Overview

- Formal Reuse Model and Framework (RC & SAD)
  - Initial Version of Formal Reuse Model
  - Adding Complexity to Reuse Model
  - Utilizing Genetic Algorithms/Experimentation

- Reusability and XML (DN)
  - Impact of XML Abstractions on Reuse
  - Issues for XML and Reuse

- Prototyping Progress in DRE (JE, SD, & SAD)
  - Graphical Representation/Editing Source (SD)
  - XML Interoperability/Cooperative DRE (JE)

- Reusability and UML with TCC (FE, XW, & SAD)
  - Integrating DRE into TCC (XW)
  - Reuse Model/Framework and UML (FE)
Formal Reuse Model and Framework (RC & SAD)

Initial Version of Formal Reuse Model

- Ph.D. Research of R. Caballero
- Paper Accepted at ICSR-7
  “Towards the Formalization of a Reusability Framework for Refactoring”
  Hard Copy Provided with All Details of Work

Long-Term

- Adding Complexity to Reuse Model
  Reusability Nodes, Dependencies, Graph, Layer
- Utilizing Genetic Algorithms/Experimentation
  Utilizing GA to Search for a Near Optimal Marking of Classes via New Formal Framework

Formal Reuse Model and Framework

- Formal Definitions of Reuse Model to Allow Automatic and Algorithmic Analysis of Couplings
- Formalization of OO Application Model
  - Defn. 1: OO Application as Three Tuple of Classes, Inheritance, and Method Couplings
  - Defn. 2: Two Classes are Pair-Wise Coupled if there is a Method Invocation Between Them
- Formalization of Reuse Framework
  - Defn. 3: Reusability Level of a Class and the Class Generality Vector for Application
  - Defn. 4: Formal Definition of Related Classes
Formal Reuse Model and Framework

Quantifying Reuse Properties

- Prop. 1: Parent of a Class is Equally General or More General than is Direct Children
- Prop. 2: Reuse Level of Class is Equal to the Reuse Level of the Least Reusable Coupled Class
- Prop. 3: Classes that Don’t Contribute to Functionality of Component have Negative Impact on Reuse
- Prop. 4: Couplings Between Unrelated Classes Hinder Reuse

Formal Reuse Model and Framework

Coupling Type Transition Matrix

- Defines Loss, Gain, or Steady State Between the Various Coupling Types
- For Example,
  
  Going from a G--->S to a G--->G Coupling Type for Related Classes is a +1
  Going from a G--->G to a G--->S Coupling Type for Related Classes is a -1
  Going from a S--->G to a S--->S Coupling Type for Related Classes is a 0
- This Matrix Defined for all Transitions Among the Eight Coupling Types in One Step
Formal Reuse Model and Framework

- Refactoring Process - Eight Step Algorithm
  1. Identify Reuse Potential - Mark Generalities
  2. Calculate Couplings Among Actual Code
  3. Identify Related Classes
  4. Determine Coupling Types
  5. Identify Undesirable Couplings
  6. Refactor by Moving Source or Destination Method, Changing Reuse Level, or Related to Unrelated
  7. Recalculate Reusability Level for Application
  8. If Reuse Factor <= 0 Go to Step 6 Else Go to Step 5 or Terminate Based on Condition

Note: Example and Details in Paper

Future Work on Formal Reuse Model

- Exploring Weightings for Transitions Among Coupling Types
  - Currently +1, -1, 0
  - What about +2 or +3 or Different Weights
- Explore and Define the Reuse Model
  - Theoretical Verification of Reuse Model
  - Use of GA to Obtain “Near” Optimal Marking
- Practical Perspective
  - Integrating Metric and Model in DRE
  - Testing with “Real” Examples and Prior Examples
  - Is there an EB Example Available?
## XML Reusability Issues (DN)

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What are Reusable Components in XML?</td>
<td></td>
</tr>
<tr>
<td>What are Dependencies in XML?</td>
<td></td>
</tr>
<tr>
<td>How can XML Dependencies be Restructured to Improve Reuse of XML Components?</td>
<td></td>
</tr>
<tr>
<td>XML Documents can be Thought of as Data Containers; Similar to Objects in an OO Language</td>
<td></td>
</tr>
<tr>
<td>How are OOP and XML Similar?</td>
<td></td>
</tr>
</tbody>
</table>

## XML Compared to OOP

<table>
<thead>
<tr>
<th>Statement</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML Schema Language allows Definition of Structure and Data Types within Documents</td>
<td></td>
</tr>
<tr>
<td>An XML Schema is Similar to the Language used to define an OOP Class (e.g., a Class in Java)</td>
<td></td>
</tr>
<tr>
<td>A Specific XML Schema is like a Class with Data Members</td>
<td></td>
</tr>
<tr>
<td>An XML Instance Document is like an Object Instance of that Class</td>
<td></td>
</tr>
<tr>
<td>The OOP Analogy Applies only to Class Data Members, Since XML Elements/Types are Data Containers with no Methods of their own</td>
<td></td>
</tr>
</tbody>
</table>
Comparing Java and XML

<table>
<thead>
<tr>
<th>How Types Defined</th>
<th>Datatype Definition</th>
<th>Datatype Instance</th>
<th>Couplings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Java used to define classes</td>
<td>Class</td>
<td>Object instantiated from class</td>
</tr>
<tr>
<td>XML</td>
<td>XML Schema Language</td>
<td>An XML Schema</td>
<td>An XML instance Document containing data</td>
</tr>
</tbody>
</table>

Basis for Reuse in XML: Elements and Types

- A **Warranty** Construct can be an Element or Type.
  ```xml
  <xsd:element name="Warranty">
    ...
  </xsd:element>
  ```
  or as a Type:
  ```xml
  <xsd:complexType name="Warranty">
    ...
  </xsd:complexType>
  ```

- Element Approach Supports Substitutability
- Can Create Elements from the Type as Needed
Declarations: Global vs. Local

- **Global Element Declarations, Type Definitions:**
  - Element Declarations/Type Definitions that are Immediate Children of `<schema>`

- **Local Element Declarations, Type Definitions:**
  - Element Declarations/Type Definitions that are Nested Within Other Elements/Types
  - Only Global Elements/types Can Be Referenced (Reused)
  - Local Elements/Types are Invisible to the Rest of the Schema (and to Other Schemas)

Design Issues: Global vs. Local

- **When Should an Element (or Type) Be Declared Globally Versus Locally?**
- Consider How a Schema Might Support the Below Snippet of an XML Instance Document:

  ```xml
  <Book>
    <Title>Using W3C XML Schema</Title>
    <Author>Eric van der Vlist</Author>
  </Book>
  ```

- **Different Schema Design Strategies Exist**
- Next:
  - Analyze Strategies (From a Reuse Perspective)
  - Consider Component Couplings and Cohesiveness of Components They Produce
Russian Doll Design
(One End of Spectrum)

- Schema Strategy: Bundle Together All Element Declarations.
- Mirrors Instance Document Structure
- Declare Book Element and Within It Declare Title Element, etc:

  ```xml
  <xsd:element name="Book">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Title" type="xsd:string"/>
        <xsd:element name="Author" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  ```

- Represents One End of the Design Spectrum

---

Russian Doll Design Characteristics

- **Opaque Content.** Content of Book is Hidden From Other Schemas, and From Rest of Same Schema
  - Impact: Book is Reusable, but None of Book’s Types/elements Are Reusable

- **Localized Scope.** The Title and Author Element Declarations are Localized Within the Book Element
  - Impact: If Schema Sets ElementForm=default =“Unqualified”, the Namespaces of Title and Author Are Hidden (Localized) Within Schema

- **Decoupled.** Each Component is Self-contained
  - Changes to Components Do Not Impact Other Components
Flat Catalog Design
(Other End of Spectrum)

Schema Strategy: Define Components as Element
Declarations, and Assemble Them Together (by
Ref'ing Them) to Create the Aggregate Component

```xml
<xsd:element name="Title" type="xsd:string"/>
<xsd:element name="Author" type="xsd:string"/>

<xsd:element name="Book">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="Title"/>
      <xsd:element ref="Author"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Flat Catalog Design Characteristics

**Transparent Content.** Components Making up Book Are Visible to Other Schemas, and to Other Parts of the Same Schema

- Impact: Book’s Types/elements Are Reusable

**Global Scope.** All Components Have Global Scope

- Impact: The Namespaces of Title and Author Will Be Exposed in Instance Documents

**Coupled.** Book Element Depends on Title and Author Elements. Changes to These Elements Alter the Book Element

- Impact: This Design Approach Produces a Set of Interconnected (Coupled) Components
Russian Doll vs. Flat Catalog

- Differences Between the Two Design Approaches:
  - The Russian Doll Design Facilitates Hiding (Localizing) Namespace Complexities
  - The Flat Catalog Design Does Not
  - The Flat Catalog Design Facilitates Component Reuse
  - The Russian Doll Design Does Not

Next:
- A Design Which Facilitates Hiding (Localizing) Namespace Complexities
- And Facilitates Component Reuse

Venetian Blind Design

- Spread Components into Type Definitions, Reuse the Types

```xml
<xsd:simpleType name="Title">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Mr."/>
    <xsd:enumeration value="Mrs."/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="Name">
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="1"/>
  </xsd:restriction>
</xsd:simpleType>
```
Venetian Blind Design (continued)

```xml
<xsd:complexType name="Publication">
  <xsd:sequence>
    <xsd:element name="Title" type="Title"/>
    <xsd:element name="Author" type="Name"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="Book" type="Publication"/>
```

- Maximizes Potential to Hide (Localize) Namespaces [Determined by the Elementformdefault "Switch"]
- Has Reusable Title, Name, Publication Type Components As Well As Book Element

---

Venetian Blind Design Characteristics

- **Maximum Reuse:**
  - Type Definitions are the Primary Components of Reuse
- **Maximum Namespace Hiding:**
  - Element Declarations are Nested Within Types, Maximizing the Potential for Namespace Hiding
- **Exposure Switching:**
  - Namespaces Can Be Hidden (Localized) in the Schema or Exposed in Instance Documents (Controlled by Elementformdefault)
- **Coupled:**
  - Generates a Set of Components Which are Interconnected (i.e., Dependent)
Schema in pdm.xsd

```xml
<xs:complexType name="Ship_class">
  <xs:complexContent>
    <xs:restriction base="edo:Edo">
      <xs:sequence>
        <xs:element name="Units" type="pdm:Ship_class.Units" nillable="true"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="Ship_class.Units">
  <xs:restriction base="mea:Unit_of_measure">
    <minOccurs "0" maxOccurs "unbounded"/>
  </xs:restriction>
</xs:complexType>
```
Venetian Blind Design Guidelines

Design Schema to Maximize the Potential for Hiding (Localizing) Namespace Complexities.

- Use Elementformdefault as Switch for Controlling Namespace Exposure via Elementformdefault:
  - "Qualified": Element Namespaces Exposed in Instance Documents,
  - “Unqualified”: Element Namespaces Not Exposed in Instance Documents.

Design Schema to Maximize Reuse:

- Use Type Definitions as the Main Form of Component Reuse.
- Nest Element Declarations Within Type Definitions.

Summary of Design Approach Impact on Components

<table>
<thead>
<tr>
<th></th>
<th>Russian Doll</th>
<th>Flat Catalog</th>
<th>Venetian Blind</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cohesion</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Coupling</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Reuse Potential</td>
<td>Low: Types hidden</td>
<td>High: Element level, with Namespace Issues</td>
<td>High: Type level, Namespace Control</td>
</tr>
</tbody>
</table>
Which Design Approach is Preferable?

- **Venetian Blind:**
  - Where Schema Requires the Flexibility to Turn Namespace Exposure on or off With a Simple Switch, and
  - Where Component Reuse is Important

- **Flat Catalog:**
  - Where Instance Document Authors Need the Option to Use Element Substitution

- **Russian Doll:**
  - Where Minimizing Size and Coupling of Components is the Primary Concern

Other Issues Impacting Reuse in XML

- Inheritance
- Substitutability
- Containers With Variable Content
- `<Import>` and `<Include>`
Inheritance

- XML: Allows for Derived Types
  - Base Attribute Points to Parent Type
  - Derivedby Attribute Indicates Whether By: Extension, Restriction, or Reproduction
- “Final” Attribute Prevents Derivation
- “Block” Attribute Disallows Child From Being Treated As Parent Type

Substitutability

- Schemas Can Declare Elements (but Not Types) That May Be *Substituted* for Each Other
  - Such Elements Must Be Declared As Global *Elements*
- Transitive:
  - If Element A Can Substitute for Element B, and Element B Can Substitute for Element C, Then Element A Can Substitute for Element C
  - A --> B --> C Then A --> C
- Non-symmetric:
  - If Element A Can Substitute for Element B, it is Not the Case That Element B Can Substitute for Element A.
Containers With Variable Content

- Designing a Container Element (E.G., Catalogue) Which Is to Be Comprised of Variable Content (E.G., Book, or Magazine, or ...)
  
  `<Catalogue>
    - Variable Content –
  </Catalogue>`

- Do We Allow the Container Element to Contain Dissimilar, Independent, Loosely Coupled Elements?

- How Do We Design an Extensible Variable Content Container? (i.e., The Kinds of Elements Which it May Contain Can Grow Over Time)

Imported vs. Include

- Complex Schemas Typically Broken up Into Separate Files for Maintainability and Reuse

- XML’s `<Include>` is Used to Bring Other Schemas Into the Current Schema, Like a Copy/paste.
  - Including a Schema Brings All its Type Definitions Into the Current Schema’s Target Namespace.

- XML’s `<Import>` : Reference Other Namespace Types w/o Bringing Them Into Target Namespace.
  - TypeDefs From Imported Namespace Prefixed With the Namespace
    - Depends on `ElementFormDefault = "(un)qualified"`
Future Work in Reuse and XML (DN)

- How do the Following Issues Impact Reusable Components and Dependencies in XML?
  - Inheritance
  - Substitutability
  - Containers With Variable Content
  - <Import> and <Include>

- How can XML Components and Dependencies be Restructured to Improve Reuse?

Prototyping Progress in DRE (JE, SD, SAD)

- Graphical Representation/Editing Source (SD)
  - Graphically Represent Classes for Generality and Related Assignment
  - Ability to Edit Source Code for “Bad” and “Okay” Dependencies

- XML Interoperability/Cooperative DRE (JE)
  - XML Interoperability to Allow DRE Marking to be Saved and Restored via XML
  - Cooperative DRE to Support Multiple Clients Working on Same Project Simultaneously
Graphical Representation
Invoking Visual Display

Graphical Representation (SD)
Initial Screen to Set Generality/Related
Graphical Representation
Generality Set for Design

<table>
<thead>
<tr>
<th>Property</th>
<th>Specific</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set All as</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Help / Close</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Couplings</th>
<th>CC1</th>
<th>Results</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Help</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Graphical Representation
Setting Related Classes

<table>
<thead>
<tr>
<th>Property</th>
<th>Specific</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set All as</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Help / Close</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Couplings</th>
<th>CC1</th>
<th>Results</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Help</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Graphical Representation
Removing Related Classes

Editing Source Code (SD)
Enabled When Doubling Clicking
Editing Source Code
Editor Appears with Line Highlighted

Editor Appears with Line Highlighted

Source Editor Features
Editing Source Code

[Image of a software interface for editing source code with menu options like File, Edit, and Help, along with features for saving, opening, cutting, copying, and pasting.

Line 1

Line 1

Jan2002. OOReuse - 43

Jan2002. OOReuse - 44
Collaborative DRE (JE)

Collaborative DRE Purposes

- Enable Teams to Collaborate on Metrics Measurement
  - Utilize Client/Server Architecture
  - Share Data Between Users
  - Separate Project Information From Class Markings
  - Store Source Code Centrally
  - Execute Metrics on “Powerful” Server
  - Use Same DRE Interface as Control Client
- Utilize XML in Redesign Efforts

Collaborative DRE is Intended to Bring DRE into a Real-World Computing Environment

Collaborative DRE Architecture
CDRE – DRE Client

**Functions**
- Client Allows User to Log into the Server
- Projects Accessed, Changed and Displayed
- File Manager to Find New Source Files
- Calculation Controlled From Client

**Requirements**
- Utilize Existing Interface (With Additions)
- Utilize Same Code Base, Data Structures
- Drill-down Capability to Class-level Only (Not Method/data Member Depth)
- Communicate With Server Via Socket Connection
- Remember: Source Code is on Server!

Main Application Window
Main Window – Setting Generality

Main Window – Setting Relations
Main Window – Choosing Simulation Options

Help Subsystem

HTML-based Documents Describe Tool & Theory

Using DRE v2.01

Adding Files to the Project

The DRE will parse selected Java files and present a list of classes contained within. The user begins by authoring the DRE with the specific file to be parsed. Simply click on the tree plus icon on the toolbar. A file open dialog box will be presented. Locate the desired file and click on “OK”. The parsed Java classes will appear in the left explorer pane.
CDRE – Project Selection

CDRE – Remote Class Selection
Functions

- Accept Socket Connections from Client
- Read and Handle Requests from Client
- Find and Parse Through Source Files
- Run Metrics Calculations
- Return Responses to Client
- Connect to Databases
- Access/Update Data in Databases
- Handle Multiple Clients
- “Authenticate” Users

Requirements

- Communicate with Client Via XML Messages Over Sockets
- Maintain Persistent Socket Connection to Client
- Operate Metrics Calculator Identical to Workstation DRE (Use Same Code Base)
- Translate Full Data Structures to Lightweight Data Structures
- Communicate with Databases Locally and/or Remotely
  - Databases Able to Be Partitioned
  - Communication via RMI
- No GUI Necessary!
**CDRE – Sample Response Messaging**

**CDRE - Databases**

- **Three Necessary Databases**
  - User DB– Authorizes Users to Use CDRE
  - Project Database – Contains Information Relevant to Each Defined Project
  - Class Database – Contains Information About the Generality/relation Markings of All Classes

- **Databases Located by a Database Finder**
  - Implementation of Data Storage Location Kept Separate From Functional Application

- **Database Interfaces**
  - Defined to Allow Plug-and-play Database
  - Default Implementation is XML-based
  - Relational Model can be Substituted
**CDRE – Users**

- **User Database**
  - Defines IDs, passwords, groups, names, etc.
  - Used for simple authentication
  - User ID is unique key

- **Users in Classes**
  - Classes contain one or more owners, who can change the markings of a class
  - Classes contain zero or more groups, whose members are all owners
  - All other users can use any class information!

- **Users in Projects**
  - Projects contain one or more owners, who can create and delete project, and change users
  - Projects contain zero or more manipulators, who can add and remove classes from project
  - Projects contain zero or more users, who can only run the metrics on the project
  - All other users prohibited from using a project!

---

**CDRE - Projects**

- **Project Database**
  - Defines Project ID, Users, Class List, Change History
  - Project ID is Unique Key

- **Project Rules**
  - Any User Can Create a Project
  - Owners Can Control Full Project Information, Including Deletion of the Project
  - Users Will Only Be Able to Access Projects Where They are Listed As Owners, Manipulators, or Users
  - Project Updates are Instantaneous (No “Save Project” Needed)
CDRE - Classes

- Class Database: Defines Class ID, Path, Package Name, Owner Lists
  - <Name, Package, Path> is Current Unique Key
  - <Name, Semi-unique-ID> is Future Unique Key

- Class Rules
  - Classes Are Added to the Database Initially As They Are Put Into Some Project
  - When Added to the Database, Classes Set the Logged-in User as the Owner
  - Any Owner Can Allow Any Other User to Become Class Owner
  - Multiple Classes with Same Name are Allowed

CDRE – Sample Request/Response

Function: Select Project
## Other DRE Improvements

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>Parser Re-Engineered From Scratch</td>
</tr>
<tr>
<td></td>
<td>Addresses Margie’s Earlier Concerns</td>
</tr>
<tr>
<td>m</td>
<td>Registration Module Added</td>
</tr>
<tr>
<td></td>
<td>Ships Registration E-mail when User First Activates DRE</td>
</tr>
<tr>
<td>m</td>
<td>Debug Facility Coded</td>
</tr>
<tr>
<td></td>
<td>Debug Levels Control Output Analysis</td>
</tr>
<tr>
<td>m</td>
<td>Command-line Parameter Analyzer Ability</td>
</tr>
<tr>
<td></td>
<td>Allows User to Alter DRE Functions</td>
</tr>
<tr>
<td>m</td>
<td>Data Structures and GUI Split Into Lightweight/Heavyweight Components (CDRE-related)</td>
</tr>
</tbody>
</table>

## Future Work: Prototyping for DRE (JE, SD, SAD)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>Simulator</td>
</tr>
<tr>
<td></td>
<td>Currently Weakest Link in DRE</td>
</tr>
<tr>
<td></td>
<td>Strengthened if Necessary</td>
</tr>
<tr>
<td>m</td>
<td>Memory Usage</td>
</tr>
<tr>
<td></td>
<td>Memory Limit - 250 Classes, 256 MB</td>
</tr>
<tr>
<td></td>
<td>Search for Memory Reduction</td>
</tr>
<tr>
<td>m</td>
<td>XML/XSL-based Reporting Facility</td>
</tr>
<tr>
<td></td>
<td>Class/Project/User Data</td>
</tr>
<tr>
<td></td>
<td>Reuse Metrics Results</td>
</tr>
<tr>
<td>m</td>
<td>Integrate CVS Storage</td>
</tr>
<tr>
<td>m</td>
<td>User Suggestions</td>
</tr>
</tbody>
</table>
### Reusability and UML with TCC (FE, XW, & SAD)

<table>
<thead>
<tr>
<th>Topic</th>
<th>Details</th>
</tr>
</thead>
</table>
| Integrating DRE into TCC (XW)                                         | - Ongoing Prototyping Effort to Integrate DRE into TCC  
- Utilization of TCC Extension Capabilities to Allow Generality and Related to be Set  
- Initiation of Complete DRE tool via Plug-In Capabilities of TCC |
| Reuse Model/Framework and UML (FE)                                     | - Investigation of Our Reuse Model and Framework at Design Level via UML  
  Establishing Generality and Related in Use-Case and Class Diagrams  
  Counting and Tracking Dependencies in Other Diagrams |

### Reuse in Togethersoft Control Center

<table>
<thead>
<tr>
<th>Topic</th>
<th>Details</th>
</tr>
</thead>
</table>
| Integrates Visual Modeling via IDE                                     | - Implements UML/Supports Java and C++  
- Source Code Editing  
- Model/Code Synchronization  
- Open API and Plugin Support (Written in Java) |
| Single Integrated Design/Development Environment with Reusability Assessment | - Incorporate G/S and Related at UML Level  
- Users can Execute Plug Ins for Each  
- Actual Properties for UML Classes Modified  
- Invoke DRE Directly from TCC |
Reusability in Together CC
Reusability in Together CC
Reuse Model and UML (FE & SAD)

Objectives
- Examine Reuse at Design Level
- Specific Focus on UML
- Concentration on Use Cases and Class Diagrams

Key Questions
- Can Use Cases have Generality Levels?
- How Do UC Generality Levels Relate to Classes?
- What Dependencies Can be Tracked for Either Warning or Counting (Metrics)?
- Incorporation into TCC and Transition to DRE

Generality and Use Cases

User Marks Generality of UCs (as Property)

Warn on Extends and Includes Depending on Generalities
- UC_A has Generality G_X
- UC_B has Generality G_Y
- UC_A includes UC_B:
  \[ G_X \leq G_Y \]
- UC_A extends UC_B:
  \[ G_Y \leq G_X \]

Use Case Generalizations
- UC_A Generalizes UC_B
  \[ G_X \leq G_Y \]

Warn on Use of Other Systems Depending on Generalities
Use Cases - Example (Step 1)

Only Warn on Use of Other System

Use Cases – Example (Step 2)

Warn on Include <Order Product>

Warn on Generalization <PayCash>
Use Cases – Example (Step 3)

Next Iteration of Previous Slight with Revised Generalities

Class Diagrams (Only Names)

Initially, Class Diagram May Consist of Only Class Names
User Relates Classes to UCs as Property
Not all Classes have to be related to UCs
For Example, a LinkedList class might not be related to any UC
A Class can be Related to Several UCs
User has to Specify Relation Between Classes as Property
Class Diagrams – Relations Outcomes

- Go from Class-UC to Class-Class:
  - Warn if there are Includes/Extends Between UCs but no Relations Between their Classes

- Go from Class-Class to Class-UC:
  - Warn, if a Class-UC Relation is a subset of Another Class-UC Relation, but there is no Include/Rxtend Between the UCs
  - Warn if a UC isn’t Related to any Class

- Warning Timing:
  - Immediate Warnings Can be Annoying
  - Warn Only on Metrics Invocation
  - Reuse Metrics Invoked at Design Time

Example: An Initial Class Diagram
User Defines Relations from Classes to UCs

- Class Order_Container => UC Place Order
- Class Order => UC Place Order
- Class Catalog => UC Place Order
- Class Item_DB => UC Place Order
- Class Item => UC Order Product
- Class Computers => UC Order Computer
- Class Cars => UC Order Car
- Class Catalog => UC Request Catalog
- Class Item_DB => UC Request Catalog
- Class Customer => UC Supply Customer Data
- Class DB_Access => UC Supply Customer Data
- Class Payment => Arrange Payment

User Defines Relations Between Classes

- Order_Container => Order
- Order => Payment
- Order => Item
- Customer => DB_Access
- Item => Computers
- Item => Cars
- Catalog => Item_DB
- Payment => Cash_Payment
- Payment => Credit_Payment
**Class Diagrams – Warnings in Example**

- **m** User has Not defined any Relations Between Classes (Order_Container, Order) and (Customer, DB_Access)
  
  Warn: UC Order Includes UC Supply Customer Data, but no Relation Between the Related Classes

- **m** Relations of UC Request Catalog are a Subset of Relations of UC Order
  
  Warn: Include or Extend Between UC Order and UC Request Catalog Missing

- **m** UC Pay Cash and UC Pay Credit don’t have Related Classes
  
  Warn: Relate Classes to UC Pay Cash and UC Pay Credit

**Example Corrections**

- **m** Add extend between UC Order and UC Request Catalog
  
  a) New Problem: Request Catalog is more general than Place Order
  
  b) Add the following Relations:
    
    a) Order => Customer
    
    b) UC Pay Cash => Cash_Payment
    
    c) UC Arrange Credit => Credit_Payment
Corrections to Alleviate Problem

User Redefines Generality of UC Request Catalog

Class Diagrams - Generality Levels

To this Point, the User has Only Specified Generality Levels of UCs and Established Relations Among UCs and Classes

Next Step is to Assign Generality Levels to Classes and Warn Users if Any Conflicts Occur

- Warn if a Class is More General than any UC it is related to

- Warn if All Classes which are Related to a UC are More Specific than the UC

  The UC May Need to be Redefined

- Warn both Immediately and on Metrics Run
Example: Initial Marking of Class Diagrams

<table>
<thead>
<tr>
<th>Order Container G1</th>
<th>Order G1</th>
<th>Item G1</th>
<th>Catalog G1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computers G3</td>
<td>Cars G3</td>
<td>Item DB G2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Customer G1</td>
<td>DB Access G3</td>
<td>Payment G0</td>
<td>Cash Payment G1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: Warnings in Class Diagram

- **m** Order Container is Related to UC Place Order
  - **q** Warn: Class is More General than UC

- **m** Payment is Related to UC Arrange Payment
  - **q** Warn: Class is More General than UC

- **m** Computers is the Only Class Related to UC Order Computer
  - **q** Warn: All Classes Related to the UC are more Specific than the UC

- **m** Cars is the Only Class Related to UC Order Car
  - **q** Warn: All Classes Related to the UC are more Specific than the UC
Generality Level Corrections

- User makes Order Computer and Order Car more specific
- User Changes Generality of Place Order to G0
- After that User Can also Change Generality of Arrange Payment to G0

Reuse as Class Diagrams Expand

- Over Time, Class Diagrams are Expanded with Additional Details
  - Attribute Types and Function Signatures
  - Associations Among Classes
- Thus, in Addition to Relations, there are Now Dependencies Between Classes
  - Seek and Warn if Bad Couplings Detected
- Perform Class-Class to Class-UC Checks again using the New Dependencies
- Warn Immediately and on Metrics Run
- Counting of the dependencies is Possible
- Display Counting Results on Metrics Run
Example: Expanded Class Diagrams

- Warn: Credit Payment has Attribute of Type Customer, but there is No Relation Defined Between the Classes
- Class-Class to Class-UC Check Doesn’t Give Any New Information

Sequence/Collaboration Diagrams

- As Design Evolves, User Creates Sequence and Collaboration Diagrams
- Both of These Diagrams Provide Additional Dependencies to be Tracked and Evaluated
- Perform Class-Class to Class-UC Checks again using the New Dependencies
- Warn Immediately and on Metrics Run
- Counting of the dependencies is Possible
- Display Counting Results on Metrics Run
  - At this Stage, the Counting Results that are Obtained May be Close to those Results that are Obtained in Future for Actual Code
  - Save and Track Results for Subsequent Use
Example: Sequence Diagram

Warning: Payment Calls a Method of DBAccess, but is Not Related to DBAccess in any way

Example: Collaboration Diagram

Warning: Payment Calls a Method of DBAccess, but is Not Related to DBAccess in any way
Future Work: UML and Together CC

Planned Work by F. Eickhoff in Spring 2002
- Transition of UML/Reuse into Together CC
- Work at Design Level - Do Not Invoke DRE
- Ability to Transition to DRE as Code is Provided via TCC

Preparation of Two Papers for UML 2002 Conf.
- http://www.umlconference.org/
- Experience Paper on TCC + DRE
- Research Paper on Reuse in UML
- Co-Authors and Input (Examples) Welcome

Future Work - Reusability - Summary

Short Term (By 5/2002)
- Formal Reuse Model
- Work on UML and Reuse
- Finishing Touches on Prototyping
- UML/XML Conference and Journal Article

Research - Through 2004
- Focus on Comprehensive Reuse Framework
- Guided Reusability Assessment/GA Analysis
- Component Repository with Semantic Search Capabilities
- Leverage Domain/Organization Specificity
- UML, XML, and Java Components