Object-Oriented Design Methodology to Facilitate Reuse

Prof. Steven A. Demurjian and Dr. Margaretha W. Price†

Computer Science & Engineering Department
The University of Connecticut
191 Auditorium Road, Box U-155
Storrs, CT 06269-3155

steve@engr.uconn.edu
http://www.engr.uconn.edu/~steve
(860) 486 – 4818

† M. Price is a Senior Engineer at Raytheon Electronic Systems in RI. Special thanks to Jeff Ellis, Dave Mattei, Jeff Robke, Yi Fang for new material added as a result of their CSE298/300 Spring 1999 project.

Motivation

Reuse Afterthought in OO Design/Development
Majority of Reuse Focuses on “Small” Components

- String Functions, Utility Routines, GUI, etc.
- Easy to Understand - Easy to Reuse
- Minimal Savings

Three Classes of Software

- **Domain-Independent**: Libraries, Utilities, etc. Most Likely to Be Reused
- **Domain-Specific**: Dedicated Software Reused in Other Programs of Same Domain
- **Application-Specific**: Uniqueness Unlikely to Be Reused
Motivation

Three Classes of Software in a Typical Software Application

- Domain-Independent: 20%
- Application-Specific: 15%
- Domain-Specific: 65%

Motivation

- Popular OO Design Methodologies: Omit and Ignore Reuse Guidelines
- OMT and UML
- Design Patterns
- Current Research Concentrates on Consumer (Reuser) and Not Producer (Creator)
- Measure Savings from Reuse
- Calculate Return on Investment
- Two-Fold Goal
- Elevate Reuse to Equal Partner Starting with Design
- Focus on Domain-and-Organization Specific Reuse
**Why Software Reuse?**

- Increase Software Productivity
- Shorten Software Development Time
- Improve Software System Interoperability
- Develop Software With Fewer People
- Move Personnel More Easily From Project to Project
- Reduce Software Development and Maintenance Costs
- Produce More Standardized Software
- Produce Better Quality Software
- Provide a Powerful Competitive Advantage

---

**Objectives**

- **Reuse as Equal Partner Starting with Design**
  - Iterative Reusability Evaluations at Early and All Stages of Design and Development
  - Production of Large Reusable *Components*
- **Capabilities of Evaluation Techniques**
  - Identify the Reusable Portions of Design
  - Estimate/Measure Reusability Automatically
  - Provide Guidelines on Improving Reusability
  - Usable for Newly Created Designs
  - Evaluation of Legacy Code for Reuse Potential
  - Integrated in a Design/Development Environment
What is a Component?

Types of Reusable Components

- Application Template
- Data Model
- Data Structure
- System Architecture
- Process Model
- Process Definition
- Prototype
- Plan Skeleton
- User Interface Skeleton/GUI
- Process Skeleton
- Utility Components
Benefits of Component-Based Reuse

- Organizational Perspective
  - Shorten Development Time
  - Reduce Costs
  - Increase Competitiveness

- Personnel Perspective
  - Increase Productivity

- Customer Perspective
  - Achieve Greater User Satisfaction Through the Production of More Flexible Products

Objectives

Design and Implementation Process

1. Define Components, Their Interactions, and Analyze Their Reusability
2. Store an Iteration of Design
3. Implement an Iteration
4. Store and Document Iteration of Implement.
5. Reevaluate an Existing Design for
   - Correcting Errors
   - New Reuse Potential
5. Reuse Existing Design with a New Implementation
Overview of Presentation

- Cultural and Social Reuse Issues
- Component-Based Design - History & Perspective
- Subjective Identification of Components
  - General vs. Specific Classes
  - Related Hierarchies to Quantify Components
- Objective Measure of Dependencies
  - Classifying Dependencies
  - Measuring Reuse Potential
- Reuse Guidelines
  - Methodological Basis for Increasing Reuse
  - Iterative Improvement in Reusability
- Evaluation and Analysis: DRE Tool/Examples
- Conclusions and Future Research

Cultural and Social Reuse Issues
Management Support

- Motorola Study: A New Reuse Program there must have Strong/Unequivocal Management Support
- Raytheon Report: Support from Upper Management Most Important for Successful Reuse
- Why? Increased
  - Cost Associated with Constructing Reusable Components
  - Communication, Coordination
  - Education, Training
- Motorola and Raytheon Facilitate by Incentives
  - Both Producer and Consumer Benefits
  - Awards Provided for Demonstrated Efforts
Cultural and Social Reuse Issues
High Initial Cost

- Reports have Indicated
  - High Start Up Costs
  - Slow Return on Investment (> 3 years)

- Best Success in
  - Starting with Small Projects
  - Distributing Components for Reuse
  - Opportunistic Reuse

- Reuse Must be Supported by
  - Libraries to Collect, Classify, and Disseminate Components
  - Ease of use for Producer and Consumer

Cultural and Social Reuse Issues
Reuse and Software Design/Development

- Lesson Learned
  - Reuse Often Avoided by SW Engineers due to Fear of Configuration Management Problems

- How is Fear Minimized?
  - Reuse as Integral Part of Development Process
  - Reuse Early and Often
  - Tools that Facilitate Producer Logging Component and Consumer Finding Component

- Summary
  - We’ll Concentrate on Technical Reuse Issues
  - Superior Techniques Will Remain Unpopular and Unused without Associated Support
Component-Based Development Process

TOP-DOWN:
To determine what is needed to satisfy this need.

OTHERS:
Consider the similarity among concurrent projects.

BOTTOM-UP:
To determine what is available to satisfy this need.

FUTURE:
Consider the possibility of reusing in future projects.

CBD: Component-Based Development

Supply
Model New
Wrap Catalog
Edit

Consume
BPM
Application Design
Application Assembly

Manage
Contact & Communicate

Repository Services

CSE300
CSE.RU-1.16
**Components vs. Objects**

<table>
<thead>
<tr>
<th>Components</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Oriented</td>
<td>Technology-Oriented</td>
</tr>
<tr>
<td>Coarse Grained</td>
<td>Fine Grained</td>
</tr>
<tr>
<td>Standards Based</td>
<td>Language Based</td>
</tr>
<tr>
<td>Multiple Interfaces</td>
<td>Single Interface</td>
</tr>
<tr>
<td>Provide Services</td>
<td>Provide Operations</td>
</tr>
<tr>
<td>Fully Encapsulated</td>
<td>Use Inheritance</td>
</tr>
<tr>
<td>Understood by Everyone</td>
<td>Understood by Developers</td>
</tr>
</tbody>
</table>
Components as Assets can Grow
What are Component Dependencies?

- Dependency Type of Components
  - Versions
  - Aggregations
  - Functional
  - Inheritance
  - Association

- What is Impact of Each Dependency on the Reusability of a Component?

Wide Range of Component-Based Products

- Select Component Factory
- Component Manager
- C++ Synchronizer
- Estimator
- Requirement Manager
- Enterprise for Java
- Enterprise for Visual Basic
- Select SE
Component Repository

Repository Browser
Hierarchy

Multiple Support

SELECT Component Manager
Repository Portability Layer

Multiple Repository Support

UNISYS UREP
Microsoft Repository
Softlab Enabler
CBD life cycle

Business Direction

General Business Requirements

Component Development

Component Requirements

Business and Harvest

Data services

Component Management

User Development

User, Business and Services

Data services

Business and

Component Management

User, Business

Data services

Component Development

Component Requirements

Business and

Harvest

Data services

General Business

Requirements

Business Direction

Scope of traditional software development project

IDC forecast CBD market

1996: $652 million

2001: $12 billion
Component-Based Tools and Software

- Software Composition Workbench
- JavaBeans
- Visual Café, Symantec
- Visual J++
- Enabler, Softlab
- Microsoft Repository
- UREP, Unisys
- Select Software Tools, Select

Web-Site References

- Reusable Software Research Group, West Virginia University
  - [http://www.cise.wvu.edu/~resolve/scw/rsrg-brochure-nov-98.html](http://www.cise.wvu.edu/~resolve/scw/rsrg-brochure-nov-98.html)
- Reusable Software Research Group, Ohio State University
  - [http://www.cis.ohio-state.edu/rsrg/index.html](http://www.cis.ohio-state.edu/rsrg/index.html)
- Select Software Tools
- Software Reuse Executive Primer, DOD
- Model-Driven Software Reuse, Extended Intelligence Inc.
Subjective Identification of Components

- Reuse Historically Occurs at Class Level
- Class as “Atomic” Component only Scratches Surface in Reuse Potential for OO
- But, Classes Interact
  - If Reuse One, Often Need Others
  - Thus, Reuse Set of Classes
- Expand Reuse from Class to Component Level
- Establish Framework for Promoting Design Reuse
  - Characterize General vs. Specific Classes
  - Quantify Related Components
  - Illustrate via HTSS and Financial Frame Applications
- Goal: Increase Reuse Potential by Understanding Classes, Components, and their Role within Appl.

General/Specific Class Characterization

- Subjective Characterization by Software Designer
- Best Estimate on Potential Utility of Class
  - **General Class (G)**
    - Those Application Classes that Facilitate Domain-and-Organization Specific Reuse
  - **Specific Class (S)**
    - Those Application Classes that are Limited to use in a Single Application
- Purposes
  - Determine Classes with Highest **Reuse Potential** for Organization’s Future Systems
  - Dependencies from General to Specific are both Non-Reusable and Hinder Reuse
General/Specific Class Characterization

General Class (G)
- Expected to be Reused in Future Applications
- Abstract Classes/Root Classes/Non-Leaf Classes in Inheritance Hierarchies
- Domain Independent/Domain Specific
- What are Some Examples?

Specific Class (S)
- Only Applicable in Current Applications
- Unlikely to be Reused in Future Applications
- Classes that Retrieve from Company Database
- Application Specific
- What are Some Examples?

High-Tech Supermarket System (HTSS)

Automate the Functions and Actions
- Cashiers and Inventory Updates
- User Friendly Grocery Item Locator
- Fast-Track Deli Orderer
- Inventory Control

User System Interfaces
- Cash Register/UPC Scanner
- GUI for Inventory Control
- Shopper Interfaces Locator and Orderer
- Deli Interface for Deli Workers

We’ll Introduce and Utilize Throughout Lecture
The HTSS Software Architecture

IL: Item Locator
DO: Deli Orderer for Shopper/Employee

CR: Cash Register
IC: Invent. Control

Inventory Control

ItemDB Global Server
OrderDB
SupplierDB

ItemDB Local Server

CreditCardDB
ATM-BankDB

Why is Item General?

What is Applicability of Item?

```csharp
class Item {
    private: // Private Data
        int UPC;
        char* Name;
        int InStock, OnShelf, ROLimit;
        float RetailCost;
    public: // Public Methods
        Item(int code, char* str, int st1, int st2, int st3, float cost);
        void CreateNewItem();
        int Get UPC();
        char* GetName();
        int GetQuantity();
        int CheckReorderStatus();
        void PrintItem();
        void UpdatePrice(float new_value);
};
```
Another General Class in HTSS

Collection Classes of General Classes are General

```cpp
class ItemDB
{
private:
    int Num_Items;
    int Curr_Item;
    Item* AllItems[Max_Items];

    int FindFirstItem();
    int FindNextItem();
    int FindItemUPC(int code);
    int FindItemName(char* name);
public:
    ItemDB(); // Constructor
    void InsertNewItem(Item* new_one);
    void DeleteExistingItem(int code);
    void FindDisplayItemUPC(int code);
    void FindDisplayItemName(char* name);
    void PrintAllItems();
};
```

Yet Another General Class in HTSS

GUI-Based Class for Supporting Inventory Control Actions Can be Domain Independent

```cpp
class InvControlGUI
{
private:
    int Curr_Option; // Current menu option

public:
    InvControlGUI(); // Constructor
    void PrintMenuSetOption();
    void ActivateController();
    void EnterNewItem();
    void RemoveExistingItem();
    void Item();
    void InvSearchQuantity();
    void InvSearchReorder();
    void GenerateAnOrder();
};
```
Specific Classes in HTSS

General Classes are Refined to Represent Particular Items, Yielding Specific Classes

Levels of General Classes

Not All General Classes Created Equally

Level of Generality Based on Role in Application

G⁰, G¹, G², ..., S

Gᵢ views Gⱼ as General if i ≥ j
Gᵢ views Gⱼ as Specific if i < j

Purposes
- Accommodate Large Systems with Multiple, Different Reusable Components
- Reusable Components can Overlap, but Still be Distinct Reusable Units
Can you Identify Different Levels of General Classes?

Extended Hierarchy with More Levels of Generality
Properties of General/Specific Classes

- Level of Generality Strictly Ordered in Hierarchy
- A Descendant of a General Class Must have an Index Greater Than or Equal to Itself or be Specific
- A Specific Class Can only have Specific Descendants

\[ G^0, G^1, G^2, \ldots, S \]

General/Specific Paradigm in HTSS

- Abstraction from HTSS to Domain Independent
- Inventory Control Application
- Separation of Supermarket Domain Specifics
- Leverage Commonalities for
  - Focused, Independent Design/Development
  - Future Products
- Relevance
  - Domain-and-Organization-Specific Reuse
  - Expand to 24 hour Mom & Pop Stores
  - Expand to Other Retail Markets
    - E.g., Auto parts, Clothing, Toy, etc.
The FinancialFrame Application

- A Commercial C++ Framework Containing 441 Classes, Proprietary of FinancialComp
- Provides Basic Functionalities of Financial System
- FinancialFrame’s Challenge - Manage Changes
  - Framework Constantly Evolving
  - New Functions and Modify Existing Functions
- Conceptual Relevance:
  - Provide Realistic Context for Our Approach
  - Domain-and-Organization-Specific Reuse

Our Purpose: Work with Existing Code

- Establish General and Specific Classes
- Characterize Related Components
- Evaluate the Goodness of G/S Characterization and Identify Potential Problem Areas
  General Components that are Specific
  Specific Components that are General

Problematic Relevance:

- Demonstrate Ability of Approach to Localize Effect of Changes
- Describe Ways to Increase FinancialFrame’s Reuse Potential
What are Components?

- ADTs as Unit of Abstraction/Conceptualization
- Classes are OO Equivalent of ADTs
- However, in Past 10 Years
  - Computing Power has Exploded
  - Application Complexity has Increased
  - Classes are Part of Inheritance Hierarchy
  - Inheritance Hierarchy Part of Application Class Library
- In Past 2-3 Years We’ve Seen
  - Emergence of Java
  - Emergence of Java Beans
  - Component-Based Development Tools
What are Components?

How are Applications Conceptualized?
- Inheritance Hierarchies Partition Domain
- Packages as Collections or Related Classes
- Collections of Classes, Packages, Inheritance Hierarchies form Application Class Library

How are Class Libraries Utilized?
- Use Individual Classes
- Use Package or Subset of Package
- Use Major Portions of Inheritance Hierarchies
- Tools Use at Most a “Few” Select Packages and/or Hierarchies
- Tools that Span Application Classes Represent Poorly Designed Software

Defining Component Concepts

A Component is Composed of One or More Classes (or Other Components) and is Intended to Support a “Constructed” Unit of Functionality

Classes Can be Utilized in Multiple Components

A Class Utilized in Multiple Components Maintains the “Same” Semantics in All of its Contexts

Our Interest Involves:
- Reusable Classes
- Reusable Components
- A Reusable Component Consists of Classes and/or Other Components that are Expected to be Reused Together in Future Applications
Related Classes and Hierarchies

- Class X is Related to Class Y if they are Related and Concept and are **Expected to be Reused Together** in Future Systems
- Class X Related to Class Y is **Subjectively Assigned** by Software Engineer (Producer)
- When Class X is Related to Class Y
  - X and All of X’s Descendants are Related to Y and All of Y’s Ancestors
  - Thus, to Reuse X or X’s Descendants, you Must Reuse B and All of B’s Ancestors
  - Class X Related to Y if Y at Same or Higher Level!
- Related Classes Promote Reuse, Since They are Expected to be Reused Together

Related Hierarchies/Reusable Components

- Two Sub-Hierarchies are Related if to Reuse One, you Must Reuse the Other
- Purpose: Identify Reusable Dependencies Among Related Classes

Reusable Component: A Set of Related Classes that are Expected to be Reused as a Group
Related Characterization in Levels of Components - Financial Frame

- Bond (G')
  - R
  - YieldModel (G'')
    - IndexBond (G')
    - Discount (G')
    - Bill (G')
    - ...

- Bond is Related to IndexBond
- When Bond is Reused, so Must IndexBond and YieldModel
- Hence, IndexBond and its Ancestor (YieldModel) are Reused!

Related Characterization in Levels of Components - HTSS

- Item (G'')
  - NonPerishItem (G')
  - PerishItem (G')
    - R
    - Environ(G')

- Does R from Environ to PerishItem Make Sense?
- Should R be from PerishItem to Environ?
Related Characterizations in Levels of Components - FinancialFrame

Classes/Sub-Hierarchies can be Related if and only if Classes (or Sub-Hierarchy Roots) are General and at the Same or Higher Level

- Root classes of strategies and other most General classes (Main) ($G^0$)
- Bond or other strategy Components ($G^1$)
- Classes specific to FinancialComp ($G^2$)
- Specific applications at FinancialComp (S)

Related Characterizations in Levels of Components - HTSS

- Root classes for Items, ItemDB, etc., which are Most General ($G^0$)
- Inventory Control/Other Components ($G^1$)
- Classes Specific to Grocery Store Domain ($G^2$)
- Specific Applications for Big Y or Shaw’s or Stop/Shop (S)

Where do Changes for Other Domains Occur?
Reusability in HTSS Domain

Root classes for Items, ItemDB, etc., which are Most General. 
(G^0)

Classes Specific to Grocery Store Domain. (G^2)

Specific Applications for Big Y or Shaw’s or Stop/Shop (S)

Inventory Control/Other Components. (G^1)

Inventory Control Tool for Ordering Items from Suppliers

Cost Accounting Tool for Tracking Net and Gross Profit

Classes for Large Supermarket

Classes for Specialty Supermarket

Classes for 24 Hour Convenience

Inventory Control

Net and Gross Profit
What are Dependencies Among Classes?

Object Inclusion: Class Contains a Instance of Another Object

Attribute Definition: Class Contains Attribute that is the Type of Another Object

Method Invocation: Class Invokes a Method Defined on Another Object

Goals

- Classify and Understand Dependencies
- Assess “Good” vs. “Bad” Dependencies
- Change “Bad” to “Good” by
  - Changing Class from S to G or G to S
  - Moving Code and/or Method Calls
  - Splitting a Class into Two Classes
  - Merging Two Classes

Reusing Sub-Hierarchies in Different Components - FinancialFrame

Dependencies Among General Related Classes

- Not a Hindrance to Reuse
- Represents Valuable Design Knowledge

Will be reused with Components utilizing FinancialComp’s database

Will be reused with Components utilizing another company’s databases
Transitivity in Inheritance and Related Relationships

**Base Case:** Related Characterization is *Transitive*, but not *Commutative*

![Diagram](image)

**Case 1:** A is *not* related to X
- Dependencies from A to X are *not* Desirable
- Recall: We Reuse X and All of Its Ancestors, But Not B’s Ancestors to Reuse A

Will be reused with Components for another domain, e.g., Toy Store

Will be reused with Components for different Supermarket Companies
An Example of Case 1 in FinancialFrame

FinancialCompMain R FinancialCompTrader
Thus, Dependencies Between are Desirable
Main is not R to FinancialCompTrader
Thus, Dependencies Between Hinder Reuse
For “Blue” Component, we Don’t Want to Have to Reuse FinancialCompTrader with Main!

Transitivity in Inheritance and Related Relationships

Case 2: X is Related to A
Dependencies from X to Both A and B are Desirable
When Reuse X, Since X Related to B, we Reuse B and All of Its Ancestors (A)
Thus, Dependencies Between X and A are Okay!
Evaluative Metrics and Methodology

Objective Measures of Dependencies

- **Object-Oriented Design**: Collection of General and Specific Classes, with Related Characterizations
- **Recall Dependencies Among Classes**
  - **Object Inclusion**: Another Instance within Class
  - **Attribute Defn.**: Attribute Type of Class
  - **Method Invocation**: Defined on Another Class
- **Quantify Dependencies for Reuse**
  - **Good**: Promotes Reuse - Leave Alone
  - **Bad**: Hinders Reuse - Try to Change
  - **Okay**: No Impact on Reuse

**Goals**
- Classify and Understand Dependencies
- Measure Reuse Potential

An Example of Case 2 in FinancialFrame

- **Class Bond** $R$ to both **IndexBond** and **YieldModel**
- Thus, Dependencies from Bond to IndexBond and YieldModel are Desirable and Reusable!
- When Bond is Reused, So is IndexBond and YieldModel!

![Diagram showing dependencies between Bond, IndexBond, YieldModel, and other classes]
### Dependencies Among Related Classes

- **Remember, G/S are Subjectively Assigned by Software Designer**
- The Two G classes are Related
- Related Classes are Intended to be Reused Together

![Diagram showing dependencies among classes](image)

### Sample Dependencies in HTSS

- **InvCont and Item are Related Classes**
- InvCont to Item Dependencies are Good: Reused Together
- Dependency from InvCont to DeliItem is Problem
  - Don’t Want to Reuse InvCont with DeliItem
  - ManagerGUI with InvCont Includes Useless DeliItem
- Dependencies from DeliIC to Item and/or DeliItem
  - Don’t Impact Reuse
  - Can Reuse Item and DeliItem w/o DeliIC
Dependencies Among Non-Related Classes

- Remember, G/S are Subjectively Assigned by Software Designer
- The Two G Classes are Not Related
- Non-Related Classes are *NOT* Intended to be Reused Together

Sample Dependencies in HTSS

- InvCont and Person are Classes that are Not Related
- InvCont to Person or Shopper Dependencies are Bad
  - Don’t Want to Reuse Person/Shopper with InvCont
  - Must Reuse - Problem!
- Dependencies from DeliIC to Person and/or Shopper
  - Don’t Impact Reuse
  - Can Reuse Person and Shopper w/o DeliIC
  - However, Poor Design if DeliIC Needs Person or Shopper!
Summarizing Couplings of Related Classes

<table>
<thead>
<tr>
<th>Type 1: Good for Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two General Classes are Reused Together</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type 3: Bad for Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>General to Specific</td>
</tr>
<tr>
<td>To Reuse, Specific Must be Included</td>
</tr>
<tr>
<td>Added Functionality with No Purpose</td>
</tr>
<tr>
<td>Change to Type 1 or 5/7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Types 5/7: Okay for Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Impact</td>
</tr>
<tr>
<td>Specific Classes Not Reused in New Application</td>
</tr>
<tr>
<td>May Improve Reuse if Changed to Type 1</td>
</tr>
</tbody>
</table>

Summarizing Couplings of Non-Related Classes

<table>
<thead>
<tr>
<th>Type 2: Bad for Reuse - Two General Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Expected to be Reused Together since Not Related</td>
</tr>
<tr>
<td>Change to Type 6/8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type 4: Bad for Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>General to Specific</td>
</tr>
<tr>
<td>To Reuse, Specific Must be Included</td>
</tr>
<tr>
<td>Added Functionality with No Purpose</td>
</tr>
<tr>
<td>Change to Type 6/8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Types 6/8: Okay for Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Impact</td>
</tr>
<tr>
<td>Specific Classes Not Reused in New Application</td>
</tr>
</tbody>
</table>
Dependencies in Levels of Components
Summarizing Related Classes

G^0 Related to G^0
G^2 Related to G^2

Dependencies in Levels of Components
Summarizing Non-Related Classes

m Dependencies Among Unrelated Classes Always
\( \downarrow \)
- Bad for Reuse
- No Impact on Reuse
### Sample Actions to Improve Reusability

<table>
<thead>
<tr>
<th>Coupling</th>
<th>Among Related</th>
<th>Among Unrelated</th>
</tr>
</thead>
<tbody>
<tr>
<td>G G</td>
<td><strong>Type (1)</strong></td>
<td><strong>Type (2)</strong></td>
</tr>
<tr>
<td></td>
<td>Good for Reuse</td>
<td>Bad for Reuse</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Move Src./Dst. to Specific</td>
</tr>
<tr>
<td>G S</td>
<td><strong>Type (3)</strong></td>
<td><strong>Type (4)</strong></td>
</tr>
<tr>
<td></td>
<td>Bad for Reuse</td>
<td>Bad for Reuse</td>
</tr>
<tr>
<td></td>
<td>Move Dst. to General</td>
<td>Move Src. To Specific</td>
</tr>
<tr>
<td>S G</td>
<td><strong>Type (5)</strong></td>
<td><strong>Type (6)</strong></td>
</tr>
<tr>
<td></td>
<td>Can Improve if Move Src. to General</td>
<td>No Impact on Reuse</td>
</tr>
<tr>
<td>S S</td>
<td><strong>Type (7)</strong> - Can Improve if Move Src./Dst. to Gen.</td>
<td><strong>Type (8)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No Impact on Reuse</td>
</tr>
</tbody>
</table>

### Reuse Guidelines

- **Methodological Basis for Increasing Reuse**
  - Designer Supplies General/Specific/Related for the Classes/Hierarchies in Application
  - Reuse Analysis Tool Calculates Couplings and Identifies Types of Reuse (Good, Bad, Okay)

- **Ideal Result: Maximize Reuse via Couplings**
  - Type 1: G to G for Related Classes
  - Type 8: S to S for Non-Related Classes

- **Extended Guidelines: Menu of Choices**
  - Different Ways to Move Couplings
  - Considers Impact of Movement on Design

- **Goal: Iterative Improvement in Reusability**
Extended Guidelines for Improving Reusability

- Core Guidelines Emphasize Local Behavior
- To Remove “Bad” or Improve “Okay” Coupling, Move Cause of the Coupling (e.g., Method Call)
  - Moving a General Method to Specific Class
  - Moving a Specific Method to General Class
- Moving Method Up/Down a Hierarchy, Which Alters the Coupling, Can Impact Elsewhere
- Expand the Analyses to Other Couplings of the Source and Destination of the Bad Coupling
  - Couplings to Source when Moving Down
  - Couplings to Destination when Moving Up
- Extended Guidelines: Menu of Choices
Extended Guidelines to Improve Reuse

Identifying the Problem

Suppose

What has Occurred?

Recall

Type 3

Type 7

Identifying the Problem

Extended Guidelines to Improve Reuse

Identifying the Problem

Suppose

What has Occurred?

Recall

Type 1

Type 5

Type 1

Type 5
**Problem and Solution**

- Focus on Core Guidelines May Ignore the Impact of a Change for Other Related and Coupled Classes
  - When a Coupling is Moved from a G to S Class
    - Examine All Existing Coupling to G Class
    - G to G Coupling Now G to S
    - We’ve Introduced a “Bad” Coupling
  - Likewise, When a Coupling Moved from S to G
    - Examine All Existing Coupling to S Class
    - S to S Coupling Now S to G
    - We’ve Introduced a “Bad” Coupling

- Solution: Extended Guidelines to Govern all Potential Scenarios for Removing “Bad” Couplings

---

**Extended Guidelines for Type 3 Couplings**

- Move Coupling Dst. to General Class or Change Dst. To a General Class
  - Type 3 to Type 1
  - May Introduce Couplings from G Dst. to Specific Classes
- Move Coupling Src. to Specific Class or Change Src. To a Specific Class
  - Type 3 to Type 7
  - May Introduce Couplings from General Classes to Specific Src.
- Change to Non-Related/Follow Type 4 Among Related
- Detailed Evaluation of Implementation
- Key Concerns
  - Local Changes with Global Impact
  - “Wrong” Choice Depreciates Reuse
Removing Type 3 Couplings in HTSS
Which Changes Make Sense?

- Change InvCont to S or DelItem to G
  - Neither Makes Sense
  - Against Design Intent!
- Move Coupling Dst. to General Class
  - Find Problem Method Call, Attribute Access, Object Inclusion
  - Move from DelItem and Item
  - Type 3 to Type 1
- Move Coupling Src. to Specific Class
  - Find Problem Method Call, Attribute Access, Object Inclusion
  - Move from InvCont and DelIC
  - Type 3 to Type 7
- Detailed Evaluation of Implementation
- Note: Maintain Application Semantics

Extended Guidelines for Type 2 Couplings

- Move Coupling Src. to Specific Class or Change Src. To a Specific Class
  - Type 2 to Type 6
  - May Introduce Couplings from General Classes to Specific Src.
- Move Src. and Dst. to Specific Classes
  - Type 2 to Type 8
  - May Introduce Couplings from G Src. to Specific Dst.
- Move Coupling Dst. to Specific Class or Change Dst. To a Specific Class
  - Follow Type 4 Guidelines
- Change to Related/Type 1/Design Impact Must be Evaluated!
- Detailed Evaluation of Implementation
Extended Guidelines for Type 4 Couplings

- Move Coupling Dst. to General Class or Change Dst. To a General Class
  - Type 4 to Type 2 - No Help
- Move Coupling Src. to Specific Class or Change Src. To a Specific Class
  - Type 4 to Type 8
  - May Introduce Couplings from General Classes to Specific Src.
- Change to Related/Follow Type 3
- Detailed Evaluation of Implementation

Summary on Extended Guidelines

- Total Alternatives for Removing “Bad” Couplings:
  - Type 2, 3, 4: Seven Possibilities Each: 21 Total
  - Type 5, 7: 3 Total
- Changing from G to S or Movement of Coupling Potential to Impact:
  - Couplings to Source
  - Couplings from Destination
- Result: Movement May Decrease Reuse Potential!
- Two-Fold Solution:
  - Design Support for OO Reuse Metrics and Evaluation within UML, Design Patterns, etc.
  - Analytical Tool for Evaluating Reuse Potential of C++, Ada95, or Java Applications/Libraries
**Evaluation and Analysis**

**Design Reusability Evaluation (DRE) Tool**
- Java-Based Tool for Analyzing Reusability
- Takes C++, Ada95, and Java Code as Input
- Works with General/Specific/Related as Subjectively Defined by Software Designer
- Analyzes Couplings to Identify, for Each Type (1 to 8), the Number of Couplings
- Allows Designer to Investigate Cause of and Correct “Bad” or Improve “Okay” Couplings

**DRE can be Utilized in Different Ways**
- Evaluate Evolving Design/Implementation
- Investigate the Reusability of Legacy Code
- Examples of Video Rental System/FinancialFrame

**Utilizing Reuse Methodology**

**Evaluate Evolving Design/Implementation**

**Constructing New Applications**
- Software Design Proceeds in Stages
- Today’s Norm: Incremental Development and Rapid Prototyping

**General/Specific Classes/Related Components**
- Assigned Initially as Classes are Generated
- Refined Throughout Increments/Versions
- G to S, S to G, etc.
- Related Components as Design Begins to Mature with Additional Details
- Use Methodology to Find/Correct “Problems”

**Video Rental System Test-Bed**
Utilizing Reuse Methodology
Investigate Reusability of Legacy Code

Reusability of Legacy Code
- Examine Legacy Code in Detail
- Talk/Contact Domain Experts with Corporate Knowledge of Code

General/Specific Classes/Related Components
- Take “Educated Guess” for G/S Classes and Related Components
- Run DRE and Find/Correct Problems
- Re-evaluate Legacy Code with Different “Educated Guesses”
- Compare/Contrast Results to Identify the “Best” way to Characterize Classes/Components

Financial Frame as a Test-Bed

The Video Rental System (VRS)

VRS is for On-Line (Browser) Rental Tapes
- Maintains Customer and Video Databases
- Tracks Borrowing/Returning of Tapes
- Logs Rented Tapes
- CGI, C++, Netscape/Explorer
- From Video Rental to Auto Parts
- Undergraduate Project: Spring 1997 - VRS-1
- Repeated as Grad Project: Fall 1998 - VRS-2

Goals:
- Demonstrate General/Specific/Related Ideas
- Incorporate/Reuse Design in Future System
- Study Effectiveness Approach in Identifying and Removing Non-Reusable Dependencies
General and Specific Classes in VRS-1

- StoreInterface (G) ▲ VideoStoreInterface (S)
- CustomerInterface (G) ▲ VideoCustomerInterface (S)
- VideoStoreInterface (S)
- VideoCustomerInterface (S)
- Customer (G) ▲ VideoCustomer (S)
- CustomerInterface (G)
- VideoCustomerInterface (S)
- CustomerDB (G) ▲ HistoryList (G)
- RentedTapeDB (S)
- Transaction (G)
- RentedTape (S)
- Item (G) ▲ ItemDB (G)
- VideoTapeDB (S)
- VideoTapeDB (S)
- VideoTape (S)
- VideoTapeDB (S)

DRE

[Diagram of Design Reusability Evaluation tool]

- Directory: /usr/local/vrs/
- General Classes
- Related Classes

Select: Calculate Couplings  Exit
DRE

DRE

DRE

46
DRE and VRS-1
Tracking Incremental Versions

<table>
<thead>
<tr>
<th>Type</th>
<th>Metr</th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
<th>V</th>
<th>VI</th>
<th>VII</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>CC1</td>
<td>8</td>
<td>54</td>
<td>79</td>
<td>109</td>
<td>166</td>
<td>167</td>
<td>193</td>
</tr>
<tr>
<td>Bad</td>
<td>CC2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bad</td>
<td>CC3</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Bad</td>
<td>CC4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Impr</td>
<td>CC5</td>
<td>0</td>
<td>19</td>
<td>28</td>
<td>39</td>
<td>94</td>
<td>92</td>
<td>75</td>
</tr>
<tr>
<td>Impr</td>
<td>CC6</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>18</td>
<td>28</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>Okay</td>
<td>CC7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Okay</td>
<td>CC8</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>21</td>
<td>22</td>
<td>26</td>
</tr>
<tr>
<td>LOC</td>
<td>410</td>
<td>1386</td>
<td>2088</td>
<td>2695</td>
<td>3784</td>
<td>3824</td>
<td>3867</td>
<td></td>
</tr>
</tbody>
</table>

Final General/Specific Classes in VRS-2 and Some Related Characterizations
DRE and VRS-2
Tracking Incremental Versions

<table>
<thead>
<tr>
<th>Type</th>
<th>Metr</th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>CC1</td>
<td>17</td>
<td>26</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>Bad</td>
<td>CC2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bad</td>
<td>CC3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Bad</td>
<td>CC4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Impr</td>
<td>CC5</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Impr</td>
<td>CC6</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Okay</td>
<td>CC7</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Okay</td>
<td>CC8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LOC</td>
<td></td>
<td>17</td>
<td>32</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>G/S</td>
<td>Cls.</td>
<td>5/1</td>
<td>7/6</td>
<td>7/6</td>
<td></td>
</tr>
</tbody>
</table>

How is it Resolved?

- DRE and VRS

One Type 3 Coupling: G to S Dependency

- What is the Problem?
- How is it Resolved?

Customer (G)

```
take_item()
```

Transaction (G)

```
calculate_currentdate()
calculate_duedate()
write_to_historyfile()
write_to_checkoutfile()
```

VideoTransaction (S)

```
write_to_checkoutfile()
```
Resolving Type 3 G to S Dependency

Customer (G)
\texttt{take\_item()} \rightarrow...
\texttt{tr\rightarrow calculate\_currentdate()}
\texttt{tr\rightarrow calculate\_duedate()}
\texttt{tr\rightarrow write\_to\_historyfile()}
\texttt{take\_item\_specific()}

Transaction (G)
\texttt{calculate\_currentdate()}
\texttt{calculate\_duedate()}
\texttt{write\_to\_historyfile()}

VideoCustomer (S)
\texttt{take\_item\_specific()}

VideoTransaction (S)
\texttt{write\_to\_checkoutfile()}

{((VideoTransaction *) tr) \rightarrow write\_to\_checkoutfile()}

The FinancialFrame Application

Initial Assumptions on G/S and Related

- FinancialFrame Composed of Multiple Algorithm Families
- All Root Classes are G(0) with Subclasses G(1)
- All Other Classes are G(0)
- Use Actual Couplings in Code to Define Related Characterization

- Bond (G^0)
- YieldModel (G^2)
- IndexBond (G^1)
- Discount (G^1)
- Bill (G^1)
- ...
Evaluation of FinancialFrame Possible Scenarios

- Dependencies from G^0 to G^1 Classes
  - Occurring from a General Class (Bond) to Classes Specific to a Component (IndexBond)
  - Such Couplings Require a Specific Class that is Not Needed to be Reused with the General Root
  - Reported as Undesirable Couplings (Type 3/4)

Revising FinancialFrame Application

- Bond is Changed from G^0 to G^1
- A Component Consisting of Bond, IndexBond, and YieldModel is Defined
- When Bond Reused, So Must IndexBond and YieldModel - Other Classes Not Needed
- However, YieldModel can Be Reused in Isolation
- Thus, Modifications to Component Only Affect Reusers of Component
Evaluation of FinancialFrame Possible Scenarios

Dependencies from $G^1$ to $G^0$ Classes
- Occurring from a Specific Class to Other Classes Specific & Unrelated to a Component
- Bond Related to $G^1$ Class Outside Component
- Such Couplings Require a Specific Class in One Component to be Reused in Another Component, Where it is Not Needed
- Reported as Undesirable Couplings (Type 2)

Reusability of FinancialFrame Components

Undesirable Couplings Identified via Either Scenario can be Removed to Increase Reuse
Results of Such Actions w.r.t. FinancialFrame
- $G^0$ Classes can be Reused with $G^1$ Classes
- When a $G^1$ Class is Modified, Only Users of Particular Component(s) are Notified
  **Bad Coupling No Longer Present!**
- Each Component Can be Reused “As is”
- Classes Outside of Component No Longer Needed

Cannot Illustrate Due to Proprietary Software
Similar Changes to VRS-2 Occur
DRE Tool - Revised Version Spring 1999

- DRE Tool Provides Calculation of Code on Reusability Scale
- DRE Calculations Based on Counting Couplings and Determining their Effect on Reuse
- Rationale: OO Developer Incrementally Runs Code through DRE
  - Coupling Counts Provide Quality of Reuse Assurances
  - Reuse Bugs May Be Identified and Fixed Before Becoming Problems
- Promotes Better OO Principles
- Results in “More Reusable” OO Designs/Software

DRE Process

- DRE Process: User Role
  - Step 1: User Selects DRE Mode
    - Currently Available in Java and C++ Mode
    - Ada95 Version May Exist
  - Step 2: User Selects Directory of Code

- DRE Process: Tool Role
  - Step 3: Tool Parses through Source Code in Selected Directory
    - Step 4: Tool Returns Class Names and Class Pairs to User Interface
DRE Process

DRE Process: User Role
- Step 5: User Selects General Classes
- Step 6: User Selects Related Classes
- 7: User Requests Calculation of Coupling Counts

DRE Process: Tool Role
- Step 8: Tool Makes Calculation Based on User’s Input
- Step 9: Tool Returns Coupling Counts to User

Can Present Calculations Only
Can Present Calculations, Possible Errors, Improvement Ideas
DRE Process

DRE Process: User Role
- Step 10: User May Select Simulation of Changes

DRE Process: Tool Role
- Step 11: Tool Outlines Possible Improvements

DRE Process: User Role
- Step 12: User Chooses Improvements

DRE Process: Tool Role
- Step 13: Tool Recalculates Coupling Values
DRE Modifications

Tool Created in Java 1.0.x by Margie Price
Most Recent Upgrade (Ellis et al., Fall 1998)
Identified Several Key Upgrades

- #1: Documentation and Comments
- #2: Simulation Tool
- #3: Characterization File
- #4: Online Help
- #5: Continued Conversion of AWT Components and Upgrade of GUI
- #6: Ada95 Integration
- #7: Data Type Reassignment
- #8: Better Separation of Concerns

#1: Documentation and Comments
- Development of DRE Developers’ Guide
  - Effective API of classes, methods comprising DRE
  - Intended for use by future DRE developers
- 300+ Lines of Commented Code Added

#2: Simulation Tool
- C++ Mode
  - Simulator repaired from previous developer bugs
    and Java version updates
- Java Mode
  - Errors identified and methodology presented to remove them
DRE Modifications

#3: Characterization File
- Storage of User’s Defined General, Related Specifications for Project
- File Storage and Retrieval Regained

#4: Online Help
- Instruction Window Replaced by Help Files
- Tips Window Removed from User Interface

#5: Upgrade of GUI
- Final AWT Objects Removed
- Main Interface Upgraded
- Simulation Front-End Redesigned

#6: Ada95 Integration
- Not Attempted

#7: Data Type Reassignment
- Attempt to Take Advantage of Java APIs
- Standard Java Naming Conventions Applied

#8: Separation of Concerns
- Further Separation Between User Interface and Functionality
- Redeployment of Methods and Member Variables to More Logical Locations
Future Directions of DRE - CBD

Marriage of Component Based Design and Design Reuse Evaluations

“What is a Component?”
- A Set of Reusable Classes that Performs a Series of Functions
- Self-Contained, General Objects that May Be Useful in Multiple Situations

Types of Components
- User-Defined
- Third-Party Open
- Third-Party Encased

User-Defined Components

General/Specific, Related Measurements:
- Hierarchy of classes is General
- Classes are Related to each other
- Classes are not Related to non-component classes

Unnecessary To Parse Component Code
- System reusability metrics independent of internal Component reusability metrics

External Couplings to Components
- One way, from Specific/General, Related/Unrelated to General, Related
- Always CC type #1,5 - Good For Reuse
Future Directions of DRE - CBD

User-Defined Components Implementation
- User Identification of Classes that Comprise a Component
- Component Classes Parsed without Coupling Calculations
- Component Classes Set to General, Related to All Other Classes
- Regular Classes’ Couplings on Objects, Methods of Component Classes Set to Type 1,5
- Simulation Tool Prevented From Redefining Component Classes

Outcome: System Reuse Independent from Component

Third-Party Open, Third-Party Encased
- Similar to User-Defined
- Internal Parsing of Couplings Unnecessary
- Always Present Good Reuse in Design

Third-Party Open Implementation
- Same as User-Defined

Third-Party Encased Implementation
- Requires Parsing of Standard Component API
- Two Approaches
  - Write separate parser for each third-party API framework
  - Assume unknown class names are Components (current design)
Coupling Counts 1,5 - Good For Reuse??

- Coupling Count Type 1
  - General Source Related to General Destination
  - Always Good for Reuse

- Coupling Count Type 5
  - Specific Source Related to General Destination
  - Price Dissertation Presents as “Improvable”
    - No harm in current situation
    - Possibility of rearranging couplings to get Type 1
  - CBD Renders Type 5 “Unimprovable”
    - Defined Component cannot be extended
    - Reason for improvement disabled
  - In CBD, Always Good for Reuse

Future of DRE - Multilevel Generals

- Realization of General/Specific as Spectrum as Opposed to Binary Choice

- Current Tool Asks: “Is this class General?” not “What level of General Class is this?”

- Coupling Count Basis of Reuse Calculation Directly Affected by Generality Levels
Future of DRE - Multilevel Generals

- Coupling Counts Based on Cartesian Product of General(G0)/Specific(Gx,S) Nature of Two Components over Related and Unrelated Hierarchies

- Multiple Levels of Generality Destroy Price-Defined Coupling Counts
  - G0-G0, Related - Type 1
  - G5-G5, Related - Also Type 1?

- New System of Metrics Is Necessary

Future of DRE - Multilevel Generals

- Possible Metrics Solutions and Problems
  - Qualitative Assessment of Reuse
    Desired Effect (Continual Code Modification)
    Perhaps Missing
  - Continue with 8 CCs, Assign Less-General Class as Specific
    Few General-General Measurements Will Result
    (Producing Bad or “Improvable” CCs)
    G5-G5 Coupling Equal to G0-G0 Coupling
  - Separate CC For Each Level of Generality
    Requires 2*n^2 CCs for n levels
    At what point is differentiation impossible?
Future of DRE - Multilevel Generals

Possible Metrics Solutions and Problems cntd.
- Ignore Multiple Generals, Calculate as Normal Point of Multiple Generals?
- Windowing Technique - Set # of CCs, Groups of Generals
  i.e. 16 CCs, 3 groups of Generals
  No distinction made in greater numbers of multiple generals
- Couplings Worth Different Amounts, Depending on Locations
  Requires complicated metrics calculations
  May produce few Specific-Specific dependencies

Future of DRE - Multiple Generals

Best solution: Couplings Worth Different Amounts, Depending on Location

Example: System with 6 Levels of Generality, G0 (Most General) through G5 (Specific)
- G0-G0 Coupling Among Related Classes Gets Maximum Value for Coupling Type 1
- G4-G4 Coupling Among Related Classes
  Still Type 1, Good For Reuse
  On reusability scale, least reusable, gets lowest Coupling Value
- Default Type as Specific Reduces Low Specific-Specific Problem
Future Directions of DRE

* CBD and Multiple Generals Are High-Level Concepts, Only to be Undertaken in Major Revamps of Current Tool
  * CBD Alteration
    * Requires Specification of Component Types
    * Requires Redesign of Parser, User Interface, and Metrics Calculator
  * Multiple Generals Alteration
    * Requires Major Overhaul of Coupling Counts Theory
    * Requires Major Redesign of Metrics Calculator and User Interface

DRE Analysis

* “Is DRE stable for production?”
  * No

* “Is DRE stable for UConn?”
  * Relatively
DRE Problems

m **DRE Written in Java**

q Problem: Continued Development of Language Has Unintentional Effect of Deprecations
Solution: DRE Tool must be upgraded often

m **Parser Incomplete**

q Problem: Coding Styles Vary, and Possibilities Are Many
Solution: Continue to test DRE on code samples and modify source when necessary
- Problem: Already has resulted in “Hack” code
Solution: Incorporate a “real” parser into DRE
- Problem: Bulky

m **Separation of Concerns Plagues DRE Code**

q Problem: Bad OO Design
Solution: Incrementally remove internal dependencies and relocate functions
- Problem: Complicated code, results in “Hack” code
Solution: Take DRE Specs and start again
- Problem: Major overhead
Solution: Ignore it
- Problem: Hard to upgrade

m **Public Use of DRE**

q Problem: Purpose of Software without Users?
Solution: Fix all other problems
Conclusions

- Comprehensive Reusability Model and Evaluation
  - General/Specific/Related Characterizations
  - Capture/Analyze Reusability of OO Software
  - Achieve Highest Gain Through Early Identification and Reusability Evaluations!

- Methodology and Empirical Verification
  - Guidelines to Encourage Reuse
  - Applicable to Reuse of Designs and Code
  - DRE Tool for C++, Ada95, Java
  - Strong Demonstration of Domain-and-Organization-Specific Reuse

Future Research

- The Future Directions of Reuse
  - Changing Company Culture to Encourage
  - Providing Sufficient Rewards for Utilization
  - Cost and Time in Establishing Reuse Program
  - Short-Term Costs vs. Long-Range Benefits

- Future/Ongoing Research
  - Collaboration on Large-Scale “Real” Examples
  - Integration of OO Reuse into UML and Associated OO Design/CASE Tools
  - Improvements and Enhancements to DRE
  - Investigation of Reusable Component Tools/Libraries that Log and Find “Best” Component