

A Generative Approach for Building Data- Oriented Information Systems

Wolfgang Goebel
CFC Informationssysteme Entwicklungsges.m.b.H.
Baeckerstraße 1/2/7
1010 Vienna, Austria
Phone: +43-1-513 88 77 Ext. 22
Fax: +43-1-513 88 62
gow@cfc.atnet.co.at

KEYWORDS:
Software Reusability;
Object-Orientation;
Object Framework;
Software Generator;
Information System.

A Generative Approach for Building Data-Oriented Information Systems

Bernhard Pieber, Wolfgang Goebel
CFC Informationssysteme Entwicklungsges.m.b.H., Vienna, Austria
{pib, gow}@cfc.atnet.co.at

Abstract

In recent years the development of object frameworks has become state of the practice for building data-oriented information systems. This approach has shown some success in improving productivity in building such systems. Using code generators has become a somewhat forgotten technology. But it is just the domain of data-oriented information systems where most of the processing consists of associative data access, that offers great potential for the use of generators. This paper describes our generative approach for building data-oriented information systems and compares it to the pure object framework approach. It shows why a pure object framework approach fails to achieve a sufficient amount of productivity gain in this domain. Our approach takes advantage of patterns which occur in the implementation of data-oriented information systems. The common object framework approach is extended by the massive use of generators which apply these patterns on a model of the application.

1. Introduction

Although many ideas for a more efficient software reuse in order to improve productivity have been presented since 1968 [6], the productivity of building computer programs is still dissatisfying. This situation is mentioned in many papers as the ‘software crisis’. Typical symptoms of the crisis are (i) software is delivered late, (ii) software quality is low, (iii) too much of software engineers’ capabilities are spent on maintenance [5].

State of the art techniques like object frameworks do improve the productivity of building data-oriented information systems but are still not enough to close the gap between the demands placed on the software industry and what the state of the practice can deliver [9].

The ideas to improve software reuse, that have been presented in the last 3 decades to find a way out of the software crisis, can be divided roughly into composition technologies and generation technologies [2]. Looking at

the common software engineering practice of building data-oriented information systems, we find that reuse strategies today are mainly based on composition techniques. The part of reuse strategy usually is reduced to the search for a suitable (class-)library or framework. Even more it can be observed that most research still concentrates on components [5]. Even though the reuse potential benefit of generative techniques has been mentioned in many papers for more than a decade (e.g. [4], [7]), the use of code generators seems to be a somewhat forgotten technology.

The approach proposed in this paper is based on a combination of object frameworks and code generators. Instead of implementing the framework code and the application code by hand, we just implement the framework code and the generator code manually. The application programmer builds a model of the application and starts the generators which produce most of the application code and the relational database table- and index structure. It is exactly the “forgotten” reuse of patterns which makes our approach very productive.

The remainder of this paper is organized as follows: In section 2 we describe the characteristics of our domain to show the potential it offers for the use of generators. The idea of object frameworks is discussed by its reuse potential. Section 3 presents our approach, discusses its benefits and compares it to the common approaches. Finally we will present related work, our conclusions and a brief survey of our future research.

2. The Domain of Data-Oriented Information Systems

The data-oriented information systems (DOIS) we build are used to administrate structured information. Most of the processing consists of associative data access. Usually the information is stored on one or more server(s). Data is entered by the clients using a graphical user interface. This data is then sent to the server(s) which store(s) it in a database. Clients which want to edit already stored information send a request to the server which provides

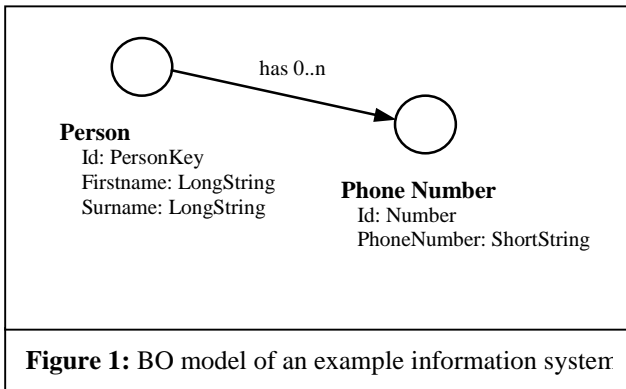


Figure 1: BO model of an example information system

the requested data. We use relational databases for storing the information on the server(s). Examples for such systems are: customer administration systems, product administration systems, etc. The description of such systems sounds quite trivial, there are no complex logics or calculations as in other domains (e.g. real time systems). The complexity arises from topics that can be grouped into 2 major categories:

Technical Issues. Examples for technical issues that have to be solved are: distributed transaction management, data replication between the servers, asynchronous communication of the data, performance issues, etc.

Technical issues have to be addressed in the implementation not for just one concrete information system, but for all systems with “the same architecture”. Therefore these issues are typical candidates to be implemented in an object framework.

Complex Structure of the Data. A medium object-oriented information system as we build, can have up to 500-1000 business object (BO) classes. The BO model is usually custom-made for a concrete information system.

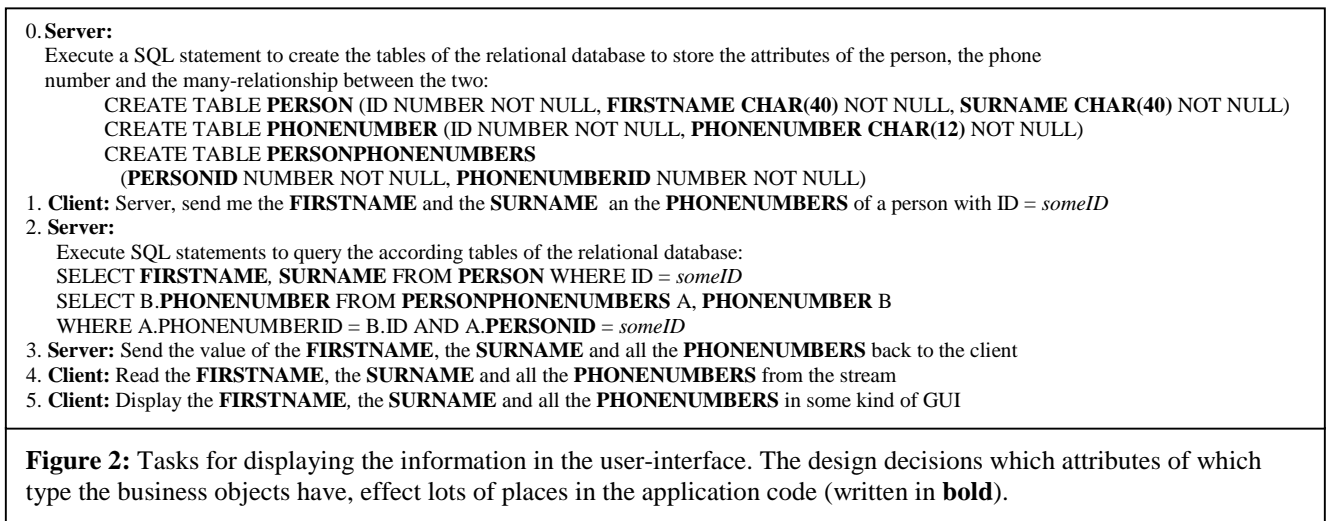
2.1. Patterns inside the Implementation

The following example illustrates the vast amount of patterns which can be found inside the implementation of DOISs: Consider a simple DOIS for processing the first name, surname and phonenumber of a person. Figure 1 shows the BO model for this information system consisting of two BOs, namely Person and Phone Number. If we take a closer look at the structure of the code which has to be implemented to build a DOIS for displaying these 2 BO classes (Figure 2 shows the code in pseudo syntax), we find lots of repetitive patterns which can be grouped into two classes as described below:

Redundant Implicitly of the BO-Model. One portion of knowledge from the BO model is spread redundantly and orthogonally across the application code. The design decisions are scattered throughout the code, resulting in “tangled” code that is excessively difficult to develop and maintain [8]. The design decisions which attributes of which type a BO class has, effect for example the table structure of the relational database, the implementation of the request for the data of the BO class, the SQL queries to retrieve the data of the BO class, the implementation of the response to send the BO class’ data back to the client and the GUI layout to display and edit the attributes. For the example in Figure 2: the knowledge that a person has a first name effects six places in the application code. The BO model is implicitly and redundantly hidden inside the application code.

Similarities in the Implementation of the BOs. Lots of similarities in the implementation of the BO classes of an information system can be found [10], or in other words: it is often “quite the same task” to

- create a database table for the BO class X or to create a database table for the BO class Y
- communicate the attributes of the BO class X or to



- communicate the attributes of the BO class Y
- query the relational database for the attributes of BO class X or to query it for the attributes of BO class Y
- display the attributes of BO class X in the GUI or to display the attributes of BO class Y in the GUI

In our example it is e.g. “quite the same task” to create a table for the BO class Person and to create a table for the BO class Phone number (Figure 2, Step 0.)

2.2. Object Frameworks

The idea of object frameworks as design-level collections of interacting objects captured significant attention recently [9], [11]. Frameworks are a form of design reuse, a kind of domain-specific architecture represented by a set of abstract classes to be subclassed by the customizing concrete application. The framework classes abstract all structures which are common to most applications in the domain. The classes of the application code are implemented “on top” of the framework classes.

Usually an application doesn’t only consist of application code in one programming language. It is e.g. common that the database stored procedures of the application are implemented in a language the database vendor provides. For the purpose of our work we name code of this kind “Other Code”. Figure 3 shows the concept of the framework approach.

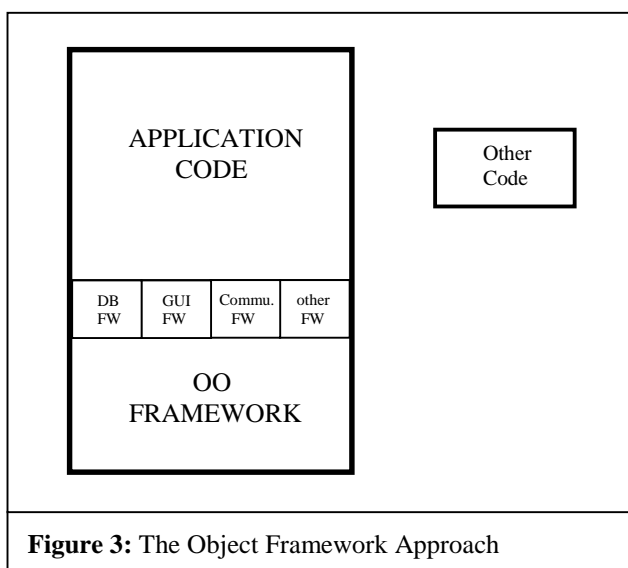


Figure 3: The Object Framework Approach

What does it mean to implement our simple DOIS from Figure 1 using the framework approach ? To achieve a well-structured design (“separation of concerns”) the framework usually consists of a number of separated sub-frameworks (typically database framework, user interface framework, communication framework, etc.).

The classes of the application code implementing e.g. the user interface are implemented on top of the GUI framework while the classes implementing the

communication between client and server are implemented on top of the communication framework. The problem is that the design decisions (like the BO model in Figure 1) cross-cut the structure of these sub-frameworks. More concrete: it is e.g. necessary to implement Figure 2, Task 5.: “Display the **FIRSTNAME**, the **SURNAME** and all the **PHONENUMBERS** in some kind of GUI” on top of the GUI framework and to implement Figure 2, Task 3.: “Send the value of the **FIRSTNAME**, the **SURNAME** and all the **PHONENUMBERS** back to the client” on top of the communication framework. The design decisions are still scattered throughout the application code that is excessively difficult to develop and maintain. Things get even worse if the design decisions are scattered throughout “Other Code” also. The implementation of database stored procedures is for example highly dependent on the BO-model of the application. Developers have to keep the BO-model, the application code and “Other Code” consistent manually.

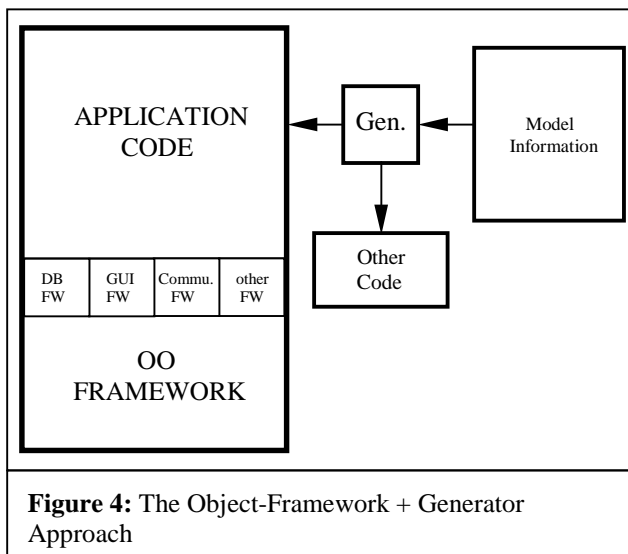
The framework approach fails to take advantage of the vast reuse potential of patterns as mentioned in section 2.1.

2.3. Generative CASE – Tools

Generative CASE-tools are the most commonly used generative technique for building DOISs. Most research concentrates on components [5] and this is reflected by the fact that software reuse strategies today are mainly based on composition techniques. Generative CASE-Tools (e.g. Rational Rose) generate some code skeletons in a target language (C++, Smalltalk) out of the BO-model of the object oriented analysis. Very few tools do any more than very simple headers-only code generation. It seems that many methodologists view the diagrams in their methods as guidelines for human developers rather than as something to be turned into precise instructions for a computer [3]. The generative capabilities of generative CASE-Tools are a nice tool to have rather than a technology for improving the productivity by an order of magnitude.

3. The Object-Framework + Generator Approach

Our approach shown in Figure 4 extends the pure object-framework approach by the massive use of generators. The generators apply the patterns as described in Section 2.1 to a model of the application. Instead of implementing the application code by hand, the application programmer builds a model of the application (writes down the model information) and starts the generators which produce the application code and “Other Code”.



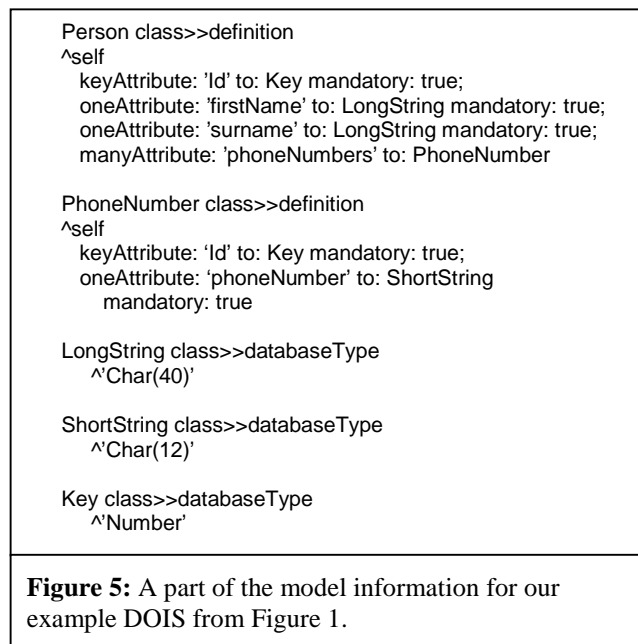
3.1. Application Code vs. Framework Code

The structure of the application code is very different from the structure of the framework code.

The *application code* built on top of the framework subclasses the framework classes for a concrete application. This code is highly dependent on the BO model, highly redundant and shows patterns like mentioned in Section 2.1. Implementing this code can be automated using generators. As the BO-model of one DOIS is usually very different from the BO-model of another, this code itself has a very small reuse factor. What can be reused are the patterns which map the model of the DOIS onto the application code. The *framework code* implements common solutions for the technical issues of our domain mentioned in Section 2. Implementing this code requires human intelligence and it is not worth the effort to abstract patterns like in the application code. We implement this code manually. The framework code has a large reuse factor when implementing several DOISs with a similar architecture.

3.2. Model Information

Model information can be seen as the knowledge which is necessary to generate the code of an application on top of the framework. The more extensive the model information, the greater the amount of application code which can be generated potentially. We've implemented an object-oriented meta-model on which the programmer can program the model information for a concrete application. The base of our meta-model are the meta-model constructs from the "classic" BO model (e.g. BO classes, one-, many-relationships between BO classes). For our example DOIS the BO model is shown in Figure 1. As this is usually not enough knowledge to generate a sufficient amount of code for an application, our meta-



model extends the "classic" BO model. The model information is provided to the generators by Smalltalk definition-classes (Figure 5 shows a part of the model information for our example DOIS).

Our meta-model is designed to hold the following model information:

"Classic" BO Model. Each BO has a class method >>definition like in Figure 5 which defines its attributes and relationships to other BO classes.

Extension of the Business-Object Model

- information if an attribute is mandatory or not. This effects e.g. if a column in a database table is allowed to be NULL or not ('NOT NULL' clause in the CREATE TABLE statement) and if a client is allowed to leave a GUI dialog without filling in a certain edit-field.
- class of an attribute
- subsets of the business-object model which are processed in one user interaction

Typically a client wants to display or edit a certain subset of the business-object model. In our example the client might want to display the lastname and the phonenumbers of a certain person in a GUI. This kind of model information effects e.g. the generated code which sends the data across the network. (not shown in Figure 5)

Other Model Information

- technical information about the used Relational Database Management System.

For the generation of the database tables and SQL-statements on these tables it is important to have the

knowledge of the available data-types and naming conventions of the Relational Database Management System inside the model information.

In Figure 5 for example, the class 'Longstring' knows its database type, Char(40).

Generally the meta-model should be designed that it is:

- well structured (from the developer's view)

It should be easy for the application developer to understand the semantics of the meta-model and to build the model of the DOIS.

- well structured (from the generator's view)

It should be easy for the generators to process the model information.

- non-redundant

This makes the model maintainable. A late change (e.g. a BO class needs another attribute) effects just one place in the model information.

- easy to extend

By getting more and more domain knowledge (identifying more and more commonalties of applications within the domain), it is possible to improve the percentage of generated code vs. manually implemented code by extending the meta-model and generators accordingly.

3.3. Generators

A generator projects a subset of the model information onto a segment of application code or "Other Code". Each generator applies a pattern on its part of the model information to generate its product. These patterns are identified during the design of the framework.

All of our generators are implemented in Smalltalk and most of them generate Smalltalk classes or methods. Usually the generated classes are subclasses of framework classes and the methods are generated for a generated class or a framework class. Therefore the generators are strongly coupled with the design of the framework – designing a framework class which is relevant for a generator identifying the according pattern and designing the generator is the same task.

The Relational Database Table Generator described in Figure 6 gives an example for one of our generators we use to build our applications. It illustrates how the generator uses simple rules for applying the patterns to the relevant model information to produce the database tables for a concrete application. These rules (for each concrete BO-class generate a table, for each one-relationship generate a column, for each many-relationship generate a table which resolves it) and the used set of model information (BO classes, one-, many-relationships between BO classes) are described in more detail below Figure 6 which gives a concrete example what and how the Relational Database Table Generator generates for the example DOIS from Figure 1.

The Relational Database Table Generator is just a simple example for one out of many (more complex) generators. In reality the rules and the necessary model information for creating tables are more complex (for database performance reasons the table structure is also dependent on the expected number of rows in a table, on the kind of queries etc.).

Currently we generate nearly the complete server application code (the table- and index structure of the relational database, the implementation of the code which reads a client's request from the stream, the SQL statements to retrieve and store the data, the implementation of the response to send the data back to the client) and approx. 50% of the client application code (the implementation of the request for the data, the implementation of reading the server's response from the stream, and some part of the GUI layout).

Our domain offers great potential for the massive use of generators. The low-complexity, data-intensive structure of the information systems we build promotes the identification of the patterns, the design of the according meta-model and the implementation of the generators.

However there are structures inside the application code where it doesn't make sense to reuse them as patterns because (i) these structures just appear once and are therefore no patterns, (ii) the pattern is very hard to identify or to implement in a generator.

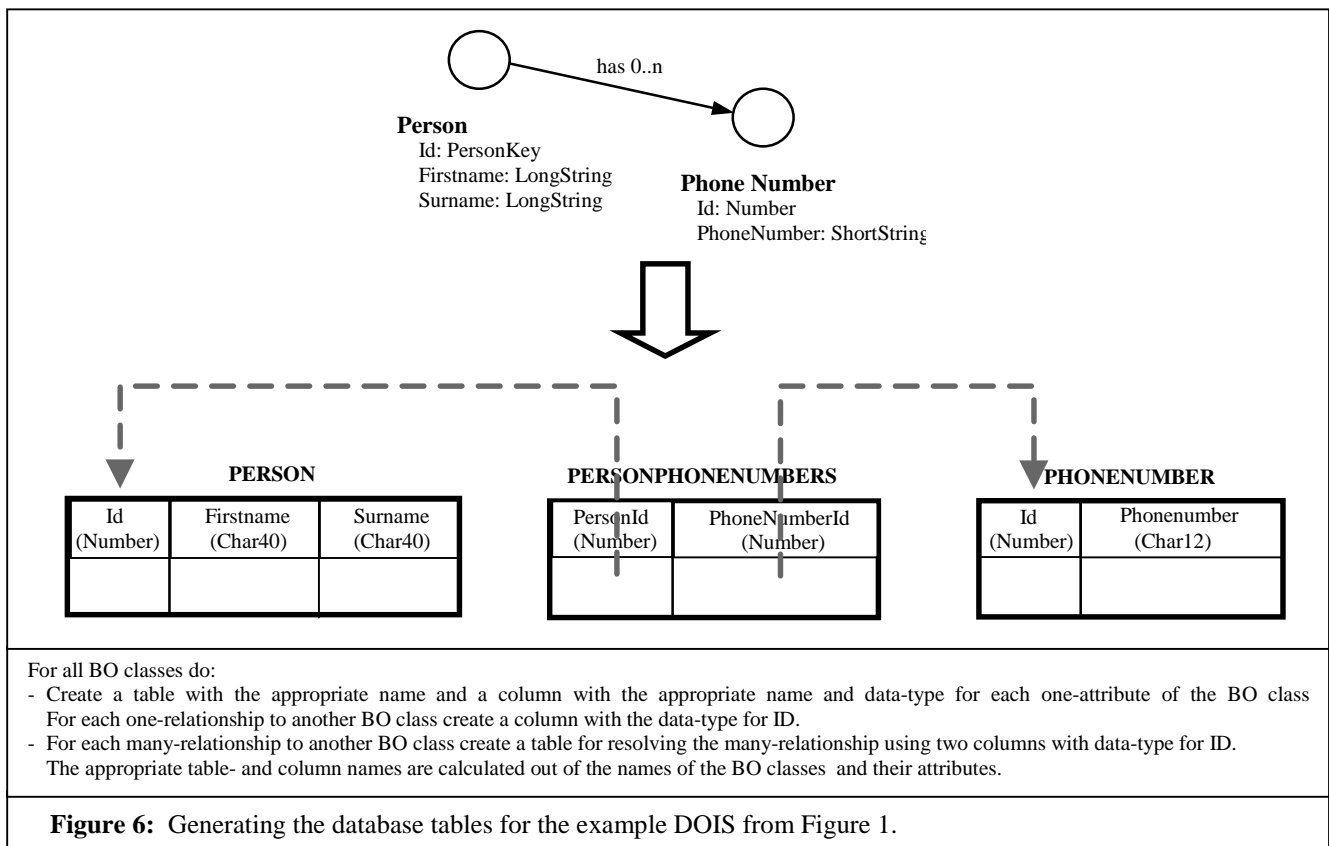
The reason for the big difference of the percentage between the client and the server code which can be generated results from a big difference in the complexity of their tasks. All a server for a concrete application does is waiting for requests, doing the according database queries and sending the responses back to the client. A server for a concrete application X and a server for a concrete application Y do very much the same job. This makes it easy to identify the necessary meta-model constructs and the patterns to be applied by the generators.

Usually the clients of a concrete application provide a much greater variety. There are GUI layouts, logics inside the GUI, etc. which aren't common for a domain but needed just by one concrete application. This fact prevents the use of generators, it is more productive to implement these parts of the client by hand.

[1] defines criteria for the use of program synthesis to have in mind when deciding whether to use generators for a certain part of the application code or not:

- „How complex is the program synthesis process ?“
- „How much effort will it take to identify and represent the necessary domain knowledge ?“
- „How much of the programming task can be automated using the program synthesis system ?“

We only use generators where it makes sense from a productivity point of view considering the above mentioned criteria.



3.4. Benefits of Our Approach

Very small amount of application code has to be implemented by hand. All the standard work is done by the generators. Only the tasks which require human intelligence have to be done by the application programmer.

Model-centered view of the application. The application programmer can concentrate on the real important thing – the model behind the concrete application he is building. It is much easier to lose the eye on the model behind a concrete application if implementing the application code by hand.

Maintainability. The model of the application is explicit and non redundant inside the model information. If a requirement changes late in the development cycle, a small change of the BO model effects just one place in the model information – the application programmer just has to change the model information accordingly and start the generators.

Meta-model keeps model information clear, well structured and extendible. The information used to specify the concrete application on top of the framework is based on a meta-model which keeps it clear and well

structured. The real productivity improvement can only be reached if the meta-model is extendible like in our approach. Simply speaking: the more model information is available, the more application code can be generated. To reach a sufficient amount of generated application code the meta-model must be custom-made for a domain rather than off-the-shelf like in the generative CASE-tools. The extendibility of the meta-model enables the incremental improvement of the amount of generated code when getting more domain knowledge.

Generation of “Other Code”. Usually an application doesn’t only consist of code in just one language. It is e.g. common that the database stored procedures of the application are implemented in a language the database vendor provides. The implementation of these stored procedures is highly dependent on the BO-model of the application. The model information is the central starting point for the generation. Any code can be generated from the model information, not just application code in a certain target language. The generation of the relational database tables for an application from its model information helps keeping the BO-model and the according relation database tables consistent. No one has to take care that the BO-model and the database structure are consistent. It seems possible that we will generate code

segments in other target languages (e.g. HTML) in the future.

4. Related Work

The goal of Gregor Kiczales' new programming paradigm called Aspect-Oriented Programming [8] is to enable programmers to express each of the different issues they want to program in an appropriately natural form. These issues are called "Aspects". Instead of manually implementing the code with all these issues "tangled" inside, the programmer expresses each of the aspects separately. Tangled code is excessively difficult to develop and maintain. The reason why these issues have been hard to capture is that they cross-cut the system's basic functionality. [8] gives the following examples for aspects in a digital library application: failure handling, synchronization constraints, communication strategy. A special form of compiler (generator) called an Aspect Weaver automatically combines these separate aspect descriptions and generates an executable form.

We see aspect-oriented programming very closely related to our work. The approaches have the same intention: to prevent the necessity of implementing non-maintainable code with design decisions tangled into. While aspect-oriented programming is a general concept with some example applications implemented using it, our approach has been used to implement applications in the very narrow domain of data-oriented information systems.

5. Conclusion and Further Research

We presented our approach for building data-oriented information systems. It shows that the pure object-framework approach fails to take advantage of the vast reuse potential of patterns which occur in our domain and therefore fails to achieve a sufficient amount of productivity gain. The problem is that the design decisions (like the BO model) cross-cut the structure of sub-frameworks. In our opinion it is the generative reuse of patterns which has a great unleashed potential to improve the productivity in building applications in our domain. Our experience with generative technology shows us that a computer program usually doesn't only have structures which can be reused by component based reuse strategies alone. Instead the domain should be analyzed focusing on the suitable combination of reuse techniques. A reuse strategy must be driven by the needs of an application program instead of adopting the software development strategy around a reuse program [1]. In our approach the code is clearly separated in a part that can be generated efficiently (application code) and a part that requires "human intelligence" to be written (framework code). This clear separation makes it possible to mix generated code and manually written code in a systematic way.

A prevalent software engineer's opinion is mentioned in [10]: "In the object-oriented world no code generation is needed, we use inheritance instead. The standard behavior in a system is just implemented once in a superclass from which all BO classes inherit. The BO classes implement just customizing parameters". We showed that this opinion is not true for our domain. The generative technique has a number of benefits compared to a parameterized approach (generation of "Other Code", concentration on the model of the application).

We are incrementally improving our meta-model adding more and more relevant knowledge to the model information. Our current estimate is that we will be able to generate the complete server- and approx. 80% of the client application code in the near future. Currently our generators are written in Smalltalk. Generators are difficult to build [4] and the implementation of the generators itself shows patterns which could be abstracted by a generator framework. We believe that building a framework of abstractions for implementing generators will decrease time and difficulty to implement them.

6. References

- [1] S.Bhansali, "A Hybrid Approach to Software Reuse", Proc. of the Symposium on Software Reusability 1995, ACM Software Engineering Notes, August 1995, pp. 215-218
- [2] T. Biggerstaff, C. Richter. "Reusability Framework, Assessment and Directions", IEEE Software, March 1987, pp. 48-56
- [3] T.Church and Philip Matthews, "An Evaluation of Object-Oriented CASE Tools: The Newbridge Experience", Proc. of the 7th International Workshop on Computer Aided Software Engineering 1995, pp. 4-9
- [4] J. Cleaveland, "Building Application Generators", IEEE Software, July 1988, pp. 25-33
- [5] L. Dusink, J. v. Katwijk, "Reuse Dimensions", Proceedings of the Symposium on Software Reusability 1995, ACM Software Engineering Notes, August 1995, pp. 137-149
- [6] M.D. McIlroy, "Mass-Produced Software Components", in J.M. Buxton, P.Naur and B. Randell, editors, Software Engineering Concepts and Techniques; 1968 NATO Conference on Software Engineering, Petrocelli/Charter, Belgium, 1976, pp. 88-98
- [7] S. Jarzabek, "From reuse library experiences to application generation architectures", ACM, Proceedings of the Symposium on Software Reuse 1995, pp. 114-122
- [8] G. Kiczales, "Aspect Oriented Programming", 8th Annual Workshop on Institutionalizing Software Reuse, WISR8, March 23-26, 1997
- [9] H. Mili, F. Mili, A. Mili, "Reusing Software: Issues and Research Directions", IEEE Transactions on Software Engineering, June 1995, pp. 528-562
- [10] M. Rösch, "Generierungstechnik für die Implementierung von Business-Objekten", OBJECTspektrum June 1996, SIGS Conferences, pp. 28-29
- [11] Object Frameworks were a main topic in the October 1997 issue of "Communications of the ACM"