ProtoGen/HL7 User Manual

Version 1.0

Gunther Schadow

1 What is ProtoGen/HL7?

ProtoGen/HL7 is a C++ application programming interface (API) to the HL7 standard for the communication of clinical data. In the terminology of C++ ProtoGen/HL7 is a class library that defines data according to the HL7 data model plus methods that provide various access to the data structure. However, ProtoGen/HL7 is not just the class library, it is also a compiler that builds this library from the textual document which is the official HL7 standard. The latter is, what the Name ProtoGen/HL7 wants to express: it is a **Proto**col implementation **Gen**erator for HL7.

An implementation of a communication standard is essentially a data structure, a builder and a parser. The data structure holds the data objects and reflects their relations among each other as assumed by the standard. The builder is to produce a valid representation of the data structure that is capable of being transported via electronic data interchange media. Finally the parser transforms this representation back into the data structure.

This implementation tries to assume models and methods of todays informatics technology. It's design is object oriented and is implementeted using C++. The code of the implementation is produced by a compiler whose input is a database which specifies the standard. The database is in turn compiled semi-automatically by scanning the text of the HL7 standard as is released by the HL7 Working Group. This method of generating the implementation directly from the textual description has the following advantages: correctness and customizability of both, the parser/builder methods and the definition of the standard itself.

Since the specifying database as well as the program code which implements the standard is not written manually but produced by a compiler, the output is assured to be correct, unless the compiler itself is incorrect. An incorrect compiler, however, will result in erroneous code but the errors tend to occur systematically rather than accidentially and are thus discovered more easily. Thus the method of ProtoGen/HL7 can be used to generate a HL7 reference implementation. On the other hand, it does reveal problems and errors in the standard definition itself, since a compiler will not guess the intended meaning of an ambiguous or obviously incorrect passage as a human programmer would probably do.

The input database for the ProtoGen/HL7 compiler is customizable and allows the user to add new data elements quite easily. For example, a special Z-segment that is used by a site can be added to the standard just by adding a few line to the database. The implementation for this special segment is generated fully automatically as is the implementation of the whole standard.

On the other hand, the methods of the parser/builder may be customized in order to provide for different encoding rules or to speed up the implementation or tune it for size. These customizations are, however, not done very often and can currently only be done with a deeper understanding of the compiler itself since this requires the compiler program to be modified. Nevertheless, there are various applications that can be worth the effort ranging from modifications to HL7 coding or processing rules up to the implementation of a completely different standard. Since the UN EDIFACT encoding rules are quite similar to those of HL7 there could be an EDIFACT module added to ProtoGen with only a little to moderate effort.

Finally, the object oriented model of the HL7 standard that this implementation assumes helps to stay compatible with the new developement in the HL7 Working Group (version 3.0) in particular as well as health care data interchange standards of the future in general, such as MEDIX.

1.1 The paragons of ProtoGen

The idea of ProtoGen is not only applicalble to HL7 but also to similar standards like EDIFACT. In fact, the idea to automatically build data structures and methods from a formal specification of the protocol is not new. In the OSI world, ASN.1 is the language in which protocols are described. Many different ASN.1 compilers exist today by which ASN.1 descriptions can be translated into C or C++ code. What might be new about ProtoGen is that it tries to compile a standard document which was meant to be read by humans rather than by computers.

Unfortunately HL7 does not comply very much to the models that the OSI proposes, even though the "7" in "HL7" is derived from the seventh layer of the OSI protocol stack, beside of this, HL7 has little in common with OSI. That is why it is not as easy as it could be to apply the methods and tools that exist in the OSI world to HL7. If we had a ASN.1 description of HL7 that could be produced by a compiler like ProtoGen we would not gain much. Most applications that use HL7 do not only use the data model that in fact belongs to the seventh layer. They also use the HL7 encoding rules which actually belong to the sixth layer. It is the problem that layer 7 and layer 6 can not be easily separated in HL7. There is on the other hand no ASN.1 compiler that would compile code for the HL7 encoding rules. From OSI's point of view there is nothing wrong with it, since the HL7 encoding rules are not qualified for encoding everything that can be expressed with ASN.1. ProtoGen was made to provide for HL7 what is already available in the OSI domain.

An other paragon for ProtoGen is rpcgen(1), a compiler for the remote procedure call protocol (RPC). ASN.1 compilers, rpcgen(1) and ProtoGen have in common that they read a specification of a protocol and transform this input into one or more files which can be used with common programming languages like C or C++. The produced files usually include one or more header (`*.h`) files, which contain definitions for the data structures and one or more implementation (`*.c`) files which contain the function definitions for encoding and decoding the data structure.

The result of such a compilation is an application programming interface to the protocol that was specified by the compiler input.

1.2 The ProtoGen compiler

Even though ProtoGen/HL7 will use the human readable standard document as it's very first input, a formal specification is produced as an intermediate step. The generation of the formal specification is quite complicated and rarely done more than once. This process is described in section "From Text to the Database" in *ProtoGen/HL7 – An implementation of HL7*.

The ProtoGen compiler is thus split into two parts: the first part reads the text and produces the formal specification, while the second part reads the formal specification and produces the application programming interface. Unfortunately the HL7 standard document is maintained with an inappropriate Word Processor and each revision tends to have a different layout. The first part of the ProtoGen compiler does have to change heavily for any new release of HL7. It seems like there will eventually be a more formal authoritative specification of HL7. It is for this reason that the core of ProtoGen is the second part of the compiler, the one that already takes the formal specification as it's input. The compiler program that comes with this version of ProtoGen/HL7 is invoked by the name '`protogen`'.

The '`protogen`' program is currently written in Prolog. The input file to '`protogen`' is also a Prolog source. You will need a Prolog implementation that can run protogen, if you need to remake the application programming interface. The '`protogen`' program is developed to run with the SWI-Prolog system written by Jan Wielemaker at the University of Amsterdam's Department of Social Science and Informatics (SWI). SWI-Prolog is a sophisticated and powerful Prolog implementation in the Edinburgh tradition, runs on many UNIX systems, and it is free software. You can get it from any good FTP server. Please use "archie" to locate a server next to you.

1.3 Refer to HL7 data elements by their names

Since ProtoGen/HL7 implements HL7, you need to know about HL7 in order to successfully work with it. On the other hand, once you know HL7 and you know the basics of how ProtoGen/HL7 works, you won't need much more than your HL7 standard document for reference. Most often you will have to refer to the HL7 document in order to see the data elements that you need for your purpose, their name, and how they relate to each other. What you do *not* need to know is the order of the data elements in a sequence, since with ProtoGen/HL7 you refer to any data element by it's name, i.e. a C++ identifier. This is very convenient and reduces error frequency.

HL7 defines standard names for its data elements. Since a HL7 data element name may consists of multiple words, protogen reduces the identifier length. This is done for your convenience as a programmer, since you do not have to type 20 or more characters for a single name, but only about 5 to 10 characters, and the identifiers are still quite expressive and readable.

2 How to build HL7 applications with ProtoGen/HL7

The object oriented programming technique as provided by C++ allows a very natural view on the HL7 data items. Any HL7 data element is a separate class in ProtoGen/HL7. Individual data types, segments or messages are special classes, which share common properties. We define the properties that are common to all HL7 classes as the class *HL7Object*. HL7Object is an abstract base class which still contains some methods and data instances including the flag that shows if an object is present or not.

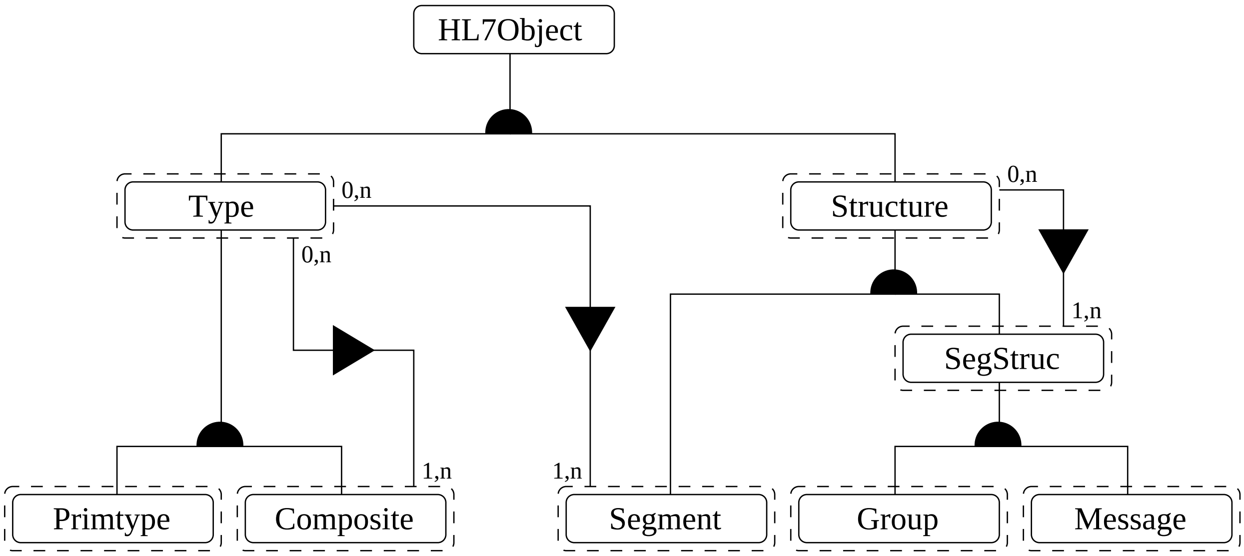

Figure 1: The hierarchy of abstract HL7 classes

There is a hierarchy of classes as shown in figure 1. Basically there is the class “HL7Object”, which defines properties that all HL7 classes have in common. The class “Type” includes the basic types (class “Basetype”) as well as the composite types (class “Composite”). All types have in common that they may assume the null value. On the other hand there are the Structure elements which include segment, messages and groups. Groups are groups of segments that may occur repeatedly in one Message. Structure objects may not assume the null value.

2.1 Naming Scheme

ProtoGen has a naming scheme that tries to keep names descriptive, readable and short.

Classes for data types and segments are given the name of their two or three characters id in upper case, with the small qualifier '`typ`' and '`seg`' directly attached to it. Thus '`NMtyp`' is the class of numerics and '`MSHseg`' is the class of message header segments. Message classes are named a similar way by attaching '`msg`' to the uppercase letters which are taken either from the list of message type ids or from the list of event type codes[1]. Note that we try to avoid using the underscore character '`_`', which tends to merely lengthen the name, readability can be acheived by altering uppercase and lowercase letters as well.

Instantiations of classes, i.e. variables as well as symbolic names of coded values are given a name which is derived from their description description. There is a simple function which produces valid C names from arbitrary text strings. The function is given three arguments. the text string, the *threshold length* and the *truncate length*. All words in the string are concatenated with each first letter in upper case and all other letters in lower case. The words are truncated to the truncate length if they are longer than the threshold length. Typically the threshold length is greater than the truncate length, thus allowing short words to be completed while longer words to be truncated to a short length. For example for the standard values threshold 5 and the truncate length 3, a description like "Patient Visit - Additional Info." becomes to '`PatVisitAddInfo`'.

This produces names which are sufficiently unique in most cases. However there are ambiguities in table values, which require a refinement of the name assembling algorithm. These ambiguities occur, when there are single words which begin with a common prefix like the latin preposition "intra" or the greek quantifier "milli". Thus "intravenous" and "intradermal" both become '`Int`'. This can be overcome by splitting such composita into two seperate words ("intra venous" giving '`IntraVen`').

2.2 The HL7 base class

Properties which are common to all HL7 objects are inherited from the class HL7Object. The HL7Object contains abstract functions that must be provided by any class that can have instances. In addition, the class HL7Object knows about the type of object of the inheriting class. The "type of object" is expressed as a pair. First there is the *subclass* that tells the abstract class that an object belongs to. For example, the ACK message has the subclass "message". The *type* tells what exact type an object is, i.e. whether it is an NM data type or an MSH segment, etc.

`HL7Object::subclass_t HL7Object::subclass() const`

The function telling the subclass of an object. The type "HL7Object::subclass_t" is an enumeration and has the following possible values:

[1] actually we merge these two lists into one

`HL7Object::primtype`
: object is a basic data type

`HL7Object::composite`
: object is a composite data type

`HL7Object::repfld`
: object is a repeated data type

`HL7Object::segment`
: object is a segment

`HL7Object::group`
: object is a group of segments

`HL7Object::repstrc`
: object is a repeated segment or group

`HL7Object::anyseg`
: object is a segment that can be of various types

`HL7Object::message`
: object is a message

`HL7Object::delimiters`
: object is the set of delimiters

`int HL7Object::type() const`

Returns the type of the object. The int can be transformed to a coded value using the appropriate code for the subclass.

`bool HL7Object::ispresent() const`

This is a predicate that tells if an object is present or not. A newly created object and it's parts are generally not present. An object that is not present can be present only if some value is written into it.

`result HL7Object::unset()`

Make an object and it's part to be not present. The result is always `SUCCESS`. Actually the unset() function should be declared `void`.[2]

`result input(istream&)`

`result output(ostream&) const`

These functions are declared as pure virtual, since any class that inherits HL7Object must define an input and an output function. These functions take iostreams as parameter which must be prepared for the use of HL7 (s. section 2.6 [Streams], p. 9).

---

[2] This will be done when the code is revised for the use of C++ exceptions

Finally there are the operators `>>` and `<<` defined, which input or output a HL7Object respectively. These are declared as friend functions to the HL7Object and don't do much more than calling the `input()` or `output()` function respectively.

2.3 Repetition of Types and Structures

Repetition is modeled by the class templates `repfield` and `repstruc`, depending on whether the repeated class is a `Type` or a `Structure`. Each repeatition is numbered from zero onwards. You can access the repeatition by the array reference operator `[]`. This returns a reference to the repeated object, thus you can read and modify the selected object. There is no upper limit for the number repetitions. The following example shows how you would print successive "COMMENT" fields from an NTE segment:

```
NTEseg nte;
repfield<TXtyp> rcom = nte.getCom();
int i;

for(i = 0; rcom[i].ispresent(); i++)
  {
    cout << (char *)rcom[i];
  }
```

2.4 Optionality

Since the HL7 specification demands a tolerance in respect to missing or unexpected objects, we do not care much about whether a segment is marked as "required" in the specifications. It is rather up to the application to reject incomplete messages, while unexpected objects should just be ignored.

2.5 Methods that select parts of a HL7 class

`const` *component type*`&` `get`*component*`()`

> Selector(s) or extractor(s) one for each component of public relevance. Extractors are canonically named `get`*component*`()`, where *component* is the name of the component that the extractor will access. The component is returned by constant reference as the value of the function. Thus you can use the `get`*component*`()` function in any term except you may not modify the returned object.

`result set`*component*`(`*component type*`&)`

Modifier(s) one for each component of public relevance. Modifiers always set an object to be present.

If you want to modify a component of an object, you have to `get`*component*`()` the object first, assigning it to a variable. Then modify this variable and finally write the variable back to the outer object with `set`*component*`()`. It would be not correct to allow modifying access to parts of an object directly, because this would violate encapsulation of data. On the other hand it results in quite ugly code. Suppose you want to set the first name of a patient in an ADT message. Instead of writing something like this:

```
ACKmsg ack;

ack.PatIde.PatName.FirstName = "Peter";
```

You must break the assignment term down into the following sequence:

```
ACKmsg ack;
PIDseg pid = ack.getPatIde();
PNtyp   pn = pid.getPatName();

        pn.setFirstName("Peter");
       pid.setPatName(pn);
       ack.setPatIde(pid);
```

2.6 Send and receive messages with iostreams

Any HL7 application will at some point receive a HL7 message from somewhere, at an other point a message is sent to somewhere. Sending or receiving of messages with ProtoGen/HL7 means reading and writing them to and from a stream. The stream may be linked to a file or a socket, ProtoGen/HL7 does not care about this. Since C++ iostream stream abstraction is used, you can send and receive HL7 messages like you would read or write any other I/O in C++ using iostreams. For example:

```
istream is;
ostream os;
ACKmsg ack;

is >> ack;

os << ack;
```

This reads an acknowledgement (ACK) message from the stream `is` and outputs it on an other stream `os`.

One would expect a HL7 stream to come as three classes: class *hl7istream* derived from class istream, class *hl7ostream* derived from class ostream, and class *hl7iostream* derived from class hl7istream and class hl7ostream. ProtoGen/HL7 will provide this in future releases, however until then you have to take a slight inconvenience: Any iostream that is dedicated to transport HL7 messages must be prepared to do so. This is done according to the following example:

```
iostream hl7ios;
xios hl7xios(hl7ios);
```

An object named `hl7xios` is declared to be of class `xios` and initialized with an ordinary iostream object. The `xios` object is not used after it is constructed. Any HL7 communication is sent via the `iostream` object after it is thus prepared. This is certainly a little confusing but will be cleaned up in a future release of ProtoGen/HL7.

An `iostream` that is prepared to transport HL7 messages can be in one of two modes that decides how HL7 messages are to be encoded: *hl7er* or *debug*. If a HL7 message shall be sent or received, the stream must be in the "hl7er" mode, which means that the HL7 objects are encoded using the HL7 encoding rules. The "debug" mode is used for a structured output that can be used in log files. The debug output can be configured in xios.h during compile time of the HL7 class library. It is not possible to read back a message that has been written using the debug mode.

The modes are set using the manipulators `hl7er` and `debug`. A freshly prepared HL7 stream is always in "hl7er" mode.