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

1 Introduction

2 Motivation

3 Architectural Analysis
   - Table
   - Codeunit
   - Form
   - Report

4 Modularized architecture

5 Conclusion

6 Future Work
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

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

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

Constant
1. Name \(\in\) String
2. \(\Sigma : String \rightarrow_{\text{fin}} \text{SimpleType} \times \mathcal{P}(\text{Property})\)
3. Fields \(\equiv\) \(\text{dom}\Sigma\)
4. PrimaryKey \(\in\) Fields\(^+\)
5. Indexes : Fields\(^+\) \(\rightarrow\) \(\mathcal{P}(\text{Fields})\)

6. TableRelation : Fields \(\rightarrow\) TableRelationExp
7. \(\Sigma_{\text{FlowField}} : String \rightarrow_{\text{fin}} \text{FlowFieldExp}\)
8. \(\Sigma_{\text{FlowFilter}} : String \rightarrow_{\text{fin}} \text{SimpleType}\)
9. \(\text{dom}\Sigma \cap \text{dom}\Sigma_{\text{FlowField}} \cap \text{dom}\Sigma_{\text{FlowFilter}} = \emptyset\)

Per instance
10. Built-in methods
11. Vars : String \(\rightarrow_{\text{fin}}\) Type
12. Methods : String \(\rightarrow_{\text{fin}}\) Procedure

13. Mutators
14. Iterator

(table name)
(signature/table schema)

(non-empty primary key definition)
(table indexes and Sum Index Field definitions)

(table relation definitions)
(FlowField definitions)
(FlowFilter definitions)
(non-overlapping definitions)

(“triggers” in NAV terminology, e.g. OnInsert, OnDelete, etc.)
(user-definable instance variables)
(user-definable methods, “procedures/triggers” in NAV terminology)
(built-in methods for updating state, e.g. set a FlowFilter)
(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:

  \[ \sum_{r \in \sigma_{F_1=v_1 \wedge \ldots F_{i-1}=v_{i-1} \wedge F_i \in [v_i;v_i']}^{(T)}} \pi_F(r) \]

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

| 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 |...

- \( \sum_{r \in \sigma_{G/L\text{ Account No.}=1020 \wedge \text{Posting Date} \in [2008-07-01;2008-12-31]}^{(T)}} \pi_{\text{Amount}}(r) \)
Also supports count, average, minimum and maximum

We present data structure (augmented search tree) with complexity $O(\log n)$ for update of $T$ and for calculating range sum queries

\[
\begin{align*}
(1020,2008-07-01) \\
\sum_{\text{Amount}} &= 1000
\end{align*}
\]

\[
\begin{align*}
(1020,2008-01-01) & \\
\sum_{\text{Amount}} &= 300 \\
& \quad \begin{array}{c}
\quad r_1 \\
r_3
\end{array}
\end{align*}
\]

\[
\begin{align*}
(1020,2008-12-01) & \\
\sum_{\text{Amount}} &= 700 \\
& \quad \begin{array}{c}
\quad r_2 \\
r_4
\end{array}
\end{align*}
\]
Table (SIFT)

- Current solution in Microsoft SQL Server has complexity $O(\log n)$ for updates and $O(n)$ 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 ⇒ 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

1. Introduction
2. Motivation
3. Architectural Analysis
   - Table
   - Codeunit
   - Form
   - Report
4. Modularized architecture
5. Conclusion
6. Future Work
C/AL (imperative programming language, Pascal like)
- Statements (w/ side effects), expressions
- (Almost) strongly typed
- Typed database access (!)
- Strict type annotations $\Rightarrow$ code duplication (cf. upgradability/customization)
- Strict type annotations $\Rightarrow$ unnormalized tables ("pseudo polymorphism")
- Proposed solution: Polymorphism/sub typing
Outline

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

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

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

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

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

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

- Tools to support modularization (dependency analysis, code refactoring)
- 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!