Architectural Analysis of Microsoft Dynamics NAV

Tom Hvitved
hvitved@diku.dk

Department of Computer Science
University of Copenhagen

3gERP Workshop, November 18th 2008

Outline

1. Introduction
2. Motivation
3. Architectural Analysis
   - Table
   - Codeunit
   - Form
   - Report
4. Modularized architecture
5. Conclusion
6. Future Work

Introduction

- Deeper investigation of *one* ERP system: Microsoft Dynamics NAV (formerly Navision)
- Small and Medium-sized Enterprises (SMEs)
- More than 57,000 customers worldwide
- More than 2,700 certified partners worldwide
- More than 1,500 certified add-ons (*verticals*)
- (approx 1,000,000 lines of code!)

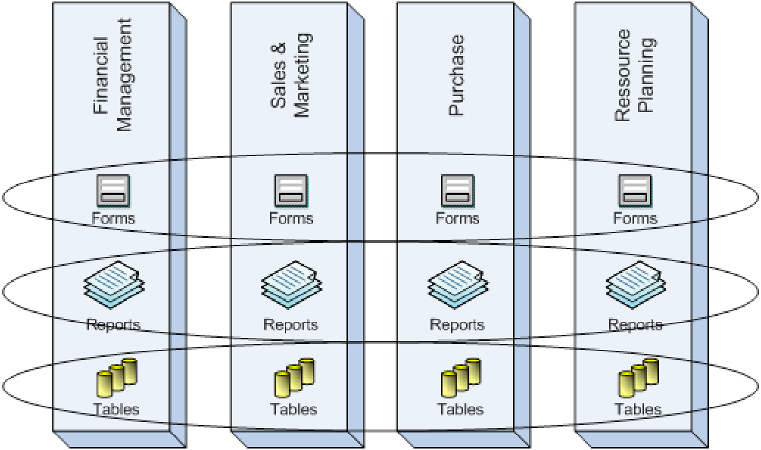
Outline

Motivation

- Hands-on experience with a real-world ERP system (within the 3gERP project, *evolutionary* approach)
- Provide a computer scientific description of NAV
- Address *upgradability* and *performance* issues
- Ideas for a modularized architecture
- Challenge: *Backwards compatibility* (NAV supply chain)

Outline

Architectural Analysis



Architectural Analysis

- Object based analysis
- NAV object $\simeq$ class (OOP)
- NAV object types:
  - Table
  - Codeunit
  - Form
  - Report
- Provide class interface "schema" for each NAV object type
- Useful for translation to e.g. C#
- Microsoft Dynamics NAV 5.0 W1 SP1, Microsoft SQL Server 2005

Outline

Table

Constant	
1. $\mathit{Name} \in \mathit{String}$	(table name)
2. $\Sigma : \mathit{String} \rightharpoonup_{\text{fin}} \mathit{SimpleType} \times \mathcal{P}(\mathit{Property})$	(signature/table schema)
3. $\mathit{Fields} \stackrel{\text{def}}{=} \text{dom}\Sigma$	
4. $\mathit{PrimaryKey} \in \mathit{Fields}^{+}$	(non-empty primary key definition)
5. $\mathit{Indexes} : \mathit{Fields}^{+} \rightarrow \mathcal{P}(\mathit{Fields})$	(table indexes and Sum Index Field definitions)
6. $\mathit{TableRelation} : \mathit{Fields} \rightharpoonup \mathit{TableRelationExp}$	(table relation definitions)
7. $\Sigma_{\text{FlowField}} : \mathit{String} \rightharpoonup_{\text{fin}} \mathit{FlowFieldExp}$	(FlowField definitions)
8. $\Sigma_{\text{FlowFilter}} : \mathit{String} \rightharpoonup_{\text{fin}} \mathit{SimpleType}$	(FlowFilter definitions)
9. $\text{dom}\Sigma \cap \text{dom}\Sigma_{\text{FlowField}} \cap \text{dom}\Sigma_{\text{FlowFilter}} = \emptyset$	(non-overlapping definitions)
Per instance	
10. Built-in methods	("triggers" in NAV terminology, e.g. `OnInsert`, `OnDelete`, etc.)
11. $\mathit{Vars} : \mathit{String} \rightharpoonup_{\text{fin}} \mathit{Type}$	(user-definable instance variables)
12. $\mathit{Methods} : \mathit{String} \rightharpoonup_{\text{fin}} \mathit{Procedure}$	(user-definable methods, "procedures/triggers" in NAV terminology)
13. Mutators	(built-in methods for updating state, e.g. set a FlowFilter)
14. Iterator	(an iterator for traversing data in the table. Key features: `FIND`, `INSERT`, `MODIFY`, `DELETE`)

Table (triggers)

- Table “triggers” are not triggers as known from active databases. Problem if used for validation purposes (invariants)
- `OnInsert`, `OnModify`, etc.
- Actually GUI triggers

Table (SIFT)

- Sum Index Field Technology is used to support *range sum queries*:

$$\sum_{r \in \sigma_{\mathtt{F}_1 = v_1 \wedge \ldots \wedge \mathtt{F}_{i-1} = v_{i-1} \wedge \mathtt{F}_i \in [v_i; v_i']}(\mathtt{T})} \pi_{\mathtt{F}}(r)$$

- $\mathtt{Amount} \in \mathit{Indexes}(\mathtt{G/L\ Account\ No.}, \mathtt{Posting\ Date})$

	G/L Account No.	Posting Date	Amount	...
$r_1 =$	1010	2008-05-01	100	...
$r_2 =$	1020	2008-07-01	600	...
$r_3 =$	1020	2008-01-01	200	...
$r_4 =$	1020	2008-12-01	100	...

- $$\sum_{r \in \sigma_{\mathtt{G/L\ Account\ No.} = 1020 \wedge \mathtt{Posting\ Date} \in [2008-07-01; 2008-12-31]}(\mathtt{T})} \pi_{\mathtt{Amount}}(r)$$

Table (SIFT)

- Also supports count, average, minimum and maximum
- We present data structure (augmented search tree) with complexity $\mathcal{O}(\log n)$ for update of `T` and for calculating range sum queries

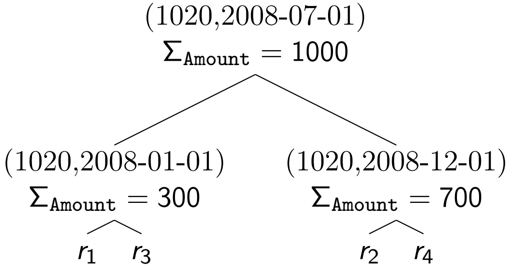


Table (SIFT)

- Current solution in Microsoft SQL Server has complexity $\mathcal{O}(\log n)$ for updates and $\mathcal{O}(n)$ for range sum queries. Only supports sum, count and average.
- Uses materialized (indexed) views
- Programmer specifies SIFT indexes

Table (relations)

- NAV supports complex table relations
- Not maintained by DBMS and not invariants: No referential integrity
- Conditional table relations $\Rightarrow$ unnormalized database design
- Proposed solution: Only allow SQL relations (invariants, checked by DBMS)
- Unnormalized database: Harder to upgrade/customize (e.g. "items" table has 175 columns)
- Proposed solution: Normalize database by introducing *joins* (*views*)

Table (FlowFields)

- Tables contain derived data (FlowFields/FlowFilters)
- Utilizes SIFT
- Derived data should be separated from “raw” data
- Proposed solution: *Views*
- *Backwards compatible*

Outline

Codeunit

- C/AL (imperative programming language, Pascal like)
- Statements (w/ side effects), expressions
- (Almost) strongly typed
- Typed database access (!)
- Strict type annotations ⇒ code duplication (cf. upgradability/customization)
- Strict type annotations ⇒ unnormalized tables ("pseudo polymorphism")
- Proposed solution: Polymorphism/sub typing

Outline

Form

- Form can be *bound* to a single table
- Easy (and automatic) integration with data
- Easiest solution for compound data (multiple tables): Make *one* product table
- Implicitly encourages unnormalized database design

Outline

Report

- Report = Data set + post processing
- Post processing = C/AL
- Data set = *pairwise join* of multiple tables:

$$T_1 \bowtie_{p_1} T_2 \bowtie \cdots \bowtie_{p_{n-1}} T_n$$

  p_i only mentions T_i and T_{i+1}
- Current solution: nested looping
- Proposed solution: (indexed) joins

Outline

Modularized architecture

- Encapsulation/abstraction is desirable (well-known in CS)
- Today: Logically related code spread across multiple NAV objects
- Needed: Code refactorization
- Today: Denormalized tables, sparse
- Needed: Decomposition to normalized tables (views provide means for backwards compatibility)
- One reason for denormalized database: *history problem*. Copying of data is OK, but data schema should be reused, not copied.
- Claim: Will make customization/upgrading easier

Modularized architecture

- First approach: Module = collection of existing NAV objects
- Did not work (cf. previous slide)
- Modularization is *necessary* (weak coupling)
- Current code base: 1 MLOC
- High level of interdependency (“spaghetti”): On average each object has 10 dependencies (not taking all dependency types into account!)
- Remove code duplication (ITU student project)

Modularized architecture

- Immediate benefits: Easier to maintain, extend, customize
- Future possibilities:
  - Module contracts (stateful types)
  - Aspect oriented programming (cf. Sebastien Vaucouleur)

Outline

Conclusion

- Lack of (formal) documentation
- Performance issues ("straightforward" to solve)
- No modular design (harder to solve: database decomposition + code refactorization + elimination of duplicated code)
- Claim: Modularized architecture will lower TCO (MS development, partner customization, upgrades)

Outline

Future Work

- Tools to support modularization (dependency analysis, code refactorization)
- Code analysis relies on a formal grammar – provided in our analysis.
- Investigate possibility of using *updatable* views
- *Incrementalized* views instead of SIFT (FunSETL, Michael Nissen)
- Student projects at DIKU

Thank You!