UMAP: A Universal Layer for Schema Mapping Languages The Implementation

Florin Chertes

Technische Universität Wien, Vienna, Austria
Institut für Informationssysteme
FlorinChertes@acm.org

1 Introduction

Umap a new universal layer for schema mapping languages which provides a unified abstraction and middleware for high-level visual mapping languages was presented in [1]. The main Tools used for the Umap implementation were:

- as operating system (OS), Windows XP (WIN32),
- as Uml modeling tool, the well known and appreciated product Sparx Systems' Enterprise Architect 9.1 (EA) and
- as integrated development environment (IDE) for C++, the product Microsoft Visual C++ 2010 Express (MSVC).

No specific WIN32, EA or MSVC features were used making the implementation OS, modeling tool and IDE independent. The implementation is dependent on standard Xml, standard Uml-Ocl and the target programming languages like standard C++11. At the same time the C++ features used are limited to those that are to be found in languages like Java and XQuery.

First we designed with the help of the modeling tool the classes representing the source, the target and the mapping. We augmented them with the functions for reading and writing the data from and to the Xml files and the functions for the mapping. The mapping functions were further defined with the help of the Ocl expressions. The model was saved in a file of type EAP (Enterprise Architect Project).

Second using the modeling tool we generated the definition and the implementation files of these classes. The programming language C++ uses for each class two files: a header file defining the class and an implementation file for the member functions of the class. The mapping function were implemented using the Ocl expressions as definition.

Third, using the IDE we created a project, a MSVC specific artifact with a main function, a C++ specific artifact. The project was saved in a file of type SLN (Microsoft Visual Studio Solution). We added the generated files to the project. The generated files are OS (Windows, Linux) and IDE (MSVC, Eclipse) independent, this means that we could as well choose on Linux the Eclipse as IDE. The MSVC IDE could immediately create an executable. The

main program was extended to use the topmost classes from the model: the source, the target and the mapping class.

Last, the implementation was executed. In this way the input file, a XML file defining the source, used in the CLIP presentation, was transform in another XML file, the target, defined by the mapping.

2 Implementation of the CLIP feature: Mapping with Context Propagation

All the files, implementing the classes from the diagram Fig. 1 (presented in [1, Fig. 2]) are stored in the archive file ex205_reg_emp.zip, Fig. 2. Each class from the class diagram hat its implementation files.

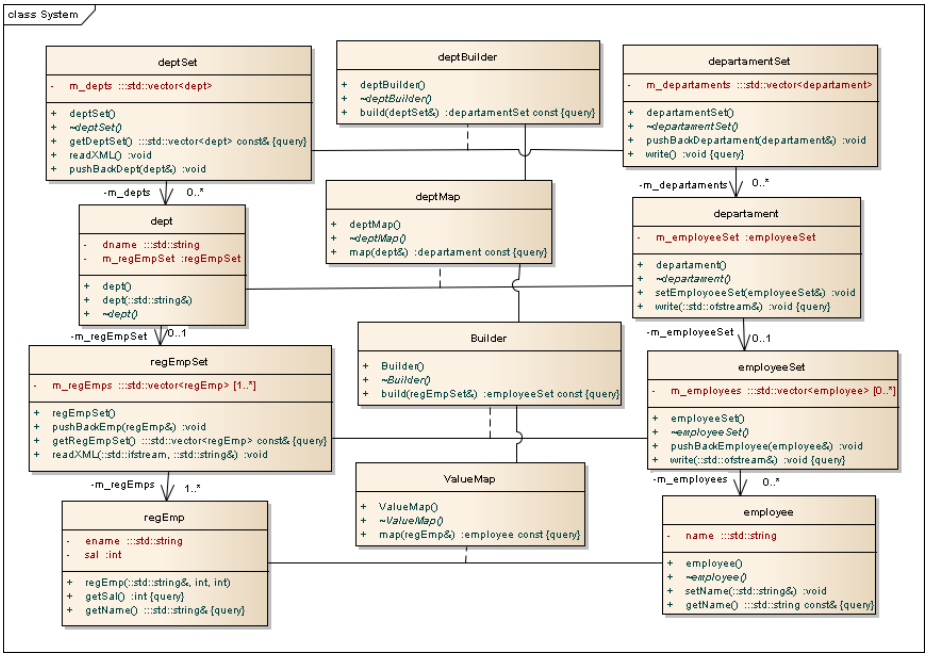

Fig. 1: Mapping with Context Propagation, the class diagram

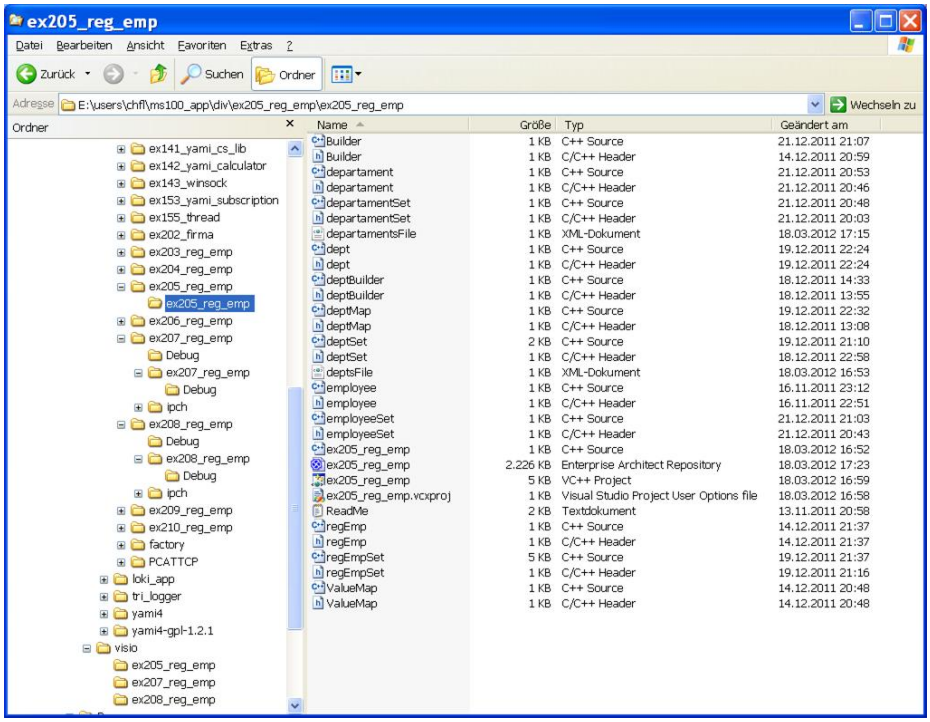

Fig. 2: Mapping with Context Propagation, the implementation files

The main function, Fig. 3 triggers the mapping. It uses:

- an object of the class *deptSet* named aDeptSet representing the source,
- an object of the class *deptBuilder* named builder representing the mapping and
- an object of the class *departmentSet* named aDepartmentSet representing the target.

The lines *11-12* create an instance of the source and read the input from the XML input file, Fig. 4. The lines *14-15* create an instance of the mapping object and use it to map the source to the target. The line *15* executes the mapping. The mapping object, using the function *build* takes as argument the source-object and produces the target-object. The line *17* writes the target-objet to an XML output file, Fig. 5.

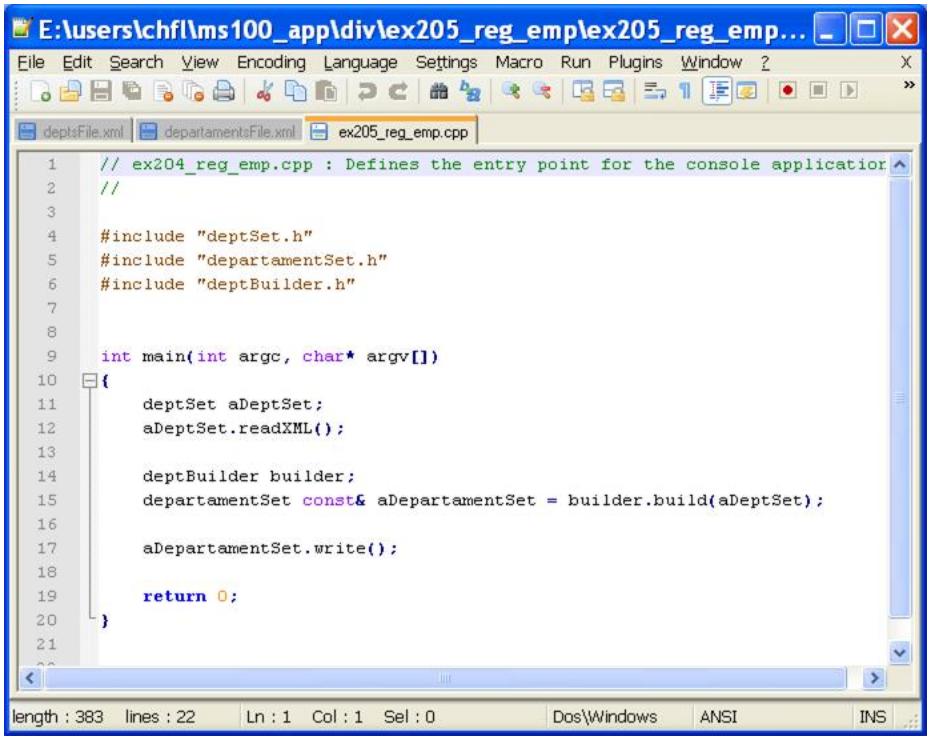

Fig. 3: Mapping with Context Propagation, the main function

Fig. 4: Mapping with Context Propagation, the input

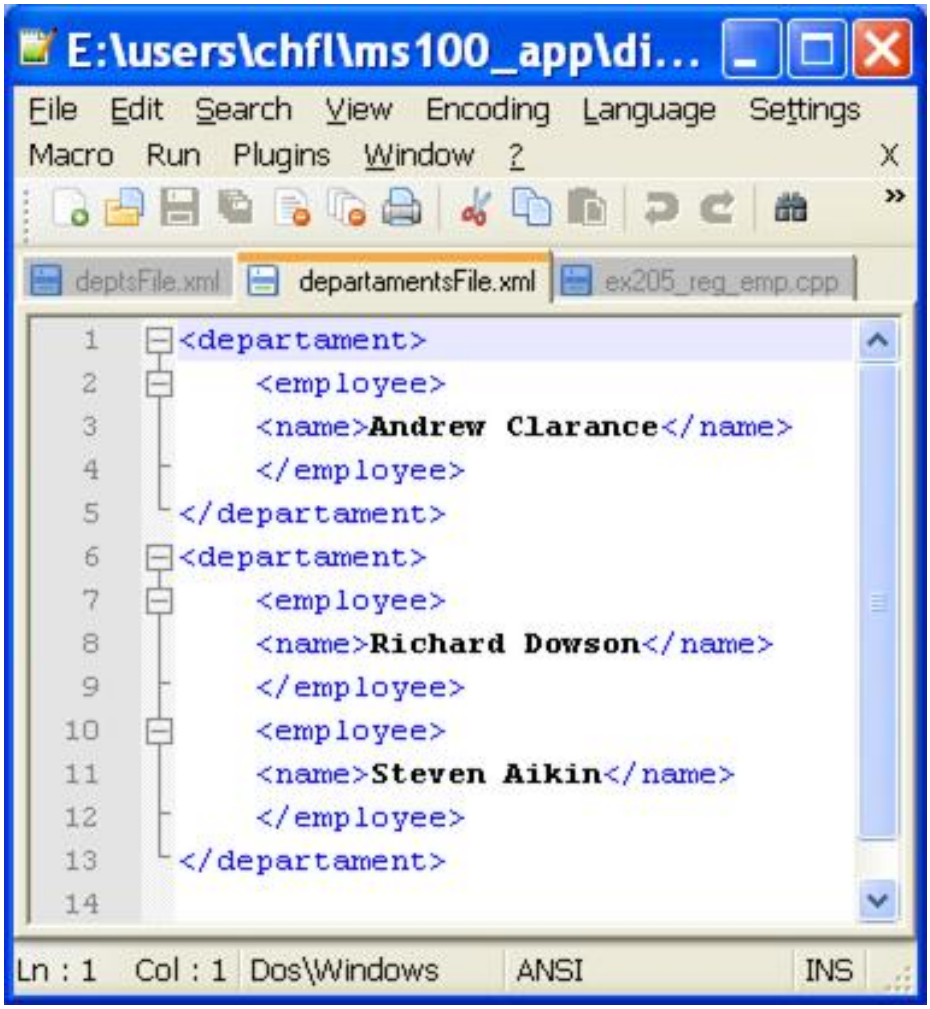

Fig. 5: Mapping with Context Propagation, the output

3 Implementation of the CLIP feature: Mapping with Join

All the files, implementing the classes from the diagram Fig. 6 (presented in [1, Fig. 4]) are stored in the archive file ex207_reg_emp.zip, Fig. 7. Each class from the class diagram hat its implementation files.

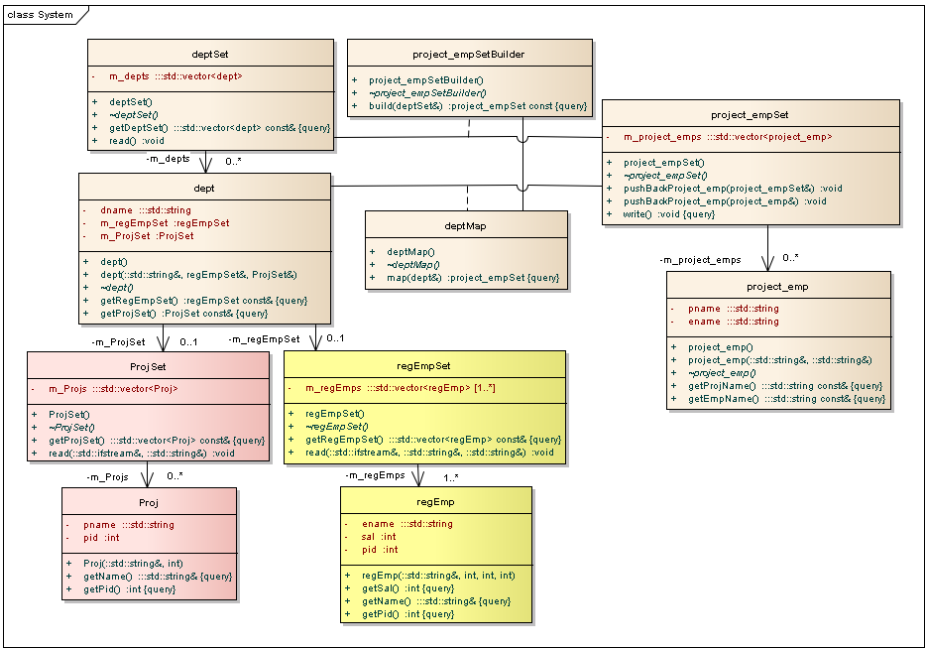

Fig. 6: Mapping with Join, the class diagram

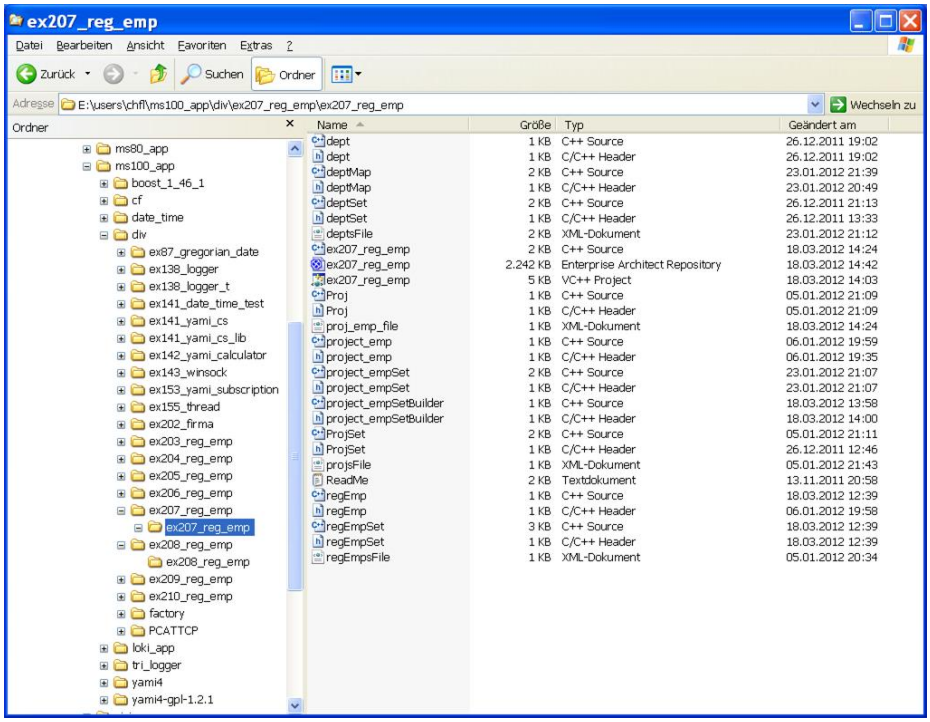

Fig. 7: Mapping with Join, the implementation files

The main function, Fig. 8 triggers the mapping. It uses:

- an object of the class *deptSet* named aDeptSet representing the source,
- an object of the class *project_empBuilder* named builder representing the mapping and
- an object of the class *project_empSet* named aProject_empSet representing the target.

The lines *10-11* create an instance of the source and read the input from the XML input file, Fig. 9. The lines *13-14* create an instance of the mapping object and use it to map the source to the target. The line *14* executes the mapping. The mapping object, using the function *build* takes as argument the source-object and produces the target-object. The line *16* writes the target-objet to an XML output file, Fig. 10.

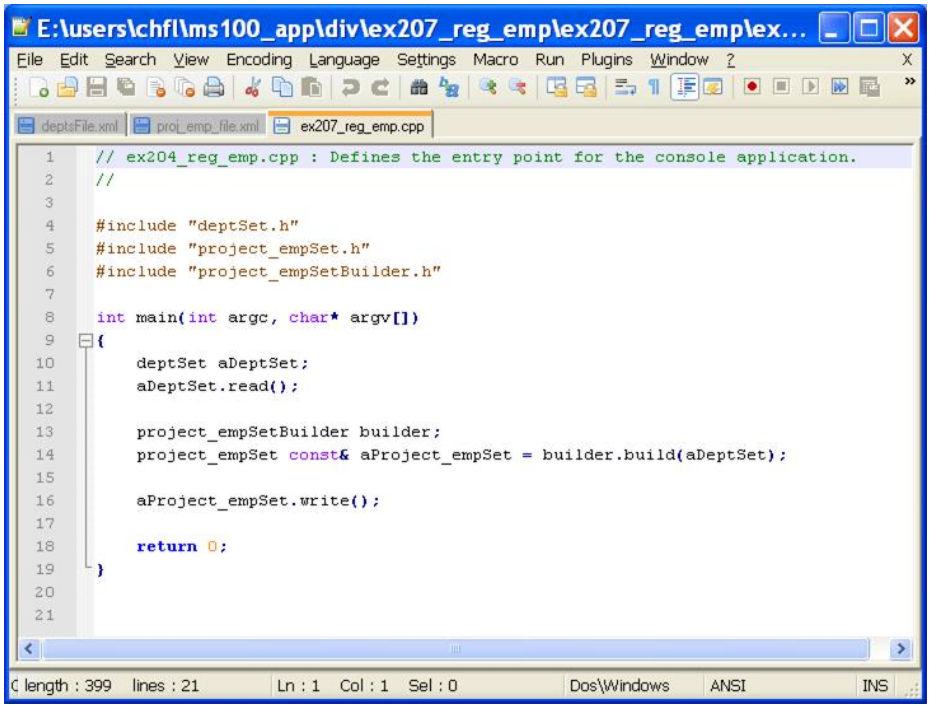

Fig. 8: Mapping with Join, the main function

Fig. 9: Mapping with Join, the input

Fig. 10: Mapping with Join, the output

4 Implementation of the CLIP feature: Mapping with Join and Grouping

All the files, implementing the classes from the diagram Fig. 11 (presented in [1, Fig. 5]) are stored in the archive file ex208_reg_emp.zip, Fig. 12. Each class from the class diagram hat its implementation files.

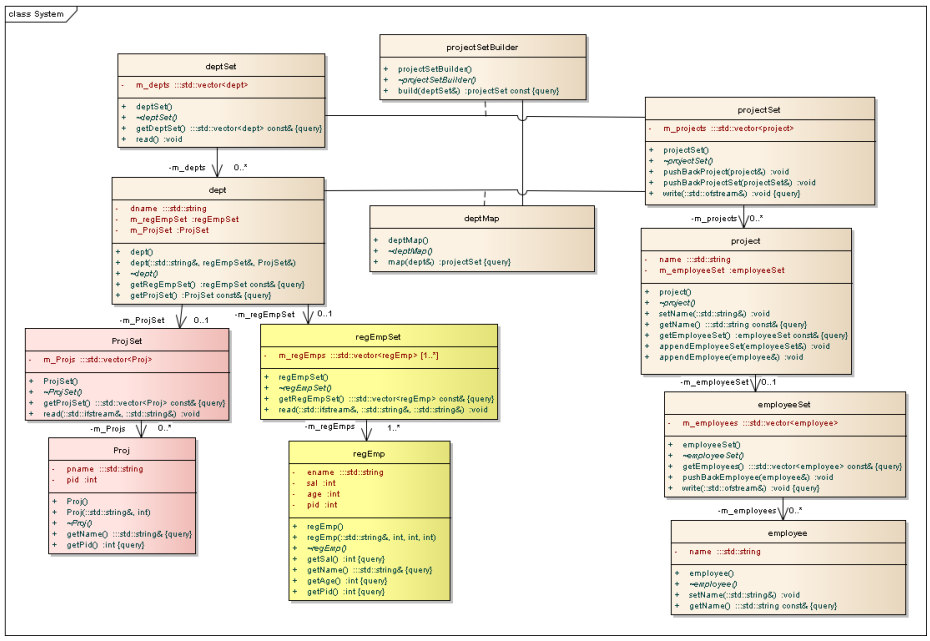

Fig. 11: Mapping with Join and Grouping, the class diagram

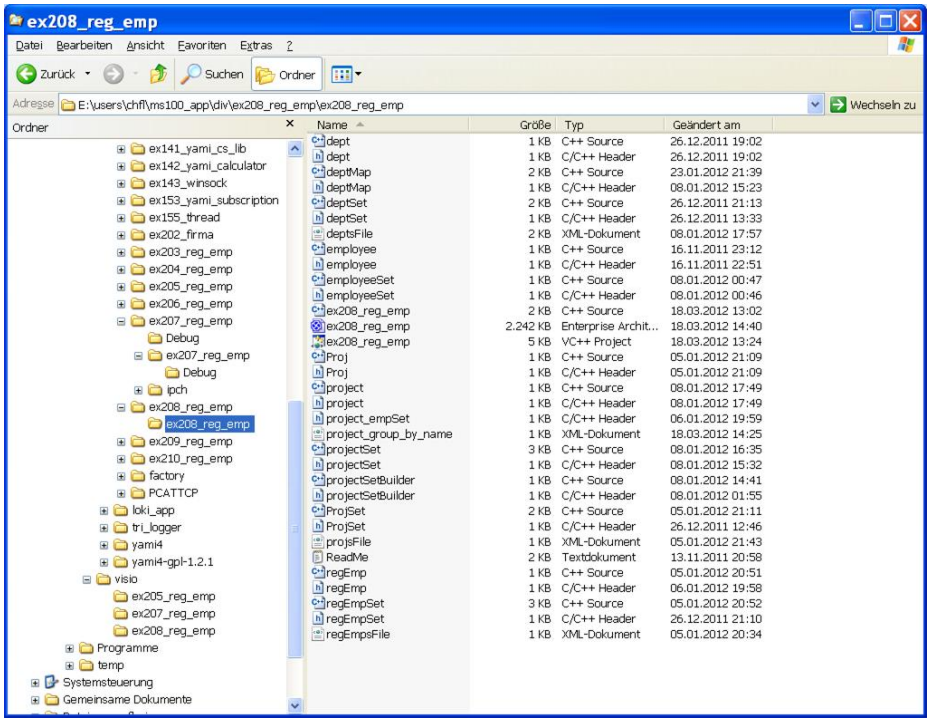

Fig. 12: Mapping with Join and Grouping, the implementation files

The main function, Fig. 13 triggers the mapping. It uses:

- an object of the class *deptSet* named aDeptSet representing the source,
- an object of the class *projectSetBuilder* named builder representing the mapping and
- an object of the class *projectSet* named aProjectSet representing the target.

The lines *11-12* create an instance of the source and read the input from the XML input file, Fig. 14. The lines *14-15* create an instance of the mapping object and use it to map the source to the target. The line *15* executes the mapping. The mapping object, using the function *build* takes as argument the source-object and produces the target-object. The line *18* writes the target-objet to an XML output file, Fig. 15.

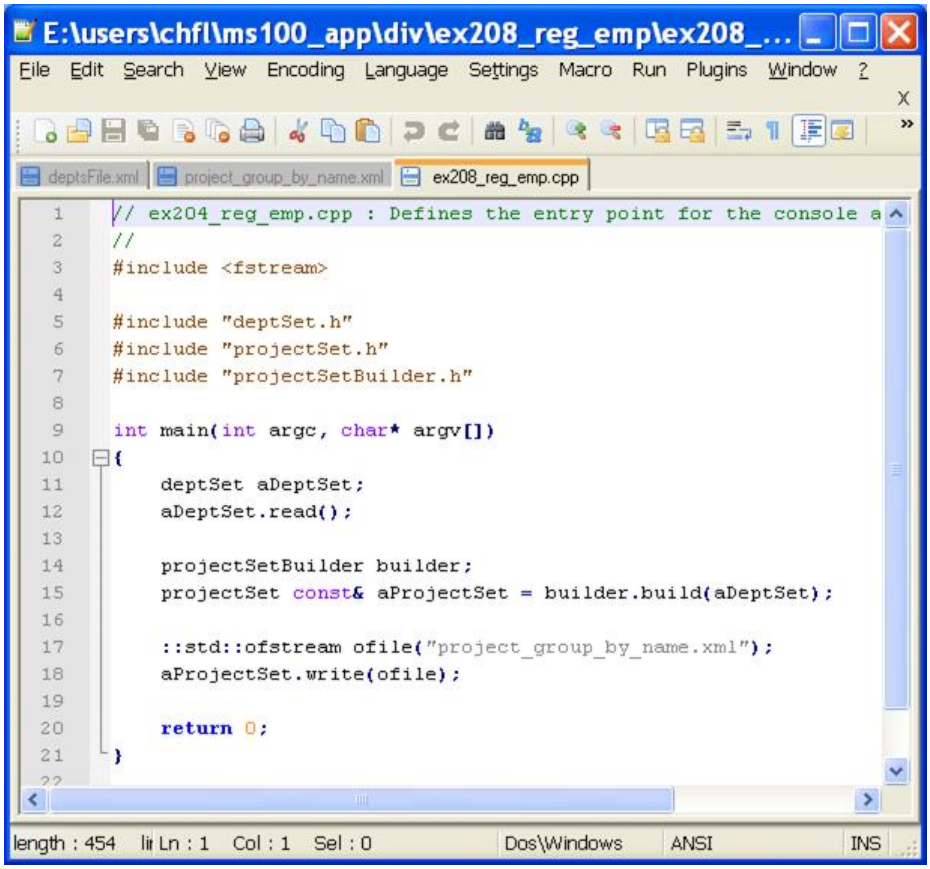

Fig. 13: Mapping with Join and Grouping, the main function

Fig. 14: Mapping with Join and Grouping, the input

Fig. 15: Mapping with Join and Grouping, the output

5 Conclusion

In this document we presented the UMAP implementation. The complete implementation translates all seven Clip [2] main use cases to UMAP. We presented here only the most important three. As future work we plan to implement interfaces and semi-automatic compilation to several targets for additional schema mapping languages and tools.

References

1. Florin Chertes. DBAI-TR-2012-76. Technical report, DBAI, Institute of Information Systems, Vienna University of Technology, 2012.
2. Alessandro Raffio, Daniele Braga, Stefano Ceri, Paolo Papotti, and Mauricio A. Hernández. Clip: a visual language for explicit schema mappings. In *ICDE 2008*, pages 30–39. IEEE, 2008.