

ClientCard: flexible record-based data storage and communication

Josh Carter

September 11, 1998

1 Overview

ClientCard was designed from the ground up to be a universal and flexible data container which automatically communicates with a back-end database. The need for this class spans a broad range of applications, especially in the vertical market space which DataRover targets. ClientCard provides a stable, extensible, and easy-to-use code base for these applications to be built on.

This document explains the design and implementation of the `ClientCard` class. This class holds information about one *record* of data, for example a medications record for a patient in a health care application. Multiple cards can also be linked together to form an indivisible *metarecord*, such as a health care visit note, which contains many sections. Much of this data storage ability is directly inherited from Magic Cap's `Card` class.

Most significantly, ClientCard adds the ability to assemble all of a card's data and send it to a remote server. This is accomplished by collecting all the data fields on the card, querying each for its contents and what it represents, and then sending this to a network stream. ClientCard also handles the reverse operation: creating a record with data received from a remote server.

2 Data organization and storage

This section discusses how data is stored in a ClientCard, and how collections of ClientCards can act together to represent a larger metarecord. I assume moderately intimate knowledge of the Magic Cap `Card` and `Form` classes; this is important since ClientCard leverages these classes heavily.

2.1 Data storage with Cards and Forms

ClientCard inherits directly from `Card`, and uses the existing Card/Form data storage mechanism for storing data. As with the traditional mechanism, most viewable objects “on the card” are really stored on the shared `Form` object instead. As the card is shown to the user, a temporary copy of the form is

installed, and this form is destroyed when the card is hidden. During this process, the contents of every data element on the form (*form element*) is packed and unpacked from the card's extra data. As such, the only data stored in each card instance is the *content* of the form elements. The form elements themselves, being on the Magic Cap form, are shared.

Each specific type of record, for example a medications record, will have its own unique form which contains various form elements arranged in a particular manner. The name of each form (e.g. "*medication*") is significant, since it describes the overall nature of the form. This name is easily accessed via ClientCard's `CardType()` attribute.

The decision to base ClientCard on Magic Cap's Card/Form mechanism was fairly straightforward: the mechanism is an efficient data container which automatically packs and unpacks data as needed. Alternate implementations would be challenged to rival the already established, efficient, and well tested mechanism. I did not see a compelling enough reason to reinvent this particular wheel.

2.2 Collections of ClientCards forming a metarecord

Cards have a closely linked relationship between how data is stored and how data is viewed by the user. Each card stores data for itself, and this data is seen onscreen as one viewable card. While tricks can be played within this limitation, e.g. the `ModalClientCard` class, it does not scale well to representing larger records. There are situations in which a record will be very large, requiring many onscreen cards to represent it.

ClientCard handles the large record problem by allowing multiple cards to be linked together to form a *metarecord*. When communicating with a remote server, each metarecord is processed as a single record, but internally the ClientCard class is processing several card instances. This design permits greater flexibility in data representation than enforcing one-card-per-record, and also opens the possibility of partial record updates when communicating with a server; since each card in the metarecord has its own unique `CardType` identifier, a future upgrade could easily handle updating only parts of a metarecord instead of processing the entire thing.

Each metarecord has a *top-level* card which acts as the master card for the entire collection. If a reference is stored to the metarecord, it should point at this top-level card. This card, in turn, contains the information to access any other *subcards*. The subcards are stored in a Magic Cap `StackOfCards`, and the subcards for any particular card are accessible via the ClientCard's `Subcards()` attribute. The subcard hierarchy can have several levels, forming a tree of cards. No card, however, should be referenced in multiple subcard lists, nor should other types of graphs be formed. This limitation could be removed for future revisions without much trouble, but it has not yet been necessary to do so. Arbitrary graphs of cards could be useful for creating an adventure game, however: *you are in a twisty maze of cards, all alike...*

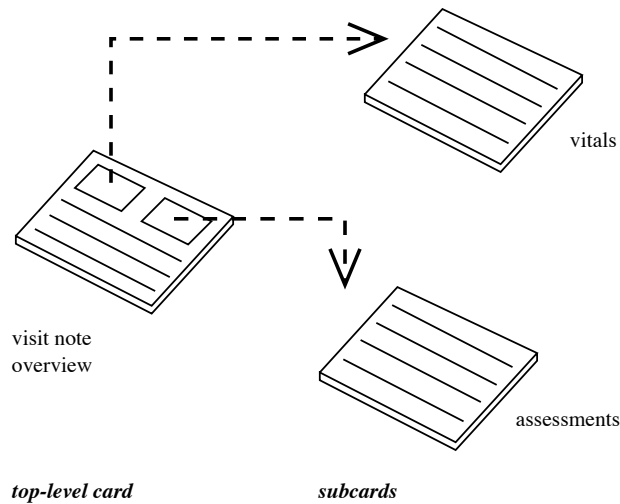


Figure 1: one metarecord composed of three cards

In general, application code will not need to crawl the subcard tree manually. Several utility methods facilitate access to various parts of the metarecord. First, any subcard can be found by its card type using `FindSubcard(type)` called on the top-level card. Second, the entire tree of cards can be iterated over using `EachClientCard()`. Searching and iteration can be limited to specific parts of the card tree by calling these methods on nodes further down the tree; only that node and its descendants will be processed.

3 Data serialization and communications

While ClientCard's data storage and organization capabilities are useful in themselves, ClientCard's real power lies in its communications ability. Data residing on a DataRover is not very useful unless it can be transmitted back to a database, and data sitting in a remote database isn't very useful to a worker in the field unless they can download it to the DataRover. Solving this problem is ClientCard's specialty.

In a most general sense, ClientCard knows how send and receive data through a stream. Each card is identified by its `CardType()` attribute, and the server uses this attribute to associate individual cards with its own tables. The exact association is decided by the server and does not necessarily reflect database table organization. This extra level of indirection allows the DataRover client and the remote server to use completely different data organization schemes; whatever best suits the application.

3.1 Identifying data fields with ClientElement

Each form element on a ClientCard which is intended for communication must be tagged with a *key* specifying what it represents. It must also have a consistent interface for getting the contents of the element, formatted in a manner understandable by the database. These two attributes, `ElementKey()` and `ElementValue()`, are defined by a mixin class named `ClientElement`. Many subclasses implementing the `ClientElement` interface exist, including text fields, date pickers, choice boxes, and sliders. `ClientCards` can be built with a mixture of `ClientElements` and other types of elements on the card's form, but only the `ClientElements` will be transferred to and from the remote server.

The key for each `ClientElement` is typically a text object stored in a field. Given that the elements reside on the form, they and their keys are all shared, minimizing memory expense. The values, on the other hand, are typically assembled from the contents of the element. For a text field, the value is simply the field itself. For a slider, it is a text representation of the slider's level.

3.2 Collecting and searching ClientElements

All the client elements for a given `ClientCard` and its subcards can be assembled into an `ObjectList` using the `ClientElements()` read-only attribute. This list can be searched by key using the `FindByElementKey(keyName)` intrinsic method. Before transferring a card, `ClientCard` will get the element list and reuse it as needed during the transfer, since collecting the elements can be a moderately expensive operation with very large metarecords.

Before the data on the card can be read or modified, the Magic Cap form needs to be installed on it. Installing the form allows access to the card's data on an element-by-element basis, which is essential to the above methods. Given that elements can span many cards, `ClientCard` adds a feature for installing and removing the forms on all cards in a metarecord: `SetFormInUse(true)` called on the top-level card will install all forms, and `SetFormInUse(false)` will restore them to their original state.

Important: the `ClientElements()` attribute is only useful when cards have their form installed. Be sure that `SetFormInUse(true)` has been called before calling `ClientElements()`.

3.3 Preparing to transfer cards

`ClientCard` does not specify the transport mechanism for its cards, other than a stream implementing the Magic Cap `Stream` interface. Cards can be transferred over streams ranging from an ethernet link to a serial cable, greatly enhancing its usability in various customer situations. A reliable data link such as TCP/IP is strongly recommended, however, since `ClientCard` does not implement an error detection/recovery scheme; it is assumed that any modern application would be using a reliable stream mechanism, and additional redundancy is unnecessary.

Also worth noting, transfers do not necessarily need to use a network. Cards can be transferred through memory (via `MemoryStream`) or a file system just as easily, opening a wide range of options for an application to exploit. For example, huge numbers of cards could be serialized to an ATA flash card, and read back on demand to implement an object-based virtual memory system. Cards could also be transferred into memory streams and post-compressed, allowing many more cards to be stored in main memory.

3.4 Sending a ClientCard

Once the application has established a stream to send the card over, it simply needs to call `SendCard()` on the card, and pass in the stream. `SendCard()`, in turn, calls `StartSending()`, `SendElements()`, and `FinishSending()` on the card. A running number of bytes transferred is also tallied and passed into `FinishSending()`, since it might be useful information. `SendElements()` is the most interesting method: it collects all the elements to send and calls `SendOneElement()` on each one. These methods are arranged to provide subclasses with several different override points, since they will almost certainly need to send some protocol-specific information.

`ClientCard`'s implementation of `StartSending()` and `FinishSending()` are short but essential: they ensure the card's form is installed and removed properly. As such, any override of `StartSending()` should call the inherited method *before* anything else, and any override of `FinishSending()` should call the inherited method *after* everything else.

The concept of `SendElements()` is simple, and so is its implementation. It gets the `ClientElements` list, and iterates over the contents calling `SendOneElement()` on each. This method will also put separator characters (or strings) between each element as it is sent. Subclasses should override `GetFieldSeparator()` to use a different separator.

`SendOneElement()` is responsible for sending the key value of an element, the key/value separator, and then sending the value. If the value needs to be encoded, for example to escape troublesome characters, `SendOneElement()` will do so. Subclasses may also override `GetKeyValueSeparator()` to use their own separator string.

3.5 Receiving a ClientCard

Building a new card from a data stream is more complex than sending a card, but `ClientCard` is designed to make the process as painless as possible for the application developer. There are a few issues which must be addressed: deciding what type of card to build from the data, and then finding a place to put the newly created card. The application will need to provide this logic.

When the application has an incoming card ready in the stream, it should call `ReceiveNewCard()`, a `ClientCard` class operation, on the appropriate subclass used by the application. This operation will in turn call `NewClientCard()`

to create a new card of the appropriate type for the incoming data (a sometimes nontrivial problem, which will be addressed shortly). After the card is created, the process of receiving the card is very similar to sending a card: `StartReceiving()` is called to prepare for receiving, `ReceiveElements()` sticks the data in the right places, and `FinishReceiving()` is called to complete the process.

3.5.1 Buffering and processing incoming data

Throughout the receiving process, a data buffer is passed among the methods to allow efficient reading from the data stream. This secondary buffer is necessary since data cannot be easily “unread” from a stream, and it is impossible to read variable-length data without the threat of overreading except by making single-character reads. Single-character reads are horribly inefficient, and greatly complicate the parsing of incoming data. The data buffer used by `ClientCard`, on the other hand, allows large block reads and easy parsing by the client code.

The data buffer is a C++ class called `NiftyString`. This string class is designed to tokenize data very quickly with no memory overhead. Its methods can break apart lines with any convention of ending, tokenize by separator characters or arbitrary-length strings, and do a few other flips and tricks. This class has been used to implement several application protocols already with very good results.

`ClientCard` uses its `FetchMoreData()` intrinsic method to fill the data buffer. This method will read any pending data from the stream into the buffer, up to the maximum amount the buffer can hold. Any pending data in the buffer will be retained, and the new data is appended to the end. Note, however, that any pointers returned from previous tokenizing methods will be invalidated, because the data buffer is compacted in `FetchMoreData()`.

`FetchMoreData()` should be called before the first access to the data buffer, and should be called periodically afterwards when the buffer runs low. Typically, `NiftyString::HasLine()` and `HasToken()` are used to check if another chunk of data is available, and if not, `FetchMoreData()` is called. `FetchMoreData()` will return false when no more data is available.

3.5.2 Creating the new card

`NewClientCard()` presents an interesting challenge: Applications will typically have several types of cards – medications, visit notes, etc. – so when a new card is waiting in our network pipe, someone needs to decide which kind of card it is. `ClientCard` itself doesn’t know about the specific protocol being used to transfer the card, so its `NewClientCard()` operation is empty. This must be overridden by a subclass.

Most reasonable protocols will specify what they are sending before they send it. This behavior is essential for implementing `NewClientCard()`. Most implementations will be able to figure out the type of incoming data from exam-

ining a header prepended to the data. Once the record type is determined, the application will typically search a list of prototype cards for a match (one prototype for each type of card), and clone the matching card. If a match cannot be found, `NewClientCard()` should return `nilObject`, and the card receiving process will stop. Applications may want to implement some type of other fallback process for this situation.

3.5.3 Receiving the data

The default implementation of `StartReceiving()` and `FinishReceiving()` do the same job as `StartSending()` and `FinishSending()`: they install and remove the form for the card and any subcards. The same rules for calling the inherited methods from an override hold.

`ReceiveElements()` breaks apart separate elements and then breaks apart the key and value in each element. The key/value pair is then passed into `ReceiveOneElement()` for processing. `ClientCard`'s implementation will search the list of client elements for a matching key and then call the matched element's `ElementValue()` attribute setter. While this logic is usually appropriate, subclasses may override the method for special cases.

3.5.4 Putting the card somewhere

`ClientCard` has no knowledge of how the application organizes its cards, and where it might want to put the new one, so it doesn't do anything with the card after `FinishSending()`. The application should either get the reference returned from `ReceiveNewCard()`, or override one of the receive methods to store the card somewhere. The former case is probably easiest to handle, but may vary depending on the application. For example, the required information may be contained in the data header processed by `NewClientCard()`, so `NewClientCard()` could file a reference to the new card before returning.

4 Future directions

`ClientCard` already provides a great deal of functionality which is immediately useful to a wide variety of applications, but like so many things it has room to grow. The most significant addition to `ClientCard` would be an efficient searching mechanism. A more "blue sky" feature would be data compaction of cards in memory which still retains searching capabilities.

The existing `ClientCard` class does not have a very fast mechanism for examining card data due to its use of Magic Cap's Card/Form mechanism, and this packing/unpacking system precludes quick searching of cards. Improving this situation does not necessarily mean rearranging the data format of the card; the card could instead work with the elements in the form item list to quickly decode its data for searching purposes. I estimate this mechanism would be fairly easy to build, run at very acceptable speeds, and would not incur extra memory overhead for stored cards.

The searchable compressed card is a more novel idea, and might warrant consideration. Previously, General Magic paid a company (whose name I don't recall offhand) to develop this exact feature, and they did it admirably. The company's scheme compressed cards *extremely* well, and they could search content very quickly without decompressing the entire card. It was a remarkable piece of technology, and I believe it would be in General Magic's best interests to recover it for use in our applications.

While the latter improvement would be a potentially significant undertaking which may not be required for most applications, the first improvement (adding fast searching) will probably be very useful, and should be implemented.