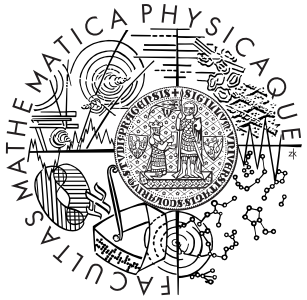
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Marek Hanes

MS SQL Application Development Framework

Department of Software Engineering

Thesis supervisor: RNDr. Michal Kopecký, Ph.D.
Study programme: Informatics
Specialization: Database systems

Prague 2011

I would like to thank my family and my friends for the support during my studies and to thank my supervisor. His advices and support have been invaluable.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date 5.8.2011 Signature

Název práce: Nástroje pro vývoj MS SQL aplikací
Autor: Bc. Marek Hanes
Katedra / Ústav: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D.

Abstrakt: Tato práce pojednává o vývoji databázových aplikací a snaží se najít možnosti zefektivnění nejčastejších problémů. Cílem práce je návrh a implementace modulárních nástrojů, které zjednodušují vývoj databázových aplikací a zabraňují nezkušeným uživatelům použití nebezpečných SQL příkazů nebo výrazů. Příkladem mohou býti insert příkazy bez seznamu sloupců, nebezpečné XPath výrazy a tak dále. Nástroje mimo jiné umožňují

- manipulaci s historickými tabulkami umožnujíc verzování dat a vracení nežádoucích datových změn
- asynchronní a paralelní zpracování SQL příkazů, správu chyb a podporu logování, monitorování změn schémat a též ladění procedur a funkcí

Spolu s prostředky na manipulaci dat umožnují nástoje jednoduché zveřejnění uložených procedur v podobe webových služeb. Nástroje jsou doprovozeny programátorskou a uživatelskou dokumentací umožňující další vývoj.

Klíčová slova: vývoj aplikací, nástroje, modulární design, bezpečné výrazy, omezení

Title: MS SQL Application Development Framework
Author: Bc. Marek Hanes
Department: Department of Software Engineering
Supervisor: RNDr. Michal Kopecký, Ph.D.
Supervisor's e-mail address: kopecky@ksi.mff.cuni.cz

Abstract: The thesis deals with a database application development and tries to find ways to optimize the most common problems encountered. The goal of this thesis is to design and develop a modular framework that simplifies the database application development and prevents inexperienced users from using unsafe SQL statements and/or expressions. The example of such a statement can be the insert statement without explicit column list, unsafe XPath expression, etc. The framework provides among others

- manipulation with history tables allowing versioning of data and reverting unwanted data changes
- asynchronous and parallel SQL execution support, error management and logging support, monitoring of schema changes as well as procedure and function debugging

Together with means of data manipulation, the framework provides the simple way of publishing stored procedure as web service as well. The framework is accompanied by well-written programmers and users guide to allow its further development.

Keywords: application development, framework, modular design, safe statements, constraints

Contents

1. Introduction

Database systems are very interesting systems to deal with. There is much progress in the field. Many commercial as well as free products contain many new features every year. Nevertheless most senior database professionals only use the core features, which have been in existence for 10 years and more. Arguably these new features are buggy in first couple of releases and after this period they may still exhibit performance issues. Whatever the reasons, the final result is that many programmers use only limited subset of the query language and engine features.

From the programmer's point of view, the comfort of working with SQL language is comparable to coding in C language without preprocessor. There are many situations where performance reasons require programmer to duplicate large sections of code. In larger system with 200 tables or more this can be very confusing for inexperienced developers. Moreover this approach tends to make maintenance of the code rather hard, as further changes in one SQL statement need to be propagated to many equal or similar ones spread among the application. There are ways to share common code using views and functions, but these are static in their nature and do not have any context. Database engines allow execution of procedural languages, which help with the tasks where SQL is just impractical[1]. These must be properly and seamlessly interfaced with the rest of the system, since switching from procedural to SQL code and back adds overhead and complexity for developers.

Last, but not least, there is a security aspect. If database systems are to face an attacker directly, vast majority of applications just ignores this issue. Database is used by single account with full rights and there is no effort to provide an ability to recover from the attack.

This thesis analyzes problems specifically in Microsoft SQL Server database engine in versions 2005, 2008 and 2008 R2. The engine allows execution of .NET 2.0 assemblies which can be used for procedural code[11, 1]. Simply put, developer just needs set of tools that create thin abstraction over system catalogue and give him or her ability to enforce patterns and use code templates. It is the author's opinion that roughly 50% of the database code can be pre-generated or permanently generated from system catalogue.

This thesis is based on pre-existing tools author used to manage database code. There are several commercial tools available to handle development[4] and deployment[3]. However, their integration has proven to be somewhat problematic and the solution was far from ideal. These tools are not considered during analysis phase since they solve only singular problems and are not suitable for the goals of this thesis. Author has also incrementally developed SQL-based ad-hoc toolchain to enhance development and deployment before, but the resulting performance and overall design features were not satisfactory. Since no existing acceptable solution was found, a new database framework had to be built to address all previously detected issues.

The next chapters enlist most of the recognized problems, analyze them in more detail and try to provide viable solution. Later on, author will design structure which holds system catalogue data and additional developer metadata,

and providers methods to solve the problems in questions.

The chapter 2 covers initial thoughts about how the development process should look like in ideal case. Negative restrictions are specified as well to clarify the goals. Specification chapter then contains more detailed descriptions of individual use cases and features. Following chapter named design and implementation covers the project layout and responsibilities of each library or utility. User documentation and programmer documentation describe the project functionality from the programmer's and user's points of view. Results chapter compares the initial goals with the resulting implementation.

2. Analysis

Let's put ourselves in the shoes of typical junior developer. This developer has theoretical knowledge of the database workings, understand notions of tables, views and procedures. He or she can write queries of different complexity and interface the database with the next application layer. He or she has to face several problems:

- problem to remember whole system catalogue, dependencies between objects and all constraints
- problem to follow naming rules and coding conventions
- problem to write complex queries, so that they will work outside happy day scenario
- problem being not reliable and leaving code unfinished without reporting what he or she omitted
- problem avoiding injected bugs through copy-pasting SQL code

The situation may seem very bad, but our junior is not alone. There are senior developers, which can write more complex queries correctly and can keep the system catalogue up to defined standards. However these persons are overworked and so our main goal is to catch and precede majority of problems caused by junior developer(s) and increase the productivity. This should be done either automatically or manually, but certainly before they can manifest in production environment. This chapter follows the flow of a new feature requirement in larger system as it is processed, implemented and deployed. Then a feature change request will modify the application since out improvised client changes his mind frequently.

2.1 New feature

Let's have a small example of a simple application feature and try to walk through the development process in its ideal form. A client requires new register of incoming invoices. Each invoice may contain several items. Project will therefore contain two new tables: `invoice` and `invoice_item`. Following steps are required:

- creation of tables - either directly to the database or to the external metabase
- description - semantical meaning of each column and value restrictions
- completion - primary keys, unique constraints, foreign keys and other constraints need to be created according to the specification.

In majority of cases, the last step can be almost completely automatized. Primary keys can be placed automatically on identity columns. Unique constraints can be placed automatically on column sets based on their metadata attributes.

Foreign key creation based on custom database scalar data types will be also automatic. All we need is a standardized definition of table that holds enough information to support these mechanisms. More on this is discussed in next chapter.

2.2 Interface

Direct access to the tables is considered a bad practice for the purpose of this example. The database need to contain stored procedures to list, create, update or delete data in our `invoice` and `invoice_item` tables.

Procedural interface should provide following features:

- strong-typed contracts
- logging
- security awareness
- integration and publishing

Interface contracts, defined in a shared place, will serve as safeguard of what database can return and what application can expect. Contract should contain named results, pre-defined data types, both for input and output. These metadata can be now used in many ways. The application can contain automatically generated wrappers for the interface. Parts of database code can be pre-generated according to the contract and further checked for compatibility. Nothing can break without developers knowing about it. There is no direct access to the tables, so restricting usage of interface should provide sufficient security barrier. Metadata can be used to provide automatic and detailed log of interface usage. This can be later reviewed or replayed for debugging, or user behavior analysis. In case application layer cannot access database directly, interface can be automatically published as a webservice[12].

2.3 Processing

Developers now have to implement the application code and the database code. Let's assume that application code is compiled and not scripted. Developers use compile->fix error approach, until project is consistent and runnable. The same should be enforced for database code. System should force developer to write consistent code, which obeys given contracts and is lexically and syntactically valid. All this should be made locally before modifying the database, since database may be shared with other developers.

In case the developer works in test-driven environment, situation can go even further since his work is not finished until everything is checked out. Strong typed interface should give a solid degree of stability and reduce developer's mistakes to a minimum. Written database code should be checked for unsafe or bad practices, for example unsafe statements, bad procedural flow and invalid scope access. This has one main purpose and that is partially substitute senior developer's review capability. Whole process has to be fast, so developer is encouraged to use the

system during every step of development and not only at the end for checking. The goal is to detect errors early in the development phase and not in testing phase.

2.4 Testing

Feature testing can be done by entirely different person, so there are few additional requirements for:

- merged event logging
- relevant and complete information
- readable presentation

Logging recognizes two types of events. User level actions (button clicks, ...) or execution actions (procedure calls, ...). During a single user action, more than one execution actions can be made and therefore more than one error can occur. These action type logs can exist separately, however, by merging them together, tester and developer gain the overall picture of what really happened during an error. Log presentation should be unified and human readable. Simply put, in whole stack trace, usually the most important part of data is the first file name and line number of the user code.

The goal for a tester is not writing long or meaningful bug descriptions. Developer should recognize the source of the bug by consulting the log. Tester's role is to confirm use case functionality and stress the system to unspecified conditions. Very important aspect is the reproducibility. Database-wise, with a support of historical data, this can be achieved easily by reverting the data changes to a point of failure.

Automated tests are not covered since their availability greatly depends on scriptability of application layer and are usually not written directly against database layer.

2.5 Deployment

Fully developed and tested code now needs to follow these steps:

- review with visualization
- merging and integration
- deployment
- smoke test

Responsible person, usually designated senior developer, needs to push the feature to the production environment. Since final responsibility cannot be put to junior developer, using a code review the senior transfers the blame for potentially bad code onto himself or herself. Review therefore must be accompanied with entire testing history and communication between developers and testers. Then

he or she can check for common weak spots and concentrate on problematic or more discussed code parts. These parts are the most-likely suspects to contain hidden bugs or problems.

Deployment usually concerns more than one feature. Merging them into production code can result in inconsistent code. All these problems need to be fixed and then automatically check against the same rules and restrictions as developer follow. Let's assume that integrity verification is sophisticated and for the most cases the only remaining step before the deployment itself. The system should enforce code quality to the point where only a integrity check and smoke test is required to successfully deploy the application features.

2.6 Feature change request

Even though the client has approved application specification, when real data are entered in our new tables, changes in schema could be required. In our example, let's say that entries stored in `invoice_item` are still quite unstructured and entered values need to be stored in more separate columns.

Only required step should be modifying metadata and supplying description for the changes (ideally directly in the metadata). Since we changed table and procedural interface definitions, whole compilation process should break automatically. Appointed junior developer fixes compilation errors until application and database code checks out.

The goal is to enforce occurrence of compilation and integrity errors for every change in metadata in the largest scale possible. If the change in the database structure will not cause any error in the application layer, the change could remain unnoticed and the old code may work wrongly. Change request implementation should only consist of fixing errors. In case of refactoring or more complex change, this should be the killer feature of the whole framework - allowing any developer to make changes into existing code without having previous experience with it.

2.7 Redeployment

Redeployment follows the same rules as deployment of a new feature, with the exception of existing data. Database developers have to consider migrating existing data from an old to a new version of table structures. This is not always easy and in extreme cases it has to be executed with no application downtime.

It is not a goal of this project to provide seamless data migration support, however when needed, the framework should allow for this extension.

2.8 Alternatives

There are several frameworks available[4, 10], that try to eliminate problems of development of database application. Most common problem being eliminated is unnecessary duplication of similar code, mainly by pre-generating of procedures from table definitions. Combined with the use of schema-binding[6], this provides partial viable solution, but still rather limited.

2.8.1 Do-It-Yourself approach

This variant is then most common. Every common part of database code is manually copied and every new project has to face same problems over and over. This approach does not constrain the development part of the application life as much as the maintenance part. Any changes in a long running and not actively developed project are problematic and costly. Since none is really sure, whether application would break or not, every small change usually requires significant re-testing.

Main disadvantages are higher time requirements and occurrence of many errors in linking application and database layer together.

2.8.2 ORM

In the end we cannot fundamentally change the nature of database engine. Table data area just a table data, not objects, and we have to treat them that way. Object abstractions can be used, but we cannot sacrifice our options.

Main problems with objects are:

- performance
- transaction processing
- versioning and data migration
- security

Especially in OLTP environment[16], table structure does not follow logical object structure. Denormalisation and precomputing of statistical data are main tools how to deal with performance requirements in multi-user databases. ORM principle is not compatible with this approach and it suited better for smaller or single user databases. The framework can be easily modified to support ORM functionality, however it is not the goal, nor it is considered a good practice. Main disadvantages are limited variability and systematic problems like caching data in application server.

2.8.3 Linq

Linq[14] is very useful feature, but also very dangerous one in the hands of junior developer.

Main problems are:

- some part of database code is in database while other parts are in the application
- some popular Linq implementations does not support basic features such as joining[2]
- database code is non-procedural in its nature

Junior developer is always tempted to take shortcuts when working with database. He or she mixes procedural code with non-procedural at the application level, which will almost certainly result in very bad application performance. Linq code is also harder to subject to a static integrity check. Main disadvantages are limited variability and database code scattering.

2.8.4 NoSQL

Quite young are the Non-SQL databases[15]. They target the applications with massive number of active users. This framework is RDBMS oriented and in no way tries to compete with specialized solutions. It is also very unlikely that for example a bank would create information system using key-value storage.

2.9 Summary

Although this project tries to solve many problems at once, which are in more detail described in next chapter, there are few key aspects that were taken into consideration.

Key goals of this project are:

- transfer workload away from senior developer
- reduce repetitive work
- automate common operations

Key features of this project are:

- creation of database objects is accompanied with semantics documentation and functional metadata
- majority of problems must be solved in compile time
- deployment must be accompanied with approval process
- everything should work fast, even on large projects

3. Specification

This chapter describes in more detail several framework features. It starts with the preview of system roles and their use cases and is followed by detailed feature enumeration in its full-featured version. Several specialized features are not to be implemented in the scope of this thesis. They are enlisted to provide the bigger picture.

3.1 System roles and use cases

- Supervisor

  Usually it is a senior developer. He or she is responsible for the project as a whole.

  - project creation - database schema & configuration
  - project modification - mainly structural
  - database architecture design (conceptual model)
  - code rules definition
  - code review
  - deployment - configuration & smoke tests

- Developer

  Usually it is a junior. He or she does not do any major decisions. For the most part, he or she writes the code that cannot be automatically generated.

  - database schema adjustment
  - code writing
  - simple bug fixing
  - writing tests (if available)

- Tester

  He or she verifies whether application follows specification.

  - use case testing
  - special condition testing
  - bug reporting

- Administrator

  He or she manages application on client's side.

  - system errors processing
  - preliminary error type guessing[1]

[1] In case of a configuration type error, it is most likely his or hers fault.

  - recovery from backup

- Hacker

  Hacker is the only person who is allowed to modify the production data directly. He or she has very good knowledge about application.

  - production data modification (according to defined scenarios)
  - application error correction

- System

  - static code checks
  - static configuration checks
  - data backup
  - inter-project communication

3.2 Requirements

Due to pre-existing applications, which were considered to be migrated using this framework, several requirements were made.

- application consists of multiple co-operating databases spread across multiple servers - Due to different backup configurations, pre-existing applications contain separate databases for temporary data like session or task queues, log data, binary material data and main data.

- application interfaces with other framework based application, so the integration must be as seamless as possible. Pre-existing applications are written either in customized ASP.NET or written using special web development framework. They both use different kinds of automatically generated procedure interfaces. The latter partially uses metadata like enums for user inputs.

- configuration of multiple sibling application should be centralized - Several pre-existing applications contain shared database code, which needs to updated on every database once a change is made. It is vital not to lose track of new versions of database code.

- database may be used for log rotation - Applications contain separate database for each quarter of the year. Table partitioning could not be used due to environment storage constrains.

- database have different backup configuration depending on their workload types
  Database workload modes:

  - read-write - normal database operation

  - append-only - specialized database operation, custom backup consideration are required
  - read-only - former append-only database (for example rotated log database)

  Database backup scenarios:

  - no backup - pure temporary database (for example web application sessions)
  - full backed up - used for read-only databases
  - logshipped - different server keeps and up-to-date standby copy of database. Author considers this to be a default backup scenario for most of non-mission-critical databases.

- database code is under revision control, however there is no need for integration. Configuration and schema files are to be organized in such manner, so they do not require any revision control system.
- out-of-process database services must be scalable - In case database requires transaction-less procedural computation (for example manipulation with images), this can be implemented as a database service supported by external processes. Total count of these processes is not limited and can be facilitated by several machines.
- support for MSSQL 2005, 2008 and 2008R2
- support for integrated security[8, 5]
- integration with application framework can be indirect using metadata exports - Direct access to database object collections may be impractical to integrate with other frameworks. In that case an automatically generated metadata export intended specifically for integration is to be used.
- use Glacier design patterns[2] - The main principles are reducing usage of `System` namespace and categorizing assembly types in order to enforce design patterns.

3.2.1 Configuration

Application configuration must honor following restrictions:

- modular design - Each non-essential module should be registered in configuration and replaced later on.
- named references are used to register database instances, database templates, CLR assembly references, database schema references

[2]Further explained in programmer documentation

```
app_schema = file1.xml file2.xml file3.xml
app_clr_assembly = file1.dll

app_database_template = app_schema app_clr_assembly

development_database = app_database_template
production_database = app_database_template
```

- XML format with schema

The configuration should be centralized in order to define configuration master. Application will use configuration directly or use an automatically exported subset of the data.

3.2.2 Logging

Framework logging capabilities must allow following:

- centralized log management and storage - It is not possible to store log data for longer periods of time at hosted environments with limited resources. The solution is moving log data to an offsite location where it can be stored cheaply. It is also possible that offsite can became offline due to unreliable internet connection.
- user events are logged - For user behavioral analysis and detection of user interface performance problems.
- execution events are logged - For bug reproduction and database performance analysis.
- events can be merged together to keep context - For bug report coherence and completion.
- log retention period can be upto 3 years - Mainly due to contractual obligations.
- multiple application share the same log management and storage - The applications share the same support crew and produce similar log data.
- same user can use multiple applications
- only one log entity per user
- development and testing logs are to be separated from production logs - These data do not need to be kept for long period of time.
- exception and error handling has to be standardized - In order to optimize the log data presentation, error types must be categorized and have to contain relevant data in one place.

3.2.3 Object model

Database objects have to be represented in following manner:

- support for all native SQL object types - for compatibility and migration reasons. Framework may be deployed onto existing project and needs to support its structure. Its purpose may only be to support schema integrity checks or to support asynchronous execution.
- support for creating complex object types - specialized or generated objects. User may want to enforce special functionality on subset of database objects. New object type can restrict functionality of base object type or enhance it by generated auxiliary objects (for example see enum types).
- type aliasing - Basis for any advanced functionality is strong typing. User defined types can be used to categorize columns or to allow advanced manipulation.
- enum types with inheritance - A scalar type, derived from char(1). Metadata will contain list of available values with their semantics. Enums can be combined using inheritance. When used in column, an appropriate check constraint will be automatically generated.
- enhanced table
  - automatic history keeping - Auxiliary table will be automatically generated to hold data of previous versions of the records. A trigger will be generated to fill this table. Structure of auxiliary table will be automatically kept up-to-date with the source table.
  - generated utility triggers - Displaying statistics about inserted and deleted data to database console as a visual confirmation, that DML execution modified the table entries.
  - generated utility views - The view of all but large sized columns for ad-hoc queries, so user do not pointlessly transfers large amount data. The view with all XML columns converted to text from as a workaround for linked server restrictions[7].
  - automatic foreign key generation - When primary key is created over column set with aliased types, this can be used as a basis for foreign key generation. Column set forms a type signature, which can be matched against every table. A positive match will mean a reference is required.
  - automatic disjoint identities - For code development and ad-hoc queries, it is a very useful feature to have the identity columns use disjoint sequences. This way, wrong column join or filter will most likely result in empty dataset. It is prevention for typing errors in `update` or `delete` statements.
- enhanced procedure - partially generated

  - generated correct logging implementation - every procedure call will be logged. This will be provided automatically. SQL statement can of course be logged from application layer, this however will not cover direct execution, ad-hoc execution or maintenance tasks.
  - generated correct transaction handling - Transaction may become valid but uncommitable[9]. This requires special exception handling and can be source of problems if not done properly.
  - workspace table editing - Simplified editing of table data. The procedure code fills memory tables with desired `valid` and `invalid` data. Automatically generated code then propagates changes to table entries. This mechanism can be applied to referenced tables with logical columns and can greatly reduce user code complexity.
  - contracted strong type input and output - The generated procedure interface in application layer guaranties correct input. The procedure input will be automatically parsed into generated memory tables and scalar variables[3]. Output is not returned directly, but inserted into generated memory table. The output is returned in a way that allows the procedure random debug output that does not interfere with interface contract.

- partial object loading - Underlying procedure/function code be only loaded when needed. This is crucial in order to effectively access object collection on remote databases. The database code should be loaded only after objects hashes do not match.

- detailed relation and dependency keeping - In order to correctly apply schema changes or to detect references on missing objects, it is important to parse as much dependency information as possible. Given the table column, the full version of this feature should allow listing of all places in the database code where it is being read or written to.

3.2.4 Object model manipulation

- creating objects from XML definition files - There should be several schemas to support database objects.
  - `catalog` - XML schema to store pure system catalog without additional metadata.
  - `master schema` - XML schema to store enhanced tables and generated procedures
  - custom user schemas - In case the application requires custom object types, they must be contained in a separate schema. This schema can be instructed do contain any of types from `catalog` or `master schema` schemas using XML namespace references.
  - creating objects from database system catalog - Database must be able to store all metadata within itself. This allows full-feature comparison of two databases without the need of any XML schema files.

[3] In author's experience, the input parsing errors are by far the most common.

- creating objects programmatically - Used for generated objects. There is a strong chance, that in order to guarantee fast performance for very large projects, a specialized UI application will be needed to keep object collection in memory at all times. Schema manipulation would be done using UI, so there is no need to re-parse every schema file. It is not goal of this to work to optimize work with database schema to this degree.

- introspection - relations and dependencies - Provides basic support for exports of metadata.

- object collection construction hooks - Support for generating objects. The object processors must be hooked into object collection building process in such a way, that does not hinder the performance. The preferred way is to hook the addition of a object to the collection. The processor than constructs full-outer-join-like data structure that compares desired object definitions and existing objects. The changes are afterwards processed to generate missing objects, or report inconsistent objects, or report orphaned objects.

- object collection comparison - Framework has to allow comparing XML schema files with database and two databases together. Both XML schema files and system catalog of database must be complete sources of object metadata. The comparison will consist of full-outer-join by the object names, followed by matching the object hashes.

3.2.5 Changeset manipulation

- reporting inconsistent objects - During process of object generation or object specialization an inconsistency may occur. Most common source is manual alteration of system catalog. These differences need to be reported before a changeset can be generated.

- reporting orphaned objects - When a source of generation object is removed manually from the system catalog, any dependant generated object becomes orphaned. These objects need to be deleted before a changeset can be generated.

- generating SQL scripts to apply changeset to target database

- support for user guided script generation - In case a table or a column is renamed and it already contains data, dropping and recreating the object is not acceptable solution. It is not goal of this project to detect these changes using heuristics. User should be allowed to fill in additional information manually.

3.2.6 Standard library

- extending included system functions

  - string manipulation - Mainly join aggregate functions, which are very practical for both database code and ad-hoc queries.

  - date manipulation - Mainly temporal vector sum. The operation takes set of (`key`, `value`, `from`, `to`) and returns (`from`, `to`, `key`, `value sum`). Returned dataset contains disjoint (`from`, `to`) intervals that form continuous interval starting from minimum `from` ending at maximum of `to`.
  - repository utils - Access to SVN and GIT commands by invoking console applications. Preexisting applications require reading several files directly from repositories. File system access must be reasonably secure.
  - secure filesystem access - Read/write access to file system subdirectories. The access must be restricted for every database. Security configuration should be transparent and well defined.
  - secure network access - Support for client side of UDP, TCP, SMTP, HTTP, and SOAP protocols. Same restrictions as for the filesystem apply.

- session emulation - The database execution has no reasonable notion of sessions. Using `context_info` mechanism, this can be emulated. Main reason is to transfer data between separate commands which belong to the same sql process. Data will contain execution start of set of commands in order to keep same stamp in history tables, user logon data and data for merging log events together.

3.2.7 Database services

The MS SQL database engine in current version has been enhanced with features such as SQL Agent, Database Mail, SQL Broker or procedure publishing. These are however not suitable for universal use and usually require more work than do-it-yourself solution. Database engine supports execution of CLR assembly, however it cannot support any isolated long-running code. Database can exist without application layer and provide functionality to external application using standardized protocols.

- asynchronous execution - This feature can be emulated using integrated SQL Agent. There are however security[4] and practical[5] issues. Framework must allow execution of SQL statements in a background thread. There is also a need for throttling background execution since preexisting applications are able to generate several hundreds of background execution requests in small amount of time. The statements can be generally divided into fast and slow work queues. Number of queues should reflect machine's ability to process concurrent requests.

- network services

  - SOAP service - Simple way of publishing procedural interface using standardized protocol[12].
  - WWW service - Simple way of publishing procedural interface, mainly for use with AJAX[13].
  - TCP and UDP socket services - For specialized services like communication with embedded devices.

- debugging and performance tuning - Microsoft SQL Server Management Console allows tuning of single statements to a reasonable degree. There is however no support for tuning high performance computation functions. In case where SQL scalar or table function is being called in a loop, it is important that function has low running time for all possible inputs. Imagine system in production where all code has already been tuned up. Code reuse caused creation of several nested high performance functions. If users start reporting slow responses, though their reports are subjective in nature, it is necessary to collect objective data and identify which function is really responsible for the slowdown. Since functions are nested together, the only way to do this is collect debug output from every function. Scalar and table functions have no output parameters, therefore this task require quite a lot of manual labor. This problem should be eliminated by calling CLR socket functions and collecting performance or debug data from within procedures and functions without the need to modify any interfaces. Reporting mechanism should not add much overhead to the execution.

- computation and snapshot service - Several preexisting application require specialized search structures which cannot be emulated by database structure. There is need to facilitate these structures, ensuring incremental updates and effortless scalability.

- enhanced monitoring - Support crew, that is responsible for stability of applications, needs to monitor currently running tasks across several machines. Each part of framework, that runs background tasks, should produce

[4]There is no good way to set database account permissions to run SQL Agent jobs without compromissing whole engine instance.

[5]Starting jobs works when job is in started or stopped state. However when job is in starting state, which can be achieved rather easily, `sp_job_start` raises uncatchable exception, that can only be handled using unsafe clr assembly code. For more information see System.Thread.ResetAbort()

interface allowing to examine currently running tasks and integration into monitoring software.

3.3 Summary

The majority of functions will be contained within shared libraries. Many will be apparent only to developer using the framework within a real application. From the outside, framework will be controlled using simple console application. This may induce the feeling of simplicity, however, the total extent of the specification cannot be completed within one man-year. It is not the goal of this thesis to present fully-working application, but to create an extensible skeleton structure which can demonstrate the viability of the design and can perform non-complex database operations.

4. Design and implementation

This chapter will describe several important design features of the project. The most noticeable are object model that is used to represent database objects and used design patterns.

4.1 Environment

For CLR assembly to be deployable in default safe mode, these conditions must be met:

- no unsafe code - restricted manipulation with mainly with arrays
- no access to external resources - for example local and remote filesystems as well as network socket access.
- no static fields in classes.

Using assembly in unsafe or external mode can be achieved, however, this severely compromises database security. Moreover, there are practical problems with unsafe assemblies[1].

Access to external resources needs to be further restricted and concentrated into single database. The framework uses `msdb` system database, which is already eligible for unsafe and external assemblies. There is also a problem of privilege escalation. Dynamic queries executed from SQL or CLR code does not use privileges of a owner of procedure or function. Calling user account must have enough permissions to execute dynamic code directly. This means that unsafe CLR procedures cannot be called directly from safe CLR code. All accounts that use network or filesystem resources need therefore full execute access to unsafe procedures. Unsafe procedures need to maintain their own security restrictions since database interworkings are not able to facilitate the security aspect. These restrictions are stored in configuration file stored outside the reach of any database code.

The last condition is the most severe. As a direct result, singleton classes cannot be used in a classic manner. Each of these classes need to propagated manually to every class that uses them. The framework tries to concentrate most of functionality to `Environment` class. This restriction also applies to all dependency assemblies. In case of third-party assembly, a modification and recompilation may be required.

4.2 Log

Framework supports unified logging into console, database procedure and windows event log. The logging is implemented using `Glacier` library which restricts log messages and exceptions to only own classes and interfaces[2].

[1] If database administrator dettaches database and reattaches it under different owner account, all unsafe assemblies priviledges may be revoked and assemblies need to be redeployed.

[2] `Glacier` library was initialy developed as part of this framework, but later in development was separated into own project.

Log messages and exceptions where designed to be fully serializable and deserializable using XML format. Only own exception classes are used and all external exceptions are being converted to custom structures. These steps are part of a greater goal to create a unified centralized log storage.

4.3 Libraries

Most of the code is written C# since the nature of the code is mainly procedural. Several database routines were predecessors of this project. They were mainly written in T-SQL. Since there were many problems using nonprocedural language[3], a procedural language had to be used.

- `Glacier.Common` - Shared library to substitute or wrap functionality of `System` namespace.
- `Meander.Common` - Shared library containing configuration classes, database objects model classes and many SQL utility classes.
- `Meander.Local` - Provides advanced functionality when libraries are used within desktop environment.
- `Meander.SqlClr` - Provides advanced functionality when libraries are used within MSSQL engine[4]
- `Meander.Console` - Console application that can be used to compare and deploy databases, check database integrity and import existing databases for use with the framework.
- `Meander.EndPoint` - Provides features, that cannot be implemented within MSSQL engine. Asynchronous and parallel execution, dynamic compilation of .NET code, function performance debugging and monitoring.
- `Meander.Tools` - CLR library that enhances database with string, binary, date and many more utilities. Contains all procedures that can be deployed and used in safe mode.
- `Meander.ToolsEx` - CLR library that enhances database with unsafe procedures. Provides access to filesystem and network resources.

4.4 Object model

The model tries to keep as many classes immutable as possible. Object data are loaded from XML files and SQL catalog into the same representation which can be found in code under definition classes. These are immutable and do not contain any dependencies since they are cyclical in nature and it would be impractical to

[3]The biggest problem by far was inability of scalar function to produce sideeffects and modify any data outside the scope. All data must have been returned as output which is very ineffective for complex computation.

[4]MSSQL engine does not allow dynamical compilation of .NET code. This need to be facilitated by external process.

compute them immediately. When all definitions are created, framework builds an object collection. First, definitions are converted into object builders, which are used to compute relations, dependencies, hashes and other metadata. After everything is computed, builders are converted into final objects and a immutable object collection is constructed [5]

Comparing two databases takes as an input two database collections and produces a changeset. The changeset consists of a set of differences between the two collections with computed scripts, prerequisite checks and metadata updates. Scripts must be generated in correct order using object dependencies. This is a non-trivial task, however, the most challenging problem is to support object renaming tables and columns and avoiding dropping and recreating tables. This is problematic mainly because table may already contain data. Although recreating can be done safely, it requires application downtime and more importantly can take considerable amount of time. When database is in full backup mode, operation forces engine to produce considerable amount of data into log.

These problems are addressed using ordered set of rules handling collection difference processing. The framework contains basic set of rules, which are able to successfully create objects into an empty database. These rules allow for simple object modifications. Rules also allow external data input. External data may contain information specifying which objects were renamed[6]. Rules are aware of this information and use them to generate more optimized scripts. Specialized rules are created to process differences that require recreating of objects. Changing data type of a column, which has a check constraint bound to it, requires dropping the constraint, altering the column and recreating the constraint. For every situation framework can contain specialized rule in order to achieve the best result possible. This approach is also applied to avoid dropping and recreating tables. It reduces the necessity of drop only to enforcing the order of columns and few operations on identity columns. External data may also contain information how to transform data between structural changes of the database.

Databases, that contain several hundred procedures and functions, face performance problems during comparison. Loading every SQL module[7] may mean transferring several megabytes over a substandard network connection. It is vital to hash these modules in order to avoid long load times. Hashes are applied to every object and are used to detect differences. Database may also contain virtual objects. These may contain metadata designated for further processing[8]. SQL modules and virtual objects are loaded only after a difference is found.

4.5 Enhanced tables

Tables contain usually these types of columns

- primary key column - usually an identity column

[5]The reader may have noticed that it is impossible to create set of immutable containing a reference cycle. This is achivied by using indirect index references.

[6]supplied by combination of user interface input and heuristics

[7]definition of a procedure, function or trigger

[8]XML definition files are great place to store logical non-database objects. Metadata are then stored in single place and it is easier to maintain single data master.

- logical key column - textual representation of record (for example sequenced invoice number with user defined prefix).
- key column - has function of both primary and logical key column - can be used in simple tables, where there is no distinction between the two.
- attribute column - only holds data. Its value can be changed any time, unlike key column with are usually considered immutable.
- auxiliary column - computed or temporary columns.

Majority of tables can be represented using these column types. Framework can automatically create primary key constraint and create unique index on logical key columns. In certain cases this model can reduce complexity of adding data to referenced table structure. Consider situation:

- database code tries to insert records into master and detail table.
- master table has primary key over single identity column
- master table has set of columns creating logical key
- detail table references identity column of master table
- developer tries to insert multiple records into both master and detail tables

Using T-SQL, inserting into detail table requires a manually written join with master table to match logical key values to previously created ID's in master table. When more than two tables are involved, developer is required to write multiple joins which can be source of many bugs.

Inserting into detail table should be done using logical key of master table instead of primary key. This can be achieved by not inserting data directly. Records will be inserted into pre-generated table variables. Table variable replacing master table will follow the definition of master table. Table variable replacing detail table will follow the definition of detail table with the exception of column referencing master table. This column is to be replaced with columns used in logical key of master table. This method is for the purpose of this thesis call workspace mode.

This way there is no need to join master table when inserting detail table records. Records will be inserted into real tables later, after all memory table are filled. Logical key joins are pre-generated in order to avoid developer mistakes. This greatly simplifies the way how to edit multiple tables in normal form. This approach has been proven very effective to reduce developer's time while adding negligible performance overhead.

Another important operation is `update`. To create optimal statements, every update should check in `where` clause whether it actually modifies the data in order to prevent only touching the data[9]. Creating full-proof comparison expressions for every attribute column, that takes into account nullability is very time consuming.

[9]Touching data means updating table record without any changes. This operation generates transaction log entries and causes triggers to fire over unchanged records, which in turn creates unwanted data in history tables

Figure 4.1: Workspace data manipulation example

Table `person`

id	name	birthday
1	John Doe	1.1.1980
2	Jane Doe	1.6.1980

Table `contact`

id	person_id	type	value
1	1	email	john.doe@gmail.com
2	1	skype	john.doe
3	2	email	jane.doe@gmail.com

Standard commands:

```
--@person_input and @contract_input table variables contain data application wants inserted

insert into [person[ ( [name], [birthday] )
select  pi.[name], pi.[birthday]
from    @person_input pi

insert into [contact] ( [person_id], [type], [value] )
select  p.[id], ci.[type], ci.[value]
from    @person_input pi inner join
        @contact_input ci on pi.[element_id] = ci.[parent_element_id] inner join
        /* element_id & parent_element_id are columns supplied by input parser */
        [person] p on pi.[name] = p.[name]
```

Commands using workspace editing mode:

```
insert into @person_valid ( [name], [birthday] )
select  pi.[name], pi.[birthday]
from    @person_input pi

insert into @contact_valid ( [person_name], [type], [value] )
select  pi.[name], ci.[type], ci.[value]
from    @person_input pi inner join
        @contact_input ci on pi.[element_id] = ci.[parent_element_id]
```

Having one memory table for both new and updated data also eliminates problems with manually checking whether logical record key already exists[10]. Detailed description of workspace editing mode is available in User documentation.

The given example may seem trivial, but the difference becomes apparent when logical keys consist of several columns and 3 or more tables are edited at once. It has been authors experience that workspace mode increases developer's capability to write database code from five to tenfold.

Performance problems become noticeable when memory tables contain several thousands of records. The reason is rather obvious and that is using both primary key and logical key at once in multiple generated statements. Since only one of them can be clustered, engine is required to process data in unsorted form, which causes overhead. While this can be battled using query optimizer hints[11], there is no need to concern the senior developer with optimalisation when contents of memory tables are limited to hundreds of records.

[10] T-SQL contains `merge` statement, however it is impractical to use and has several limitations.

[11] The biggest improvement was achieved using general `merge join` preference.

4.6 Contracts

Communication between several components of framework cannot be direct and requires serialization and deserialization of data. It is vital to concentrate both parser and generator into same class in order to maintain consistency. This is a necessity for remote procedure calls. Framework contains contract classes whose only role is to serialize and deserialize data. These are placed into shared libraries and used by both caller and recipient of remote call. The goal is to break application at compile time when modifications are made. Classes can be easily subjected to unit testing to provide quality assurance.

4.7 EndPoint

EndPoint component is used for features that cannot be executed within MSSQL engine. The database registers services with can in turn be performed by several endpoint instances. EndPoint uses M:N model in relation to databases. EndPoint instances can be easily organized to provide computation farm.

This configuration requires reasonable monitoring support. Each database service and their respective running jobs must be accounted for and must be monitored for irregular behavior. One of the irregular behaviors is job being created and destroyed so fast that, the problem cannot be detected by consulting the list of active jobs. Monitoring therefore needs to work more like a accumulator. Information about database services and jobs need to logged for a limited amount of time to provide adequate output. These data can be also easily represented in graph form.

5. Programmers documentation

5.1 Introduction

As was mentioned earlier, the framework was during the development split into two. The separated part consists of utility and wrapper classes. These are used as storage containers, XML serialization and deserialization, logging and exception processing. The goal is for production code to use `System` namespace as little as possible, since it is author's belief, that many classes within this namespace are not safe to use by junior developers.

The source code is organized using pre-defined coding style and classes are complemented with metadata attributes, used for code ownership[1] and design patterns[2].

Since most of the code is within libraries, test-driven-development model was used. To successfully run tests, you need an instance of MS SQL Server engine with three databases. For more information please see `Common` class in `Meander.Tests` library.

It is worth mentioning that for any assembly to be deployable with safe permissions-set, no static fields can be created, which prohibits usage of singleton pattern. This chapter describes the most interesting parts of the source code. For further implementation details, please refer to the supplied generated documentation.

5.2 Configuration and environment

All libraries and executable are targeted to use Microsoft .NET Framework 2.0 which makes the projects buildable by both Microsoft Visual Studio 2005 and Microsoft Visual Studio 2008. The latter is the only one actively maintained, although reverse-conversion may be performed manually. The projects can be build without any external utilities or libraries. In order to run tests or execute example applications, you will need access to a SQL Server instance, preferably MS SQL Server 2008 or MS SQL Server 2008 R2[3]. In order to successfully run `EndPoint` application, you need to have ability to open TCP ports and access these directly ports from SQL server engine.

Database engine need to contain these databases with full owner access:

- meander_tests_1 - Main testing database
- meander_tests_2 - Auxiliary testing database
- meander_tests_3 - Auxiliary testing database
- msdb - System database

[1] See `Maintainer` attribute

[2] See `TypeCategory` enum

[3] MS SQL Server 2005 contains several bugs in XML processing, which the framework currently does not take into account during deployment.

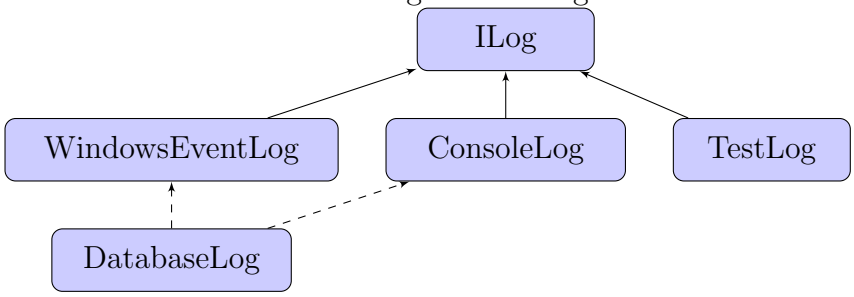
Database names can be changed in `Common` class in `Meander.Tests` project. In default test configuration, MS SQL engine should be default local instance, accessible by `(local)` and EndPoint test will use `127.0.0.1:14331` address.

5.3 Logging

Whether it is a user message or a warning from within shared library, there is a need for common interface to report several kinds of messages. The concept is to create shared library that solves these problems:

- unified interface should allow to output log data to many destinations
- log messages should be accompanied with meaning full data[4]
- exceptions should be logged without losing important information over heterogeneous stacks
- exception classes must be sealed not allow junior developers to create dozens of custom application exceptions
- log representation must be in human readable form, which includes limiting stack to only meaningful entries and highlighting parameters[5]
- some log messages need to be shown to application user and therefore must undergo translation and must presented in UI.

Figure 5.1: Log interfaces



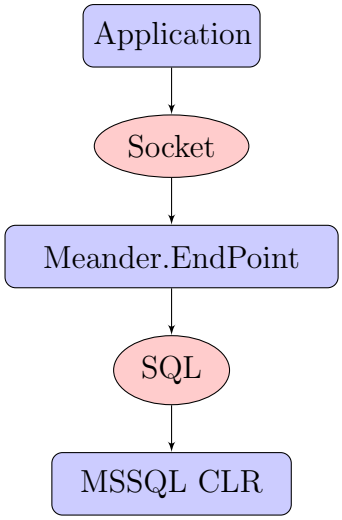
Common `ILog` interface provides ability to report ad-hoc messages and to report exceptions. The log data output can be redirected to system console, where it is printed which restricted character buffer with coloring ability. For services, log can be redirected directly to Microsoft Window's Application event log. This is much better alternative than logging into text files, where each service needs to configure log rotations. To help test-driven-development, log data can be reported by using `TestContext` class, where test is automatically stopped when exception or error occurs. The `TestLog` can be used to expect predefined messages to test rainy-day scenarios.

[4]The default assumption is that containers does not contain sensitive data, therefore `KeyNotFoundException` is allowed to include which key has not been found and in some cases also included list of all the keys in the container for debugging purposes.

[5]When exception occurs, developer needs to see the last non-framework and non-runtime stack entry and highlighted parameters that caused the exception.

In order to provide central log management, `DatabaseLog` can be used, the class serialized log data and store them into common storage. The storage can sustain multiple applications and allows for merging several events together. The class can be configured to fallback log data storage to console or windows event log, in case of a fatal database error.

Figure 5.2: Heterogeneous stack



When exception is thrown in the last node of the chain, in MSSQL CLR code, framework tries to transfer all the relevant data back to the source application. During this processing, the exception must be accompanied with all the stacks and must be serialized and deserialized several times. Transporting exception data over socket is an easy task, however the problem is with the SQL stack. When a exception is thrown, reading `SqlPipe` gives `Meander.EndPoint` only `Message` and `StackTrace` properties of the exception. All the data need to be stored into exception message and therefore the whole exception needs to be serialized into a single string within exception class's creator base class initialization list. For more detail see nested `Input` and `Data` classes. Since constructing exception class is rather difficult, this effectively discourages junior developers to cause exception class polution.[6]

`Glacier` library implements several types of exceptions. Each type has designated purpose, so developer is always able to choose one quickly.

- `NullArgumentException` - for simple input checking
- `InvalidArgumentException` - for input checking, provides description of the problem and the invalid value.
- `UserException` - exception to be shown to the application user
- `CompilerException` - error accompanied with location and debugging information
- `CompositeException` - encapsulates set of multiple exceptions, so more than one error reported during compilation or verification process

[6]When developers are monitored or payed by the number of lines they write, they tend to create exception subclasses for every error type.

- `IntegrityException` - assertion checks within the code, designed mainly to check sanity of computation outputs
- `RuntimeException` - wrapper for native and nested exceptions
- `NotImplementedException` - mark not implemented sections of the code, forces developer to define severity of the missing code
- `NotSupportedException` - mark invalid state, mainly used for default branches of switch statements

5.4 Patterns

Every class in .NET code has been assigned a type category using a system type attribute. These information are processed in order to maintain several code restrictions.

- Immutable - the type contains only read-only data, data cannot be changes in any way and class is thread-safe
- Helper - the type contains only static members
- Container - the type represents long living object, that can change its content over time
- Adapter - the type encapsulates third-party class
- Builder - the type instance is used only for limited period of time.

The helper types are required to be static, with only static members. No constructor can be present. The adapter is a fallback option, upon which no restrictions can be placed since it by definition interfaces foreign code. The builder is supposed to be a temporary type, whose only purpose is to create another object (in most cases immutable or container). The builder references should not be stored in heap and should only live for short period of time. The container marks a dynamic structure, for example `List` or `Dictionary` or any other type that logical represents living object, that changes contents. The immutable marks type to be thread-safe and for read-only access only. All used interfaces and base class must be also marked immutable, all fields must be immutable. In order to access read-only data effectively, read-only interfaces such as `IMap` or `IDictionary` are provided for the common containers[7]. Therefore, they do not allow modifying the data and are also considered immutable and thread-safe. The immutable or container types must be XML writable, so they can be easily used as exception input.

A unit test controls type designations and the metadata. Each class must have be designated a maintainer. The nested classes inherit maintainer from parent class.

[7] As opposed to the `IList` or `IDictionary` interfaces from `System.Collections` namespace. .NET Framework requires developer to create read-only collection manually

5.5 Object

The term object in Meander framework is meant to represent its counterpart in SQL Server. Object types and their properties are designed to be compatible with MS SQL Server system catalog. The framework will in its full specification contain all the parsing and object storage capability of MS SQL engine.

The object model has been through several iterations. The model was rewritten every time a required operation on the model was either difficult or impossible to execute. These constraints have been discovered during the refactoring:

- The resulting objects need to be immutable. The builder type is also an option, but too much effort needs to be done to prevent undesired manipulation with objects. The best found structure to support building the objects consists of an immutable object definition, an object builder and an immutable built object. The definition contains object attributes in scalar form (string)[8], no direct references are created between the definitions. The built object contains direct references to other built objects such as parents, containers, children, members, forward or reverse dependencies. These references are constructed using a builder class. The object collection build is described in more detail in next section.

- For usability reasons, object references must be in form of direct properties. Indirect referencing using object names or indexes, can leave the collection potentially broken.

- The object needs to storage additional metadata. Almost every object type contains data, that cannot be stored within the system catalog. This data can be stored using MS SQL's external attributes, however because of a bad performance, a custom table is used as storage.

- The objects need to be hashed. Certain object types need to be loaded only partially, in order to prevent unnecessary delays. The procedure or function definitions should be transmitted after it has been confirmed, that content is different[9].

- There is no need to handle generated and user-created objects differently. Only difference is that the user-created object hashes user inputs, where generated object hash metadata of referenced objects and the version of the generator.

- It is bad idea to allow for object collection modification. If the object collection were of a container type instead of immutable type, either the collection would allow to exist in broken state when it is not usable or every modification would have to be executed in precise order and the logic would have to be too complex. It is impractical to implement object model where any reference can be broken and every part of code must check for validity. This would greatly hurt the extensibility, because it would be very difficult to implement object handling correctly.

[8]For example, when a column definition is created, it only contains indirect string reference to its parent table.

[9]In larger applications, the size of sql source code can reach upto several tens of megabytes

- There needs to be a virtual object. The object specialization may require storage of object types that does not exist in MS SQL object model. For further detail see class `ProcedureBlockModule`.

- The construction of object definitions from XML schema files and SQL catalog differ in several ways, every user-created or generated object's definition must be fully constructible from XML schema file, SQL catalog or programmatically during generation.

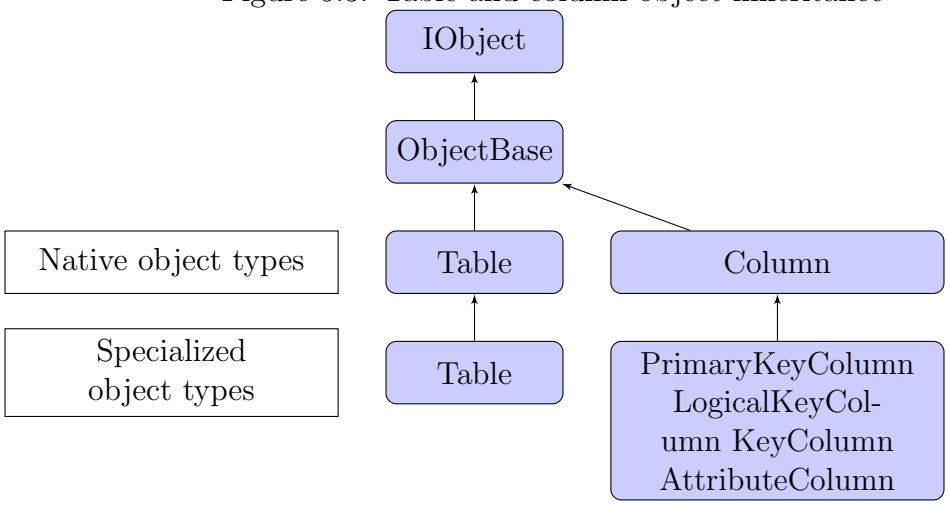
- The object references cannot be resolved all in one pass. In order to resolve dependency references, all objects must have already resolved parent, container, member and child references. Since several object types add their dependencies into hash input, the hashing can be performed only after all the dependencies have been resolved.

- During the construction of the collection, an error may occur on object data. Either some object is missing altogether or its definition does not match what is expected. These errors cannot be solved programmatically, since during the building of object collection, the collection has no notion of created changes in SQL database. Its only role is to create a coherent set of objects, regardless of the further use.

Object names have been standardized using `ObjectName` class. The class is a utility class to safely handle qualified object names and enforce case-insensitive comparison for object names. Object identification is accompanied with `ObjectType` enum value. The combination of type and name forms a unique key, although it is applicable only to columns and parameters of table or inline table functions. Most objects are required being uniquely identified only by their name.

The collection build process allows plugins. These are called object processor are used to inspect, validate or export object data, or to append new objects to the collection. More about processors is described in next section.

Figure 5.3: Table and column object inheritance



Note: The object type PrimeryKeyColumn may be confused with index column object type (`sys.index_columns`), but the sub-class specifies semantic use of the column.

The object class inheritance provides ability to freely specialize object without having problems with further use. When the framework later tries to generate change scripts between the two object collections, generally no specialized change-set code is required to support existence of specialized objects.

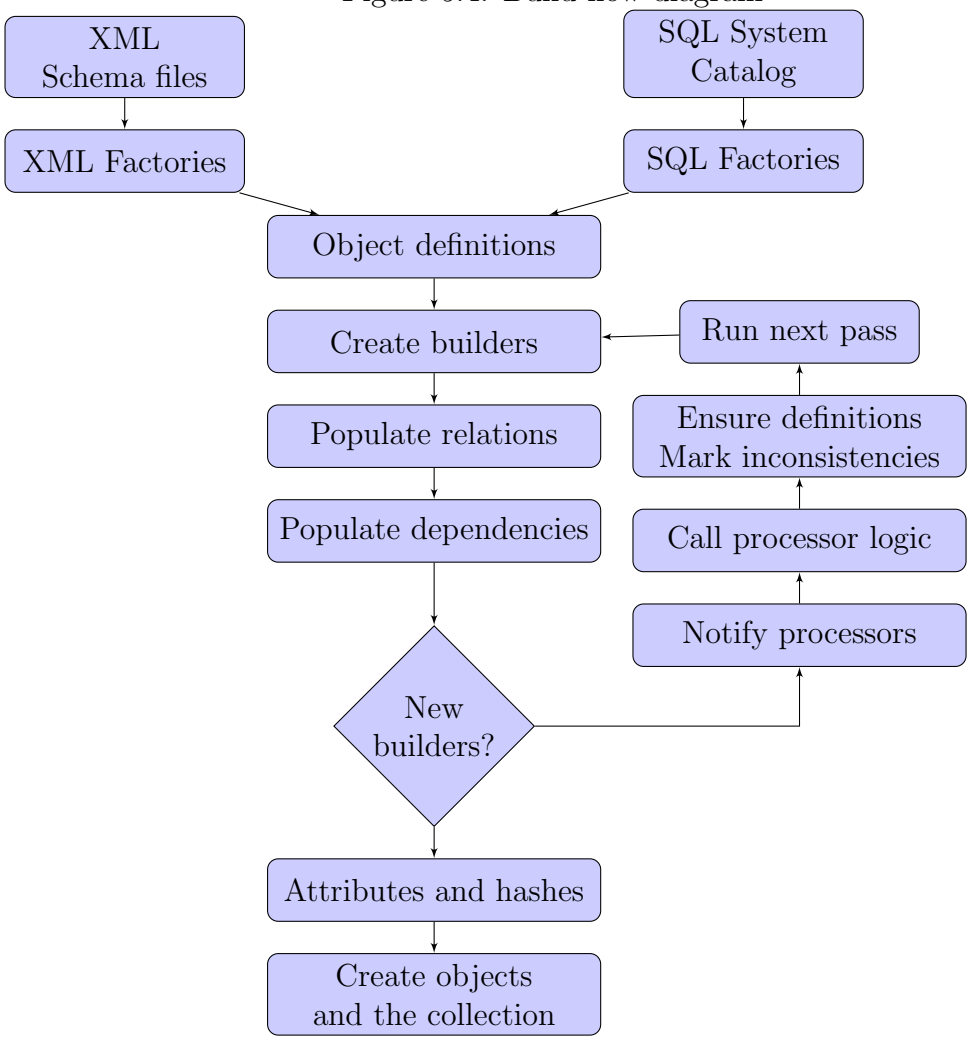
The framework works with two set of objects. Their internal names are `Catalog` and `MasterSchema`. The `Catalog` objects follow the SQL Server engine object model as closely as possible in order to support the whole model. The `MasterSchema` objects are optimized for rapid application development and provide usability and productivity. There are several occasions where `MasterSchema` is not able to hold a specific object configuration[10]. `MasterSchema` is implemented by sub-classing the `Catalog` objects. Developers can create their own schema sub-classes or replace object system altogether. Only requirement is the support of `IObjectDefinition`, `IObjectBuilder` and `IObject` interfaces.

5.6 Building objects

Every object type is subjected to the same building processes. The process takes for an input set of object definitions. The result is immutable object collection.

[10]For example having primary key non-clustered and logical-key clustered

Figure 5.4: Build flow diagram



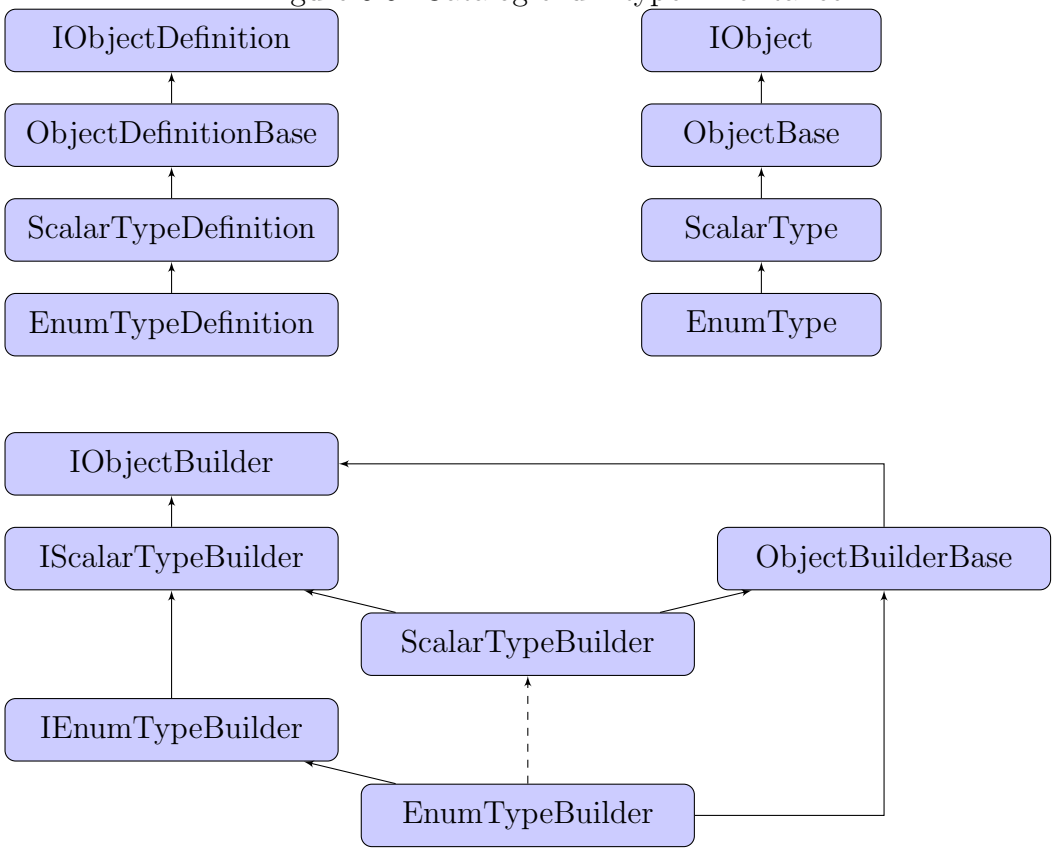
This flow satisfies conditions defined in section 5.5. The processors handle object builders which have already successfully resolved relations and dependencies. Processors are designed, not to cause any performance problems and have O(n) complexity with relation to the number of new builders in each pass. Processors internally create full-outer-join-like structures, which can be marked dirty if a change occurs. After notifications about new builders are complete, processor logic handles all dirty entries. When processor generates new definitions, these are converted to new builders and processed as any other new builder. For XML input, this means that processor handles generated objects twice. In first pass, the processor generates definitions of missing generated objects. In second pass, it verifies that constructed builders match desired definitions. For SQL input, generated definition are already supplied by SQL factories. This time, usually only verification pass is required.

5.7 Custom objects

Custom object can be created easily. There are several components required to achieve this goal:

- definition sub-class
- builder class
- object sub-class
- XML factory support
- SQL factory support

Figure 5.5: Catalog enum type inheritance



The inheritance can be seen on the example. Definition and object follow direct inheritance, whereas the builders follow different inheritance model. The builder sub-classes manually create base builder class and reimplement all required methods. In most cases, the methods just call the equivalent base builder methods. In few cases, the subclass restricts available features. This scenario may have been supported by direct inheritance, however, it become apparent that many programmer's errors occur when builders are created using copy-and-paste[11]. Generic arguments supplied to the `ObjectBuilderBase` guarantee, that any of these errors are detected during-compile time.

[11]New builder needs different definition and produces different object. The definition and object class types are referenced multiple times whithin the builder code and one or two referencies always remained unchanged, without a compilation error

5.8 Custom processor

The processors can perform multiple functions:

- object introspection
- object generation
- rule enforcement
- metadata export

The processor must implement `IObjectProcessor` interface and supplier the specialized constructor. For future compatibility reasons, processor should not use builder classes, but use builder interface instead. Since builder's inheritance is interface-based, this is the only way to ensure, that processor will correctly recognize every new user created object types.

The processor is given `IObjectProcessorOutput` interface to produce output. It can report warning or errors, create new and check existing builders using `EnsureDefinition` method and report orphaned[12] or inconsistent objects[13]. When working with generated objects, it is important to take into account, that not all relations or dependencies may be resolved successfully.

The processor may be used to generate additional source code. The source code is used to simplify interfacing developed application with the database. The source code may either be directly created from processor, or it can be performed into two phases. First phase is the export of required metadata and second is the external source code generation. This allows for wide range of integration possibilities.

5.9 Building changeset

When source and target object collections are constructed, they can be compared to enlist differences and generate list of actions to propagate differences from the source onto the target. The differences are detected on the object level using hashes. The differences are then aggregated to the most common object parent and form changes. The parent-child relation is defined for example between tables and columns or between functions and parameters. Decision between container-member type relation and parent-child relation is based on available the DDL statements. Create table statement also creates columns so the relation is parent-child, whereas the table trigger is created separately and table is considered not changed when trigger is added or removed, so the relation is container-member.

Changes contain parent object from the source and the target collection. Change can be converted into SQL scripts that modify target database, however and order of DDL statements must be enforced. Changes need to topologically sorted honoring the fact that sort order is reversed when object are being dropped in contrast to the order of object creation.

[12] For example, when table with dependent history table is manually deleted, the history table becomes orphaned.

[13] For example, attributes of generated objects are manually altered.

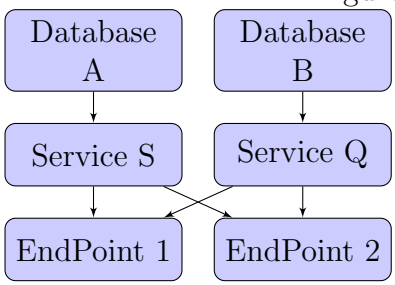
Change does not procedure only DDL statements. A prerequisite check may be required to successfully execute the statements. For example setting nullable column to not-nullable column requires checking where any nullable data exist in the table. Prerequisite checks may be applicable for unique index creation or manipulation with enum types.

It has been mentioned that generating correct changes may require user input to support operation such as renaming the table, renaming the column or more complex table structure conversion. The class structure has been designed to support this requirement, however no workable implementation is ready.

5.10 Endpoint service model

The database service model only needs to solve one problem - the security. Consider following configuration:

Figure 5.6: Endpoint configuration



Database A and B are considered hostile. They must not be able to read each other's table data. This can be achieved using database permissions. The databases must also not be able to access services of the other database. This needs to be enforced by using safe RPC call accesslists and secret access tokens. Access token is pre-shared key between the database and the endpoint. It is not readable from the outside and can serve as a sufficient security measure. RPC call accesslists is only an additional security mechanism in case the access tokens were compromised.

5.11 Workspace table editing

As was mentioned earlier, it is preferable to edit table indirectly using specialized memory variables. Standard `insert`, `update`, `delete` or `merge` are replaced with insert into valid and invalid memory tables. The valid memory table contains record data that should be inserted or updated. The invalid memory table marks records for deletion. Full implementation of workspace edition requires solving these problems:

- nesting - When editing multiple referenced table in one code block, The primary key columns must be substituted with logical key columns to an unlimited level. While it is easy to generate memory table definitions, it is not easy to generated DML statement fragments that map logical key columns back to primary key columns.

- KeyColumn with identity - When edited table does not contain separate primary and logical key and the column holding key values is a identity column, measures must be taken to support workspace editing. An additional virtual logical key must be supplied to successfully match master and detail tables with the newly generated identity values.

- performance - Workspace editing is best suited for smaller input sizes. In case you require editing of several tens of thousands records, an automatically generated code may add significant overhead. The problem lies with the necessity to re-sort data when joining for primary key values. No one can guarantee that primary and logical key will preserve the same order. Merge join hint supplied to every statement has been proven a viable solution, however processing that amount of data within user initiated actions is considered a bad practice, since the whole application seem slow and unresponsive. The action should be either executed in background, or a table design should be considered.

6. Results

This work achieved to provide a functional set of developer tools. The current implementation supplies:

- fully configurable and extensible environment
- working object model that represents larger part of SQL database.
- CLR assembly deployment
- working mechanism to compare and deploy databases
- error management and logging capabilities with log merging
- automatic foreign key generation
- history tables
- generated procedures with strong-typed input and output contracts
- simplified workspace editing mode
- working interface with application level
- rudimentary dependency detection
- basic asynchronous execution
- scalable multi database support
- practical T-SQL tools

Much effort was dedicated to the purity of the design. It has been the previous experience of the author, that if there is something design-wise wrong with the framework project, it will certainly turn into blocker problem in near feature. Not all functionality has been transferred from the previous tools chains and is yet to be implemented. During the development several priorities have to be shifted. The framework needed to be functional for use for a real project.

Following functions have been omitted and their implementation is pending.

- advanced functionality in generated procedures
- full-featured workspace editing mode
- network services
- T-SQL lexer and parser for complex statement and expression verification
- full CLR support

6.1 Effectivity

Creating database objects using local XML files is much less time consuming than clicking into SSMS[1] or typing DML commands manually. SSMS is easily usable for small databases, but once the database contains hundreds of tables and procedures, it becomes difficult to work with. The gap becomes even greater when database is on a remote location, where SSMS performs very badly due to long latencies.

The most common error in SQL is typing error in object names. This is partially eliminated by code completion in SSMS, however, the IntelliSense system becomes unusable for combination of large database(in number of objects) and long scripts. In current version it contains several bugs, where it either stops working altogether when XML functions are used, or it provides the user with internal system objects instead of user created objects. It is best suited for ad-hoc queries, but not for procedure development. MSSQL engine does not compile procedures until all objects exist. If you misspell table name, you will find out that procedure is broken only by running it. You can create schema-bounded procedures which compile upon creation and modification, but it is not recommended during development, since it adds too much overhead when refactoring is needed. The framework detects breakage before deployment and importantly, checks the older code if it has not broken. Furthermore invalid joins, invalid expressions can be detected, which are not detected by intellisense nor schema binding[2]. Their occurrence is quite frequent and they are very hard to find, especially when database is full of similar id values.

6.2 Adoption process

There is definite learning curve needed to work with the framework. An initial configuration is needed, best executed by senior developer or DBA administrator. Many database developers are accustomed to develop procedures directly against database engine. The procedure is being written until it contains no lexical errors. After that, it is being debugged by repetitive executions. This process needs to be adjusted by moving development away from management console. The procedure should reach database only when it is in usable state.

The framework is best suited for new projects. Integration with existing code is available using import, but using advanced framework features may require refactoring. Imported schema files usually require additional metadata like descriptions and comments.

The framework is designed to interface with custom web-based framework. This has been achieved using metadata export. This way it can integrated with any kind of project. Several applications are already being written using this configuration and so far no major problems have been reported by the developers.

[1] Microsoft SQL Server Management Console

[2] Once the T-SQL parser will be implemented

7. Conclusion

It is the author's opinion, that the thesis fulfilled its goals. The resulting application is able to support database development and is able to interface with application layer. Although the work is far from completion, the project is on the good way to become a stable part of database developer's toolset.

Compared to the other database frameworks, the main difference is the philosophy. The database layer should remain responsible for transactional integrity and it should not be used as mere storage container. The framework considers database layer to be a complete and self-sustaining part of an application. It should have a well defined interface with the rest of the application. This all is possible and applications can go even further. Developers can create customized database objects and database can take over data-bound functionality from the application layer.

The framework can be extended in many ways. The design allows for the object model to be substituted completely in order to be targeted to other database engines. The extension can even further and create unified representation of database schemas with full procedural support. There also room for improvements in user interface. Mechanisms used within the framework can be used to provide superior code-completion.

A. User documentation

A.1 Introduction

This framework is allows you to:

- manage multiple database's source code
- share database code between databases
- improve database code quality and stability
- speed up database development

This manual will help you set up initial configuration and show you how to use several of the main framework features. It is designated for application and database developers. This manual presumes you are familiar with creating MSSQL databases as well as XML files with schemas. Within the manual you will be instructed to create XML files, modify XML files and create MSSQL databases and access MSSQL databases. Before you start, please read first the glossary section. The manual is accompanied with demo application. Several features will be demonstrated using the demo as an example.

A.2 Glossary

- Schema file - a XML file containing full or partial definitions of various database objects
- CLR - Common Language Runtime, mainly refers to integrated .NET execution support within the database engine
- Assembly - a library or executable written in any .NET language, usually ment for usage within the database
- Database template - set of schema files and assemblies, that provide complete set of database objects
- Instance - a database running in MSSQL engine.
- System catalog - set of database views, that describe database objects.
- Schema - refers to a set of objects sharing the name prefix[1]. It can be in form of system catalog subset or union several schema files.
- Catalog schema - set of framework objects, that represent database objects as closely as possible

[1] Level 1 object name

- MasterSchema - set of framework objects, that represent database objects in a way to provide faster development. Object definitions require less input data than catalog schema and they derive the rest of object attributes automatically, which in turn restricts a few options.
- SQL module - definition of stored procedure, scalar function or table functions
- SQL module header - region of a SQL module from `create` or `alter` keyword upto `as` keyword.
- SQL module body - region of a SQL module from `as` keyword upto the end of SQL module.

A.3 Demo

A console application written in C#, that exhibits several framework features. Demo project directory contains these files:

- `Configuration.xml` - main framework configuration file
- `App/App.csproj` - project file
- `App/Constants.cs` - helper project file
- `App/Program.cs` - main project file
- `App/Interface.cs` - automatically generated database interface
- `Invoices/Schema.xml` - definition of database objects
- `Invoices/partner_store.data.sql` - SQL module fragment
- `Invoices/partner_store.output.sql` - SQL module fragment
- `Invoices/invoice_new.data.sql` - SQL module fragment
- `Invoices/invoice_list.output.sql` - SQL module fragment
- `Invoices/invoice_detail.output.sql` - SQL module fragment
- `Invoices/invoice_delete.data.sql` - SQL module fragment

Several other framework files are referenced and are located in parent directories. Listed files demonstrate configuration of single database template. The template exists in three instances: `demo_dev`, `demo_test` and `demo`. Database procedures can be accessed using automatically generated interface. Application itself does not contain any logic. Its main function is only to execute several procedure calls.

Object definitions can be within `Schema.xml`. File declares database schema `invoices` which contains 3 tables and 4 procedures. Procedure bodies are partially generated. Non-generated fragments can be found in SQL files in the same directory.

In order to successfully run demo application you need to:

- create empty `demo_dev`, `demo_test` and `demo` databases on `(local)` server.
- build release configuration of Meander solution - located under name `Meander-vs10.sln` in main directory. Demo application is built within the solution.
- deploy database code using following commands from within `Schema/Demo` directory:

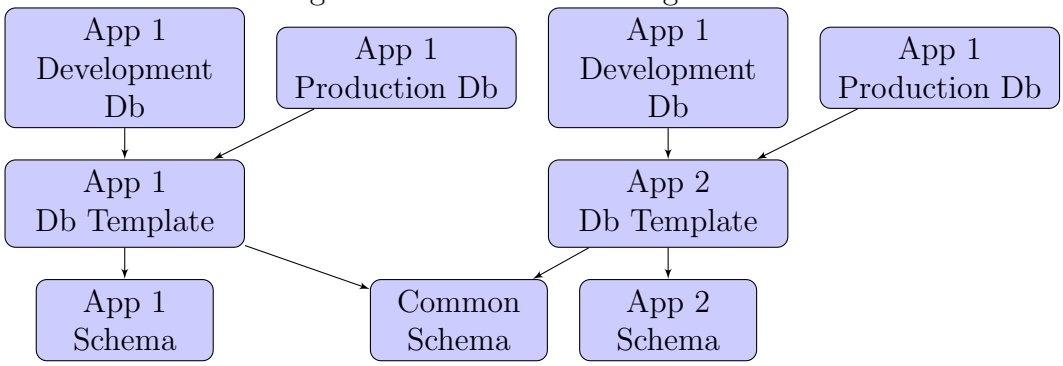
```
..\..\River.Meander.Console.exe . deploy all
```

- set Demo project as startup application and run the project.

The details about configuration and usage of console application are available in next sections. Sections contain several snippets from the demo project.

A.4 Configuration

Usual framework configuration contains following relations:



To create a initial configuration you must:

1. create own schema file - will contain information about database objects
2. create configuration XML file - will contain information about databases
3. reference framework schema files and own schema files in configuration

For initial configuration, please create an empty schema file:

```
<?xml version="1.0" encoding="utf-8" ?>
<c:Schema Name="invoices" xmlns:c="urn:river:meander:catalog:v1" >
</c:Schema>
```

Next, please create a minimal configuration file `Configuration.xml`:

```
<c:Meander.Configuration xmlns:c="urn:river:meander:configuration:v1">
  <c:Instance Name="demo_dev" Template="demo" ServerName="(local)" />

  <c:Template Name="demo">
    <c:IncludeSchema Reference="Meander/Types"/>
    <c:IncludeSchema Reference="Meander/Build"/>
    <c:IncludeSchema Reference="Meander/Tools"/>
    <c:IncludeSchema Reference="Meander/EndPoint"/>
    <c:IncludeSchema Reference="Meander/Log/Frontend"/>
    <c:IncludeSchema Reference="Meander/Log/Backend/Storage"/>
    <c:IncludeAssembly Reference="Meander/Tools"/>
```

```
    <c:IncludeSchema Reference="Demo/Invoices" />
    </c:Template>

  <c:SchemaReference Name="Demo/Invoices" FilePath="Invoices/Schema.xml" />

  <c:SchemaReference Name="Meander/Types" FilePath="../Meander/Types.xml"/>
  <c:SchemaReference Name="Meander/Build" FilePath="../Meander/Build.xml"/>
  <c:SchemaReference Name="Meander/Tools" FilePath="../Meander/Tools.xml"/>
  <c:SchemaReference Name="Meander/EndPoint" FilePath="../Meander/EndPoint.xml"/>
  <c:SchemaReference Name="Meander/Log/Frontend" FilePath="../Meander/Log.Frontend.xml"/>
  <c:SchemaReference Name="Meander/Log/Backend/Storage" FilePath="../Meander/Log.Backend.Storage.xml"/>

  <c:AssemblyReference Name="Meander/Tools" FilePath="../../Bin/River.Meander.Tools.dll" PermissionSet="Safe"/>

  <c:RegisterAssembly Name=".">
    <c:RegisterFactory ClassName="River.Meander.Catalog.SqlFactory" />
    <c:RegisterFactory ClassName="River.Meander.Catalog.XmlFactory" />
    <c:RegisterFactory ClassName="River.Meander.MasterSchema.SqlFactory"/>
    <c:RegisterFactory ClassName="River.Meander.MasterSchema.XmlFactory"/>
    <c:RegisterProcessor
      Name="AutomaticForeignKeys" ClassName="River.Meander.Catalog.Processors.AutomaticForeignKeys"
    />
    <c:RegisterProcessor
      Name="EnumCheckConstraints" ClassName="River.Meander.Catalog.Processors.EnumCheckConstraints"
    />
    <c:RegisterProcessor
      Name="DefinitionDependencies" ClassName="River.Meander.Catalog.Processors.DefinitionDependencies"
    />
    <c:RegisterProcessor
      Name="UnsafeStatementDetection"
      ClassName="River.Meander.Catalog.Processors.UnsafeStatementDetection"
    />
    <c:RegisterProcessor
      Name="HistoryTables"
      ClassName="River.Meander.MasterSchema.Processors.HistoryTables"
    />
    <c:RegisterProcessor
      Name="Procedures"
      ClassName="River.Meander.MasterSchema.Processors.Procedures"
    />
    <c:RegisterProcessor
      Name="CSharpInterface"
      ClassName="River.Meander.MasterSchema.Processors.CSharpInterface"
    />
    <c:RegisterNameProvider
     Name="Standard"
     ClassName="River.Meander.Definition.StandardObjectNameProvider"
    />
  </c:RegisterAssembly>
</c:Meander.Configuration>
```

This configuration requires you to have a database project directory, which contains files:

- `Configuration.xml` - configuration file
- `Invoices/Schema.xml` - project schema file
- `../../Bin/River.Glacier.Common.dll` - framework shared library
- `../../Bin/River.Meander.Common.dll` - framework shared library
- `../../Bin/River.Meander.Tools.dll` - framework shared library
- `../../Bin/River.Meander.Console.exe` - framework application
- `../Meander/Types.xml` - core of standard library
- `../Meander/Build.xml` - core of standard library

- `../Meander/Tools.xml` - standard library
- `../Meander/EndPoint.xml` - standard library for EndPoint services support
- `../Meander/Log.FrontEnd.xml` - standard library for logging
- `../Meander/Log.Backend.Storage.xml` - standard library for log storage

Framework files are part of the distribution. Several more can be found at the same location. Configuration file contains these XML element types:

- `RegisterAssembly` - reference to a .NET Assembly containing classes needed to construct object model of databases. When no customization is needed, section of the file can used as shown in this manual.
- `SchemaReference` or `MasterSchemaReference` - named alias for a schema file. Specifies relative file location.
- `AssemblyReference` - named alias for a CLR assembly file. Specifies relative file location. In case an assembly references other assemblies that also need to be deployed, these must be contained within the same directory as the deployed assembly. Assembly deployment is governed using the assembly version. It is necessary to manually increment version or use automatic versioning.
- `Template` - named configuration of schema and assembly files. Template should contain a valid and complete set of database objects in order to be successfully deployed
- `Instance` - named reference to a MSSQL database. In case integrated security cannot be used to access the database for full owner access a `ConnectionString` attribute can be specified.

In case you need to maintain more databases that share the same name, you can easily distinguish database instances using prefixes or postfixes. However, now, you have to remove `ServerName` attribute and supply full connection strings to given databases using `ConnectionString` attribute.

In case only a portion of database should be under the control of the framework. `Instance` can specify `IgnoreSchema` and `IgnoreAssembly` options. Using them hides whole schemas and can be used to provide backward compatibility.

Complete XML configuration file options are defined within `Configuration.xsd`.

A.5 Database objects

Standard work of database developer consists of creating and modifying database objects. This is done by modifying respective XML files. After modifications are complete, you check the consistency of database schemas. This section contains examples of representation of several basic database objects. Next section will show how to propagate those changes onto the database itself. Please note, that there are two sets of database objects `Catalog` and `MasterSchema`.

A.5.1 Data types example

```
<?xml version="1.0" encoding="utf-8" ?>
<c:Schema Name="demo_schema" xmlns:c="urn:river:meander:catalog:v1" >
  <c:Type Name="record_id" Definition="int"/>
  <c:EnumType Name="record_state_enum">
    <c:Item Id="A" Ident="Active"/>
    <c:Item Id="X" Ident="Cancelled"/>
  </c:EnumType>
</c:Schema>
```

Now `demo_schema` will contain an int-based scalar type `record_id` and char-based enum scalar type `record_state_enum`. These types can later used as data types of table columns or procedure parameters. It is a good practice to define limited set of data types used across the database. It can save you problems with potential data loss, precision loss or conversions.

A.5.2 Table example

```
<?xml version="1.0" encoding="utf-8" ?>
<c:Schema Name="demo_schema" xmlns:c="urn:river:meander:catalog:v1" >
  <c:Table Name="record">
    <c:Column Name="id" TypeName="record_id" Identity="disjoint" />
    <c:Column Name="value" TypeSchemaName="meander" TypeName="identifier" />
    <c:PrimaryKey Name="pk_record">
      <c:Column Name="id"/>
    </c:PrimaryKey>
  </c:Table>
</c:Schema>
```

Now `demo_schema` will contain also table `record` with columns `id` and `value`. Here you can see an identity column using automatic disjoint sequences[2]. Since `value` column references data types from a different schema `TypeSchemaName` attribute has to be specified.

A.5.3 Procedure example

```
<?xml version="1.0" encoding="utf-8" ?>
<c:Schema Name="demo_schema" xmlns:c="urn:river:meander:catalog:v1" >
  <c:StoredProcedure Name="proc">
    <c:Parameter Name="id" TypeName="record_id"/>
    <c:Definition>
<![CDATA[
begin
  set nocount on

  --TODO insert code here
end
      ]]>
    </c:Definition>
  </c:StoredProcedure>
</c:Schema>
```

Now `demo_schema` will contain also stored procedure `proc` with single input parameter. Procedure definition is supplied without the header[3]. Procedure body can be stored externally by using element `External` instead of `Definition`. The

[2] Alternatively you specify identity in the same format as in create table statements.

[3] Everything that follows after `as` keyword. It is a deliberate design limitation of the framework, that comments before the `as` keyword are not permitted and have to be moved to the body of the procedure. Many developers misuse the fact that procedure can contain text before the header and after the end of the body to store auxiliary statements or comments.

body must be saved in the same directory as the XML file, named in our instance `demo_schema.proc.sql`.

As was shown here `urn:river:meander:catalog:v1` schema follows very closely MSSQL database structure and object names. You should have no problem creating other object types not shown in this manual. Elements are defined within `Catalog.xsd`.

A.5.4 Enhanced table example

The XML schema shown so far is used to represent database objects as closely as possible to their form in real database[4]. Now you will see a `MasterSchema`. A different XML schema, that allows you to create specialized types of objects. Please note that only one type of xml schema can used to represent the target database schema. `MasterSchema` also uses different configuration elements. In order to register it in the configuration, please use `RegisterMasterSchema` and `IncludeMasterSchema` configuration elements.

```
<m:Schema
  Name="demo" Description=""
  xmlns:m="urn:river:meander:master-schema:v1"
  xmlns:p="urn:river:meander:procedure:v1"
>
  <m:Table Name="test" Description="">
    <m:Options Identity="false" History="false" Referencable="true" />
    <m:KeyColumn Name="id" Type="test_id" Description=""/>
    <m:Column Name="value" Type="meander.identifier" Description=""/>
  </m:Table>
</m:Schema>
```

The first thing, you may have noticed, is a `Description` attribute on every database object. It is accompanied with optional `Comment` attribute. Their function is to provide semantical documentation of the objects. Attribute values can be used for automatically generated documentation.

New table `test` have considerably more options. Columns have to be designated one of following types:

- primary key column - non-nullable column, that will be part of primary key.
- logical key column - nullable column, that will be part of unique index.
- key - non-nullable column, that is both primary and logical key[5]
- attribute column - nullable column, that stores table data.

In case your table cannot be represented in this manner, you have to use previous schema.

`Options` XML sub-element allows you these options:

- `History` - when true, a table holding record modification history will be created. Using a generated trigger, each insert, update or delete will make of record in the history table allowing you to review previous versions

[4]It is also necessary for providing backward compatibility.

[5]When primary key values are defined by user input or table is append only with no separate logical key.

of the records. When schema does not specify `HistorySchemaName` attribute, the same schema is used to contain the history table under name `other_schema.test_history`[6]. History table also contains information about user account that modified the data. Please see `meander.session_push` in Tools section for more details.

- `Identity` - specifies whether primary key column or key column is an identity column. When true, there can only be one primary key column or key column. Identity is disjoint by default.

- `Referencable` - specifies whether primary key or key columns should be used as a signature for generating foreign key references.

- `Immutable` - when true, table record cannot be modified or deleted, only added. When referenced by a table with history, column data type is not changed to `meander.id`, but kept to provide reference integrity.

A.5.5 Enhanced procedure example

```
<?xml version="1.0" encoding="utf-8" ?>
<m:Schema
  Name="invoices" Description=""
  xmlns:m="urn:river:meander:master-schema:v1"
  xmlns:p="urn:river:meander:procedure:v1"
>
  <m:Procedure Name="proc_test" Description="">
    <p:Input>
      <p:Scalar Name="id" Type="meander.id"/>
      <p:Vector Name="idents">
        <p:Scalar Name="ident" Type="meander.identifier"/>
      </p:Vector>
      <p:Scalar Name="id2" Type="meander.id"/>
      <p:Vector Name="idents2">
        <p:Scalar Name="ident" Type="meander.identifier"/>
      </p:Vector>
    </p:Input>
    <p:Block Name="data">
      <p:Edit TableName="test"/>
    </p:Block>
    <p:Block Name="output" />
    <p:Output>
      <p:Scalar Name="result_id" Type="test_id"/>
      <p:Vector Name="result_idents">
        <p:Scalar Name="ident" Type="meander.identifier"/>
      </p:Vector>
    </p:Output>
  </m:Procedure>
</m:Schema>
```

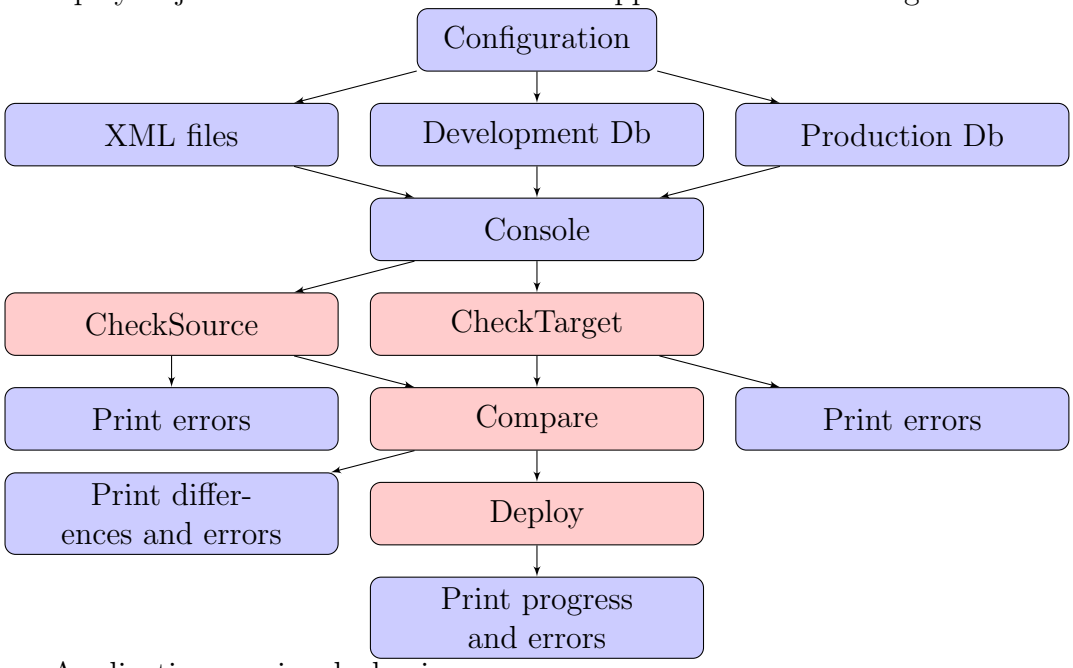
The main difference between standard stored procedures and enhanced procedures is pre-defined input and output. I/O contract is specified in procedure definition. Body of the procedure is partially generated. You can specified non-generated regions using `Block` element. Definition of block is external only a should be stored under name `invoices.proc_test.data.sql` where `data` is the name of the block. The procedure can contain more than one block depending on the desired use. It is a good practice to separate data table modification from generating procedure output. `Vector` input and output are translated into table variables whereas `Scalar` input and output are translated into simple variables.

[6]Naming conventions can be modified, please see `IObjectNameProvider`

Since generated procedure body is quite large, it is best viewed directly in SQL server after deployment. In the procedure body, you will clearly see several sections. For every `Block`, there is a named region created.

A.6 Console

Once database objects are assembled, you can use console application[7] to compare or deploy objects to the database. Console application follows diagram:.



Application can invoked using:

```
River.Meander.Console.exe <Configuration XML file> <Command> <Instance> <Switches>
```

`Configuration XML file` is a relative path to a xml file with configuration. In case current working directory contains only one configuration file a shortcut "." can be used. `Instance` parameter is matched against `Name` attribute of `Instance` elements in configuration. Value "all" can be specified to execute command against all instances.

Available `Command`s are:

- `check-source` or `cs` - Creates object collection from xml schema files of the specified instance. Prints any problems with the schemas[8].
- `check-target` or `ct` - Creates object collection from database catalog. Prints any problems with the schemas.
- `compare` or `co` - Creates changeset between XML schema files and database catalog of the specified instance. Prints differences and problems with the

[7] Application is designed to be very simple. It is planned that the framework will be completed with UI application, that will superseed Microsoft SQL Server Management Console. Therefore application is designed with functionality in mind.

[8] By far the most common error is referencing the non-existent object because of the typing error.

schemas. May fail if the desired operation is not supported[9]. In case a `--verbose` switch is specified, every change is printed by name, otherwise only cummulative statistic is shown.

- `deploy` or `de` - Creates changeset between XML schema files and database catalog of the specified instance and executes prerequisite checks and generated scripts. May fail when prerequisites are not met, for example setting column non-nullable when it already contains null values. Prints any errors and problems that ocurred during the deployment.

A.7 Tools

`Meander` database schema is supplied with several utility functions and procedures. Next follows the listing of more notable ones.

- `meander.print` - Prints text data into SQL console. It solves problem, that standard print command shows statements delayed and cannot print texts beyond 8KB.

- `meander.applock_acquire` and `meander.applock_release` - Wrapper for application locks. When in transaction, you can create your own exclusive locking mechanism.

- `meander.sequence`* - allows customized sequence generation. It is designed for integer and string sequences with dependent factors such as current year. The sequences are global within the database and can be shared among many tables.

```
declare @year [meander].[id]
set @year = datepart(year, getdate())

declare @id [meander].[id]

exec [meander].[sequence_get_next_id]
  @context1 = 'noris_commission.commission', --used only for specified table
  @context2 = @year,                         --every year sequence resets
  @format = '{C2}{V:#####}',                 --output id is a concatenation of context2 and
                                             --padded current value
  @next_id = @id out
```

- `meander.regexp`* - utility functions for text manipulation.

```
select  s.[index],
        s.[output]
from   [meander].[regexp_split](
          'A|B|C|D',--input data
          '\|',--pattern
          1 --options: 0 -> case-sensitive, 1 -> case-insensitive
        ) s
```

- `meander.hash`* - utility functions for hashing. Integrated `hashbytes` functions has problems with string truncation.

```
select  [meander].[hash_sha512](
          0xDEADBABE
        )
```

[9] At some point, senior developer may disable automatic table recreation.

- `meander.hex` - converts binary input into hex-formatted string output
- `meander.session`* - initializes session data such as session start, current account name and log merge guid. It is necessary to initialize session before calling enhanced procedure or modifying table with history table.

```
exec [meander].[session_push]
  @entity = 'john.doe',
  @session_start = @session_start
  @merge_guid = @merge_guid

/* insert code here */

exec [meander].[session_pop]
```

In case you use SQL accounts:

```
exec [meander].[session_push_native]

/* insert code here */

exec [meander].[session_pop]
```

- `meander.string_join`* - an aggregate function, that concatenates string values. Aggregate output is limited to 8KB. Functions are available in several variants.

```
select  a.[class],
        meander.string_join_comma(a.[member]) [members]
from    (          select  'Class'    [class],
                           'Member1'  [member]
        union all  select  'Class'    [class],
                           'Member2'  [member]
        ) a
group by
        a.[class]
```

- `meander.temporal_vector_sum` - calculates sum aggregate function over time-aware keys.

```
select  *
from    [meander].[temporal_vector_sum](
          '          select  123 [group_id],  --int key representing group-by-key
                             0 [vector_index],--position is vector
                             2 [count],  --value
                             convert(datetime, ''2011-01-01'') [valid_from],
                             convert(datetime, ''2011-10-01'') [valid_to]
          union all  select  123 [group_id],
                             0 [vector_index],
                             3 [count],
                             convert(datetime, ''2011-06-01'') [valid_from],
                             convert(datetime, ''2011-12-01'') [valid_to]
          ', 1
        ) a
```

Output time intervals are continuous with second precision spanning from minimal `valid_from` to maximal `valid_to`.

Bibliography

[1] Comingore D.; Hinson D. (2006): *Professional SQL Server 2005 CLR Programming: with Stored Procedures, Functions, Triggers, Aggregates, and Types*. Amazon.

[2] Developer Express Inc. (2011), *Specifics of joining data from multiple tables in XPO*.
URL `www.devexpress.com/Support/Center/p/K18431.aspx`

[3] Red Gate Sofware Limited (2011), *SQL Comparison SDK*.
URL `www.red-gate.com/products/sql-development`

[4] CodeSmith Tools LLC (2010), *CodeSmith GENERATOR*.
URL `www.codesmithtools.com/product/generator#templates`

[5] Sledge O.; Spenik M. (1999): *Microsoft SQL Server 7.0 DBA Survival Guide*. Amazon.

[6] Microsoft, *CREATE FUNCTION*.
URL `msdn.microsoft.com/en-us/library/aa258261(v=sql.80).aspx`

[7] Microsoft, *Limitations of the xml Data Type*.
URL `msdn.microsoft.com/en-us/library/ms187107(v=sql.90).aspx`

[8] Microsoft, *SSPI*.
URL `msdn.microsoft.com/en-us/library/aa380493(v=vs.85).aspx`

[9] Microsoft, *XACT_STATE (Transact-SQL)*.
URL `msdn.microsoft.com/en-us/library/ms189797.aspx`

[10] Simple Talk Publishing (2011), *SQLServerCentral.com*.
URL `www.sqlservercentral.com`

[11] Ben-Gan I.; Kollar L.; Sarka D.; Kass S. (2009): *Inside Microsoft SQL Server 2008: T-SQL Querying*. Microsoft Press.

[12] W3C, *Simple Object Access Protocol*.
URL `www.w3.org/TR/soap12-part1/`

[13] Wikipedia, *Ajax (programming)*.
URL `en.wikipedia.org/wiki/Ajax_(programming)`

[14] Wikipedia, *LINQ*.
URL `msdn.microsoft.com/en-us/netframework/aa904594`

[15] Wikipedia, *NoSQL*.
URL `en.wikipedia.org/wiki/NoSQL`

[16] Wikipedia, *Online transaction processing*.
URL `en.wikipedia.org/wiki/Online_transaction_processing`