

# A Framework for Process Automation Systems

## The Telegram Pattern

Wolfgang Narzt  
Johannes Kepler University Linz  
Altenbergerstr. 69  
A-4040 Linz, Austria  
Wolfgang.Narzt@soft.uni-linz.ac.at

### ABSTRACT

Frameworks are considered as a fundamental base for building powerful applications in object-oriented systems. In connection with design patterns, they provide an architectural guide for frequently appearing problems. Although, frameworks and design patterns can be regarded as state of the art in modern Software Engineering [5], they have not made their entrance to the industrial market, yet. Many process automation systems [2] are still developed with traditional programming languages like Fortran or C, which restrict the reuse of software to a low level, whereas object-oriented techniques allow to reuse the whole logic of a system.

This paper presents a part of a framework for process automation systems [4], which has been developed in the course of an EU-project. It deals with the difficulties in telegram-based communication when telegrams (data streams) are transferred between different platforms and converted to the data format of the target system. The problem is modeled by a new design pattern - the *Telegram Pattern*.

The *Telegram Pattern* has been implemented with C++ and the CORBA [3] implementation Orbix. Its applicability within the framework has been validated by tests under real-time conditions at a hot rolling mill in Austria. It fulfills the requirements efficiency, flexibility, adaptability, reuse and portability and can henceforth be considered as a platform for the development of future process automation software.

### Categories and Subject Descriptors

D.1.5 [Software]: Programming Techniques—*obj.-oriented Programming*; D.2.11 [Software Engineering]: Software Architectures—*Patterns*

### Keywords

Framework, Design Patterns, Telegram Pattern, Process Automation Systems

### 1. INTRODUCTION

Consider two distinct processes with the needs of mutual data exchange, but implemented in different programming languages and running on different systems. This scenario often appears in process automation systems. In order to solve the communication problems, most systems use telegrams, i.e. byte streams of predefined format. A written document containing detailed information about the contents of the telegrams can be regarded as the interface description between the two processes.

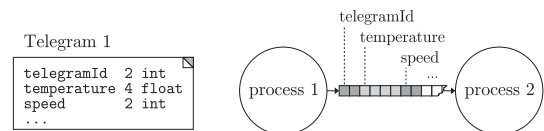


Figure 1: Telegram Description

The left part of Figure 1 shows a typical telegram description defining the names, the number of bytes and the types of the data sequentially contained in the telegram. Knowing this information, the byte stream sent from process 1 can be interpreted by the receiver (right part of Figure 1).

On the first glance, the problem seems quite trivial with a simple telegram description as the key to the solution. The problem is harder, though, when we take a closer look at it: First of all, it is necessary to distinguish between ASCII and binary telegrams. Whereas ASCII telegrams are quite easy to handle, binary telegrams are typically shorter, but they depend on the different representation of the data types caused by different operating systems. One operating system might store integer types in two bytes, whereas the other expects four bytes for them. Furthermore, the byte representations for numbers differ between little endian and big endian machines [6]. It is also possible that types and units do not match. The sender process transmits a certain value in millimeters as integer, and the receiver process needs it as float in meters or as a fraction of inches.

There are many further examples of communication barriers between telegram sender and receiver longing for a general implementation guide. The *Telegram Pattern* provides such a guide and acts as a bridge between the two processes. In the following, the *Telegram Pattern* is introduced step by step, continually increasing its capability to solve the problems mentioned above.

## 2. PARSING THE TELEGRAM

The first task of the receiver process is to read the incoming telegram byte by byte and to restore the information contained in it. The experienced object-oriented programmer probably thinks of an architecture where *Telegram* objects parse their corresponding byte streams themselves. So, there should be a *Telegram* base class providing the interface for parsing a byte stream. Every kind of telegram is then represented by a subclass, which implements the interface due to the structure of the telegram. We also create a member variable for every value contained in the telegram and try to assign the correct value during the parsing process.

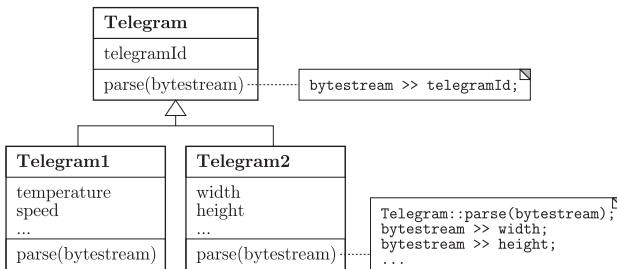


Figure 2: Telegram Class Hierarchy

Figure 2 shows an idealistic view of the problem with a simplified implementation of the parsing method. Actually, the implementation must be more powerful to solve the problems mentioned in section 1. Nevertheless, it is quite comfortable and should not be overloaded with parsing details. It should remain as simple as it is, and the process of reading the byte stream should be shifted to an external class.

There is already a design pattern for this kind of problem – the *Strategy Pattern*. It defines a family of algorithms, encapsulates each one, and makes them interchangeable [1]. This is exactly what we need here: We define a family of algorithms for the different ways of reading a byte stream, encapsulate each one in a class, and make them interchangeable for the already defined *Telegram* classes.

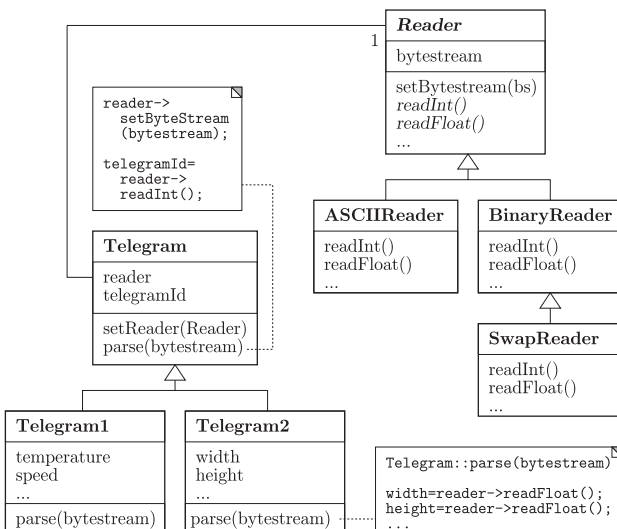


Figure 3: Strategy Pattern for Reading Telegrams

An abstract base class *Reader* provides the interface for all kinds of reading operations, which are implemented as separate methods in the derived classes. Reading an integer from an ASCII stream is different from reading it from a binary stream, and again different when the byte order is swapped. The detailed implementation of these algorithms is irrelevant, though, and not shown here. What is important, however, is the extensible class hierarchy consisting of various *Reader* classes, which enable a further simple implementation of the parsing method (see Figure 3).

This architecture appears as a flexible solution for parsing the incoming byte stream. A new telegram can be handled by deriving a new *Telegram* class and implementing the parsing method without considering the type of the byte stream. This task is delegated to the *Readers*, which can be individually assigned or exchanged – even at run-time.

## 3. EXTRACTING THE TELEGRAM ID

Although the strategy introduced in section 2 provides an adaptable approach of software design for telegram communications, it still neglects one important aspect: When the sender transmits several telegrams of different structure, the very first task of the receiver is to identify which telegram has just arrived. This task should be done at a central point within the receiver, let us call it *TelegramHandler*.

A *TelegramHandler* sequentially receives different kinds of telegrams and has to find out which *Telegram* classes can handle them. In order to distinguish between the byte-streams, every telegram carries a unique identification (the telegram id) in the first bytes of the stream. But this fact raises the next problem: Different software vendors use different telegram types, which makes it impossible to find out how many bytes should be read and how they should be interpreted in order to extract the correct id.

Consequently, the area of responsibility for a *TelegramHandler* has to be restricted in a way that one *TelegramHandler* can only handle telegrams of the same kind, and the same type and size for the id. So, if a process e.g. receives ASCII and binary telegrams, it has to provide *two TelegramHandlers* in order to interpret the telegrams correctly. The connection to the *TelegramHandlers* has to be established once in the beginning, to ensure that one *TelegramHandler* only receives one type of telegrams. Figure 4 shows a first attempt of a design for a corresponding class.

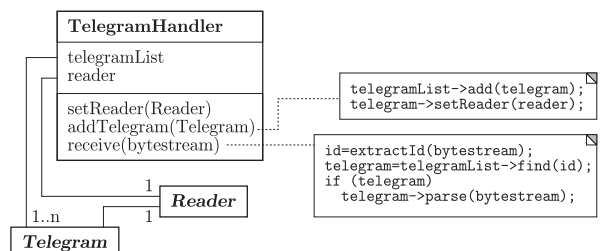


Figure 4: Attempt of a TelegramHandler Class

Due to the fact that every *TelegramHandler* is now restricted to one type of telegram, it correspondingly references only one *Reader* object. It can have several differently-structured

*Telegrams*, though, with a prototype object of each *Telegram* stored in the *telegramList*. In order to ensure the same type also in the implementation, every *Telegram* object added to the *TelegramHandler* is immediately initialized with the same *Reader* object (see top right implementation in Figure 4). The *TelegramHandler* does not create *Telegram* objects. It uses the prototype objects for parsing; and they must be processed before the next byte stream arrives.

A *TelegramHandler* must at least have a method for receiving a byte stream. The first action within this method is to extract the telegram id from the stream, in order to select the corresponding *Telegram* object, which can parse the rest of the byte stream. The code for this should look somehow like the one in the right bottom corner of Figure 4. Every *TelegramHandler* extracts the same amount of bytes from the stream. The implementation does not show any details on how this is done. Afterwards, it searches for a suitable prototype *Telegram* object (with the same id as just extracted) and delegates the parsing task to it.

There is a small problem with this first attempt, though: What data type does the extracted telegram id have? Sometimes, the id may be an integer and sometimes a string, or whatever. It doesn't even matter if the *TelegramHandler* itself extracts the telegram id or if this task is left to the *Reader*. None of these classes can provide a common solution. So, the only way out of this dilemma is to encapsulate the basic data types into wrapper classes and use them for representing the telegram ids (see Figure 5).

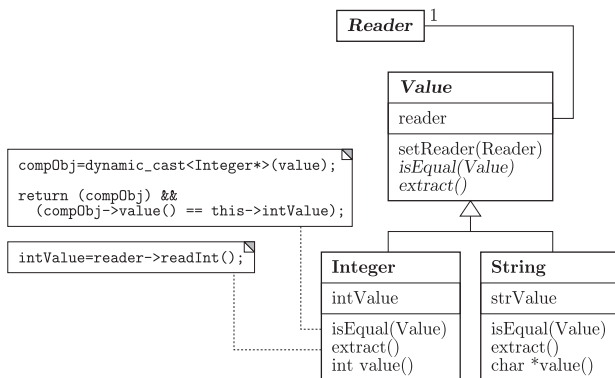


Figure 5: TelegramId Class Hierarchy

A telegram id object should be able to read its contents automatically out of the byte stream, so the wrapper hierarchy should be equipped with a virtual extraction method, individually implemented in the derived classes. In order to read the correct amount of bytes from the stream, every *Value* object uses a *Reader*, which is already able to handle different byte stream formats (see the code on the left bottom side of Figure 5). This appears to be a very flexible solution because the *Value* class (like the *Telegram* class in Figure 3) is independent from the type of the byte stream.

Additionally, the abstract *Value* class provides the interface for comparing itself to other *Value* objects. The implementation on the left side at the top of Figure 5 shows the comparison of different *Values*: Two *Value* objects can only be compared when they are of the same type. It makes no sense

to compare *Integer* and *String* values with each other. Thus, the first task within the comparison method is to check if both objects have the same type, and afterwards, if their contained values are equal.

Now, with this additional design reflection, we should be able to solve the entire parsing problem. Figure 6 summarizes the architecture developed so far and focuses on the major changes.

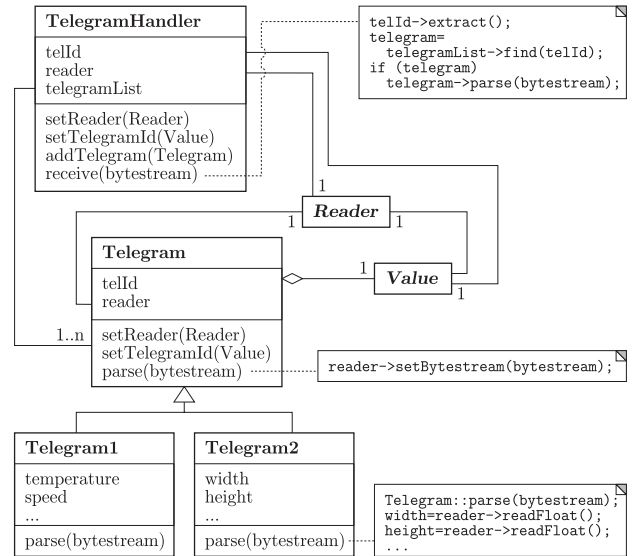


Figure 6: Pattern for Extracting the Telegram Id

We can identify the previously introduced *TelegramHandler* in the top left corner as the central point for receiving telegrams. This *TelegramHandler* in a first step needs to know the type of the telegrams, and what kind the telegram ids are. Therefore, it is initialized with a distinct *Reader*, as well as a *Value* object representing the telegram id type (which itself immediately receives the same *Reader*). The different *Telegrams* added to the *TelegramHandler* now carry their own *Values* for the telegram id, expressed through the aggregation connection between the two classes. All the rest concerning the *TelegramHandler* is the same as shown in Figure 4. The implementation of the telegram receiving method (top right hand corner of Figure 6) has slightly changed, though. The *telId* member object itself can extract the correct telegram id out of the byte stream. It can then be compared to the telegram ids of the stored *Telegram* objects using the *isEqual* method (which is not shown in the implementation here). When the suitable *Telegram* object is found, it can parse through the rest of the telegram. Because of the fact that the telegram id is already extracted, it is not necessary anymore to read it again within the parsing method, as it was shown in Figure 3. The bottom right corner of Figure 6 shows the implementation of this method without considering the telegram id.

Following this strategy, the first task within the telegram traffic (i.e. reconstructing the contents of a transferred telegram) can be solved. However, this is just the initial part of the issue. The succeeding section sheds some light on the next upcoming problem – the conversion of telegrams.

## 4. CONVERTING THE TELEGRAM

Until now, the design guidelines concentrated on the interpretation of the telegrams, facing the difficulties concerning data formats, byte order, etc. This section now makes the first step towards preparing the data for the receiver.

Reflecting the statements of the introduction section 1, there is a frequent difference in the data types and units between sender and receiver. Whereas the sender process transmits one value as integer in millimeters, the receiver process prefers to calculate with the same value as float in meters.

Initially, it has to be clear, that the telegram description (introduced in Figure 1) must be enlarged in a way, that not only the types and units of the original telegram are contained, but also those which are used at the receiver side.

telegramId	2	int	-	int	-	1
temperature	4	float	°C	float	°C	1
width	2	int	mm	float	m	0.001
speed	4	float	m/s	long	mm/s	1000
pressure	4	float	kg	float	N	9.81
...						

Figure 7: Extended Telegram Description

Figure 7 shows an example of how the extended description could look like: The first entry is the name of a value, followed by the number of bytes, the data type and the unit of this value. The next column specifies the preferred data type within the receiver, analogously followed by the units used there. The different units between sender and receiver force a transformation of the values, which can be recognized by the last column. This column acts as a kind of documentation and orientation guide for the developer to multiply the incoming values with the correct factor.

On the first glance, it seems as if there is no special pattern for this kind of problem. The software developer just has to go through the telegram description step by step, and individually implement the data conversions and multiplications for each entry and for each *Telegram* class. Figure 8 describes an approach of a solution for this problem.

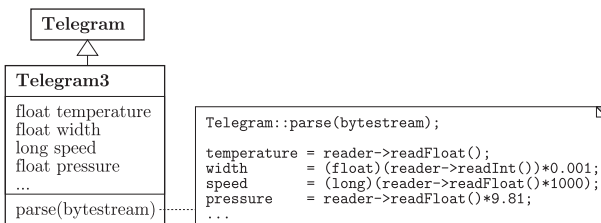


Figure 8: Telegram Conversion

Every derived *Telegram* class now carries member variables of those types needed at the receiver process, in contrast to former implementations, where they all had the original type transmitted by the sender. The implementation on the right side of Figure 8 contains type conversions and multiplications, due to the telegram description of Figure 7.

Actually, the implementation in Figure 8 is just an application of the *Telegram Pattern*. The programmer can assume a surrounding architecture with flexible methods to extract

the telegram id and to resolve type conflicts. Nevertheless, this architecture cannot be regarded as the final design of the *Telegram Pattern*, yet. Any component, which is interested in the values of a *Telegram*, exactly needs to know the member variables of the *Telegram* class. It would be more convenient to iterate over a generic list of *Values*, instead. So, it is only natural to replace the basic data types of all telegram entries with suitable *Value* types and take advantage of their extracting features. In the following, this generic *Value* list is continually introduced, accompanied by a mechanism to systematically do the conversions.

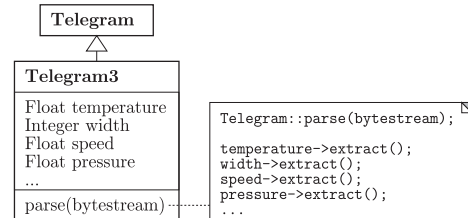


Figure 9: Generic Parsing Implementation

Neglecting the necessary conversions in a first approach, Figure 9 shows the advantage coming into existence by using the generic *Value* types. (Note, that the data types for the telegram entries temporarily again correspond to the sender types.) The reading commands within the parsing method no longer distinguish between different data types, because the *Value* objects themselves know how to read the contents.

Coming back to the conversion problem, we are confronted with the question how to handle the currently extracted *source Values* in order to convert them to their *destination* type. Obviously, we can solve the problem by introducing accompanying *Value* objects representing the destination types in the telegram, which receive the source values and automatically transform them, like in the following example:

```

srcType->extract();
destType->assign(srcType);
  
```

The data conversion is consequently shifted to the *Value* classes, which all have to provide a virtual *assign* method.

Before we take a look at the implementation, we have to consider the remaining multiplication factor for the completion of the telegram conversion. It is important *when* the multiplication takes place. Assume an integer source type which should become a float by the factor 0,001. In this case, the conversion takes place first and is followed by the multiplication; otherwise, information would be lost. It is the other way round, when a float should become an integer by the factor 1000. Now, the multiplication must be executed before the conversion, in order not to lose information. This fact can already be observed in Figure 8. The code in the parsing method should not be affected by this problem, though. So, the best place for the multiplication is the *assign* method of the *Value* class, where the manipulation sequence can individually be handled in the subclasses.

Figure 10 shows the new methods, which enable automatic type conversion in respect to the correct transformation order. Unfortunately, this hierarchy is now faced with a

tight dependency among the classes, providing a conversion method for every wrapped base type. But luckily, there are only a few basic data types, which have to be encapsulated, so that the number of conversion methods is low and fix.

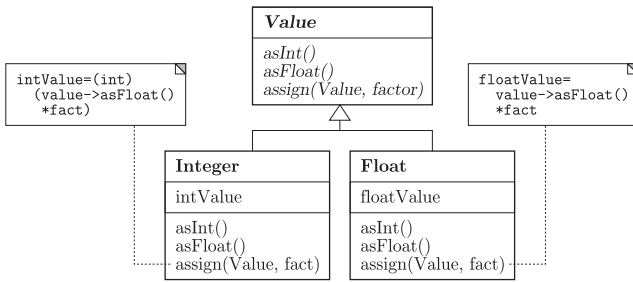


Figure 10: Automatic Value Conversion

Those conversion methods, which are the reason for the mutually dependent classes, need *not* to be touched anytime later, even when a special customer-defined data type is introduced and demands the derivation of a new class. In certain respect, every new (simple) data type represents a standard type; so, every new type should lead to a subclass of one of the existing non-abstract *Value* classes and inherit the conversion functionality. The only difference between standard type and customer-specific type may be a special byte representation, which forces the programmer to override the extraction method in this class, but nothing else.

With the aid of these new features, the parsing routine in the *Telegram* has now changed to a repeating sequence of reading source *Values* and assigning them to destination *Values* together with a multiplication factor:

```
srcTemperature->extract();
destTemperature->assign(srcTemperature, 1);
srcWidth->extract();
destWidth->assign(srcWidth, 0.001);
```

A closer look at the four lines of code shows that the implementation of the parsing method has become generic, not telling anything about the data types behind the variables, as it has done before (see Figure 8). The programmer is nearby released from the laborious task of implementing converting routines when he overrides the parsing method. This is the point, when we can finally introduce the previously mentioned generic *list* for the entries. For each telegram entry, this list carries a record containing the source *Value*, the destination *Value*, the multiplication factor and – to be still able to identify the entry – the value name. The major advantage of a *Value* list is that other classes, which are tightly coupled to the *Telegram*, do not need to know the exact structure of the *Telegram* class, anymore; they can refer to the contents e.g. by iterating over the *Value* list.

Figure 11 summarizes the outcome of the current design reflections. The record mentioned before is illustrated by an own *Descriptor* class. For every entry in the telegram description, the programmer now has to instantiate a *Descriptor* object with corresponding source and destination *Values*, the correct factor and the name, and sequentially add the records to the *Telegram*. The implementation of

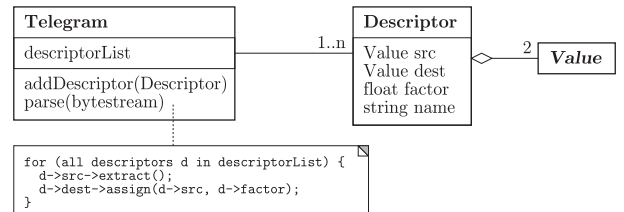


Figure 11: Telegram Conversion Summary

the parsing method now looks like the one at the bottom of Figure 11 and is equal for every telegram. This leads to a drastic, but advantageous change in the design because the *Telegram* class no longer serves as a base class for other telegrams, but can commonly be used for all kinds of telegrams, without the needs of deriving a new subclass.

## 5. SUMMARY

The *Telegram Pattern* provides a solution scheme for telegram-based communication systems. It supports the software developer to find the telegram id out of a byte stream, parse the byte stream and prepare the extracted data for the specific needs of the receiver process.

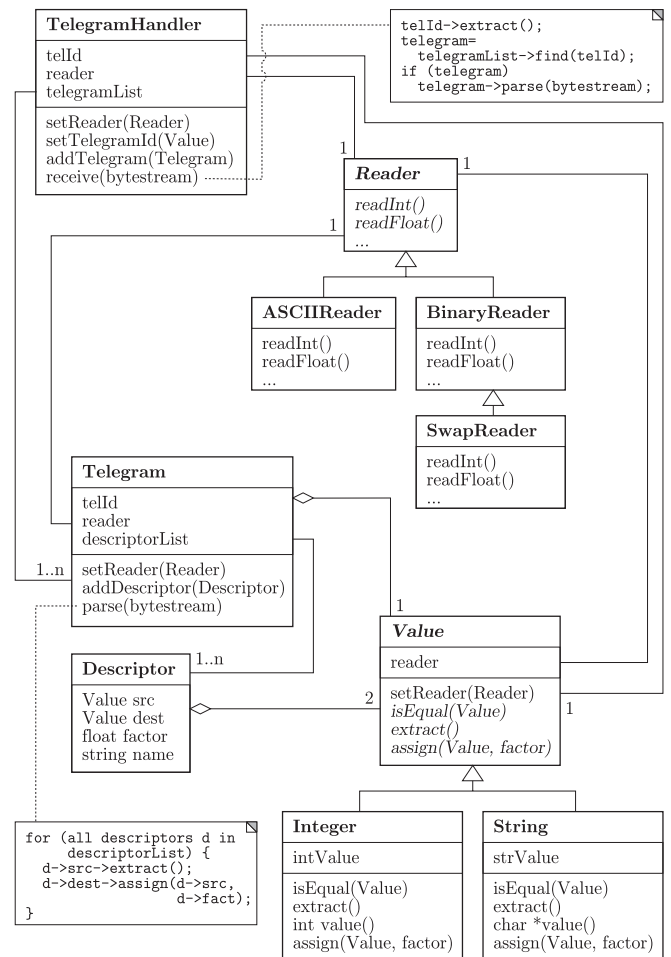


Figure 12: Telegram Pattern, Class Diagram

Figure 12 summarizes all classes and collaborations of the *Telegram Pattern*, which have been successively developed in the previous sections. The central class is the *Reader*, which offers generic methods for parsing a byte stream. All the other classes depend on this *Reader*, which can be observed by the association connections to the *Reader* base class. The data flow starts at the *TelegramHandler*, which receives a byte stream through its *receive* method. The first action in this routine is to extract the telegram id with the aid of the generic *Value* class (top right corner of Figure 12). Next, the parsing routine of the *Telegram* class is called, which itself provides a generic implementation of how to read and convert the contents of the byte stream (bottom left corner of Figure 12). Therefore, each *Telegram* carries a *Descriptor* list containing information about the parsing and the converting process. A detailed description of the classes, however, can be found in the previous sections 2 to 4.

In addition to the class diagram, the *Telegram Pattern* is also expressed by an object diagram, outlining a distinct scenario in the telegram traffic domain (see Figure 13).

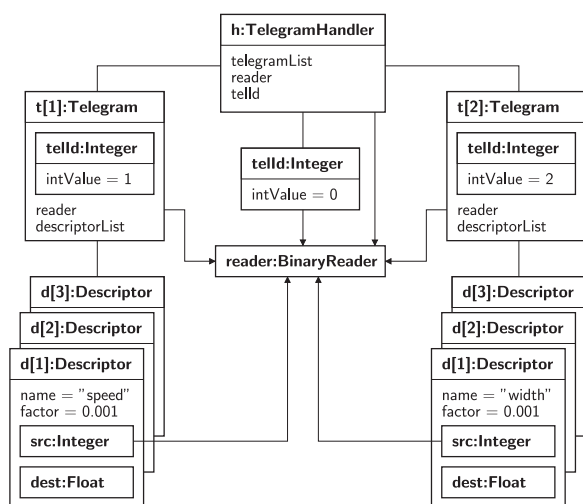


Figure 13: Telegram Pattern, Object Diagram

Let us assume that a telegram-based communication system transmits binary telegrams. Consequently, the receiver process has to provide one *TelegramHandler* as the interface to the “outer world”, with a *BinaryReader* connected to it. All the telegrams are identified by a unique integer value at the beginning of the byte stream, so the *TelegramHandler* additionally references an *Integer* object (with no distinct value initialized), in order to correctly extract the id. For this purpose, the *Integer* object itself uses the same previously mentioned *Reader* to examine the contents of the telegram. The *TelegramHandler* also carries a list of prototype *Telegrams* together with their *Integer* objects for the telegram id, which are now equipped with the same integer value as expected in the byte streams. In this case, only the *Telegram* objects reference the *Reader*. It is not necessary to initialize the *Integers* with a *Reader* because these objects are never used for parsing; they are just needed for comparing their value with the one extracted out of the telegram. Furthermore, every *Telegram* has a list of *Descriptors* containing information how to read and convert the incoming

byte stream. Again, the *Value* objects within the *Descriptors*, which are responsible for reading (i.e. the *src Integers* – see Figure 13), refer to the same *Reader* for proper parsing.

When a telegram arrives, the *Integer* object, which is connected to the *TelegramHandler*, extracts the identification out of the byte stream and assigns this value to its “inner mind”. This object is then compared to the other *Integer* telegram ids, in order to determine the *Telegram* object which can parse through the rest of the telegram and simultaneously convert the contained data for further computations. The *Telegram Pattern* does not create new *Telegram* objects. It just uses its prototype objects and continually “fills” them with new data. When a *Telegram* should be kept longer, it has to be copied into a permanent object.

Although the *Telegram Pattern* has particularly been developed for process automation systems, it can also be applied for many other problems with a similar structure. Due to the close relationship to the *Strategy Pattern*, the area of responsibility for the *Telegram Pattern* is quite similar: It is used when many related classes differ only in their behavior, and when one class should be configured with one of many behaviors [1]. What is different from the *Strategy Pattern*, though, is the possibility to automatically configure the expected behavior by using a special identification mechanism, which also abstracts from the concrete implementation.

## 6. CONCLUSION

The *Telegram Pattern* provides a powerful solution model for telegram-based communication as part of a framework for process automation systems. The author [4] describes four additional design patterns for this domain, which deal with the automatic distribution of the converted telegrams (*Dispatcher Pattern*) and provide an event-based architecture (*Condition Pattern*) for data requests (*Request Pattern*) and commands (*Instruction Pattern*).

## 7. REFERENCES

- [1] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [2] R. Lauber, P. Göhner: *Prozeßautomatisierung, 3. Auflage*, Springer Verlag, Berlin-Heidelberg, 1999.
- [3] T. Mowbray: *The Essential CORBA*, John Wiley & Sons, 1995.
- [4] W. Narzt: *Design Patterns for Process Automation Systems*, Dissertation University Linz, July 2000.
- [5] G. Pomberger, G. Blaschek: *Software Engineering. Prototyping und objektorientierte Software-Entwicklung*, Carl Hanser Verlag München Wien, 1996.
- [6] P. Rechenberg, G. Pomberger: *Informatik-Handbuch*, Carl Hanser Verlag München Wien, 1999.