MaramaDSL

Based on Karen Li research proposal (presentation) at VASE workshop, ASE 09

- Aim of section:
  - Examine a new DSVL meta tool platform
    - Currently as an extension to Microsoft DSL Tools
- Contents
  - Design-for-reuse and design-by-reuse via patterns
  - Pattern specification in meta model
    - Structural vs. behavioural
  - DSVL pattern examples

Comparing Marama and MaramaDSL

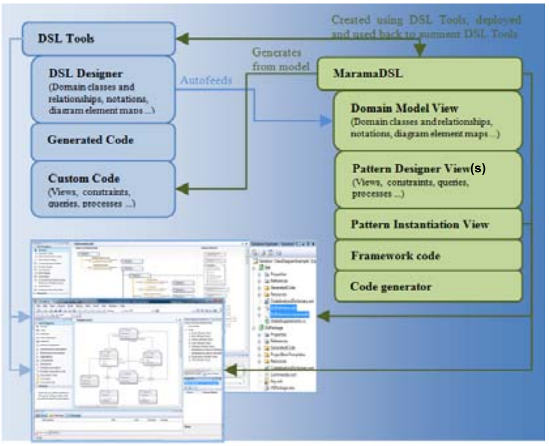
- Similarity
  - Raising the level of abstraction of knowledge representations in domain-specific software applications
  - Visually specifying DSVL meta models, notations, views, constraints, critics, event handlers, model transformation ...
  - Automatically generating DSVL environments based on specifications
- Difference
  - Marama generates DSVL environments from MULTIPLE integrated meta-DSVL based specification models
  - MaramaDSL aims to generate DSVL environments from a SINGLE pattern language model (with linked structural and behavioural perspectives)
    - Augmenting DSVL meta tools with pattern specification, instantiation and reuse

Motivation

- Common recurring problems and solutions exist in DSVLs
  - Model element composibility, cardinality, mutability, multiple views and their interoperability
  - Model organisation, constraints and metrics, value dependency, queries and workflows, diagram layout management, importing/exporting, tracing and debugging
- Design-for-reuse and design-by-reuse meta patterns
- The need for a pattern modelling language in parallel with DSVL meta model
- The need for pattern instantiation tool support (should help with creation) for DSVL model instances
- High level pattern-oriented cross-domain reuse optimistic/optimal approach to remove barriers to use

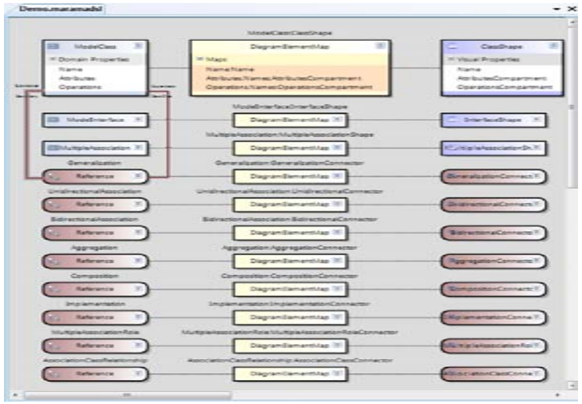
MaramaDSL Architecture

- *Microsoft DSL Tools* based meta tool
- Complements *DSL Designer* with multiple designer views, framework code and code generators
  - Current integration via importing DSL Designer models and generating code back to DSL projects
- Working together with *DSL Tools* integrating pattern specification and usage into DSVL development process



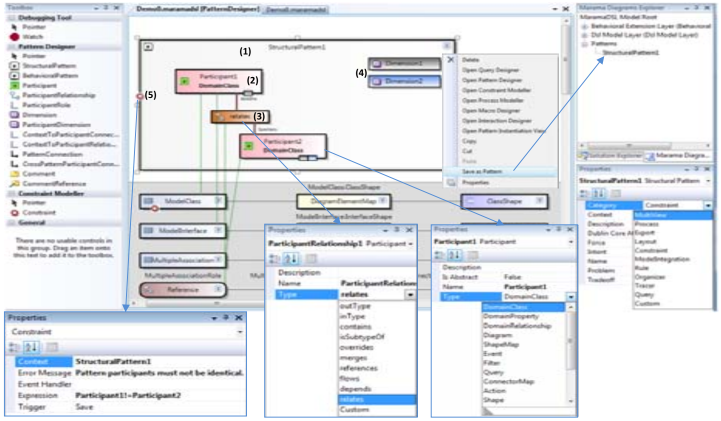
Domain Model View

- Displays a DSVL meta model (including defined domain classes, relationships, shapes, connectors and mappings), imported from a DslDefinition.dsl



Structural pattern specification

- Via a generic entity-relationship based language



Structural pattern specification

- Pattern modelling layer
  1) Structural pattern model
     - Categories: design pattern, multi-view, model constraint, model integration...
  2) Participant
     - Types: domain class, domain relationship, domain property, shape, connector, map...
  3) Participant relationship, source and target roles
     - Types: isSubTypeOf, contains, references...
  4) Dimension
     - Complements participant relationships with explicit multi-dimensional participant role cardinality constraints
  5) Constraint
     - Similar to MaramaTatau; can be added to all pattern elements
- Domain model layer
  - Filterable domain model elements imported from a DslDefinition.dsl
- Cross layer
  - Domain context bindings of pattern participants and relationships; integration of a pattern specification with a DSVL meta model

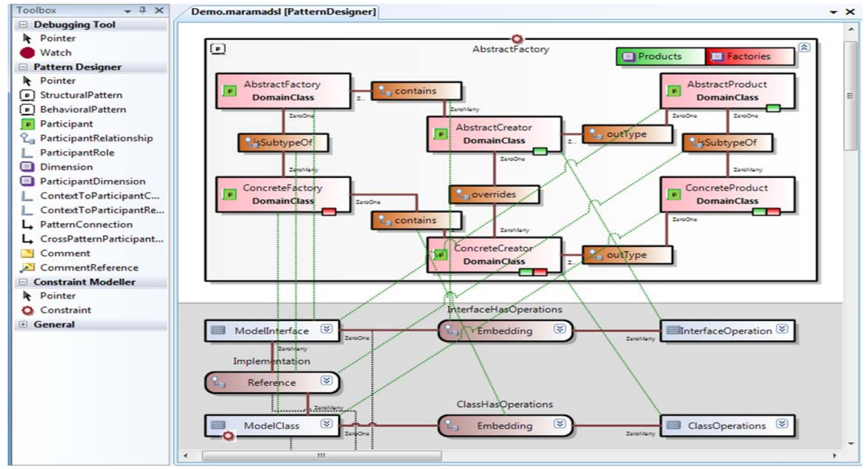
Pattern specification reuse

- Design-for-reuse
  - A pattern can be saved context-free (with all the context references removed), appearing in the Patterns Explorer Tree
- Design-by-reuse
  - A pattern from the Patterns Explorer Tree can be accessed and used on a different DSVL meta model (via a simple drag-drop and followed by context configurations)
    - Direct application on a domain model
    - With some adaptation – add/modify/remove participants or relationships at a DSVL client

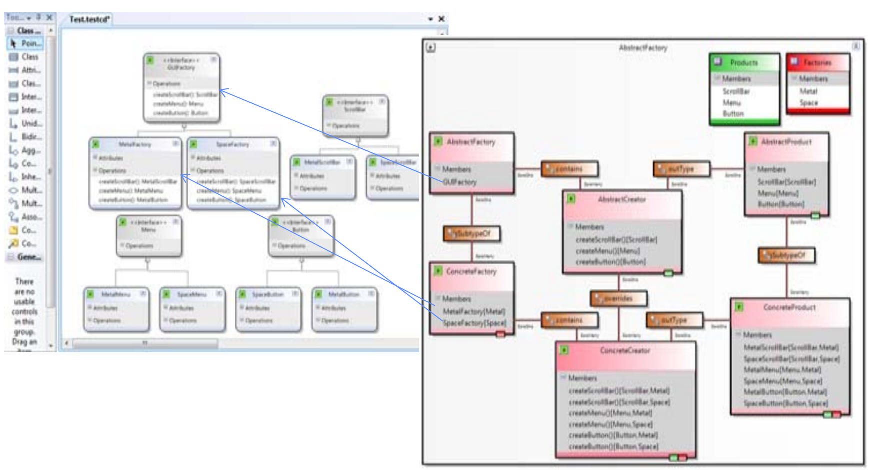
Structural pattern instantiation

- Context bindings at meta model level
- Pattern instantiations at instance level
  - Selection of DSVL model instance elements as runtime pattern participant members
  - An auxiliary Pattern Instantiation View (based on pattern specification – reuse design level abstraction)
    - Accepts direct input of pattern participant members
    - Allows cross-diagram drag-drop from DSVL model instance
  - Has generative creation or modification effects on DSVL model instance

Abstract Factory specification on UML meta model

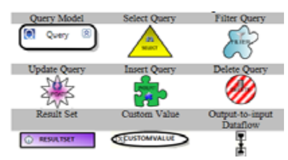


Abstract Factory instantiation on UML model instance

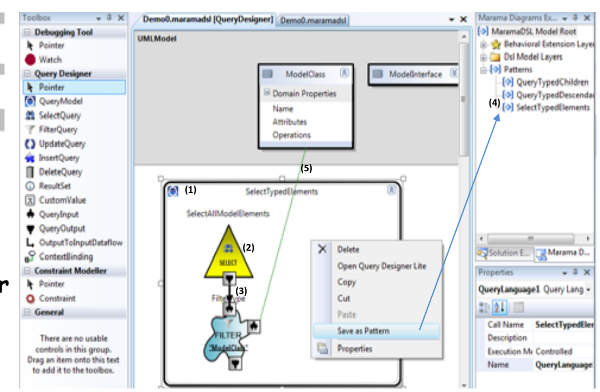


Behavioural pattern specification

- Currently via a generic dataflow based DSVL query language
  - CSI Academy project (Brian Webb and Tony Ly)



1) Query model
2) Query element with ports
3) Dataflow
4) Patterns explorer tree
5) Context binding



What behavioural patterns

- Visual analytics tasks in DSVL
  - Retrieving model data of interest to create visualisations
  - Detecting and removing conflicts
  - Refactoring
  - Differing and merging …
- Recurring querying need - Need a query language at appropriate abstractions
  - SQL lacks appropriate abstraction for manipulating DSVL model elements
  - OCL, Eclipse Query Model etc require too much of DSVL end users to learn, understand and code
  - Existing visual query languages address only basic selections

Query pattern specification

- Query modelling layer
  - Query model
  - Query element: SELECT, FILTER, UPDATE, INSERT and DELETE
  - Result element
  - Custom value
  - Dataflow
- Domain model layer
  - Shared btw structural and behavioural pattern modelling views
- Cross layer
  - Domain context bindings to query components

Our approach to development

- Four steps:
  1) a bottom up development of typical query (visual analytics) examples
  2) examination of the implemented code - largely domain-specific, with much repeated task logic
  3) use of reflective and refactoring techniques to "generify" code with reusable query artefacts - vocabulary for simpler query composition
  4) visual language design based on this vocabulary

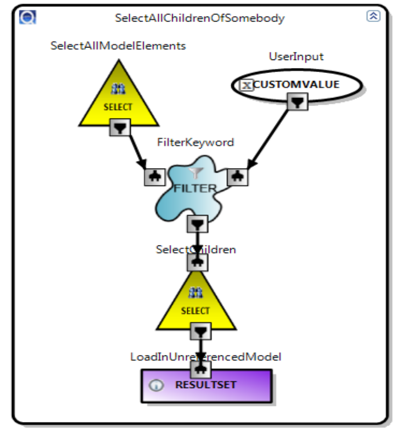
The vocabulary

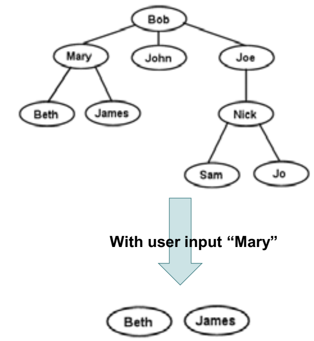
- A set of querying building blocks
  - Of SELECT, FILTER, UPDATE, INSERT and DELETE types, to retrieve data, set filtering criteria, and alter model/view elements
    - E.g. SelectAllModelElements, SelectChildren, FilterType, InsertModelElement, UpdateParent, DeleteLoop …
  - Parameterized with query context (models, views, model elements or visual symbols) and criteria (typed values)
  - Each has a returning result state as the output, for display or piping for further query construction
- A set for RESULTSET rendering

Our visual query language

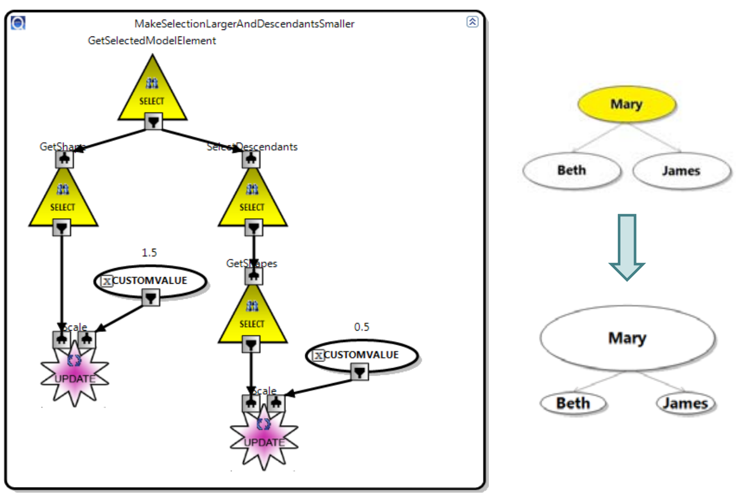
- For visual analytics of domain models, for non-programmer DSVL end users
- Represents the generalized query elements, and defines their interaction
- Allows complex queries to be composed from sub queries
- Allows queries to be abstracted for ease of reuse and reconfiguration
- Supports user-defined, automatic or self-controlled interactive query execution
- Design rationale - Moody's Physics of Notations theory

Example #1 Select elements of interest

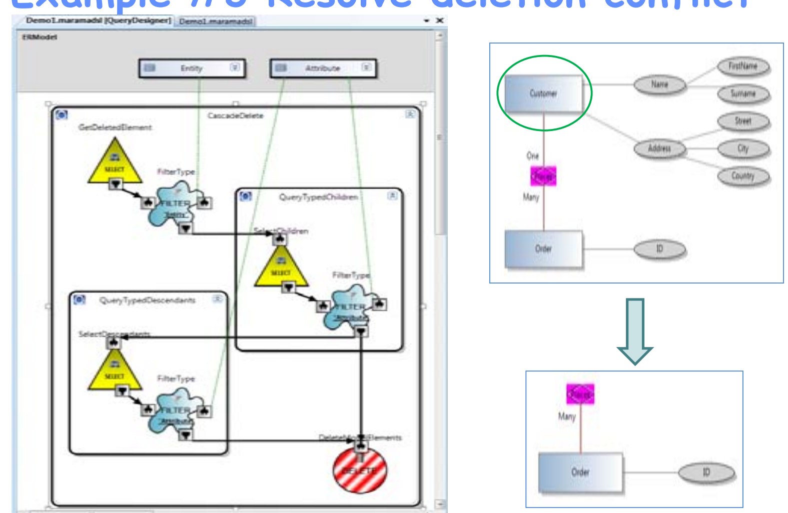




Example #2 Create visualisation effect

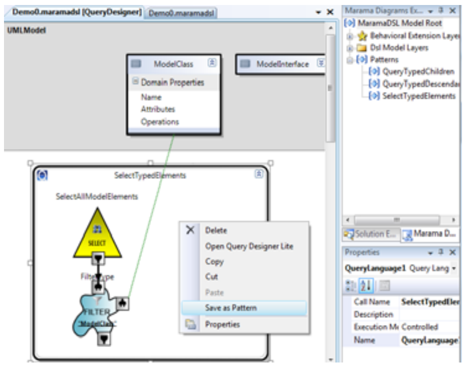


Example #3 Resolve deletion conflict



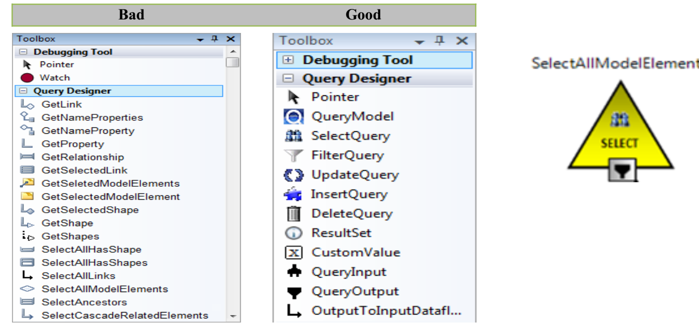
Applying Principle of Cognitive Integration

- An orthogonal domain model layer representation for meta model integration and easy context binding



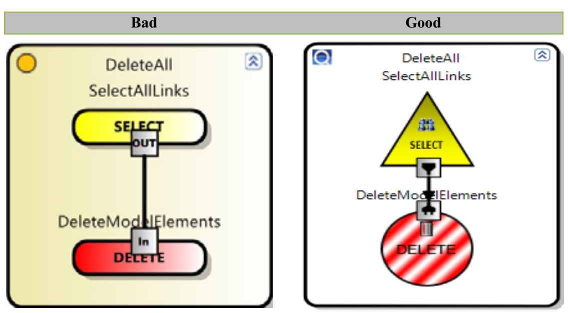
Applying Principle of Graphic Economy

- A cognitively manageable number of graphical symbols
- But heavy text Dual Coding



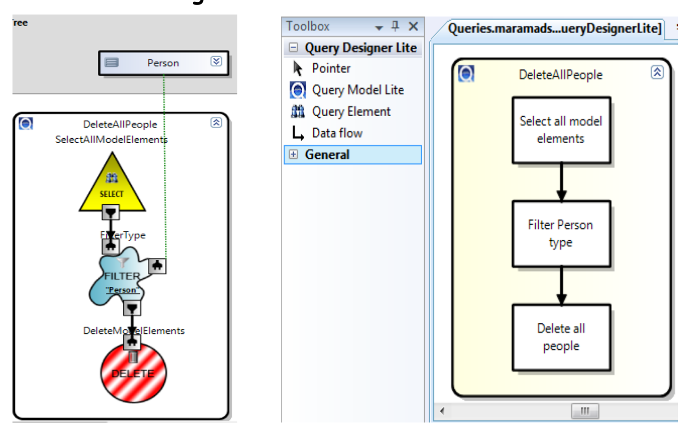
Applying Principle of Semantic Transparency

- Icons suggest semantics



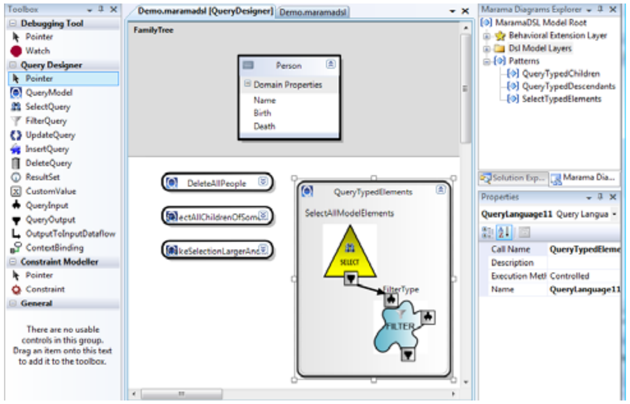
Applying Principle of Cognitive Fit

- Expert vs. lite view - 2 perspectives/dialects, one detailed and the other high level



Applying Principle of Complexity Management

- **Query pattern reuse via drag-drop, collapse/expand and form-based filtering show/hide mechanisms**



Tradeoffs

- Graphic Economy to be dominant
- Used "symbol overload" for Graphic Economy effect, which resulted in negative Semiotic Clarity
- Reduced graphical complexity (Complexity Management) by increasing Visual Expressiveness, but limited the number of symbols for Graphic Economy
- Heavily relied on text to distinguish query elements
- Sufficient visual distance (Perceptual Discriminability) and expressiveness through multiple channels: shape, icon, colour and texture, but didn't use the full range of visual variables (Visual Expressiveness)
- We chose to improve effectiveness for novices via Perceptual Discriminability, Complexity Management, Semantic Transparency, Graphic Economy and Dual Coding, trading off cognitive effectiveness for experts.

Evaluation - cognitive dimensions

- Equivalent expressiveness to domain-specific code written with APIs, but a lower *abstraction gradient*, augmented understanding, reduced effort, and a much shallower learning curve via better *closeness of mapping*
- Requires *hard mental operations* and *premature commitment*, but adding *abstractions* in the form of pre-defined query patterns reduces complexity and *diffuseness*
- Reduced *error proneness*, but requires proactive model checking
- Allowed *progressive evaluation*, but requires a compile-and-run cycle for the generated code
- *Terse* symbols, clear *role expressiveness*, but with *verbose* textual labels for expressing query building blocks
- Layout as a *secondary notation*
- Diagram insert *viscosity* problems occur, and require automatic layout to mitigate
- *Hidden dependency* and *visibility* mitigated by *juxtaposition* of orthogonal layered views and dual coding of custom values

Conclusion

- Ultimate goal – facilitate sharing of design knowledge among DSVL communities
- So plenty of to-dos (your contribution welcome – Summer Scholarship/PG/Hons/Msc projects available – supervised by John and Karen):
  - Model quality assurance via pattern applications
  - Pattern validation (completeness, consistency, soundness)
  - Force analysis/balancing of patterns in Pattern Language
  - Design decision support via higher level visual metaphor patterns
  - Automatic capture of design knowledge
  - Pattern publish and discovery – use semantic web technologies
  - ...