

DATA TYPE MANAGEMENT IN A DATA MINING APPLICATION FRAMEWORK

Lauri Tuovinen, Perttu Laurinen and Juha Röning

*Department of Electrical and Information Engineering, P.O. Box 4500, FIN-90014 University of Oulu, Finland
tuokku@ee.oulu.fi, perttu@ee.oulu.fi, jjr@ee.oulu.fi*

Keywords: Data mining, Application framework, Pipes and filters, Data representation, Data type, Interoperability.

Abstract: Building application frameworks is one of the major approaches to code and design reuse in object-oriented software engineering. Some frameworks target a particular application domain, adopting a number of domain-specific problems to be addressed by the framework in such a fashion that there is no need for application developers to devise solutions of their own to those problems. When the target domain is data mining, one interesting domain-specific problem is management of the data types of model parameters and data variables. This is not trivial because the framework must be able to convert parameter and variable values between different representations, and it would be preferable to have these conversions take place transparently, without involving the application programmer. This is not difficult to achieve if the framework restricts the programmer to a predefined set of allowed data types, but if such a restriction is undesirable, the framework needs an extension mechanism in its type management subsystem. Smart Archive, a framework for developing data mining applications in Java or C++, includes such a mechanism, based on a type dictionary document and a type renderer programming interface. These make it possible to handle even highly complex values such as collections of instances of programmer-defined classes in a variety of platform-independent representation formats. The benefits of this approach can be seen in how the framework interfaces with databases through data sinks and in how it exports and imports application configurations.

1 INTRODUCTION

Reusing designs and code is considered desirable in software engineering. An application framework—a software skeleton that can be specialized into different applications by plugging in a comparably small quantity of new code—provides both a reusable design and a body of code that implements the design in a reusable form. A good framework is therefore a valuable commodity in software development, and in certain areas a thoroughly indispensable one; one would hardly consider writing a graphical application for a modern desktop operating system without adopting a framework such as MFC, Swing or GTK+ to take care of handling user interface elements and events. (Fayad and Schmidt, 1997)

The purpose of an application framework is to support a family of applications, often defined by a particular type of functionality required. Smart Archive (Laurinen et al., 2005; Tuovinen et al., 2008) is a framework for applications that use data mining to extract knowledge from a large body of data. It pro-

vides an application design based on the pipes-and-filters architectural style, breaking down the solution to a data mining problem into a sequence of transformations that can be implemented independently. The transformations (filters) are encapsulated in components, which may also contain a data sink for storing the transformation results into a file or a database.

Smart Archive is intended to be highly generic, adaptable to a wide variety of data mining problems without having to modify the framework code. One aspect of this generic nature is the manner in which the data types of variables and parameters are handled. Clearly it would make the framework less generic if only a small fixed set of data types could be used. On the other hand, there needs to be a way of controlling the types used in Smart Archive applications in order to ensure interoperability between the framework and other application subsystems such as databases. Interoperability between different implementations of the framework is another important issue, as Smart Archive is being concurrently developed for two programming languages, Java and C++.

A concrete example of how the type management problem affects application development can be seen by considering the case of storing the results of a data transformation into a database table. In order to accomplish this an application has to generate a table structure with suitable column types and a sequence of SQL queries to write the data into the table. This, in turn, requires a mapping from application platform (e.g. Java) types into SQL types and conversions of values into strings that can be appended to an `INSERT` query. Without type management this involves a considerable manual effort.

Smart Archive solves this problem by introducing an extensible data type engine. The engine offers a range of commonly used default types and is open to the addition of new types by developers using the framework. The extension API ensures that each new type created by application programmers satisfies certain conditions designed to enforce interoperability. Thus the use of data types in Smart Archive is unrestricted without also being uncontrolled. The primary contribution to the state of the art is an abstract type system covering a number of concrete data processing and storage platforms; the usefulness of the type system is highlighted by important framework features that could not be implemented without it.

Section 2 describes how applications are developed using Smart Archive and briefly reviews other data mining frameworks. Section 3 explains in detail how the Smart Archive type engine works. Section 4 discusses the practical implications of type management in Smart Archive, demonstrating how the type engine benefits the framework. Section 5 discusses the findings, pointing out strengths and opportunities for future work as well as known shortcomings and possible remedies. Section 6 concludes the paper.

2 DEVELOPMENT WITH SMART ARCHIVE

Applications based on Smart Archive are built from components. A component, in this context, is very specifically defined as an instance of a class that implements the `Component` interface, which is one of the core elements of the framework. A component class may have some distinguishing features of its own, but the functionality of a component is determined by the filter and sink it contains. Simply put, a component is a receptacle into which different filters and sinks can be plugged as the application developer wishes.

Similarly, a filter is an instance of a class that implements the `Filter` interface. A filter receives its input from its parent component and likewise sends its

output to the parent component; in between the data is transformed by the filter's processing algorithm. The interface between the filter and the component consists of groups of logically interrelated data variables called channels: input channels that the filter requires to be supplied and output channels that the filter supplies. For example, the filter could implement a sensor fusion algorithm, in which case each input channel would deliver the data generated by one sensor.

Finally, a sink is an instance of a class that implements the `Sink` interface. A sink acts as an abstraction layer between a component and a persistent data storage system, typically a relational database. When a component has a sink, all of the output produced by the component's filter is delivered to it, and the sink executes the operations required to write the data into persistent storage. Alternatively, the component can retrieve data previously stored into the sink, in which case the sink assumes the role of the filter, producing data that becomes the component's output.

Components are linked together with pipes to form a directed graph with a single root node. Raw input data is inserted into the graph at the root and then pushed through each component in turn in an order dictated by the inter-component links (pipes). Thus the data is refined in a stepwise manner until the desired result of the application is obtained. This style of software architecture is natural for data mining applications, which generally progress in steps following a process where each step brings the data closer to a form that can be interpreted and applied.

D2K (Automated Learning Group, 2003), KNIME (Berthold et al., 2006) and RapidMiner (formerly YALE) (Mierswa et al., 2006) are data mining frameworks that, like Smart Archive, employ the paradigm of assembling independent components into solutions. Each provides a graphical interface for selecting, configuring and coupling components, which is a convenient way of creating processing sequences. Smart Archive trades off some of this convenience for the ability of developers to decide exactly which parts of the framework are used in an application and how, allowing applications to be tailored in a greater variety of ways. The greater degree of control implies a greater number of things to learn and to keep track of, but it still leaves the option of hiding the greater complexity behind a comparably simple interface, so in a sense this is a best-of-both-worlds approach.

Another major difference is that only Smart Archive is implemented and maintained in more than one programming language, making cross-language portability of applications a consideration that the other frameworks do not need to address. An exception to this is the XELOPES library (Prudsys

AG, 2007), which uses a single platform-independent model, mapped to multiple platform-specific models, to achieve portability. However, the focus and underlying paradigm of XELOPES differ so much from the frameworks discussed above that it does not really make sense to think of it as a parallel to them.

As reported in (Tuovinen et al., 2008), Smart Archive has been successfully used to develop data mining applications in diverse application domains. Thus the framework is not merely a proven concept but a practical foundation for the implementation of real-world applications.

3 TYPE MANAGEMENT IN SMART ARCHIVE

In Smart Archive, every data type used for parameters or variables in applications must have a corresponding entry in the type dictionary. The type dictionary is an XML document containing the following information about each type:

- abstract type name
- type renderer class
- concrete type names
- type parameters (if applicable)

An excerpt from the dictionary is shown in Figure 1.

The abstract type name is what is used in application code to identify the type when specifying parameters and variables. An abstract type is realized by one or more concrete types, each in a particular context. One context is the framework itself, and concrete types in this context are types in the implementation language of the framework. In the other currently implemented context—relational databases—concrete types are SQL data types. It is possible to have more than one concrete type per context by making the choice of type dependent on type parameters.

Besides translating type names between different contexts, the type engine also converts values between different representations. Each type has an internal representation, dictated by the framework implementation language, and two external representations, SQL string and XML element. Conversions between representations are handled by classes that implement a special interface, known as type renderers. The framework includes a default renderer that provides a number of commonly needed basic data types. For types not supported by the default renderer it is possible to build an extension by coding a new renderer and writing entries for the new types into the type dictionary. The framework, when started,

```
- <type name="double" renderer="archive.datamining.DefaultRenderer">
- <concrete-types>
  <ctype context="Java">java.lang.Double</ctype>
  <ctype context="SQL">double precision</ctype>
</concrete-types>
</type>
- <type name="char" renderer="archive.datamining.DefaultRenderer">
- <concrete-types>
  <ctype context="Java">java.lang.Character</ctype>
  <ctype context="SQL">char(1)</ctype>
</concrete-types>
</type>
- <type name="string" renderer="archive.datamining.DefaultRenderer">
- <parameters>
  <param type="categorical" required="no">maxLength</param>
</parameters>
- <concrete-types>
  <ctype context="Java">java.lang.String</ctype>
  <ctype context="SQL" with-params="maxLength">varchar(%maxLength%)</ctype>
  <ctype context="SQL">text</ctype>
</concrete-types>
</type>
- <type name="boolean" renderer="archive.datamining.DefaultRenderer">
- <concrete-types>
  <ctype context="Java">java.lang.Boolean</ctype>
  <ctype context="SQL">tinyint</ctype>
</concrete-types>
</type>
- <type name="date" renderer="archive.datamining.DefaultRenderer">
- <concrete-types>
  <ctype context="Java">java.sql.Date</ctype>
  <ctype context="SQL">date</ctype>
</concrete-types>
</type>
- <type name="time" renderer="archive.datamining.DefaultRenderer">
- <concrete-types>
  <ctype context="Java">java.sql.Time</ctype>
  <ctype context="SQL">time</ctype>
</concrete-types>
</type>
- <type name="array" renderer="archive.datamining.DefaultRenderer">
- <parameters>
  <param type="character" required="yes">elementType</param>
  <param type="categorical" required="yes">arrayDepth</param>
</parameters>
- <concrete-types>
  <ctype context="Java">java.lang.Object</ctype>
  <ctype context="SQL">text</ctype>
</concrete-types>
</type>
```

Figure 1: An excerpt from the type dictionary, showing entries for some of the types supported by the default renderer. The `ctype` elements indicate the concrete Java and SQL data types corresponding to each abstract type.

will load all type renderers and place them into a data structure where they can be accessed quickly.

Figure 2 illustrates what happens when the framework needs to convert a value from one representation to another. First, a ‘render’ message is passed to the type engine; the message contains the abstract type name, the current and target representation formats and the value itself, an `Object` in Java. The type engine uses the type name to look up the type renderer in charge of the type in the type dictionary. It then passes the ‘render’ message to the renderer, which returns the converted value. Finally, the return value is passed to the routine that generated the original ‘render’ message. The return value, another `Object`, can then be cast into its runtime type—in this case, `String`, as the purpose of the rendering is to obtain a value that can be appended to an SQL query string.

In the case of compound values such as lists and maps the rendering mechanism is applied recursively: a render operation invoked on such a value triggers nested operations to render the elements of the compound value. Figure 3 shows the result of this when a map object is rendered as XML. The initial render operation generates the hierarchy of XML elements and then uses recursion to fill in the contents of the key

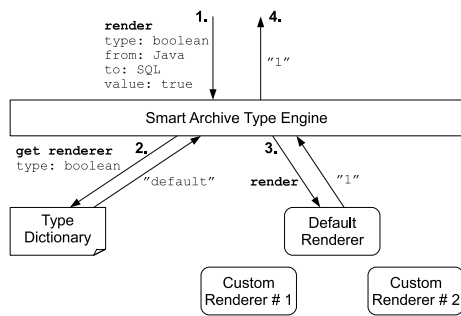


Figure 2: The processing sequence of a rendering operation. The type engine receives a message (1.) and does a type dictionary lookup (2.) to find out which type renderer to call. The message is then forwarded to the renderer (3.), which performs the conversion. Finally, the converted value is returned (4.) to the routine that invoked the type engine.

```

- <parameter name="param4">
- <value type="map" key-type="string" value-type="boolean">
- <elements>
- <element>
- <key>
- <value type="string">foo</value>
</key>
- <value>
- <value type="boolean">>true</value>
</value>
</element>
- <element>
- <key>
- <value type="string">bar</value>
</key>
- <value>
- <value type="boolean">>false</value>
</value>
</element>
</elements>
</value>
</parameter>

```

Figure 3: A string-to-boolean map object rendered as XML by the type engine.

and value nodes. The XML representation is also used as the SQL form of these data types.

D2K, KNIME and RapidMiner each manage the data types of variables in such a way that, while the implementation approach varies, the overall effect is largely the same. A concrete difference, however, is that D2K, KNIME and RapidMiner each delegate the responsibility for handling variables in external contexts to the components that interface with those contexts. For example, while each framework provides a component whose function is similar to that of the data sink in Smart Archive, the conversion of values from internal to external representation is up to the components themselves. In Smart Archive the conversion functionality is provided by the framework and available also to filters and other application elements that employ some form of persistent storage.

Another notable difference is the treatment of parameters (values that control the behavior of components): in Smart Archive, unlike the other frameworks, the same mechanisms are used to handle both parameters and variables. Thus in Smart Archive ev-

ery data type that is legal for parameters is also legal for variables and vice versa. On the other hand, if one knows that a type will only be used for a particular purpose, extending the framework to support the type can be simplified considerably; in the simplest case it is sufficient to just insert an entry into the type dictionary without writing any new code.

4 THE TYPE ENGINE IN PRACTICE

To see how the type engine affects Smart Archive application development in practice, let us consider a simple example application with three components that are executed sequentially. The first component takes the raw input data, cleans and normalizes it and outputs the preprocessed data in a format understood by the second component, which extracts features from the data. The third component takes the extracted features and uses them for cluster analysis.

The outputs of the filters of the three components are written into a database using a data sink. Creating the sinks requires generating database table structures suitable for storing the outputs of the filters. The type engine makes this highly convenient because it can translate the output variable specifications of the filters into the equivalent SQL column specifications. All Smart Archive filters are required to declare their output variables, so given a filter, the appropriate sink can be generated with little extra work. In fact, in Smart Archive a single method call is enough, whereas setting up the table structures manually would easily require tens of lines of code.

When a batch of data is fed into the application, it is progressively refined by each filter and then written into a database by the associated sink. The sinks use the type engine to render as strings the variable values produced by the filters. The rendered strings are formatted such that they can be directly appended to the VALUES clause of an SQL INSERT query. The queries are automatically generated by the sinks so that the entire process is almost completely transparent; the only database-related concepts the programmer needs to address are the connection parameters.

Once the dataset has been processed by all components, the database contains not only the final results (from cluster analysis) but also an image of the data after each intermediate step (preprocessing and feature extraction). Supposing that the application operator now wants to repeat the clustering with different parameters, there is no need to start the whole process over: the results of the feature extractor can be retrieved from its sink, leaving only the clusterer to be

re-executed. Similarly, if the operator wants to try out a different feature set, the results of the preprocessor can be retrieved, leaving only the other two components to be re-executed. Thus Smart Archive can be used to implement the stepwise approach to data mining discussed in (Laurinen et al., 2004).

Finally, supposing that the operator wants to transmit the application to someone else, it is enough to send a description of the application configuration provided that the recipient also has an installation of Smart Archive. Another necessary condition is that the recipient has access to all classes used by the application. Application classes that are not part of the framework are mainly filter classes, so the latter condition is generally true if the application uses only algorithms found in common libraries. Interfaces to such libraries are among the planned future extensions to Smart Archive.

Components, filters and sinks in Smart Archive may have any number of parameters that control their functions. The parameters, like variables, may be of any data type supported by the type engine. Smart Archive allows the values of the parameters, along with all other information that defines a Smart Archive application, to be written into an XML-based configuration file. The type engine is used to render the parameter values as XML elements.

Generating the configuration file proceeds from the top of the application hierarchy to the bottom. The top-level routine generates the outline of the configuration document and iterates through the components of the application, requesting each in turn to describe itself as an XML element. Each component then requests the same from its filter and sink and inserts the results into the description it returns to the top-level routine. The top-level routine collects the component descriptions and inserts them into the configuration document. Loading a configuration into the framework is basically a reversal of this process: every component, filter and sink class also knows how to generate an application object from an XML element. The framework therefore simply needs to parse the XML file, find the specified classes and have them instantiate themselves by giving them as input the appropriate sections of the configuration description.

The generation of component descriptions, like the generation of data sinks, takes place in an entirely transparent fashion. This is because the only parts of a component description that need to be generated dynamically are the class names and parameter values of the component and its filter and sink. The former are trivial to find and the latter are handled by the type engine, so the XML operations can be implemented in a common superclass without any knowledge of the

classes that are derived from it. As long as application programmers derive their component, filter and sink classes from these standard abstract superclasses, their classes are automatically able to perform configuration export and import operations as required.

5 DISCUSSION

The interchangeability of filters and sinks is one of the most important design ideas in Smart Archive. Whether a component is transforming data with its filter or retrieving data from its sink makes no difference to the components that follow it: in either case the component is producing output that the other components process further. With sinks providing access to all the data that has passed through the application, the effects of a changed parameter or algorithm can be tested by rerunning only the affected part of the processing sequence. Compared to the tightly integrated data sink, an external database interface would not provide an adequate solution, but working with sinks could easily become a nuisance if the mappings between variables and database columns had to be done manually. The type engine prevents this.

The recursive approach adopted by the type engine in rendering compound types can also be used by application developers writing renderers for data types they want to use in their Smart Archive applications. Provided that the state of a class consists entirely of types already supported by the type engine, a type renderer for the class mostly just needs to invoke already implemented render operations and compile their output. Conceivably this could even be performed automatically by the framework, but at this time the detailed specification and implementation of such a mechanism must be deferred to future work.

A possible weakness in the Smart Archive type rendering scheme is the restriction that there can be only one renderer for each type. This could prove inconvenient, in the future if the framework is extended with a new representation for values. In this case one might want to write a new renderer to handle the new representation rather than modify existing ones, especially if one does not have access to the source code of some of the renderers one is using. The restriction may be lifted in future revisions of the framework.

About configuration documents it is worth noting that they describe applications in a format that does not depend on a particular implementation of the framework. A configuration exported from a Smart Archive Java application could therefore be imported into the C++ version of the framework. The one language-specific feature of the format—class

names—means, however, that an additional dictionary would be needed, mapping the names of Java classes to those of the equivalent C++ classes. This aspect of the C++ version of Smart Archive has not yet been fully developed, so testing the portability of configurations has not been possible.

6 CONCLUSIONS

This paper addressed the problem of managing the data types of parameters and variables in a data mining application framework. This problem is interesting and nontrivial because, on the one hand, artificially limiting the set of types a programmer can choose from is undesirable, but on the other hand, not controlling types at all may lead to problems in interoperability. Smart Archive, a data mining framework for Java and C++, employs a solution based on a type dictionary and one or more type renderers, allowing developers to extend the selection of available types while ensuring that the framework knows how to handle each type in various situations relating to conversion of values between representations.

The type dictionary is an XML document containing information on each data type, including the name of the type renderer that handles the type. A type renderer, in turn, is a class that implements a special rendering interface through which the framework uses its services. New types are created by editing the dictionary and coding a renderer. The dictionary and renderers allow the framework to keep track of any number of concrete types and representations associated with a given type, while the application programmer only needs to be aware of a single abstract type name and a single representation, namely the one used in the implementation language of the framework.

The main principles of Smart Archive application development were first introduced, along with other systems intended for the same purpose. The Smart Archive solution to data type management was then examined in detail. The practical implications of this type engine were explored by walking through a hypothetical case study. Finally, some notable strengths, weaknesses and open issues were identified and analyzed. The type engine has proved useful for implementing framework features that make application coding considerably quicker and more convenient.

ACKNOWLEDGEMENTS

The authors would like to thank the Finnish Funding Agency for Technology and In-

novation (<http://www.tekes.fi>), Rautaruukki (<http://www.ruukki.com>) and Polar Electro (<http://www.polar.fi>) for funding the research on Smart Archive in the SAMURAI project. L. Tuovinen wishes to thank the Graduate School in Electronics, Telecommunications and Automation (<http://signal.hut.fi/geta/>) for funding his postgraduate work.

REFERENCES

- Automated Learning Group (2003). *D2K Toolkit User Manual*. Technical manual, available at <http://alg.ncsa.uiuc.edu>.
- Berthold, M. R., Cebron, N., Dill, F., di Fatta, G., Gabriel, T. R., Georg, F., Meinl, T., Ohl, P., Sieb, C., and Wiswedel, B. (2006). Knime: The Konstanz information miner. In *Proceedings of the 4th Annual Industrial Simulation Conference, Workshop on Multi-Agent Systems and Simulation*.
- Fayad, M. E. and Schmidt, D. C. (1997). Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38.
- Laurinen, P., Tuovinen, L., Haapalainen, E., Junno, H., Röning, J., and Zettel, D. (2004). Managing and implementing the data mining process using a truly stepwise approach. In *Proceedings of the Sixth International Baltic Conference on Databases & Information Systems*, pages 246–257.
- Laurinen, P., Tuovinen, L., and Röning, J. (2005). Smart Archive: a component-based data mining application framework. In *Proceedings of the Fifth International Conference on Intelligent Systems Design and Applications (ISDA 2005)*, pages 20–25.
- Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., and Euler, T. (2006). YALE: Rapid prototyping for complex data mining tasks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 935–940.
- Prudsys AG (2007). *Xeli's Intro. Introduction to XELOPES*. Technical manual, available at <http://www.prudsys.com>.
- Tuovinen, L., Laurinen, P., Juutilainen, I., and Röning, J. (2008). Data mining applications for diverse industrial application domains with Smart Archive. In *Proceedings of the IASTED International Conference on Software Engineering (SE 2008)*, pages 56–61.