 v2 A04 message). For the latter, an implementation may choose, in a single interface (via a single module), to support multiple message events such as HL7 v2 A01, A04, and A05.

¹HL7 v2.x and other data exchange standards are communication protocols at the higher level of the ISO/OSI stack. Other communication protocols, such as TCP/IP, are at the lower level.

The communication interface is distinguishable from a technical interface within programming languages (e.g., Pascal, C, etc.); from object oriented approaches that deal with objects consisting of properties (attributes) and methods invoking activities/functions within these objects; and from links between libraries and executables, which, in a broader context and in a sense, are also interfaces. Different types and paradigms of interfaces are presented in this section, including an analysis of the procedures and the background knowledge necessary for implementing interfaces.

2.1.1 Types of Interfaces

First, a brief explanation of the different types of interfaces is presented. Regardless of which type is utilized, from a conformance perspective all interfaces share the same concepts and challenges [1–3].

2.1.1.1 Functional (Local + Remote)

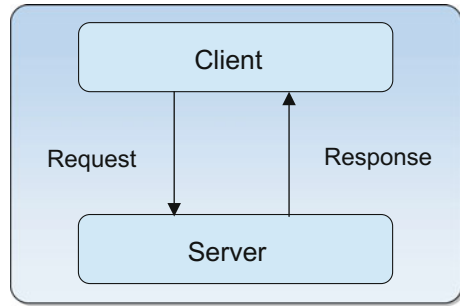
Most likely, the oldest type of interface for data exchange among disparate systems is the client/server relationship, i.e., a client system that uses common functionalities provided by a server. The client/server type of interface is based on simple functional libraries where a client makes use of the published services, e.g., to display data or to calculate certain results. The client has to know how the published services can be used, which is indicated by the function names and the required parameters, including the structure of returned results.

Typically, the client and the server reside on the same computer. In the “old days of computing”, the functionality was directly (statically) linked into the binary executable code, resulting in a single executable file. Later on, the functionality was separated into distinct files to separate algorithms from data thus enabling reuse, but all of the functionality was still part of a single executable file. Then, shared libraries were established to allow for reuse across applications, but these libraries were still located on the same machine.

Currently, client and server (Fig. 2.1) can be located on different machines. This approach is known as RPC—remote procedure calls. The latest technology used for supporting the RPC approach is called Web Services, which is an even further abstraction from specific programming languages and development environments and enables the combination of different solutions and products. In addition, with Web Services different clients may use the same server.

The client/server type of interface is used when an immediate response is desired with regard to a computed result or some other outcome on the server side. A key advantage of this type of interface is that it enables the response to be returned quickly.

Fig. 2.1 Function calls



2.1.1.2 Message-Based Data Exchange

Another type of interface involves connecting different systems to each other, typically systems that are located on different machines. This type of interface is normally used to transfer data from one system to another (see Fig. 2.2). The relevant data are collected and encoded into a message by the sender and then are transmitted to the recipient. After the data are processed by the recipient, the message may be discarded if all relevant data have been consumed; however, often (in the healthcare domain) the message is maintained for logging and auditing purposes for a specified period of time.

In the case of a simple unidirectional data exchange (e.g., data upload), the function call is sufficient. For an on-going communication or a more complex scenario, however, a response in the form of an acknowledgment becomes important. A more thorough analysis of the different types of acknowledgments is discussed later in this chapter (see the Sect. 2.4.1).

The exchange of messages requires a definition of the data format and of the method by which the data are transported. In the early days of data exchange, the encoding was accomplished by writing the data into simple files, separating discrete data elements by a specific character. Several records with equivalent structure are

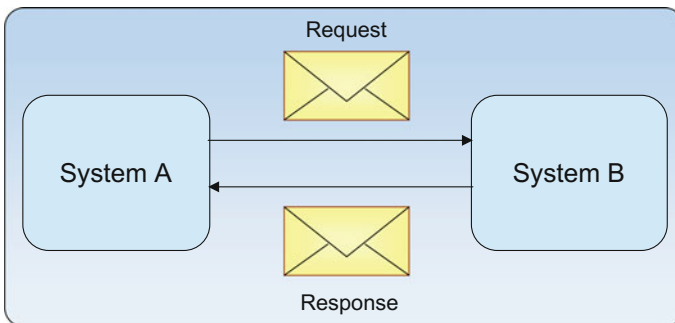


Fig. 2.2 Message exchange

represented as a set of rows. This type of file is called “comma separated file”,² and it still is in use for spread sheets. Typically, all lines in such a file have the same structure, but not the same length depending on the data. Prior to the deployment of this method, data were written into lines of the same length that specified the exact location and length of the data. This approach was also called “fixed length files”.

In the typical communication scenarios, most of the data are too complicated for a simple tabular representation (flat file) to be sufficient. For example, some concepts, such as diagnoses or clinical procedures, may repeat for a given patient. In this case, a variable number of repeat occurrences must be transmitted for these concepts while the administrative information for the patient is sent only once. For such a case, a repetition-based and/or hierarchical message structure must be defined, requiring an encoding system that allows for them. (HL7 v2, for example, provides this capability in a line-oriented format.)

In the end, the structural composition of messages can become very complex and must be managed hierarchically by appropriate separators.

2.1.1.3 Document Exchange

The same notion applies for messages and for the exchange of documents, in that document standards are necessary to define the content and format of the documents to be exchanged.

In principle, similar rules apply, i.e., documents are created by one system and sent to another. In contrast to simple messages, documents have a set of specific characteristics leading to a slightly different handling:

Persistence: The document is kept and stored after transmission. Normally both systems will have a copy of it.

Stewardship: A person or institution is responsible for maintaining it.

Accountability: A document can be signed (by a person).

Context: The context explains the environment to which the document belongs and why it has been created.

Completeness: A document has to contain all artifacts which are necessary to understand its contents.

Readability: A document should be presented to the human reader; therefore, it must contain elements that can be presented for display.

Figure 2.3 illustrates the document exchange process (the means by which the document is transported is not identified).

²The normal abbreviation is “CSV” which stands for “comma separated values”, rarely declared as “character separated values”.

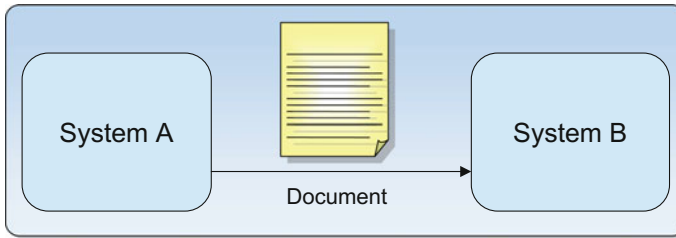


Fig. 2.3 Document exchange

2.1.2 *Communication Paradigms*

The exchange of information is normally triggered by one of the following situations:

Declarative: An event has occurred in the sending application and the sending system is used to inform one or more other systems about the current (changed) status and the associated data. A common example is the admission or transfer of a patient.

Interrogative: A system is used to obtain data from another system. In order to acquire the data, a simple request (query) is sent to the system that is assumed to have the data. The queried system will send a response. An example is the query to provide the demographic data for an identified patient.

Imperative: A system is used to trigger a specific transaction (command), such as when it is used to send an appropriate request. Orders to perform a certain task, like a lab test or radiology test, are the most prevalent kinds of requests in the healthcare domain.

Figure 2.4 highlights the different paradigms described above from a client/server perspective.³

The client is the system interested in the data/request and the server is the system that can provide the data/request.

The commonly used publish/subscribe mechanism is a combination of a request for information (subscribe) with the declarative notification (publish).

³At this point it must be noted that the systems are not interconnected in a traditional client/server architecture. The term “Server” in Fig. 2.4 indicates the availability of data, whereas the “Client” is seeking this data.

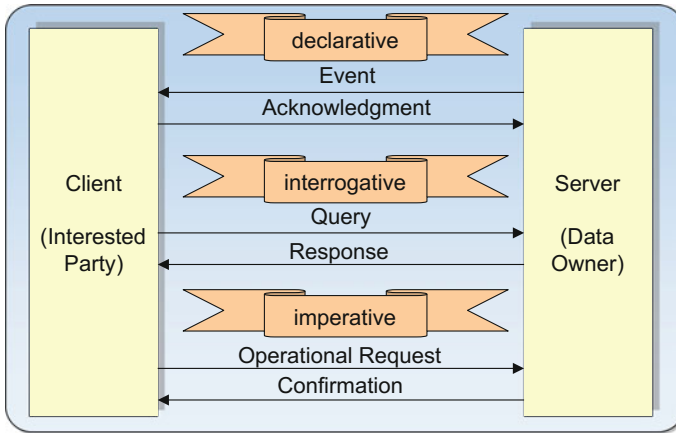


Fig. 2.4 Communication paradigms

2.1.3 Protocol Layer

Establishing interoperability requires a protocol stack supporting data exchange in a flexible and efficient way. On the lowest level is a simple technical interoperability that allows the simplest transmission protocols to be reused. Good examples are the hardware connections for TCP/IP. One may switch between twisted-pair connections, DSL or ISDN, while still using TCP/IP on a higher ISO level. Instead of just looking down to the IP in the ISO stack, one can also go into higher levels. As the lowest level protocol, IP is agnostic to the higher level contents. Consequently, two different higher level transport protocols exist that are incompatible with each other: TCP and UDP. Both provide different properties supporting data exchange in different ways.

The layers of the protocol that stack on top of TCP/IP and UDP/IP mediate among the technical and physical data exchange on the one side and the reuse of data as the foundation of interoperability on the other.

2.1.4 Levels for Interoperability

Interfaces can be established on different interoperability levels according to the layer of abstraction, as illustrated in Fig. 1.3 (Sect. 1.5).

The principle of the ISO stack applies in the same way on the application (processing) level with structural, syntactical, and semantic interoperability. For example, a syntactic understanding of what has been exchanged does not necessarily produce a correct interpretation of the contents. This interpretation must be applied manually.

With the increased interest in semantic interoperability, data will be accompanied by meta data, which will facilitate a correct understanding of the data. For example, a code such as “M” is not only placed in a certain location within the exchanged data, indicating that the correct interpretation is “married”, but the code is associated with a specific code system allowing it to be distinguished from the interpretation “male”, which might also be coded as “M” in another code system.

On the highest interoperability level we talk about organizational or service interoperability, which allows for direct calls of foreign services. This capability will become possible when real semantic interoperability is achieved.

2.2 Interfacing

When it comes to exchanging data between two different systems⁴ (see Fig. 2.5), normally two interfaces are communicating with each other (see point 1 in the diagram). One system is sending data, and the other is receiving the data. In doing

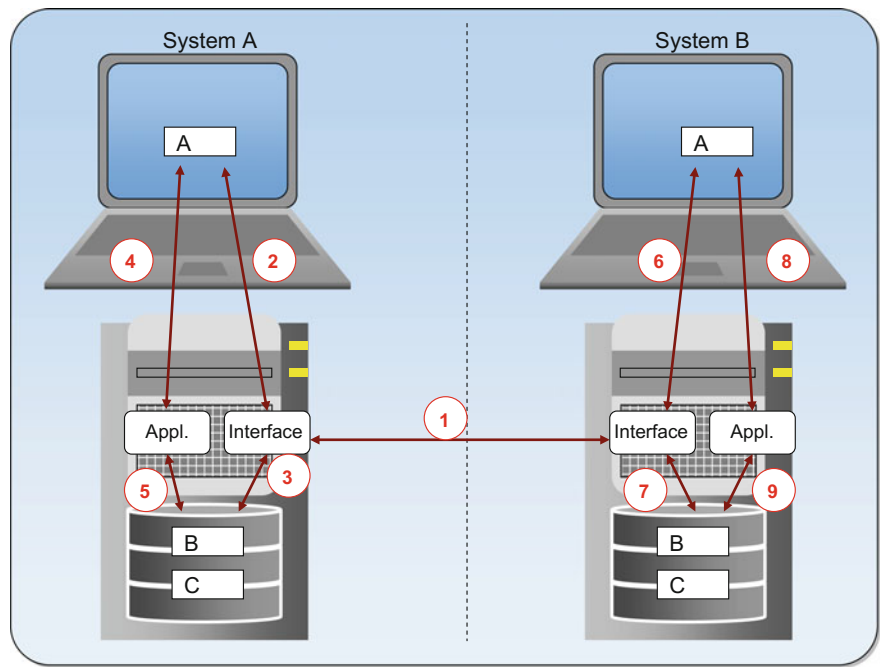


Fig. 2.5 Interfacing

⁴The arrows in Fig. 2.5 indicate the possible data flow within both applications. The accompanying numbers show the route of the data in the systems.

so, both are using the same specification, which is essential for interoperability. It is probably easy to understand that if two systems are not speaking the same language (i.e., implementing the same specification) they cannot share their data; but, as we will see later on, this prerequisite (using the same specification) is necessary, though not sufficient on its own because the appropriate understanding must be conveyed as well.

The sending system has to collect the data and convert it into the “language of the agreed upon protocol”. Normally, this task is accomplished by the sending system accessing a data repository (3). The receiving system extracts the data out of the transmitted message and stores it within its own internal data repository (7).

In most cases, the interface modules are not concerned with what the application program for the graphical user interface (GUI) is doing (4, 5, 8 and 9). Furthermore, it is quite unusual for the interface module to take the data from the screen (2) or to present it directly to the screen (6). However, the latter (6) is perhaps possible for display messages, i.e., messages sent as a request do provide the information directly from a message instance to the screen to alert the end-user. In most cases, directly providing the information from a message instance to the screen is achieved by simply showing a message box that the user has to acknowledge.

What induces a sending system to send data? Data transmission is often triggered by data being entered into the system—which could mean the trigger is associated with a real-time event—or by regularly scheduled services (cron or at-jobs⁵) that prompt an interface module at pre-defined times when data require transmission to a communication partner.

How the trigger is conveyed to the interface module—directly (e.g., by message queues) or indirectly (e.g., by specific data repository tables)—and whether the trigger is managed in the user interface of the application or interface module at all are reasonable technical questions; but they are, in principle, irrelevant questions for this book and are not discussed in detail.

2.2.1 *Impact on Interfaces*

What are the ramifications of the information flow presented in Fig. 2.5?

When talking about interface specifications, the underlying architecture of the system should not matter; nevertheless, interrelationships do exist. Whether a system is capable of supporting specific features or transmitting the information requested using a standard are important aspects to know. Therefore, the specification of standards should clearly delineate these requirements: (a) whether a system must be capable of supporting specific information (relevant for a developer), and (b) under which circumstances this information is present in a message

⁵These are services that are provided by different operating systems and that run at scheduled times to perform a certain task.

instance (at runtime). This separate delineation is typically described as implementation requirements and operational requirements. (The differences for sender versus receiver are discussed separately in this chapter.)

In a standard, implementation requirements concerning the handling of a specific element can be indicated with a simple yes/no flag. If the flag is set to “yes”, a developer has to write lines of code for the requirement. In derived specifications (i.e., profiles, see Chap. 7), this flag can only be switched from “no” to “yes”. How this indicator is realized in data exchange standards varies—Chap. 5 provides examples. How this manifests for operational requirements is a bit more complex and is explained in the next section.

2.3 Serving Interfaces with Data

A totally different question pertains to the use of a general capability of a system in combination with its interface behavior. It is usually thought to be true that specific information that is, in principle, available within the data repository of a system (e.g., data elements B or C in Fig. 2.5) can also be exchanged with a foreign system (points 3 and 1 in Fig. 2.5). Of course, this supposition is not true. This capability is determined by the (internal) directives of the corresponding system.

From an interface perspective, it makes no difference whether specific data are hidden from an interface module, whether the data are not implemented at all, or if (data privacy/security/protection) policies prohibit making use of the data. A system functioning under these constraints must be treated as though it is not capable of handling the data (for a particular use) even if another interface within the same system may very well support the data.

2.3.1 General “Capability” of a System

The capability of a system (see Fig. 2.6) acting as an information partner within a given use case is directly influenced by how the information is obtained and subsequently handled internally. Potentially, three different situations might occur:

- (1) The system’s user interface offers a field to enter or to present specific information that is not stored internally.⁶ This capability, represented as point

⁶When talking about “data repository”, we have in mind any means to store information internally, regardless of how it is achieved or whether the data are used at all. Typically, one thinks of (relational) databases, but other mechanisms (e.g. files, USB-sticks, flash memory) are possible as well. From the perspective of this book, these distinctions are not relevant.

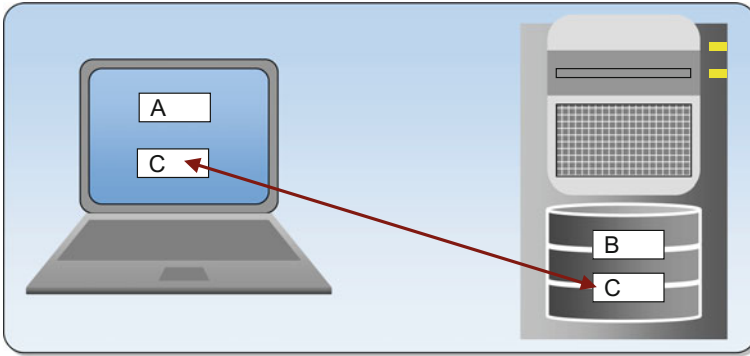


Fig. 2.6 General system capability

A in Fig. 2.6, is provided for handling temporary information and is rarely used.

- (2) The system stores specific information internally, but does not present it through the user interface.⁷ Here we do have persistent information present, represented as point B in Fig. 2.6.
- (3) The system's user interface offers a field to enter or to present specific information that is also stored internally.⁸ The arrow in the following drawing exemplifies the correlation, represented as point C in Fig. 2.6.

For interfacing, options (2) and (3) are essential in order for the system to be capable of providing specific information, given that an interface is referring to persistent information. Of course, a system may take information from an interface and provide it for display, which is also a means of meeting the requirement, but we want to abstract from this situation.

Although systems are typically implemented following option 3, the fact that a field entry mechanism is presented via the user interface does not necessarily imply that information is also entered, stored, and communicated to another system. In order to ensure accurate data entry, the user interface may contain mandatory fields that the user must complete before they can continue on with the application's workflow (e.g., name and date of birth of a patient). However, some fields are not required (e.g., address), i.e., completion is dependent on the user knowing or being able to obtain the data, and then entering the data before the system subsequently stores them. Hence, even if a system is capable of supporting the data, nothing can be communicated if the fields are not populated.

⁷System timestamps or internal identifiers are candidates for this kind of information.

⁸So whatever is typed into a certain field will be transferred—in more or less the same way—into a field of the data repository. Whether it is taken directly or indirectly, combined with other information, or separated into different data repository fields does not matter at this point.

However, in the medical domain, binary logic, meaning a piece of information is either present or absent, is sometimes not sufficient. In many situations the information cannot be provided to the user (e.g., due to the patient being unconscious) or simply is not obtainable for various reasons. Highly sophisticated applications provide some means to enter pertinent details that explain why the data were not entered, e.g., the patient was asked but was unable to answer, the patient was asked but refused to answer, or the patient was not asked at all. In some instances, knowing this information at this level of granularity is important for making informed medical decisions. Another option is to mask the information, i.e., to send something to the receiver indicating that the information is available but is not provided due to privacy policies, for example. In reality, applications allowing for such a distinction are very rare, because more sophisticated capabilities are required of the system, which will be explained later (see Sect. 2.9).

2.3.2 *Relevance for Interfaces*

As a consequence of the variety of legitimate reasons that certain data cannot be entered and a particular field in a message cannot be populated, in order to verify the general capability from an interface perspective, a system must create a message where such a data element is populated (disregarding policy issues and other regulations). If a system processes specific information, the situation must exist where this information is made available via an interface, otherwise, the system is declared not capable of conformance to this requirement. At this point, we have to remind the reader that such a statement is made based on the experiences gained during actual testing. Determining a system's ability to support a capability is easily handled during testing; see Chap. 11 for more information. Operational factors, such as regulatory requirements related to data privacy that prevent a system from sending (populating) the data, are not considered. Testing for capabilities is often in controlled environments where real patient and clinical information are not used.

As explained previously, this discussion focuses on capabilities required by an interface, which is one targeted use case of this book. A distinction has to be made between what a system can handle in general, and what the interface can convey or allows. Before an interface can handle the data, the application must be capable in general. But the opposite is not true. If the data is not made available for the interface (for whatever reasons), the application is declared as not capable even if it can handle the data internally. The "capability of a system" is decided from the communication partner's point of view. In other words, an interface behaves like a firewall, because only defined data elements are allowed to pass through.

2.4 Dynamic Behavior

Besides the static aspect of messages, i.e., how the messages are structured, the dynamic behavior is of interest as well. Dynamic behavior refers to the interaction specification for an exchange between two or more systems.

2.4.1 Message Pairs

As explained in the discussion about interfaces, information is typically exchanged between systems in both directions; a message is sent and a response is returned (see Fig. 2.7).

Most of the systems are implemented to support a simple (and single) message as a response to their initial message (“payload” in Fig. 2.7). In most cases this response indicates that the message has been accepted by the receiving application. Such a response is commonly referred to as a transport acknowledgement (ACK) in which a 1:1 message exchange is performed. However, a transport ACK does not indicate whether the message has been processed according to the rules established by the receiver; it simply indicates that the message was received. If the message can be processed and consumed, then an application ACK may be sent (note that not all use cases include the possibility of an application level acknowledgement). An acknowledgement response may not be sent at all, so a sending system may receive zero, one, or two messages in return.

Most data exchange standards limit acknowledgements to one or none at all. More explicit standard definitions support specific conditions under which a response should be returned, they include:

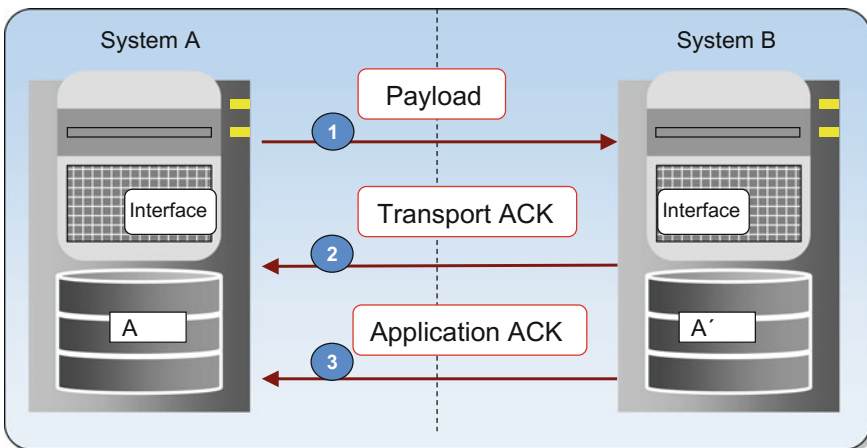


Fig. 2.7 Message pairs

- always
- in case of an error
- after successful processing only
- never

As a consequence, the sender should keep track of all messages sent and the expected responses, so in case of an unexpected behavior an appropriate notification is sent to the system administrator.

As we will see later, the situation where various recipients respond to a sender's message differently (positively and negatively) requires special treatment.

2.4.2 *Timing*

Implementers need to know about the timing of the messages, i.e., the relationship between a message and its response with regard to time interval. Some standards specify that a (single) response should be sent immediately, others state that no response is to be sent, while others, such as HL7 base standards, are silent on this aspect.

When the standard is silent, some implementers believe—or interpret it to mean—that a single response message should be sent immediately, and then they agree on this behavior with their communication partners. Of course, although this simplification is pragmatic, it leads to no support for zero response or two response messages as explained in the previous section, because a single response is made required. Such a solution also goes hand in hand with certain architectural decisions when developing the interfaces prohibiting enhancements.

As Fig. 2.8 demonstrates, this approach is then combined with a specific dynamic behavior: the transmission of a given message is repeated several times (1a, 1b and so forth) if no transport ACK is received within a certain period of time (timeout). This process continues until a response (1) is received or a maximum number of repetitions have been reached. When a data transmission fails, it is good practice to notify the system administrator immediately, because most likely the reason will influence real-life workflows.

The sequence diagram in Fig. 2.9 shows that several messages can be sent before a single response is received. Although rarely used, this behavior has to be accounted for in the design of the system architecture.

2.4.3 *Message Identification*

There are two common paradigms used in message exchange. In synchronous communication, the sending application waits for the corresponding receive (acknowledgement) before proceeding with the next message. In asynchronous

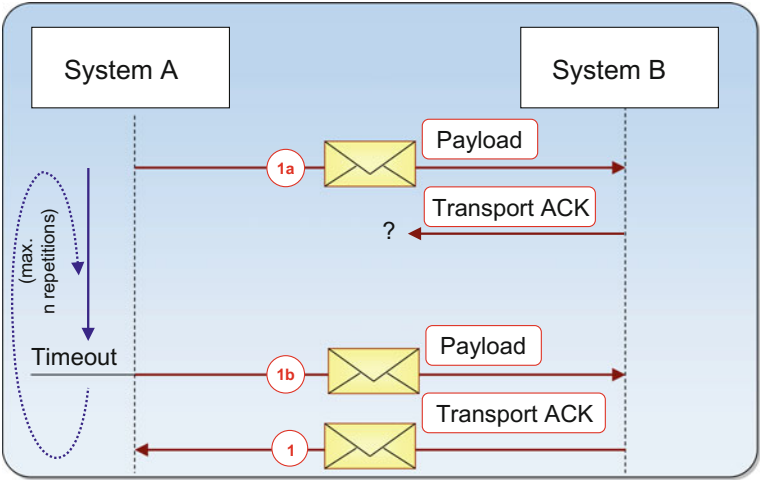


Fig. 2.8 Timeout in sending messages

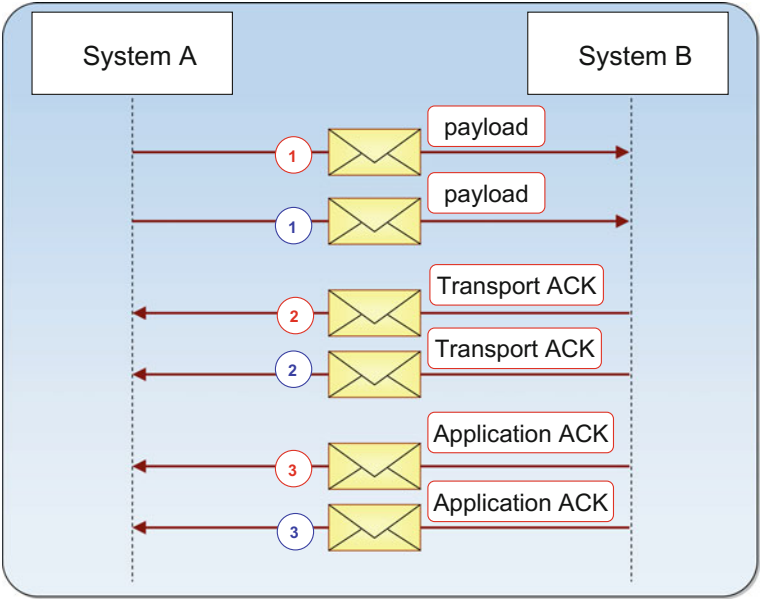


Fig. 2.9 Timing of messages

communication the sending application does not wait for the response (acknowledgement). The former has a “tight binding”, and the latter has a “loose binding” and hence the need for a message ID. A response in the form of an acknowledgment must clearly indicate to which originating message it belongs. Therefore, some

means to identify each message uniquely is required. The most common mechanism is to create a unique identifier for each message and to keep track of all messages sent. The response has to include the identifier of the originating message to which it is a response. In HL7 v2 the message header (MSH segment) contains this unique identifier, whereas the response references this identifier with the MSA segment.

2.4.4 Routing to Multiple Destinations

In the case of the declarative communication paradigm (see Fig. 2.4), the messages are sent concurrently to a set of recipients that have coordinated the need for the data. The ADT⁹ messages are a typical example. A set of systems managing patient information needs to be informed about new patients and updates to existing patient information (Fig. 2.10).

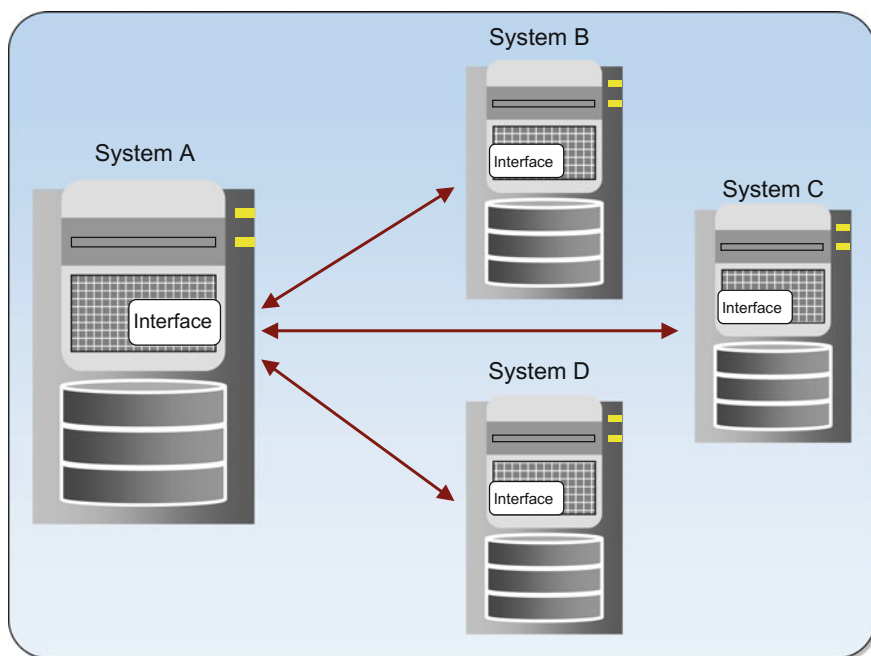


Fig. 2.10 Message routing to a set of recipients

⁹ADT is a typical abbreviation in English speaking countries and denotes “admission, discharge and transfer”. More precisely, this is taken from HL7 version 2.x and Version 3. In Germany, ADT has a different meaning and refers to a family of standards created by KBV and ZI.

Even if the internal architectures of systems are more or less identical and their ability to process the transmitted messages is comparable, there is still no guarantee that all connected systems will respond in the same manner. As such, it is important to analyze the behavior of each connected system to determine the capabilities for processing a message, i.e., will none, some, or all of the systems accept the message? This type of probing is necessary even when a system publishes a statement of capabilities, because, in practice, often what is stated and what is implemented are different.¹⁰

Some systems will accept the transmitted messages while others will not, in which case the connected systems may be operating on different internal states. When a receiving system does not accept the message, the sending system has to react by notifying the receiver's system administrator (i.e., usually through a human-mediated manual process). This notification can happen only if the sending system is aware of differing results on the application level.

This use case demonstrates the important differences between a transport acknowledgment (receipt of the message) and an application acknowledgement (processing of the message). A positive transport acknowledgement indicates that the message was received, but it does not indicate that the data in the message were successfully processed. A subsequent negative application acknowledgement could be sent by the receiver, therefore, the sending application should be capable of handling application acknowledgements in order to avoid data fidelity issues.

2.4.5 *Responsibility of a System*

Independent from the correct acknowledgments¹¹ on the transport and the application levels, a message is sent to initiate a response within the receiving system. In other words, a message is exchanged to trigger a reaction on the receiver side. The most common reaction to the transmission of a message is for the receiving system to store new data. Therefore, the appropriate reaction must be indicated by the event that triggers the message. For example, the creation of a "new patient" in the sending system should result in a new record in the receiving system.

¹⁰The American Immunization Registry Association (AIRA) has conducted such tests comparing requirements in published implementation guides to working immunization information systems. Often there is a discrepancy between what is stated and what is implemented.

In Germany, the KH-IT [4] (a professional association of head of IT departments in hospitals) has conducted studies and created a detailed workflow description about ADT messages and the corresponding expected acknowledgements and corresponding result statuses. This summary is an excellent starting point to verify test cases regarding their correctness and completeness.

¹¹In practice, integration engines are used to eliminate problems in the interpretation of the data exchange standards. These engines facilitate the use of transport acknowledgments that are sent synchronously, i.e., in an immediate 1:1 response to the initiating message. This capability hides (a) the fact that users cannot rely on an immediate response and (b) the existence and awareness of application acknowledgments.

To summarize, two different systems (as a sender/receiver pair) should operate in the same way on a user interaction by the means of exchanging an appropriate set of messages to stay in sync. For example, if the receiving system does not allow for merging of records, the internal state of the interoperating systems will be different when such an event occurs. If the sending system is capable of communicating a request to merge two records (and the trading partners are expected to do so), then the merge capability is a requirement for the set of receiving systems with which the sender interoperates. The receiving systems have the responsibility to process the messages accordingly.

2.4.6 *Event Handling*

As described above in Sect. 2.3.1 (Fig. 2.6), systems provide a set of forms (dialogs/windows) used to interact with the application, i.e., a user works with a set of forms for entering information. Upon acceptance by the user, the content of the form is stored by transferring the information into a set of data records. The structure and contents of the records depend on the application. The conventional and quite common behavior of an application is to store information about the actions a user has performed with the application. This logging is then used to trigger, i.e., create and exchange, the appropriate set of messages. The best way to handle those events is the decoupling of foreground editing (data input) from the background asynchronous processing (sending of messages).

Of course, depending on the application, the set of internal events (e.g., the various aspects within an admission process) differ from those being defined by a data exchange standard. In other words, even if an application supports a set of events, this set may intersect only with the standard-defined events (i.e., different coverage) and have a diverging granularity. In addition, the set of events to be supported is diverging between the different data exchange standards.¹² One of the most important steps in developing an interface for a specific data exchange standard is the mapping of the internal events to the events defined by the data exchange standard. The more closely the internal structure of the application mirrors the structure defined by the data exchange standard—or, in other words, the more aligned these structures are—the easier the mapping is to accomplish. (But it is not advisable to align an application with a specific single interoperability specification, because competing standards exist that must be supported.) The more finely grained the set of internal events is the easier the mapping will be. But in some circumstances, the mapping may be difficult and potentially not achievable, because of non-mendable, incompatible structures, in which case system redesign needs to be

¹²This can be observed in the administrative domain with HL7 version 2.x and Version 3: In HL7 v2.x the ADT messages cover in- and outpatient management, whereas in HL7 Version 3 home health and emergency admissions must be managed as well. Therefore, HL7 V3 has a more matured set of events with a finer granularity.

considered. In the case where an application needs to support multiple data exchange standards (e.g., HL7 v2.x, HL7 V3, and HL7 CDA[®] for sending lab results), an internal sophisticated information model that is agnostic to a particular protocol may be more advantageous.

2.4.7 Delayed Message Handling (Sender)

Certain data fidelity issues can arise as the result of delayed message handling. For example, data may be entered and collected, and then subsequently updated. In such a scenario, multiple events have occurred that affect the patient record and that need to be communicated to one or more receivers. The sender may implement message creation and transport in a number of different ways. Two common approaches for creating the message are to (1) construct the message instance immediately upon data capture from the input source, and (2) store the data in the data repository and create the message instance at a later time. Each approach has its pros and cons, and certain events that occur during the process must be accounted for in order to ensure data tracking is handled correctly (if there is a need or desire for data tracking).

A typical system architecture will separate data input (e.g., use of input forms in the user interface) and data persistence (e.g., preservation of the entered data in a data repository). In this context, the data input aspect can be thought of as the client and the data repository as the central application server. Two approaches are examined for dealing with delayed message handling that would occur when initial data are entered into a record, and then these data are updated.

In the first approach, data (e.g., a patient address, which we will call PA-1) are recorded about a patient. Upon completion of the data capture, the data are extracted from the input form and saved into the data repository, and a message instance is immediately created (i.e., close to the timing of the edits). Note that in this approach the message instance is created directly from the data that were captured. A short time later the patient record is updated to correct the patient address (PA-2). Again, after completion of the data capture the data are saved into the data repository, and another message instance is created. Sometime later (typically, the sender will send each individual message that was created immediately if possible), the messages are sent in the sequence in which they were created. The first message will contain PA-1 and the second will contain PA-2. In this case, the sender will have handled the state of the data correctly and provided the historical trace of the data to the receiver.

In the second approach, the data are stored in the data repository, but no message instance is immediately created. An indicator that the event occurred is recorded for later processing. When the update to the data is made, PA-1 is replaced by PA-2 in the data repository. Upon creation of the messages, both will contain PA-2. After the messages are processed, the state of the receiver will be the same with the first approach as with the second approach; however, a historical trace of the data is not provided to the receiver with the second approach. In order to track the data

changes, a shadow data repository could be used in the sender to store PA-1. The importance of data tracking varies depending on the use case and the data.

The first approach described above eliminates the problem of delayed message handling at the cost of decentralizing all interface issues, because support for different data exchange standards must be implemented several times. The second approach results in event-handling as a separate procedure, so that a trigger event can be stored immediately after a user interaction, but is not processed immediately. So it can happen that several subsequent edits on a single record are executed before the associated events are processed accordingly.

2.4.8 Handling of Outdated Data (Receiver)

The usual paradigm of data exchange is to send the most recent data such that recipients can maintain the correct state of the data. An implementer has the choice of supporting only the most recent data or establishing a historical trace of all data. In the case of the former, the application overwrites the data currently maintained in the system (data repository) without regard to historical data preservation.

A sending application that always sends the most recent data enables a receiving application to establish a historical trace of data. Of course, the extent of this history is tied directly to the number of events that are triggered in reaction to modifications made to the initial data that were sent. The more details (i.e., more data elements) that are maintained about a data record the better the chance of updates. Consider a scenario where a patient is admitted to hospital unit “A” on day 1, transferred to unit “B” on day 2, and finally to unit “C” on day 3. When this sequence of transactions transpired, however, an error was made. The patient was actually transferred to hospital unit “F” on Day 2. Another example would be the case of a caregiver entering a diagnosis that later was found to be erroneous and subsequently was updated. How does the receiving system recognize the *outdated (invalid)* data and what steps does it need to take to correct the data?

Some receiving applications do not track the history and always expect to get the most recent data, while other applications maintain a historical trace for the data. Therefore, it is imperative that a mechanism exists that clearly indicates whether a message contains the most recent data or *outdated (invalid)* data. Different solutions exist (or should exist) in each of the current standards to account for this situation.

One method that has been specified by the IHE Patient Administration Management (PAM) integration profile is to provide a separate customized event (a Z99 event). IHE adopted this approach because they thought that implementation of the A08 (update) event may ignore the subtlety of an *outdated (invalid)* data indicator in the message. Some of the impetus of this reasoning was that many legacy systems would not be updated (to account for the subtlety of the outdated data indicator) and thus processing of an A08 message would lead to erroneous destruction of data. Use of the special trigger event increases the possibility of correct implementation, because a clear distinction can be made between data

included in the most recent data event (“A08”) and data in the *outdated* data event (“Z99”). Systems that don’t support tracking of historical data will ignore the outdated data event (“Z99”) to maintain data fidelity. Evaluating this distinction at the event level may raise the awareness of implementers, allowing them to determine quickly (and, hopefully, unambiguously) the timeliness of received data before processing them so that a misunderstanding and destruction of the most recent data are prevented. This example illustrates one documented solution; other solutions are equally plausible.

2.5 Intermediate Message Handling

Another topic worth discussing is the forwarding of messages by a receiving system (System B in Fig. 2.11) to other systems. This section explores the impact on the data, but it does not define the target system or any receiver responsibilities in the case of misrouted messages or error handling. The correct handling of received information becomes important if the data are used as input to other applications.

What happens if information received from System A via (1) should be sent to System C via (5)? The easiest approach to use is for the intermediate application (System B) to act as a ‘pass through’, routing the message without consuming the incoming message and recreating a new outgoing message. This solution is quite uncommon and not recommended, because the routed information should be consumed and stored by the intermediate application (System B)¹³ (otherwise, no

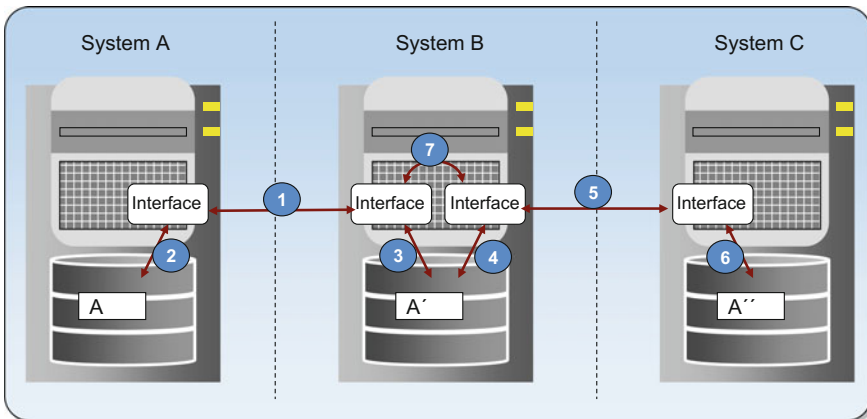


Fig. 2.11 Message routing

¹³This even applies to, and is common practice of, integration engines, although they do not act upon the data in any meaningful way.

real bridging to other data exchange standards or re-use for other use cases is possible). Therefore, the receiving interface (System B) stores the information within the data repository (3), and the sending interface extracts the data from the data repository upon request (4). Another option is the internal routing between both interfaces (7), which is also an uncommon implementation.

But even if data are sent from System A via System B to System C, it is not necessarily true that the information stays the same after it is stored in the various data repositories (A, A' and A'' in Fig. 2.11). The different interfaced modules may translate the information received in the message from the protocols required by the interface message standard into an internal format/value/code.

As has been explained in Sect. 2.3.1, storing this information internally can be defined as “being capable”. A distinction between (B) and (C), as shown in Fig. 2.6, is not necessary here. [Note: (C) would present this value to the user; (B) will keep it hidden].

2.6 Message Population

Achieving semantic interoperability requires some means of identifying misinterpretations when creating or consuming a message. However, one cannot detect whether there is a (semantic) mismatch of information on the sending side (Fields 2 and 3) or the receiving side (Fields 1 and 2) by using a given standard (represented by the cross-overs in Fig. 2.12).

Therefore, the different parties have to trust that the information populating the fields in a messages is accurate, i.e., whether a date in the field for the patient date of birth really represents the date of birth and not some other date. This issue is important, but it requires an abstract means of expressing the semantics of a field

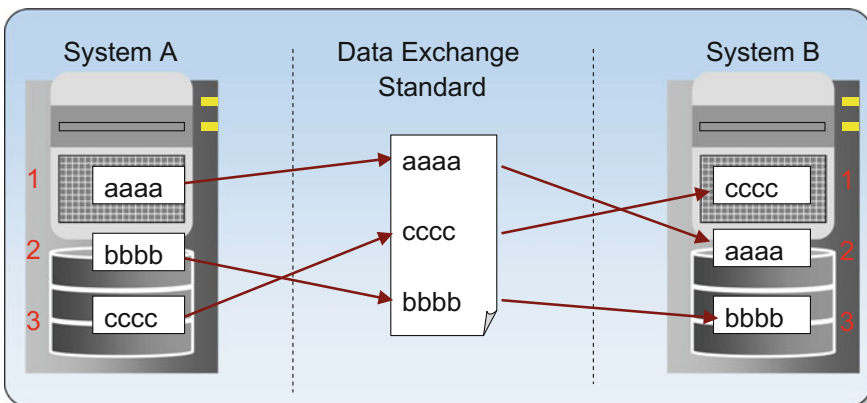


Fig. 2.12 Message population

according to the processing of the data within the application to avoid misinterpretation of data element. As explained in Chap. 1, this feature is introduced on higher interoperability levels by semantically annotating data elements. Until this feature is available, trust in the data is essential, and misusing fields for other purposes is strongly discouraged and should be prohibited.

2.7 Information Transmission

The most conventional method for data exchange is initiation of an appropriate event in response to data being imported into an application (e.g., a patient is registered). Figure 2.13 demonstrates this process with artificial elements and data, where data are provided, collected, and packaged in System A (on the left side) and sent as a message to System B (in the middle). System B incorporates the data (right hand side). The table associated with System B indicates the known state of System B after each transaction.

As can be seen in transactions 1, 2, and 3, as long as valid data are communicated by System A, the data in System B are as they are expected to be. An issue arises when data are deleted without being replaced by new data, as shown in the transaction 4.¹⁴ Element Y is deleted from System A’s data repository (as indicated

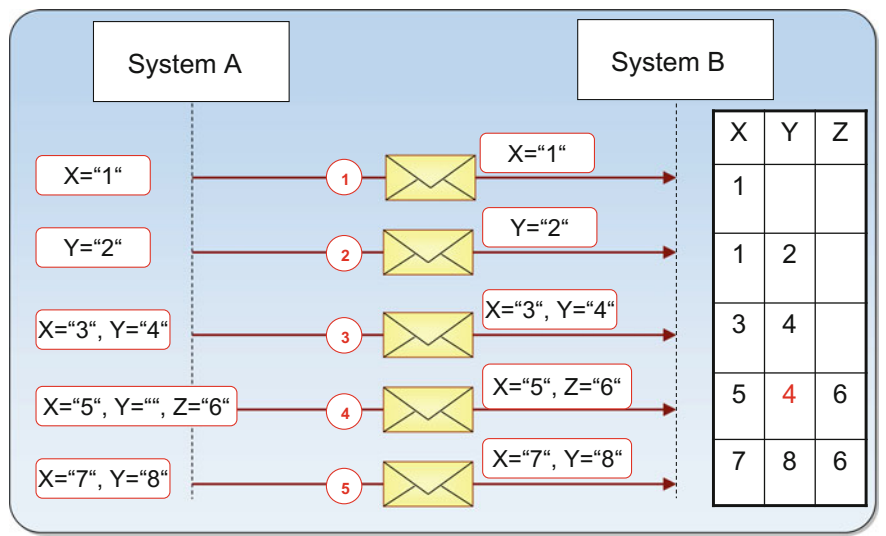


Fig. 2.13 Information transmission

¹⁴Assuming that the intent of the workflow is that the systems have matching data.

by Y = “” on the left hand side)¹⁵ because the data are no longer valid (see Sect. 2.4.6). However, since no data, nor any notification about the deleted data, are sent, the receiver does not have any indication that something has changed, and, therefore, cannot delete the erroneous data. This situation is denoted by the red “4” in the table summarizing the current values in System B. At this point in operation of these applications, Element Y should be empty in both systems so their data are synchronized.

There are different solutions for indicating to the receiver that data are no longer valid. The first option is to send a “placeholder” indicating that the data must be deleted (to be explained in the next section). Other solutions include the snapshot and update modes, which are covered near the end of this chapter.

In transaction 5 shown in Fig. 2.13, new data are sent and both applications are in sync again.

2.8 Delete Indication

HL7 Version 2.x has the notion of a “delete” indicator,¹⁶ which is two double quotes (“”).¹⁷ This indicator notifies a trading partner that specific data are to be deleted from their system because the data are no longer valid. To facilitate implementation, the sending system must be capable of recognizing the need for data to be deleted when an event is being processed. The easiest way for the sending system to meet this requirement is by maintaining the history of previous (i.e., “out-of-date”) data. The most common method for persisting historical data is the use of a “shadow table”.¹⁸ Figures 2.14, 2.15 and 2.16 illustrate the shadow table concept.

Figure 2.14 illustrates the status of a value after the initial data entry. The value is stored in the data repository, the shadow table is empty (white box), and the value is presented to the user on the screen.

Now the user enters new data (“B” in Fig. 2.15) for this particular data element, which should be stored, thus overwriting the old value (“A” in Fig. 2.15). The

¹⁵HL7 v2.x provides a notation to use for explicitly indicating that the data for a specific element must be deleted. The notation consists of two double quotes (“”). Most other standards do not handle deletion at all or provide an explicit transaction for this purpose. Furthermore, it must be noted that not sending information is different from sending a delete request, and both of these actions are different from sending null values. Each of these circumstances is explained in this book.

¹⁶HL7 v2.x is the only standard the authors are aware of that supports a delete indicator.

¹⁷Quite often, the concept of a delete indicator is misinterpreted as a null value! Also, some systems erroneously send two double quotes to indicate that the field is empty, which triggers an incorrect action by the receiver. This is especially important in case the receiver is collecting information from different sending systems.

¹⁸There are additional methods for storing historical data (e.g. SQL stored procedures) that have their advantages and disadvantages, but these methods are left to the reader to explore.

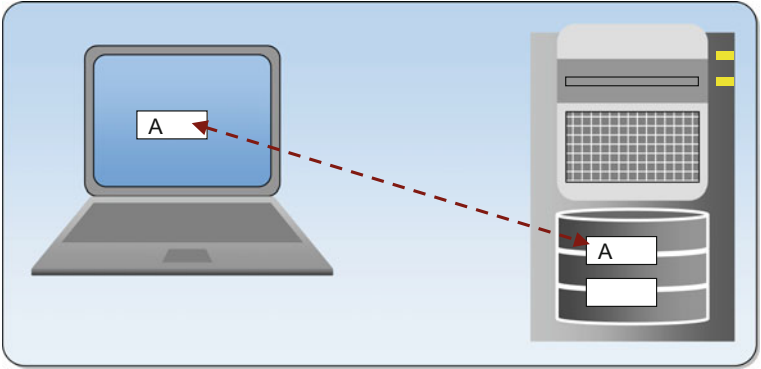


Fig. 2.14 Shadow table concept (after first data entry)

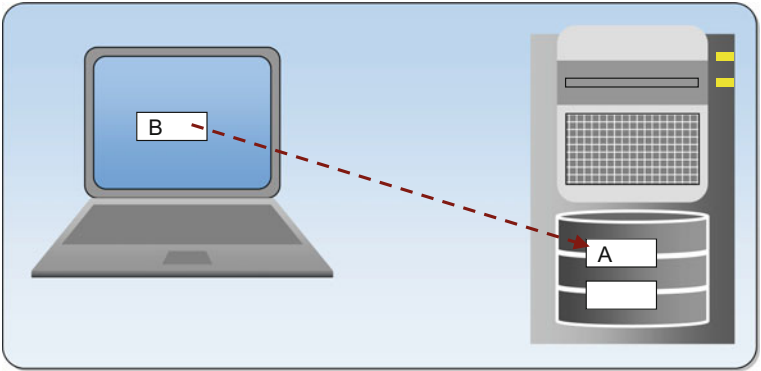


Fig. 2.15 Shadow table concept (before data update)

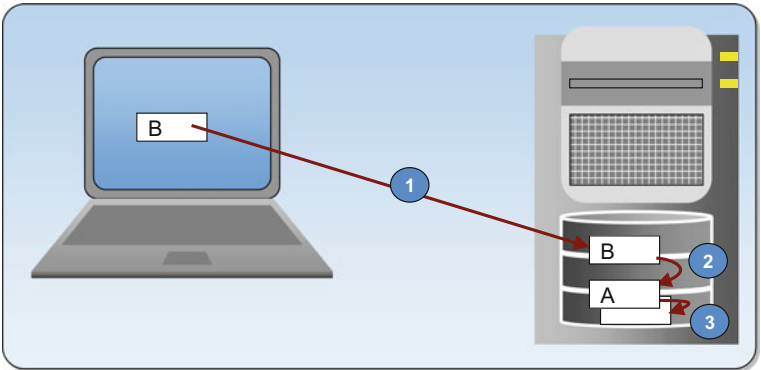


Fig. 2.16 Shadow table concept (after data update)

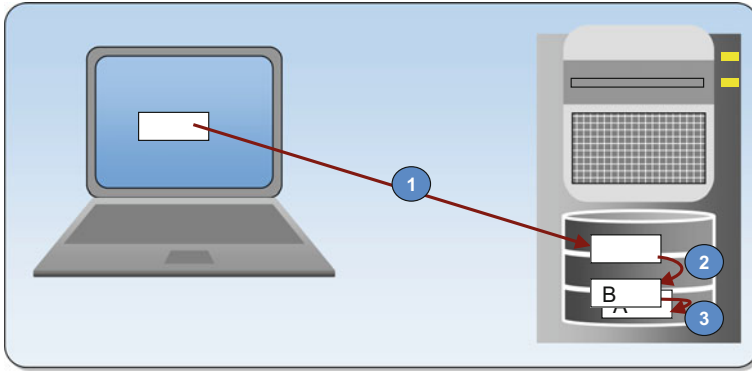


Fig. 2.17 Shadow table concept (delete data)

system becomes aware (e.g., user enters data) of an updated value (“A” in Fig. 2.15) for a particular data element (indicated by the white box on the screen containing “A”). When saving this information in the data repository¹⁹ (arrow 1 in Fig. 2.17), the following transactions must occur as an atomic process¹⁹: the previous value “A” for this element is transferred internally into the shadow table [as designated by arrow 2 (Fig. 2.17)], moving the old value “A” to the shadow table; the current (“new”) value (“B”) is transferred to the “active” location for the element, replacing the old value; and if previous data existed in the shadow table, then another layer of tracking data is employed as shown by arrow 3 in Fig. 2.16. Hence, the trigger of the whole transaction is arrow 1, but the processing sequence is 3, 2, and, finally, 1, so data loss is prevented.

Note that this set of events also can be applied when the data are to be deleted. In this case, “B” is written to the shadow table and the “active” element location becomes empty as is demonstrated with Fig. 2.17.

Without such a mechanism,²⁰ an interface module is unable to distinguish between a simple missing (empty) value and a deleted value, because the internal appearance as a record set value is identical. There is a precondition, however, on the sending side. Having the shadow table capability allows a system to identify data that are to be deleted and to send the two double quotes in the message, notifying the receiving system to execute the deletion. The receiving system can process the deletion request without having the data history and can simply react to the delete indicator.

¹⁹A quite common method employed to accomplish this process is to use stored procedures in the data repository.

²⁰The concept of a shadow table is not the only way to achieve a delete indication. Another option, for example, is to use stored procedures in common DBMS where old and new data are available simultaneously. This method allows for detecting of a deletion and storing of this information for the interface module, because in this case the old data are not necessary.

2.8.1 Legal Requirements for Data Persistence

Deleting information must be handled with care. In most settings, legal requirements (e.g., accountability and liability issues) oblige entities to keep track of a subset of data and any changes made to previously stored data.

A physician may be called upon to show the information upon which their treatment of a patient is based. As such, any change in the information must be tracked along with relevant dates and times of updates, especially in the case where the data were no longer applicable²¹ and were deleted.²² Relevant information must not simply disappear from the patient record in the data repository.

This topic is discussed in more depth with the snapshot and update modes for transactions.

2.8.2 Receiver Responsibility

Receiving applications are expected to be able to process the delete indicator in accordance with the prescribed semantics. In operation, the receiver is informed that the data sent previously by the sender are no longer applicable and that the sender is requesting they be deleted.²³

An example of data for which a sender might request a deletion is a patient address. The clerk at the front desk realizes that he typed in a wrong address and eliminates it from the input form. The form is submitted, which causes the system to move the current data to a shadow table and to remove that data from the “active” data repository. The interface module compares the empty value in the “active” element location with the value in the shadow table and concludes that data have been deleted. As a consequence, the interface module inserts the delete indicator (“”) into the message location corresponding to the address element where a value has been deleted. The receiver interprets the two double quotes (“”) as an indication to delete the address from its data repository. If the receiver wants to maintain historical records, then its “shadow table” is updated.

²¹This may include incorrect data or outdated data that are no longer pertinent.

²²A distinction must be made between logical and physical deletion of data, but this is not addressed here.

²³This arrangement is based on prior agreement between the sender and receiver; in most cases the data are deemed to be incorrect, and the receiver will delete the data. It may be, however, that the receiver system also obtains data transmissions from other sources, and that it may have stored an update already, in which case the receiver can ignore this delete request.

2.8.3 *Data Granularity*

Data are normally transmitted with varying levels of detail. An address may consist of a street, house number, city, ZIP code, and state. Deleting of the address data may occur at a high level, resulting in the deletion of the complete address, or at specific level, for example, the street or house number. The delete indicator can trigger deletion at either of these levels; i.e., when positioned on a high level (e.g., a field in HL7 v2.x), the indicator denotes the deletion of the complete address, whereas, when positioned at a more detailed level (e.g., a component in HL7 v2.x), the indicator denotes deletion of a specific component only.

2.8.4 *Impact of Order of Messages*

If two parties are exchanging information, the result in the receiving system is easy to determine,²⁴ because there is only one sender. If more than one party is sending the same type of information, the result depends on the sequence in which the messages were sent (Fig. 2.18).

The following table provides an interaction scenario with concrete values for a single data element. It is assumed that in System A or System C (both are sending systems in this context) the appropriate change has been applied and the information is sent via the appropriate transaction (1 or 2) immediately afterward. Table 2.1 presents a hypothetical sequence of transactions, data or lack of data sent, and the resulting state of the receiving system. The notation “<empty>” is used to indicate the absence of data in the transaction. The “<delete>” notation represents active communication of the delete indicator.

As one can see, empty information versus an explicit delete indication will have different results. Not sending any data causes the previous/old information to be maintained as is on the receiving side, whereas a processed delete indication will result in deleted data. (In combination with handling outdated data, the systems will be aware of the data history if necessary.)

²⁴Only if each message is a complete “transaction” on its own—which is the case for HL7 but not for DICOM. For example, the sequencing of image transmission (storage), MPPS and Storage Commitment is undefined for an IHE Modality actor, and so is the mapping between transactions and DICOM network associations (TCP connections). Therefore, the result is not always easy to determine, and, of course, with more than two systems, the results are even more complex.

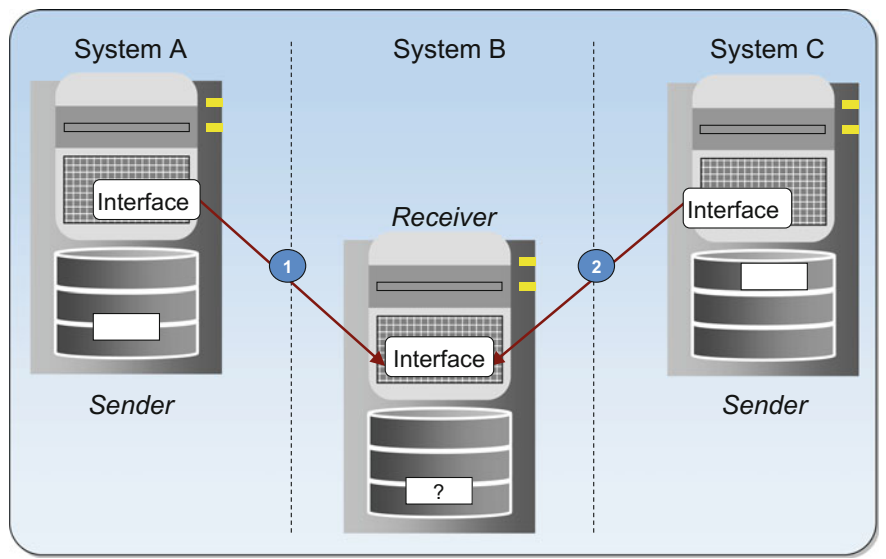


Fig. 2.18 Receiver responsibility

Table 2.1 Order of messages

Seq		Information	Comment	Result in system		
				A	B	C
1	1	"x"		"x"	"x"	<empty>
2	2	"y"		"x"	"y"	"y"
3	1	<empty>	Deleted in System A	<empty>	"y"	"y"
4	2	<empty>	Deleted in System C (without history)	<empty>	"y"	<empty>
5	1	"z"		"z"	"z"	<empty>
6	2	"x"		"z"	"x"	"x"
7	2	<delete>	Deleted in System C with history	"z"	<empty>	<empty>

2.8.5 Sender Responsibility: Impact to System Design

For a receiver, it is relatively easy to take care of the delete indication that is sent on a segment, field, or component level within a message. As indicated before, for the

sending system to detect the need for a deletion is much more complicated, because each field must be checked against a previous value. There are different methods for implementing the delete capability:

- stored procedures to update values
- shadow tables with a history of the data
- special encoding of deleted information.

The interested reader is invited to think about additional means for realizing this requirement.

To summarize briefly, implementers must be aware of and implement the delete indicator, otherwise data fidelity issues likely will occur among the communicating entities. This is especially true if the delete indicator is treated as normal data and is stored as such.

2.9 Null Flavors

An oddly “similar”, but distinctly different, concept to the delete indicator is the so-called null-flavor that indicates why certain information is not available (e.g., “asked but not known”).

HL7 Version 3 provides a very detailed explanation about the different concepts associated with null flavor. Table 2.2 provides an excerpt, and the complete definition can be found in Chap. 5.

Table 2.2 HK7[®] Null Flavors [5 MB 2013]

Concept code	Print name	Level ^a
NI	No information	0
INV	Invalid	1
DER	Derived	2
OTH	Other	2
NINF	Negative infinity	3
PINF	Positive infinity	3
UNC	Un-encoded	2
MSK	Masked	1
NA	Not applicable	1
UNK	Unknown	1
ASKU	Asked but unknown	2
NAV	Temporarily unavailable	3
NASK	Not asked	2
QS	Sufficient quantity	2
TRC	Trace	2

^a“Level” indicates the nesting of the listed codes

Some of the values listed in Table 2.2 are clear indications of why information is not provided (“ASKU”), whereas others are a matter of representation, e.g., negative or positive infinity or quantity measures.

The most common way of providing null-flavors is in the form of explicit values, as is shown in Table 2.3 (“O” and “U”).

If null-flavor code values (e.g., “O”—“Other”) are handled in the same way as individual concepts (e.g., “M”—“Male”), then, typically, within various tables different codes are defined to express the same concept (e.g., if no code matches using the values “O” or “OTH” (“other”), or the information is unknown using the values “U” or “UNK” (“unknown”), etc.). The approach very much depends on the author of the code system and/or other codes in the code system and other dependencies. The three letter code “UNK” is typically selected for “unknown” if the other codes are longer than a single character.

The way null flavors are managed correlates closely with the way the data exchange standard is designed. Introducing a separate set of concepts to be used is permissible only if an appropriate methodology is provided, as can be seen with HL7 version 3 in contrast to HL7 version 2.x.

Providing the null-flavor as a code in each code system (and, therefore, value set) where applicable is—from an implementer’s perspective—considered the easiest method, because the null-flavor code is treated the same as any other code. This method also facilitates easy implementation, because the null-flavors are maintained and updated directly in the given value set. However, this approach does not ensure conformity of the codes across different code systems, so a specific null flavor may be represented by different codes (e.g., “U” or “UNK” for “unknown”).

The described concept of null flavors is a general methodology to convey information about why certain data are missing. This methodology should be available and applicable for all data elements independent from their data type (also including non-coded data types such as String). Therefore, the first method discussed is limited to coded information when a code must be selected from a drop-down list. Introducing a methodology—as in HL7 V3—to provide null-flavors for all data types is a sophisticated approach, but it requires an appropriate enhancement of the application in general. In most cases—if not in all—there is no

Table 2.3 Table 0001—administrative gender^a (HL7 v2.7)

Value	Description	
F	Female	
M	Male	
O	Other	⇐
U	Unknown	⇐
A	Ambiguous	
N	Not applicable	

^aIn HL7 v2, the element is referred to as “Sex” and not “Gender”; however, gender better represents the concept and is used in V3 and FHIR. In this book we will refer to as Gender

means to enter (coded) null-flavor information into a conventional text field, such as fields used for entry of names and addresses. (“I have asked the patient for his name, but he did not tell me.”)

Managing null flavors for all data elements of an application implies a significant architectural enhancement that is hard to realize and therefore not realistic. From an interoperability perspective, changing one application alone is not sufficient, as the example application at the end of this chapter will demonstrate. Therefore, necessary support of null flavors for certain (un-coded) data elements shall be indicated explicitly in the implementation guides in order for a special treatment to be developed individually for the requested data elements.

Chapter 5 explains methods for creating, managing, and applying value sets and code systems effectively. One such method is to combine the null flavor code system with the code system (containing the concept codes needed for the element) into a single value set. This approach is generally, from a maintenance and management perspective, superior, because it ensures conformity of the “null” concepts. Otherwise, it is not possible to detect that “U” and “UNK” are both codes for the same concept.

2.10 Snapshot Mode Versus Update Mode

The most common interaction users have with an application is that of entering new data. As explained previously, a “new data” event may trigger an exchange of data from one system to another. All needed data could be known and entered into the system together, or bits of data may become known and entered over time as they are available. The latter scenario is typical with laboratory test results. For simple lab tests all result data components likely become available simultaneously; however, for panels, often only partial results are available initially, and the remainder of the data for the lab panel test are available later.

In the case of partial data availability, how should these data be transmitted? One option is to repeat the transmission of all the data as soon as they become available, i.e., including data already sent for this event. This method mandates for every known result component to be sent at the time the message is created regardless of when the data became known. This method is called snapshot mode. Another option is to convey in the transmitted message only the data that are new or that have changed. This option is referred to as update mode.

Before continuing with the explanation, it is worth noting that the distinction between these two modes is only necessary in case of multiple values for the same concept, for example, a set of result values providing the details to an order, or a set of diagnoses (e.g., admission or treatment, or primary and secondary). If only a single value is possible, such a distinction is not necessary. Transmitting the current marital status does not require such a determination, but if the history of marital statuses should be sent it becomes important.

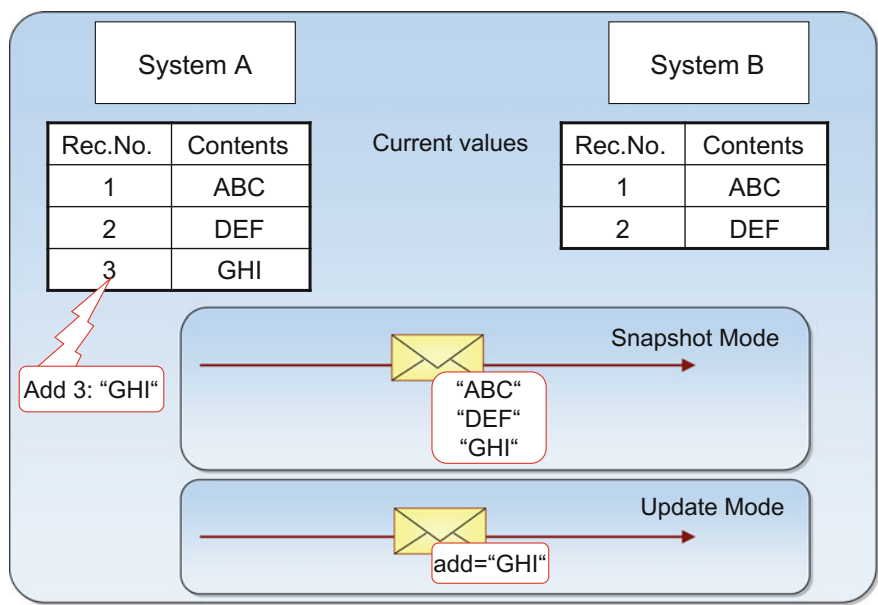


Fig. 2.19 Snapshot mode versus update mode

The decision as to which method is preferable is implementation-dependent and typically is influenced by the particular interaction. A trade-off exists between the ease of processing and the ease of messaging. Figure 2.19 illustrates the contrasting approaches.

Systems A and B have participated in a previous data exchange resulting in two synchronized records 1 (“ABC”) and 2 (“DEF”). Now a user enters data “GHI” as the 3rd record into system A. In snapshot mode, all three records are communicated, and system B replaces the existing data in the records with these new data. In update mode, system A only sends the new data “GHI” with instructions to add it to the already stored set of records. In both modes, system B will have the same data as system A.

Two requirements must be met for the update mode to work. As depicted in Fig. 2.19, the update mode requires an indicator (“add”, in this example) to signal the change. There are a variety of methods for implementing update mode indicators, and action codes are one method. Table 2.4 provides the segment action codes for HL7 v2.7.

The second requirement for using update mode is the capability to identify the data of concern. If the status of a result value should be updated, the correct data entry must be identified. The code or value of the data (e.g., “hematocrit” lab result value) or any of its attributes (e.g., the status “preliminary”) is not sufficient, because these items are subject to change. To provide another example using identifiers: For managing patients or visits, the update mode is the most common

Table 2.4 Table 0206—segment action code (HL7 v2.7)

Value	Description	Explanation
A	Add/insert	The transmitted data are new and should be added as a new record into the system
D	Delete	The referenced data should be deleted. Therefore, an identification of the data is sufficient
U	Update	The transmitted data indicates the updated information, therefore the original data must be uniquely identified
X	No change	This action code just references the data and indicates no change. Such a transmission makes sense in the case where a requirement exists that enforces the presence of data. Therefore, this requirement can be met while optimizing data processing

choice, because only a few attributes (like the address etc.) should be altered. Here, the patient or visit identifier is used, and not the name, address or any other demographic data. Each data item must have a unique and persistent identifier that will not change over time. Inserting new data or using the snapshot mode can be done without any reference to existing data; but, in case of updates or deletes, the referenced information must be clearly identified, which can, of course, not be done based on the data and its attributes. So, even when inserting new data for later updates, an identifier is required.

In principle, both modes allow for adding, updating, and deleting data. The advantages and disadvantages of each mode primarily depend on the frequency of changes or additions to the data, the message size, and the receiver's responsibilities.

If most of the data are entered at once, then snapshot mode is generally the best approach. If a single record contains a lot of details, then the update mode is more suitable. On the other hand, snapshot mode is simpler to implement given that the data referenced in the snapshot are easy to identify.

The most common processing method used with snapshot mode is to delete all the existing stored data and to add the newly received data afterwards. This approach is easier, but it may lead to a loss of relevant data in case the new submission does not cover previous information. The shadow data repository mentioned earlier only supports keeping track of the overall changes. Therefore, for update mode, the burden to identify the changes is on the sender, whereas in snapshot mode the receiver has to identify the delta.

In circumstances where the receiver is collecting data from multiple sources, using snapshot mode is risky, because one sender is unaware of the data another sender has provided. Deleting all data received previously from all senders just because new data are received from a single sender most likely will lead to a loss of important data. Therefore, in these cases the snapshot mode should be implemented to affect only data exchange among each sender/receiver pairing. In other words, in snapshot mode the receiver should only update the data received from this specific sender and leave the remainder of the data from other senders untouched, otherwise, assurance must be provided that all possible senders are aware of the same data.

Laboratory results are a bad example here, because the results are always order specific and, by this means, are clearly indicated. Better examples are procedures, diagnoses, and allergy codes, because multiple systems will collect and communicate these data.

2.11 Considerations in Application Development

The following example is used to discuss how design decisions can impact interoperability. To demonstrate this point, contrived requirements are introduced, resulting in the development of a demonstrative data repository schema (because this is an illustrative example, the scope is limited). The reader can relate this scenario to an application they are familiar with.

The requirements introduced are somewhat arbitrary, but are selected to enforce specific changes in order to exemplify different stages in developing an application.

The example shows how the schema evolves, with new requirements (shown in boxes in italics) being incorporated into design changes, which are analyzed in the Conformance Discussion section following the scenario. The evolution of the schema does not consider any practical impacts that may result in additional problems when deploying this example application or problems that may occur when converting data from an old schema to a new schema. It is intended to be an example to explain and compare the different steps of the evolution (phases) with respect to the data exchange requirements. Additionally, it highlights the impact of changes to interoperability specifications on internal application design. Such changes may not be easy for implementers, and authors of specifications should fully understand the implications when changes are sought; however, this consideration does not justify excluding needed capabilities.

2.11.1 *Introducing Functional Requirements*

We start with a very simple set of requirements:

Functional Requirement 1

The application shall be capable of managing (storing) demographic information about a person. This includes the person's given name, family name, date of birth, and gender.

The application shall be capable of managing (storing) two allergies for this person.

One implementation approach is to create a single (simple) schema without a primary key and using simple data types (see Fig. 2.20).

Fig. 2.20 Sample schema for requirement 1

Persons	
Family_name	char
Given_name	char
Dob	date
Gender	enum
Allergy1	char
Allergy2	char

A second approach is to add the use of a primary key. This approach is not derived from the original requirement, but it does not violate it. Therefore, enforcing an additional requirement is introduced.

Functional Requirement 2

The application shall provide a primary key that is independent from any other information.

Figure 2.21 illustrates the revised data repository schema that includes the primary key Id (identifier). (In the figures that follow, primary keys are marked in bold).

Next, a country-specific requirement is introduced.

Functional Requirement 3

The application shall be capable of managing (storing) information about the birthname in accordance with German law.

To understand this requirement, it must be understood that in Germany the birthname is only referring to the family name. Official registration of a newborn must be completed within 3 days after birth and is permanent (i.e., birthname). Changes to the given name are (normally) not allowed. Figure 2.22 illustrates the data repository schema that includes the birthname.

Fig. 2.21 Sample schema for requirement 2

Persons	
Id	long
Family_name	char
Given_name	char
Dob	date
Gender	enum
Allergy1	char
Allergy2	char

Fig. 2.22 Sample schema for requirement 3

Persons	
Id	long
Family_name	char
Given_name	char
Name_at_birth	char
Dob	date
Gender	enum
Allergy1	char
Allergy2	char

Now, we introduce the United States-specific requirement to manage birthnames (in addition to the German requirement).

Functional Requirement 4

The application shall be capable of managing (storing) information about the birthname in accordance with American law.

Within the United States, it is possible to change the given (first) name within a certain time frame, a rule that varies among the different US jurisdictions. The introduction of this requirement leads to a revision of the data element *Name_at_birth* to *Family_name_at_birth* and *Given_name_at_birth*. Caution must be taken when converting the data. Typically, the old attribute name *Name_at_birth* is changed to *Family_name_at_birth*, and a new attribute *Given_name_at_birth* is added. If the original attribute was used erroneously to capture the full name at birth, this information will reside in the wrong field after converting the data repository schema (Fig. 2.23).

Fig. 2.23 Sample schema for requirement 4

Persons	
Id	long
Family_name	char
Given_name	char
Family_name_at_birth	char
Given_name_at_birth	char
Dob	date
Gender	enum
Allergy1	char
Allergy2	char

Fig. 2.24 Sample schema for requirement 5

Persons	
Id	long
Family_name	char
Given_name	char
Family_name_at_birth	char
Given_name_at_birth	char
Dob	date
Gender	enum
Allergy1	char
Allergy2	char
Allergy1_date	date
Allergy2_date	date

For the next version, a minor new feature is added:

Functional Requirement 5
For both allergies, the associated date of onset shall be managed.

This request results in the schema as shown in Fig. 2.24. In the next step, we introduce some flexibility:

Functional Requirement 6
The application shall be capable of managing (storing) multiple names and identifiers. The management of those data shall be supported by name and identifier types.

This new requirement is included on purpose to show the information being divided between two additional schemas, which are introduced in Fig. 2.25. Some of the original data items are moved to the new schemas, and an additional type attribute is introduced to create the primary key of the sub-schemas. The primary key is used to link those schemas.

As can be seen in Fig. 2.25, both the given and family names for persons are migrated to *PersonNames*. The distinction between the current name and the birth name is indicated by the *Name_Type*. To support the previously introduced requirements, at least two name types must be provided: legal name and birth name.

The external identifier for the person is managed as part of the primary key in table *PersonIdentifiers*.

This schema supports the functional requirement, as multiple names can be used, but its utility is limited to names of different types. To solve this problem, the schema must be enhanced even further, as is shown in Fig. 2.26.

The sequence number as part of the primary key allows for storing multiple names comprised of given and family names of the same type. (Populating this field

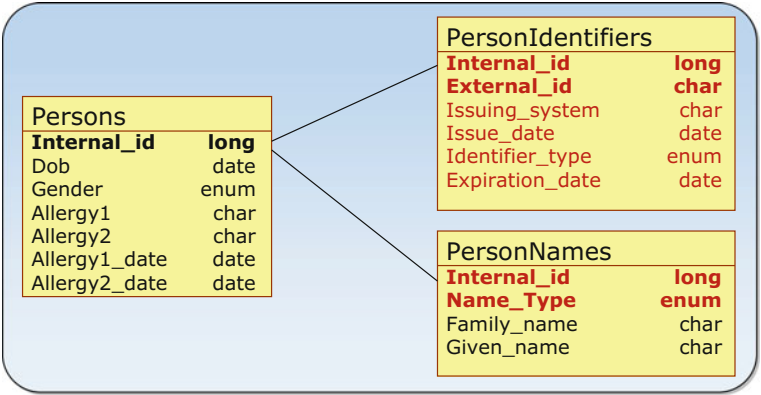


Fig. 2.25 Sample schema for requirement 6

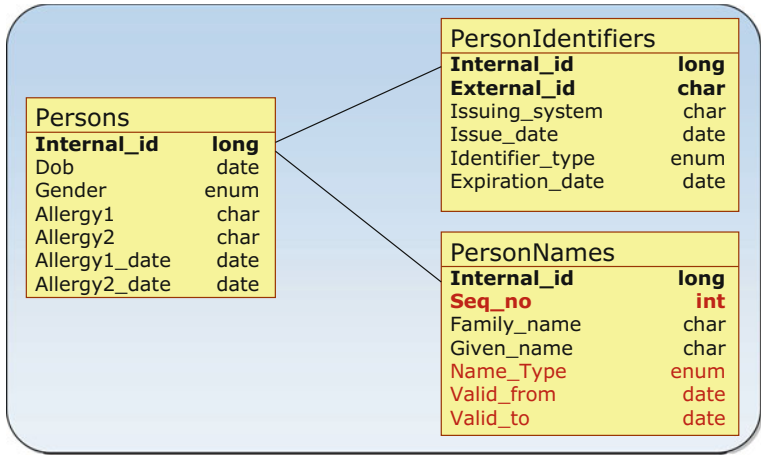


Fig. 2.26 Enhanced sample schema for requirement 6

correctly is delegated to the application.) To help with deciding which name to select, a validity range has been added. This feature is not mentioned in the requirement, but it supports proper use.

The next step should eliminate the restrictions for the maintenance of the allergies:

Functional Requirement 7

The application shall be capable of managing (storing) an unlimited number of allergies.

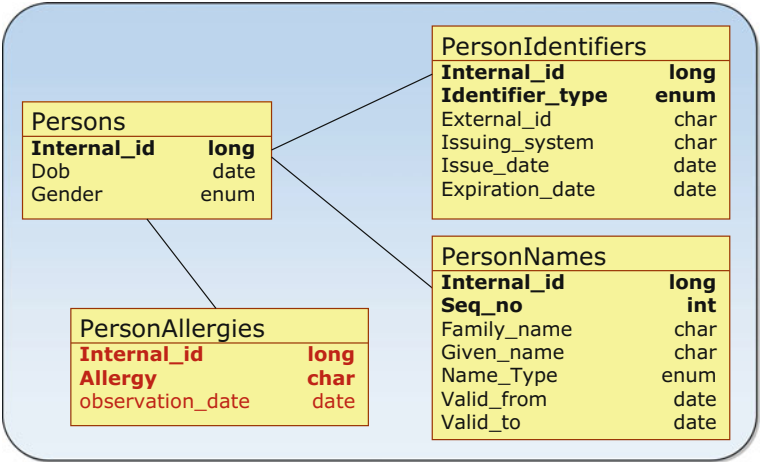


Fig. 2.27 Sample schema for requirement 7

Now the allergy information is migrated to another schema. Assuming that a person will not have the same allergy listed twice, this data element is made part of the primary key. This approach allows for maintaining an up-to-date set of the person’s allergies without including the same allergy twice in that set, which makes sense in this simple use case (Fig. 2.27).

We could continue on with development of example data repository schema, however, the versions presented thus far are sufficient for a thorough discussion of implementation and conformance implications.

2.11.2 Conformance Discussion

In the previous subsection, a simple data repository schema was developed on behalf of the application it supports. The complexity of the resulting schema structure increased with the insertion of each new requirement, reflecting typical changes and stages in application development.

An analysis of the different data repository schemas compared with their ability to fulfill requirements for data exchange is provided next. These requirements are presented in a very simple format. Interoperability aspects are also considered in this analysis. The requirements are examined with respect to the seven example data repository schemas. The check marks simply indicate the fulfillment of the requirement.

Let’s start with a simple requirement:

Data Exchange Requirement 1
The following information shall be provided (exported) on a line-oriented basis in form of a CSV file:

family name
given name
allergy 1
allergy 2

The requirement for the data to be represented as a CSV file means that all data elements must have a one-to-one relationship in order to fit into a single line. Of course, it should be noted that this kind of requirement is bad practice and does not contribute to meeting any future needs that might arise, but it aids here in comparing the different development phases with different kinds of data exchange requirements (Table 2.5).

The last data repository schema—and the most sophisticated one—can cause problems under certain circumstances (for example, if the schema is used when the ability to designate allergy 1 vs. 2 is needed) if the data exchange requirements are kept too simple (select only 2 allergies).

Data Exchange Requirement 2

(2a) A timestamp for marking the export shall be provided.
(2b) The export function shall support re-identification of the person to update previously transmitted information.
(2c) A date for each allergy shall be supported.

Table 2.5 Conformance assessment for (export) requirement 1

Schema	1	2	3	4	5	6	7
Assessment (explanation)	√	√	√	√	√	√	√ This data repository schema allows for an unlimited number of allergies to be listed; therefore, a choice must be made, e.g., to arrange the list starting with the oldest or newest allergies Another enhancement of the data repository schema could be assigning of a sequence number to manage the order of the allergies. This method of organizing these elements can guarantee that the first two entries represent the most important allergies

Table 2.6 Conformance assessment for data exchange requirement 2

Schema	1	2	3	4	5	6	7
(2b) Re-identification of person	Not supported	✓	✓	✓	✓	✓	✓
(2c) Allergy date	Not supported	Not supported	Not supported	Not supported	✓	✓	✓

The first part (2a) of this requirement that deals with the export date can be solved very easily by using the creation date attribute of the filesystem for the export file. However, the most common way (although, maybe not the best way) this requirement is handled is by incorporating this information into the filename. Adding the information to the data itself does not make sense, because it is duplicate information and, therefore, misleading. An alternative approach would be to convey the information by adding meta-data into the file, but the CSV format does not allow this addition without duplicating the information. This alternative works best with other representation formats such as XML.

Table 2.6 provides a summary of application support for the other two requirements.

The next requirement deals with export and import of information.

Data Exchange Requirement 3

The following information shall be provided (in export and import direction) on a line-oriented basis in form of a CSV file:

identifier
family name
given name
allergy
allergy observation date

The allergy and allergy observation date shall be appended to the line for as many occurrences as present in the data set.

The CSV format can be used as long as repeating information is appended at the end of the line. This simple approach allows for appending tuples of data, but its use is strongly discouraged given that more suitable approaches exist. Of course, data with more complex structures, e.g., repeat occurrences of allergies and medications, require additional control information.

The conformance assessment evaluates the third requirement against the data repository schemas for export and import support (Table 2.7).

If the data exchange requirements are expanded, simple applications may not be able to support the requirements satisfactorily.

Table 2.7 Conformance assessment for requirement 3

Schema	1	2	3	4	5	6	7
Export	Identifier not available	No allergy observation data element			√	√	√
Import	No Identification possible, i.e., this information is ignored, possibly leading to duplicate records	No allergy observation data element					√
	Unable to handle more than two allergies						

The step-by-step development of an example application, along with the evaluation of the appropriate data repository schema needed to support data exchange requirements, leads to a very simple conclusion: an application is not always conformant to given requirements, at least not without further organizational directives!

The data repository schema represents the different maturity levels of an application. The complexity of the schema depends on the set of requirements, and usually this complexity is increased over time. Each level added to the schema to support a specific requirement may prove to be too simple/immature, suitable, or too complex/powerful.

2.12 Summary

This chapter explains basic architectural components and elements that are directly related to interfaces and their behavior. All data exchange requirements are closely tied to the underlying architecture of the application.

A key point identified by the authors is that the development of an interface is not independent from the underlying application. All relevant terms, and their implications for and interrelations with an application, have been defined and explained. An example of the steps involved in application development (represented as a data repository schema) is given to illustrate the challenges of establishing an interface while dealing with changing requirements. Furthermore, this example also demonstrates that, without additional guidance and implementation and/or organizational directives, each combination of functional and data exchange requirements may lead to problematic situations.

The primary purpose of an application is to support a user in their workflow. Therefore, an application typically is not designed to match a single data exchange standard. The example presented demonstrates the difficulties of matching a single set of requirements, depending on how these requirements are formulated. Having an application match a set of distinct interoperability specifications is unlikely for a variety of reasons including

The application has to meet other requirements beyond interoperability, and they may conflict with the interoperability requirements.

There is no guarantee that multiple interoperability specs won't conflict with each other, thus making it impossible for an application to be structured in accordance with both.

Legacy data within live instances of the application may make it difficult to change the underlying structure.

Interoperability standards developers must understand the application needs of the systems that are likely to implement the standards and then work to develop standards that are compatible with these needs. Vendors should support this process by providing information about their applications' functionality to the standards developers. This collaboration will help ensure that the applications will be more in line with the end user needs relative to using the data while also supporting interoperable data exchange. Standards developers must keep in mind, however, that, while this collaboration is needed, new and evolving standards must drive the industry toward the desired state for health IT interoperability and not merely document requirements based on applications' existing capabilities.

References

1. Blobel B, Holena M: *Comparing middleware concepts for advanced healthcare system architectures*. International Journal of Medical Informatics 1997;46:69-85.
2. Blobel B, Holena M: *Comparison, Evaluation, and Possible Harmonisation of the HL7, DHE, and CORBA Middleware*. In: Dudeck J, Blobel B, Lordieck W, Bürkle T (Edrs.): *New Technologies in Hospital Information Systems*, pp. 40-47. Series Studies in Health Technology and Informatics, Vol. 45. IOS Press, Amsterdam; 1997.
3. Blobel B: *Analysis, Design and Implementation of Secure and Interoperable Distributed Health Information Systems*. Series Studies in Health Technology and Informatics, Vol. 89. IOS Press, Amsterdam; 2002, pages 12-31.
4. Bundesverband der Krankenhaus-IT-Leiterinnen/Leiter e.V., <http://www.kh-it.de>, last accessed August, 13 2015.
5. HL7 V3 Product Suite, http://www.hl7.org/implement/standards/product_brief.cfm?product_id=186, last accessed December 14, 2015.
6. HL7 V2 Product Suite, http://www.hl7.org/implement/standards/product_brief.cfm?product_id=185, last accessed December 14, 2015.

Healthcare Interoperability Standards Compliance
Handbook
Conformance and Testing of Healthcare Data Exchange
Standards
Oemig, F.; Snelick, R.
2016, XLVI, 662 p. 292 illus., 289 illus. in color.,
Hardcover
ISBN: 978-3-319-44837-4