WISR'93: 6th Annual Workshop on Software Reuse Summary and Working Group Reports

Jeff Poulin and Will Tracz
IBM Federal Systems Company,
Owego, NY
{ tracz or poulinj } @vnet.ibm.com

The Sixth Annual Workshop on Institutionalizing Software Reuse (WISR '93), hosted by IBM Federal Systems Company (FSC), took place Nov. 2-4, 1993 in Owego, NY. Almost 80 experts representing more than 60 industry, academic and government organizations worldwide gathered to share problems and solutions in adopting software reuse. The workshop began with an introductory session where participants presented current and critical issues based on position papers they submitted to the workshop. Attendees then divided into eight working groups that covered a wide range of topics, including:

- **Design for Reuse and Object Oriented Reuse Methods:** led by Doug Lea (SUNY Oswego/Syracuse CASE Center) and Bill Frakes (Virginia Tech)
- **Reuse Tools and Environments:** led by Becky Joos (Motorola) and Tim Stockwell (The Mitre Corporation)
- **Language Issues for Generic Code Architectures:** led by Larry Latour (University of Maine, Orono) and Ira Baxter (Schlumberger)
- **Hybrid Reuse with Domain-Specific Kits:** led by Martin Griss and Kevin Wentzel (both of HP Laboratories)
- **Reuse Education:** led by Ben Whittle (University of Wales, Aberystwyth & University of York) and Trudy Levine (Fairleigh Dickinson University)
- **Technology Transfer:** leg by Sidney C. Bailin (CTA, Inc.) and Guillermo Arango (Schlumberger Laboratory for Computer Science)
- **Management Issues:** led by Patricia Stump (IBM Endicott) and Terry Huber (DSD Laboratories, Inc.)
- **Formal Methods and Certification of Reusable Components:** led by Maureen Stillman (Odyssey Research Associates)

The WISR'93 workshop revealed how far reuse technology has advanced in recent years. Current research focuses not only on "reuse-in the small," or the sharing of small utilities and functions, but on "reuse-in-large." Approaches to large scale reuse include the building of programs from entire subsystems of existing software to the building of generic frameworks that represent the structure of entire classes of application programs.

This report was constructed by editing the individual working group reports. Complete versions of working group reports are available from the ftp site listed at the end of this summary.

Design for Reuse and Object Oriented Reuse Methods

Bill Frakes, Virginia Tech
frakes@sarvis.cs.vt.edu
Doug Lea, Syracuse CASE Center and SUNY Oswego
lea@sunyo.edu

The goal of this working group was to produce a set of language independent principles for "design-for-reuse", while also capturing:

- common realizations of the principles as they apply in different languages (especially C++, C, and Ada),
- the rationale for each principle, the rationale against competing principles, and
- the decision/debate process by which agreement on each principle was reached.

This work built on two previous design working groups at WISR '91 and WISR '92.

In the course of discussion, the group made progress toward defining an architectural reference model and accompanying terms and concepts providing a conceptual framework for the design of reusable components. The "3C" (Concept, Content, Context) model of component reuse was helpful in this work. The following major points of reuse design are salient:

- Descriptive and computational properties are ENCAPSULATED into COMPONENTS (objects, modules, packages, subsystems).
- A component provides access to SERVICES by possibly many CLIENT components through an established INTERFACE specification representing a single "concept."
- To fulfill its role, a component may interact with possibly many HELPER components, as constrained by an IMPORT specification or CONTRACT describing invariants necessary in carrying out its implementational "content."
- A single PARAMETERIZED description suffices to GENERATE components with the same high-level import and export specifications, but under operating different "contexts" (policies, control regimes, type substitutions, etc.).

About fifty design rules, principles, and ideas (most originating in previous WISR meetings) were grouped into further categories based on this model. The next level of grouping includes:

1. **Component Structure**
   - Identify and encapsulate commonality and variability.
   - Separate interfaces and implementations.
   - Identify and isolate context and policy from functionality.
   - Link documentation to code.
   - Link tests to code.
   - Use tools when target languages do not support sufficient interface, composition, and/or parameterization constructs.
2. **Interfaces**
   - Minimize the number of names per name space (scope).
   - Minimize implementation-dependence of interfaces.
   - Refine interfaces by extending and adding properties.
   - Optimize components via specialization.
3. **Composition**
   - Identify and minimize import requirements.
   - Identify and minimize interference among helpers.
   - Use layering to define complex components using simple ones.
   - Implement policy on top of mechanism.
4. **Parameterization**
   - Use parameterization to abstract away contextual variability.
   - Use instantiation to generate components.

A more complete description of all principles, along with operationalizations, rationales, examples, and discussion remains unfinished, in large part because terminology, concepts and classifications evolved in a bottom-up fashion during working group sessions.

Reuse Tools and Environments

Becky Joos, Motorola
joos@motorola.com
Tim Stockwell, The Mitre Corporation
stockwell@mitre.com

Scope

This group focused its attention on the issues of integration and interoperability of tools and repositories, as well as required and recommended functions and services to be provided. The work built upon 1992 Tools working group.

Goals

- Define tool functions required,
- Share experiences and information about tools and environments, and
- Establish a mechanism for continuing collaboration on topics of interest.

Tool Types

The group concentrated on tools for creating and supporting the development and use of reusable software components/assets. Three categories of tools were defined: tools for the creation of reusable components/assets, tools for the management of reusable components/assets and, tools for the utilization of reusable components/assets. Functions/activities were listed for each category.

```
 Creation                      Management       Utilization
-------------------------------------------------------------
- domain modeling            - acquisition      - asset
  requirements                                    determination
- software architecture      - acceptance
  development and            - certification    - asset
  identification                                  development
- component or generator     - access control   - asset tailoring
  development                - cataloging       - asset selection
- general software           - metrics for      - asset use
  engineering integration      tracking
  tools for evolution        - data modeling
  applications
- object oriented analysis   - library metrics
- documentation              - configuration
                               management
```

After establishing the desired functions, a search was conducted for tools that are available. Information was gathered from STARS CFRP, ALOAF, NATO, Paramax, and SofTech.

Reuse Tools Survey

The following lists provides a reuse tool classification and examples of tools, their developer and a reference point.

1 Asset Creation Process Family
1.1 Domain Analysis and Modeling

- ElvisC, Bailin, WISR'93

1.1.1 Reverse Engineering

- SoftKin, Hislop, WISR'93

1.1.2 Knowledge Acquisition

- ElvisC, Bailin, WISR'93
- KAPTUR, Bailin, WISR'93

1.1.3 Technology and Requirements Forecasting

- ElvisC, Bailin, WISR'93

1.1.4 Modeling

- MAST, Bailin, WISR'93

1.2 Software Architecture Development

- OPTEC, Kojima, WISR'93

1.3 Software Component Development

- Hendrix, Bailin, WISR'93
- Penelope, Stillman, WISR'93
- Energize Programming System, email from Aarne Yla-Rotiala
- XRT Widgets, email from Aarne Yla-Rotiala
- C++ Standard Components, email from Aarne Yla-Rotiala
- OST-LOOK, email from Aarne Yla-Rotiala
- Zinc Application Framework, email from Aarne Yla-Rotiala
- XVT Development Solution for C, email from Aarne Yla-Rotiala

1.4 Application Generator Development

- Sinapse, Baxter, WISR'93

1.5 Asset Evolution

2 Asset Management Process Family

2.1 Asset Acquisition

- SRL REUSE LIBRARY PROTOTYPE, ASSET_A_290, ASSET Repository

2.2 Asset Acceptance

- SRL REUSE LIBRARY PROTOTYPE, ASSET_A_290, ASSET Repository

2.3 Asset Cataloging

- EXTRACT, Castano, WISR'93
- WAIS, Gopher, World-Wide Web, Stockwell, WISR'93

2.4 Asset Metrics Collection

- ADAQuest, Arya, WISR'93
- SRL REUSE LIBRARY PROTOTYPE, ASSET_A_290, ASSET Repository

2.5 Asset Certification

- AdaWise, Stillman, WISR'93
- Penelope, Stillman, WISR'93

2.6 Library Operation

2.6.1 Library Support Procedures

2.6.2 Library Access Control

- WAIS, Gopher, World-Wide Web, Stockwell, WISR'93

2.6.3 Configuration Management

- CMA, ASSET_A_243, ASSET Repository

2.6.4 Asset Interchange

- WAIS, Gopher, World-Wide Web, Stockwell, WISR'93

2.6.5 Reuse Promotion

2.7 Library Data Modeling

- RLF, ASSET_A_442, ASSET Repository

2.8 Library Metrics Collection

- WAIS, Gopher, World-Wide Web, Stockwell, WISR'93

2.9 Library Evolution

3 Asset Utilization Process Family

3.1 Asset Requirements Determination

3.2 Asset Identification

- CodeFinder, Henninger, WISR'93
- RECAST, Castano, WISR'93
- RLF, ASSET_A_442, ASSET Repository
- RSR, ASSET_A_220, ASSET Repository
- SRL REUSE LIBRARY PROTOTYPE, ASSET_A_290, ASSET Repository
- WAIS, Gopher, World-Wide Web, Stockwell, WISR'93

3.3 Asset Selection

- RECAST, Castano, WISR'93
- RLF, ASSET_A_442, ASSET Repository
- RSR, ASSET_A_220, ASSET Repository
- SRL REUSE LIBRARY PROTOTYPE, ASSET_A_290, ASSET Repository
- WAIS, Gopher, World-Wide Web, Stockwell, WISR'93

3.4 Asset Tailoring

- RECAST, Castano, WISR'93

3.5 Integration of Assets with Application

- RECAST, Castano, WISR'93
- RLF, ASSET_A_442, ASSET Repository
- CMA, ASSET_A_243, ASSET Repository

Language Issues for Generic Code Architectures

Larry Latour, University of Maine, Orono
larry@gandalf.umcs.maine.edu
Ira Baxter, Schlumberger
baxter@austin.slcs.slb.com

An important idea for reuse of design is:

Generic Architecture: a system of components with rules specifying consistent compositions.

Such architectures can be used to implement instance systems by choosing compatible components, and combining them by using the composition mechanism implied by the rule structures. These architectures are especially useful when supported by tools to manage the selection and composition of components.

The working group met with the initial intention of determining sets of features and mechanisms useful for constructing systems from generic architectures. Discussion led the group to examine several real systems built by some of the group participants, with the hopes that identifiable concepts would emerge. Further examination suggested that the composition mechanisms themselves were crucial, and therefore interesting to examine in detail. The results of this examination appear in the sequel in the discussion on refinements.

The systems considered were:

Sinapse: Generates numerical solvers for partial differential equations [Kant91].

Predator (P++): Database system generator [Batory92] (language incorporating composition principles [Batory93]).

RESOLVE(/Ada): Language and discipline for describing and reusing software architectures [RESOLVE93], [WOZ91], [Murali93] (for Ada components [Hollingsworth92]).

Details as to the operation of these systems is deferred to the references. Implementors of these systems were present in the working group so that hypotheses about system mechanisms could be formulated and verified.

Transformation Systems as an Explanatory Model for Generic Architectures

One of the major difficulties in comparing such systems is the diversity of terminology, mechanisms, and representations used. In an effort to find commonality, it was proposed that the group use a *transformational* model of program generation to provide standard vocabulary, concepts, and mechanisms, and attempt to map relations between generic architecture systems via the standard. Part of the reason for this proposal is that one of the systems (**Sinapse**) is transformational, and so a straightforward mapping already existed.

A transformational model provides a number of concepts and mechanisms:

Specification language: used to specify desired properties of the desired instance system. Such a language may consist of atomic symbols representing concepts to implement (e.g., "database"), or expressions of constraints over desired results (e.g., $<$ In,Out $>$:In=Out*Out) or some domain specific notation with implicit intent (a partial differential equation with the implicit requirement to solve it).

Synthesis mechanisms: convert specifications into program abstractions with constraints. This requires the system to choose some set of mechanisms, which when composed, implement the specification. As an example, the system having both an integer enumeration and summing components might notice that the sum of integers 1..n is n*(n+1)/2, and use that fact to generate an integer square root algorithm by enumerating successively larger values of n, summing, and comparing. Synthesis in effect generates a refinement.

Refinement (transformations): map high-level abstrac- tions to abstractions at a lower level. An example refinement is the conversion of the abstract notion of matrix inverse into more concrete code using multiply nested loops for a Gauss-Jordan procedure.

Optimizations (transformations): replace configurations of abstractions by "simpler" configurations of abstractions at the same abstraction level. As an example, the SQL query:
SELECT cost $<$ 50 AND month=March FROM invoices can be to optimized to SELECT cost $<$ 50 FROM (SELECT month=March FROM invoices)
assuming the statistical fact that there are usually fewer records from March than records with low cost value.

Control: guides application of transforms (any of synthesis, refinement or optimization) to produce the implemented system. Usually a control mechanism chooses among several legal transforms, each introducing certain levels of performance, to optimize for a performance property, such as execution speed or storage requirements.

To produce a desired program, a transformation system accepts a specification, and then, under the guidance of the control mechanism, repeatedly chooses transforms from a library to incrementally convert the original specification into a final specification, representing the desired program. (We remark that any of synthesis, refinement, and/or optimization can be implemented using rewrite rules or arbitrary procedures).

A correspondence to generic architecture systems is obtained when one realizes that any principled composition of instance components must itself have well-understood properties, and therefore both the components and the composition mechanism must have well-defined properties. Oversimplifying, the key property required of an implementation I for a specification S, is that important properties M (for "meaning") of the specification are preserved: M(S) $<=$ M(I), i.e., the implementation must have at least the properties required by the specification; this is precisely the notion of *refinement* used in transformation systems. Any generic architecture system must somehow combine components c1, c2,... to form an I for its input specification S.

The question is then, how can the example systems be described using the conceptual framework of transformation systems?

Example Systems viewed Transformationally

A correspondence between example system mechanisms and that of transformation systems was established and is summarized in the following table:

	Specification	Synthesis	Refinement	Optimization	Control
Sinapse	Partial differential equations (PDEs). Implicit requirement that PDEs be solved.	Map PDEs to finite-difference solver.	Individual application of bundle of consistent transforms.	Many explicit array and scalar simplifications; code motion, common subexpression elimination.	Internal sequencing of refinements and decision resolution. Decision objects have resolution heuristics.
Predator (P++)	Component name (implies functionality) + signature.	N.A.	Substitution of named component instance.	Procedural transforms for query optimization.	User-supplied refinement sequence given by "type" expression
Resolve - Ada	Model-based predicate calculus description over math domains (sets, tuples, etc.).	Refinements manually synthesized from existing components	Generic instantiation constrained by [Goguen86/89] style views.	Manual; rare. Believes refinement choice gives sufficient performance.	Manual; user-supplied transformation sequence

Table 1: Transformational view of several generic architecture systems

Discovering how the instance systems fit into the transformational perspective led to a number of insights about how or why the various systems worked. As examples:

- System "inputs" mixed different kinds of knowledge needed by a transformation system. This suggests that generic architecture systems are likely to have mixed input information whose parts can be classified, which helps explain why such input information is needed to enhance automation or output quality. The sample systems mixed, to some degree, specification and control knowledge in their inputs:

  - Functional specification:
    Sinapse: PDEs
    Predator/P++: Name/signature of desired system function
    RESOLVE: Predicate calculus specification of behavior supplied by a component.
  - Control knowledge:
    Sinapse: Decision resolutions controlling synthesis process
    Predator/P++: Explicit specification of choice of refinement of sub-components
    RESOLVE: Not in specification [user interactively selects from catalog]

- One can't hide from control knowledge; it is present in all the systems in different forms.
  - Sinapse has explicit control knowledge, built into the system
  - Predator/P++ uses supplied control-knowledge to apply component refinements
  - Resolve leaves choice of refinement to the user. But he must choose.

- The key notion of refinement appears in each system in different forms, with differing levels of robustness. How robustness is ensured is important if we wish to trust systems produced by generic architecture schemes.
  - For each Sinapse abstraction, there are multiple sets of transforms that refine the abstraction consistently; set members are applied incrementally.
  - For each Predator abstraction ("realm"), there is a set of components, each being a monolithic refinement. Each refinement may introduce new abstractions which must be further refined; if the new subcomponents are refined correctly, then the refinement is itself correct.
  - RESOLVE specifications are monolithically refined only when Goguen-style constraints are satisfied.

  Sinapse has multiple sets of transforms for which selected subsets must be applied consistently. Consistent transforms are applied individually, interleaved with other actions, i.e., the refinement step is *not* monolithic. This is accomplished by establishing an explicit intent, recorded by the resolution of a design decision, and only selecting transforms which are marked as consistent with that intent. A similar method was used for the Draco transform system [Neighbors84] in which applied transforms asserted they provided certain properties by posting a tag symbol representing those properties, and potentially applicable transforms required the presence of transform-specific tag symbols to become truly applicable. This method for consistent refinement is documented in [Katz93].

  Some Predator components, while having the same conceptual specification and identical signatures by virtue of being members of the same "realm", do not always provide the functionality required by a true refinement. This is handled by a (proposed) mechanism which notes when a particular subcomponent is used, and ensures that only compatible components are used to refine other parts of that component. The transformational perspective allowed us to recognize that this was similar if not identical to the Draco tagging method. So two systems, Sinapse and Predator, use essentially the same idea present in a third. This hints that this idea may be somehow fundamental, or least a standard heuristic.

  A second consequence of the transformational perspective is an insight into how refinements are packaged. Predator packages refinements as entire, consistent components, ensuring that refined Predator specifications lead to working code. The price for this is that if one has n transforms, A, B, C,... N, which taken individually are not refinements, but which taken in pairs (often) form consistent refinements, they must be encoded as (up to) N^2 monolithic refinements. There is a trade-off between flexibility (application of smaller transforms in a larger variety of ways) vs. reuse of useful compositions (application of a larger transform known to be useful). In order to factor monolithic refinements into smaller transforms, one must be able to express the constraints between the smaller transforms in order to have the option of the trade-off. It would convenient if sets of transforms chosen to form an incomplete refinement somehow constrained further choices to compatible transforms in a manner less ad hoc than the Draco tagging method.

  RESOLVE has such a mechanism. Components have import/export interface specifications and an implementation. Components depend on explicit properties rather than directly on other components. Refinement of a module part by use of a consistent component then creates a composite which has its own import/export specifications, which continue to reflect the component's property dependencies. This method appears to scale well: rather than worrying about the interactions of all the (exponentially large) possible subsets of N components, one can instead simply state necessary formal properties. The price for this is that one needs a theorem prover to verify that properties match instead of a simpler tag matching mechanism.

- Optimizations may be required for efficient code generation.
  - Sinapse offers many because of the performance demanded by supercomputing tasks. The Sinapse authors believe in the Draco model of a repeated refine-then-optimize cycle to remove redundancies introduced by juxtaposed refinements, and to perform optimizations at high levels of abstraction where they can be easily detected.

- Predator uses relatively simple optimizations that appear mostly in the form of partial evaluation of conditional expressions. The most complex optimizations (and the most important) are those involving query optimizations, which involve the selection of which data structure to traverse and the factoring of predicates to avoid repeated subpredicate evaluation.
- RESOLVE does currently not provide any automated method for optimizing a particular composition of refinements beyond inlining. The RESOLVE philosophy is that the best performance gains are to be had by choosing the most appropriate refinement at each step, rather than *a posteriori* optimization of a completed series of refinements.

This implies that the choice of each refinement, a manual process in RESOLVE, is an important way of tailoring the performance of the resulting product [Murali92].

It is remarkable that these insights were obtained even though two of the systems examined were designed without a transformational model in mind.

Unresolved Issues

In a single day, one cannot realistically explore the value of a model; at best, one can determine there is some utility. A number of obvious issues remain:

- Does a transformational model apply to other generic architecture systems?
- More detailed models exist (c.f. [Baxter90], covering explicit performance constraints and providing considerable flexibility in the explicit statement of control knowledge). How well do they hold up, and what, if any, extra insights might they provide?
- How does a transformational view explain architecture of components for systems?
- How does it explain architectures of the generated systems?
- How does the composition mechanism affect the components of generic architecture?
- Are there alternative hypotheses that explain the commonalities better?

The main value of the transformational perspective is to allow comparison of features of the mechanisms used. Such a perspective can be used to determine the robustness of the composition process, how complete the specification language is for both problem and components, and to determine what mechanisms a particular generic architecture system may need.

References

[Batory92] Batory, Don and Sean O'Malley. The Design and Implementation of Hierarchical Software Systems Using Reusable Components. ACM Transactions on Software Engineering and Methodology, 1(4) 355-398, October 1992.

[Batory93] Batory, Don, V. Singhal, M. Sirkin and J. Thomas. Scalable Software Libraries. Proceedings of CM SIGSOFT 93.

[Baxter92] Baxter, Ira. Transformational Maintenance by Reuse of Design Histories. Ph.D. Thesis, University of California at Irvine, Department of Computer Science, 1991. Available as University Microfilms International document #9109634.

[Goguen86] Goguen, Joseph, Reusing and Interconnection Software Components, IEEE Computer, 19(2), February, 1986.

[Goguen89] Goguen, Joseph A. Principles of Parameterized Programming. In Software Reusability, Volume I: Concepts and Models, Biggerstaff, Ted J. and Alan J. Perlis, ACM Press, New York, New York,1989, pp. 159-225.

[Hollingsworth92] Hollingsworth, Joe. Software Component Design-for-Reuse: A Language Independent Discipline Applied to Ada, Dept. of Computer and Information Science, The Ohio State University, Columbus, Ohio, 1992.

[Kant91] Kant, Elaine, F. Daube, W. MacGregor and J. Wald. Scientific Programming by Automated Synthesis. In M. Lowry and R. McCartney, editors, Automating Software Design. AAAI Press, 1991.

[Katz92] Katz, Martin and Dennis Volper. Constraint Propagation in Software Libraries. Journal of Software and Knowledge Engineering, 2(3): pp. 355-375, September 1992.

[Neighbors84] Neighbors, James. The Draco Approach to Constructing Software from Reusable Components, IEEE Transactions on Software Engineering, SE-10(5), September 1984.

[Murali92] Sitaraman, Murali. Performance-Parameterized Reusable Software Components. International Journal of Software Engineering and Knowledge Engineering, 2,(4):pp. 567-587, October 1992.

[Murali93] Sitaraman, M., Lonnie R. Welch and Douglas E. Harms. On Specification of Reusable Software Components. International Journal of Software Engineering and Knowledge Engineering3(2):pp. 207-229, June 1993.

[RESOLVE93] The Reusable Software Research Group at The Ohio State University, RESOLVE Reading/Reference List, available via anonymous ftp from ftp.cis.ohio-state.edu in the directory pub/rsrg as the file RESOLVE-refs.txt, 1993.

[WOZ91] Weide, Bruce W., William F. Ogden and Stuart H. Zweben. Reusable Software Components. In Yovits, M.C., Advances in Computers, Volume. 33, Academic Press,1991.

Hybrid Reuse with Domain-Specific Kits

Martin Griss and Kevin Wentzel
HP Laboratories
griss@hpl.hp.com, wentzel@hpl.hp.com

Introduction: What is a Hybrid Kit

The goal of this group was to explore the hypothesis that high-payoff hybrid reuse can be systematically addressed in the form of domain-specific kits, and that appropriate methods and technology should be developed to support this integrated approach to delivering reuse.

The leaders first described HP's notion of a kit, meant to be the "complete" and "coherent" packaging and delivery of (several different) reusable work products to simplify application building. A kit is comprised of compatible, domain-specific frameworks, components, glue language and supporting tools. The notion of "hybrid kits" was discussed in the domain-engineering WG at WISR'92. We built on this work by describing kit "style" as ranging from purely compositional to purely generative. We identified hybrid reuse, combining compositional and generative reuse, as having the greatest payoff. In this mode, generators and builders can be used to produce a variety of workproducts (such as the glue language, parameters for components, customized components, and data-tables).

We illustrated the concept and range of kits styles by discussion the Calculator Construction Kit and a prototype HP labs software-bus based kit for Task-list management. The group discussed the notions of frameworks, components and glue languages, which workproducts could be reusable, where to place the domain-specificity, and the differences between generators (automated generation or configuration from specifications) and builders (graphical tools to aid in manual composition of components).

Kit Taxonomy

A prototype kit taxonomy and comparison table from HP was described. This identifies a variety of kit attributes (such as style, domain-specificity, openness and completeness), and is intended as a taxonometric framework to compare the features of kits, and suggest alternative ways of delivering reusable workproducts. The use of the analysis table was illustrated by comparing several systems (such as Genesis, the Calculator Construction Set, and several HP systems). Graphs were used to highlight useful clusterings by subsets of the attributes (openness, completeness and domain-specificity).

Kit Development Process

The leaders described HP's proposed development process incorporating "domain engineering" and "kit engineering", building on some of the discussion of the Domain Engineering Workshop at WISR'92. In the HP process, domain engineering includes the overlapping phases of domain analysis, domain design and domain implementation, focusing primarily on application functionality. We have introduced a set of parallel and overlapping phases of kit engineering, called kit analysis, kit design, and kit implementation, focusing primarily on the style and technology of application development to be used by kit users, following a customized application engineering process. The discussion brought forth issues relating to how domain analysis, modeling and engineering need to be constructed so as to lead effectively to hybrid reuse, how much the various phases of domain engineering and kit engineering overlap and/or are distinct, how some domain engineering techniques are now including part of what HP calls kit engineering and whether it makes sense to split up these activities at all.

Several draft papers on HP Labs notions of kit process and kit concepts were distributed. One of these, "Hybrid domain-specific kits for a flexible software factory" by Martin Griss and Kevin Wentzel, will appear in the proceedings of SAC'94, Phoenix, Arizona, Mar'94. For more details or a copy of this paper, contact either of the WG leaders.

Case Study Analysis

As homework, and in breakout groups, the working group participants used the kit analysis table as the starting point to collect information about 13 kit-like systems. In the process, many improvements were identified, and a revised table was proposed.

As motivation for developing, and using the table, the following reasons were suggested; each could motivate a different level of detail:

- Sales job – explain to people why they should kits in general, or a particular kit
- Start DA of kit styles – a tool, or reference diagram to allow kit designers to pick mechanisms and styles used as classification/reference
- Kit selection – outline of a catalogue of kits, allowing kit-users to elect

Original HP Kit Taxonomy Table

Revised Kit Taxonomy Table

Use as starting point for DA of kits: Collect and analyze information about various kits, value of features etc.

GENERAL:

Kit Name:		
Domain:		
Domain Specificity:	How specific to a particular application domain is the kit?
Style:	Are applications built from the kit in a compositional, generative or hybrid manner?
Completeness:	Can the complete application be built from the kit?
Openness:	How easy is it to add new functionality to applications developed using the kit?
Target User:	Who is expected to use the kit: application developers, system integrators, or end users?
Granularity:	How big are the kit components? Range from functions to processes.
Quantity of pieces:	How many components are there in the kit?

```
Kit Name:
Style:
Purpose:
Application User:
Features of Merit:
   (coherence, ease of use, evaluability of
   results, distance from problem space,
   ... "compression")

Notes:
   Interoperability with other kits
   Sub-kits used
```

ANALYSIS: A two dimensional (matrix) model was chosen with kit elements on the vertical axis and element attributes on the horizontal axis.

```
 "kit elements" (work products list:)
    components
    architecture
    framework
    domain specific language
    glue
    generator
    builder
    tests
    end user document
    maintenance document
    kit use documents
    domain model
    feature set
    rationale
    generic application
    glue language

 Element (workproduct) attributes:
    present?
    completeness
    openness
      (includes openness of applications
      and of the kit itself)
    domain specificity
    binding time
    source
      (provided in kit, created,
       written by user)
    Application Engineering Life Cycle stage
      (How/when this element is used? --
       problem, solution, implementation)
```

Kits analyzed by participants

"Kits" analyzed include:

- Predator
- Boulder design Environments
- Marvel/Oz
- Robot Control Software
- Personnel Mgt Kit (MGIB)
- MacApp
- CCL
- Process/1 "Anderson"
- DSSA-ADAGE
- Movement Controller
- PARTS (DEC)
- KIDS

Revised Kit Process

The discussion on the kit development and use process lead to several observations and some suggested modifications:

- DA constrains kit analysis/design.
- KA constrains domain design.
- Some methods of DA/DE include parts of KE.
- Distinction between DE and KE is fuzzy. What are some distinct work products?

The working group suggested that there was so little independence in implementation that separating kit and domain implementation may not make sense. The constraint observation lead to a new model of development in which Domain Analysis and Kit analysis overlap, both affecting kit and domain design then merging in implementation.

Conclusions

- Kit Taxonomy table was useful to collect and analyze cases.
- Developers could use a table like this to suggest extra elements in kits.
- Seems like KE/DE separation has validity. (There are distinct, yet interrelated WP and processes.)

Next Steps

- Kit Taxonomy
  - collect and analyze additional kits.

- do DA like analysis of kits.
- refine as a reference model.
- use to find design rules/criteria (for particular reuse scenarios).
- develop kit glossary, with terms such as "kit", "framework", "domain".

- Kit Process
  - define distinct WP of Domain Engineering and Kit Engineering.
  - identify steps in domain analysis/engineering process(es) as part of DE/KE.
- Overall
  - link Taxonomy & Process to relate to kit design.

Questions and suggestions when the final results were presented at the plenary included:

- Need to look at several different kinds of systems.
- Are kits distinct from Application Generators or Generic Architectures?
- Why distinguish DE/KE?
- Look at Ed Berard view on Kits, because he used the term before.

Reuse Education

Ben Whittle, University of Wales, Aberystwyth & University of York
ben@minster.york.ac.uk
Trudy Levine, Fairleigh Dickinson University
levine@sun490.fdu.edu

The original premise of the group was that at the present time people need to be educated in the techniques of reuse, but that in the future reuse will be an integral part of the curriculum. The group agreed that there is a need to design re-education courses for the generation of software engineers who were not taught about reuse; this topic was suggested as a starting point for next year's discussions. The focus for this year was to try and decide where reuse would fit into a software engineering curriculum of the future. The group looked at education principles and ideas from other engineering disciplines to define the skills and thus the curricula focus for reuse in future software engineering courses.

This resulted in preliminary descriptions for the following courses.

1. Introduction to Software Engineering – mainly design with reuse.
2. Software Design – design for reuse, must push standardization.
3. Domain Specific Courses – e.g., compiler courses, data bases, real-time control systems.
4. Management – general management will include reuse management.

The group assumed certain prerequisites and co-requisites in terms of technical writing and mathematics courses. Courses which bear a strong resemblance to these ideas have been developed by Murali Sitarman at West Virginia University. None of the courses is beyond the reach of current technology and in many ways the approaches are less radical than introducing a specific course on reuse, which is a technique rather than an area of systems development.

Goals:

The goal of working group was to focus on the integration of reuse into the education of Software Engineers.

Perspective: Reuse should and will become an integral part of Software engineering as opposed to a separate add-on concept. Therefore, reuse should be incorporated into all courses for software engineering (in the long run) rather than separate courses on reuse. We defined courses in the context of this long term vision of reuse.

Assumption: Software engineering will be a separate discipline from computer science and will be related to it in the same way that traditional engineering is related to natural science.

Assumption: Software engineers will, in addition to the courses described below, take course(s) in theoretical computer science and mathematics. For example, mathematics courses would cover set theory, theory of relations, functions, recursion, graph theory, predicate calculus, and automata theory.

Activities

The working group presented its results as four course descriptions for the software engineer:

- Course 1, for both computer science and software engineering students
- Course 2, for both computer science and software engineering students
- Domain-specific courses. These courses each focus on one domain, examining standard software systems for that domain.
- Course on management for software engineering students. Covering topics such as economic issues, risk analysis, and team dynamics.

Each course takes 1 to 2 semesters or terms.

In addition to the computer science and mathematical skills the following software engineering skills are supplied by our courses:

- Communication skills
- Managerial skills (people, product, process)
- Standard ways for: specification, representation, documentation, style, testing, qualifying, analyzing.
- Methods for problem solving
- Algorithms, building systems, etc.,
- Customizing
- Modeling
- Metrics
- Domain-specific knowledge

The remainder of this section presents an outline of the four courses.

A First Course in Software Engineering

The course adopted design principles from other engineering disciplines to software engineering to strengthen reuse. The course also includes an evolutionary approach to assignments and teaching and emphasizes the construction of software rather than the writing of code. In addition the course should: teach language concepts and provide a context to the above material with a brief overview of computer science history, and a brief overview of "machines" (as appropriate).

At the end of the course students should be able to design and construct a solution given

- a problem
- a set of components (design, code, documentation, test cases, etc.,) that could be used

and be able to explain their solution.

Thus the material for course one includes introductions to:

- Understanding a specification, building from sample specifications (with reuse) and building toward specifications (to reuse).
- Problem solving by assembling components for customizing and analyzing good examples of design by reasoning by analogy, trial and error with learning, etc.
- Standard set of views of: Specification; Representation; Documentation; Style; Assessment; (Peer) Code review; Testing; Qualification; Analyzing.
- Customizing (tailoring), retrieving from (domain specific) "library", and building generic code.

A Second Course in Software Engineering

The second course, which has the first course as a pre-requisite, strengthens and deepens the idea of engineering through the following structure:

- How to build reusable components: Data abstraction (ADT, encapsulation, inheritance), Control abstraction (external modules, iterators).
- How to create systems from (reusable) components: Introduction to systems: properties, system architecture, problem decomposition, why decisions are made among multiple (existing) alternatives.
- Introduction to metrics: complexity, measurement.
- Design rationale: Why you make design decisions, performance considerations, etc., and design (for reuse).
- Introduction to repositories: extraction, variants.

Domain-Specific Courses

After the first two courses a number of domain specific options can be taken. The working group considered that domain-specific developments would have progressed to the extent that in many domains it would be appropriate to take a course in construction of software in that domain. For example, many compiler courses explain the principles of compiler theory and allow the students to build simple compilers using meta-compiler tools such as LEX and YACC. We envisage this approach to toolkit development extending. In domains where the technology has not advanced to the level of providing toolkits then appropriate generic frameworks and architectures for software construction from suitable components will be likely. The majority of software engineers will be engaged in the construction of these components into systems rather than with a "start from scratch" implementation. There are a number of domains where this is already possible; these include: compilers; database systems; real-time control systems, business systems.

In addition these courses would cover appropriate aspects of:

- Terminology
- Standard problems and solutions
- Architecture
- Application generators
- Customer interface
- Test suites
- Maintenance
- Repository maintenance

Higher-level Courses

The group gave an initial consideration to more generic courses that could be available to students. These would likely include: Management issues, Communication skills, Economic issues, Risk analysis, Team dynamics, Trade studies, Modeling, Verification, and Domain analysis (depending on the state of the art).

References

These are the references suggested by the group which have a similar approach to Software engineering education.

Frakes, William B., "A Graduate Course on Software Reuse, Domain Analysis, and Re-engineering," Virginia Tech.

Gray, J. "Teaching the Second Course in Computer Science in a Reuse-Based Setting: A Sequence of Laboratory Assignments in Ada," 11th Annual National Conference on Ada Technology, 1993 pp. 38-45.

Proceedings of the 1st Reuse Education and Training Workshop, Sept. 23-24, published June 18,1993, available from source.asset.com.

Proceedings of the 2nd Reuse Education and Training Workshop, October 25-27,1993, contact C. Lillie at ASSET.

Sindre, G., Karlsson E. and Stalhane, T., "Software reuse in an educational prospective," Proceedings of the 1992 Software Engineering Education Conference, Norwegian Institute of Technology, Springer-Verlag, 1992.

Smith, K "CARDS Application Engineering with Domain-Specific Reuse," course description vol I and II, available from source.asset.com.

James C. Spohrer and Elliot Soloway, "Novice Mistakes: Are The Folk Wisdoms Correct", Communications of the ACM, 29(7), July, 1986, pp. 624-632,

Gerhard Fischer, Andras C. Lemke, Raymond McCall and Anders I. Morch, "Making Argumentation Serve Design," HCI Journal, April, 1991.

D. Partridge (ed.), "Artificial Intelligence and Software Engineering," Ablex Publishing Corporation, Norwood NJ, 1989.

Norman E. Gibbs and Richard E. Fairley, "Software Engineering Education: The Educational Needs of the Software Engineering Community," Springer-Verlag, New York, 1987.

Technology Transfer

Sidney C. Bailin, CTA, Inc.
sid@cta.com
Guillermo Arango, Schlumberger Laboratory for Computer Science
arango@austin.slcs.slb.com

The meeting of the Technology Transfer (TT) group started with a discussion of reuse TT experiences in the participants organizations. Clear differences in management approaches, scope of the efforts, and funding styles between industry vs. government organizations was apparent. A list of issues was identified:

- Does Reuse require special approaches to TT?
- How is it done (in the cases where it works well)?
- Who pays for TT?
- How to speed it up?
- How do we develop advocacy?
- How do we measure success, ROI in TT?
- Who are the customers, targets, users?
- What do we transfer?
- What drives TT?
- What are the roles of people in the TT process?

Given the time constraints, only the first three issues were discussed in some detail.

Does Reuse Require Special Approaches to TT?

Is there any reason to think that the TT process applied to other technologies may not apply to Reuse? Participant opinions were divided.

Those who argued for the need of a special TT approach pointed out that reuse is not a "technology". It involves the application of well-established engineering principles–the application of best principles and practice, case-based reasoning–to software engineering activities. In the established engineering communities, reusability is part of the culture, like the principles of physics or mathematical notations. That culture is propagated through educational programs, training, engineering standards, codes of practice, and so on. Following this argument, rather than investing on technology transfer, we should invest in education and on consolidating good "software engineering theories" and "domain-specific theories" that could serve as a reliable foundation for the profession (by analogy with physics, materials science, structural analysis, and so on). Having redefined the TT problem as one of (i) building reusable theories and (ii) teaching people good engineering, the group could not agree on the question of how to reliable build reusable theories (i.e., What is the scientific method?). Software engineering research and the practice of domain analysis will continue to produce many of these. Practical validation and market needs will probably drive a Darwinian process of selection/evolution as it has happened to computer architectures, programming languages, design notations, and so on. An interesting question not addressed by the group was: How could we speed up the selection/evolution process?

Those who argued against the need of special TT approaches justified their answer on the observation that "good" TT approaches used by other disciplines do not seem to depend on what is being transferred but on:

- the structure of the organizations participating the transfer,
- funding schemes,
- the products or processes to which the technology applies,
- the economics of the markets of those products or processes, and

- the level of sophistication of the user community.

Do we have any "good" processes that could be shared? A number of known processes (e.g., IBM, HP, ICASE, SPC, Schlumberger, DSRS) were proposed as possible candidates. Those processes, however, are customized to the conditions listed above and have not been validated extensively.

The discussion touched briefly on "What is transferred in Reuse TT?" There was agreement that the two major components of the transfer are: (i) systematic processes for building on what exists, and (ii) methods and tools for building reusable software and for assisting in reusing it.

How is it done (in the cases where it works well)?

The consensus on this issue was refreshingly crisp and down to earth: "You can take the horse to the water but ..." The common denominator of all successful experiences is **pull**. Successful TT takes place when there is a **pull** from the receiving users. This factor dominates all other factors.

Based on this observation, Reuse TT approaches should first:

1. Identify pull, or
2. Generate pull

Only then, attempt to transfer the technology.

To identify **pull** requires first identifing who the users are. (We distinguish between Users–those who will ultimately apply the reuse technology– and Customers–those who pay for it.) These distinction helps account for critical difference in perceptions and interesting politics. Most successful transfer cases appear to be driven by **grass roots** efforts. To identify and encourage such movements is a powerful means to facilitate the introduction of reuse technology. Participative software process improvement programs offer a good framework for creating and identifying opportunities.

Pull can be stimulated by using techniques such as TQM, the SEI Capability Maturity Model, ISO 9000 certification process, or the Malcom Baldrige Award. Interestingly, we do not seem to be exploiting one of the most common techniques used to generate pull in our economy: marketing. The marketing community has developed a number of analysis tools and communication tools to:

- identify opportunities–who are the users, what are their perceptions and needs, what is the source of their buying power,
- measure ROI, and
- generate pull for products and services.

These are key aspects of TT. There is a wealth of experience and techniques used by the marketing community that we have not even started to exploit for the transfer of technology.

A first lesson that we can learn from marketing, is that marketing does not come for free. It is quite expensive. Many TT projects fail precisely because they did not identify the extent of the investment needed to make TT work. Schemes for funding marketing projects can also be applied to TT projects.

Who Pays For TT?

The group discussions would repeatedly come back to the point of "Who pays for it?" Funding is a central issue in TT. First, because TT does not come for free. Second, because funding patterns drive how the organizations participating in the process operate and how the process is managed.

There are a number of funding schemes that are appropriate in different organizational settings:

- If TT is structured using a marketing strategy then it is traditional that marketing (thus TT) gets paid by the customer. If customers have been persuaded that sufficient value is delivered, they will pay for it. A number of joint-venture schemes are possible to facilitate mixed funding between sources and recipients of technologies.

- When government organizations are involved, the TT process is not regarded as a money-making operation. The goal is to recover only the operational costs. Government organizations act as third-party funding sources to "prime the pump." The issues are: how to determine time-to-recover of costs and how to recover the costs. Schemes that are applied or may be applied shortly involve: subscriber fees for users of code libraries, fee-for-service-rendered contracts, transfer TT project to external "technology brokers" who are profit-and-loss groups.

- Some organizations become TT agents by transferring a technology as value added to their own products or services. An example, is the access and use of the Internet made available to students by universities as value added to their own educational services.

- In industry, internal R&E laboratories usually have TT responsibilities. The funding of such organizations has a substantial influence on what, when, and how technology is transferred. A number of different "tax schemes" are used to fund such groups. Central funding at the corporate level encourage longer-term technology development rather than technology transfer. Funding from distributed sources close to the profit-and-loss centers makes their managers more aware of R&E expenditure and more demanding about ROI. These schemes tend to encourage TT over technology development and shorter-term projects. Funding groups have a strong influence about what gets transferred, to whom, and how the process takes place. Mixed schemes enable trade-offs between investing in technology development and TT.

- In the US a number of Consortia with industry or mixed industry-government membership (e.g., MCC, SPC, SEI) have been in operation for many years. The results have been mixed. These groups appear to have succeeded in TT (in Reuse as well as other software engineering domains) only in those cases where they operated very closely to the users.
- In Europe, the EEC has funded for more than a decade a number of government and government-industry funded cooperation projects (e.g., ESPRIT, ESSI, EUREKA). By and large these programs have succeeded in creating networks of collaborators and multiple channels for technology transfer between partners. Several projects have focused on Reuse (e.g., Knosos, Reboot, MACS). The ROI, however, is not easy to determine.

A conclusion from the analysis of funding schemes suggests that successful TT projects tend to be those where the customer (funding source) and the users of the technology being transferred are close and share objectives.

The old saw, "when there is a will there is a way," also applies to Reuse TT. Many successful TT projects take place without official funding, with participants working out of their own initiative, on their time, in skunk-works mode. Such projects are usually carried out through the collaboration of networks of individuals sharing common professional interests. The Internet has a very positive effect in supporting the social processes key to the development and operation of such groups.

Recommendations on TT

There was consensus on the following key recommendations:

1. Identify and use existing pull or generate pull.
2. Position customer as close to the user as possible.
3. Benchmark what we have: methods, tools, TT Schemes.
4. Reuse successful TT from other disciplines, in particular, borrow principles and techniques from marketing.

Formal Methods and Certification of Reusable Components

Maureen Stillman,Odyssey Research Associates
maureen@oracorp.com

Formal methods and certification of reusable components were discussed using a model of system and component "reuse manufacturing." Reuse manufacturing was defined in terms of the following three groups:

1. **Producers** – create reusable systems and components Producers are interested in creating high quality systems and components for reuse.
2. **Brokers** – the marketers and sellers of reusable components Brokers want to let the consumers know what they have to sell and an accurate indication of the quality of the system or component.
3. **Consumers** – obtain and use the reusable components Consumers are interested in what the component will do for them and also want an accurate indication of the quality of the component.

The discussions in the working group were in the context of the interests of these three groups. We focused our discussions on areas of interest to the group as outlined below.

- **Metrics**: What kind of metrics are being used in reuse repositories? How are these metrics calculated by the producer or broker? How do these metrics aid the consumer in choosing a component?
- **Local certifiability, modularity, scalability, reuse after modification.** How does the producer manage the task of certifying a large system? How does the consumer manage the task of modifying reusable components and ensuring their previous level of quality and assurance?
- **More connections are needed between certification, verification, reuse and programming languages.** How do the producer and consumer weigh and select among the myriad of tools and techniques available given that there is no standard prescribed methods to produce high quality reusable components? Should we start to address this by redefining our programming languages?

Scope

The working group focused its discussion on the role of formal methods and certification as applied to reuse. The definition of formal methods included all tools and techniques that were mathematically based. The group used a broad definition of certification which included the use of documentation, testing methodology and tools, metrics, structured walkthrough, etc. In short, any method or tool that could be used to aid in increasing the quality or reliability of the reusable system or component was included in the scope of the discussions.

Goals

The goal of the group was to explore current accomplishments and research issues in applying formal methods and certification techniques and tools to reuse.

Formal Methods – State of the Art and Practice

There are an increasing number of formal methods tools and techniques that are available for use by the reuse community. There have been a number of successful applications of formal methods to the software engineering of government and commercial systems documented in the literature.

The application of formal methods to reuse repositions or to internal reuse projects has been a rare occurrence, but there are a few existing efforts. We believe that increased efforts in the area of technology transfer of formal methods is necessary. The creation of high assurance reusable components may also be driven by consumer demands for high quality.

This is an especially important issue for those in the areas of building trusted software, safety critical systems and high assurance systems.

Certification – State of the Art and Practice

As collections of reusable software assets have become increasingly widely available in recent years, concerns have grown about how to ensure that such assets conform to the quality requirements of the systems that will be using them. Asset certification has been proposed as a means of providing the software engineer with a level of confidence that an asset will conform to the quality requirements of the domain in which systems are being built and of the organization creating these systems. While a number of certification schemes have been proposed, there is still disagreement as to what it means for an asset to be "certified" and how such a scheme is best implemented.

The certification schemes that have been proposed to date fall into three categories - levelled, ranked, and enumerative. In a levelled scheme, assets are assigned quality "levels" according to the amount or type of quality assurance processing that has been applied to them. For example, a certification process might term a component that has had no quality assurance performed on it a "level 0 component," one that compiles without error might be a "level 1 component," one that has been unit tested might be a "level 2 component," one that has been deployed and displays a mean time to failure within a certain bound might be a "level 3 component," and one whose correctness has been formally verified might be a "level 4 component".

Levelled schemes suffer from two main drawbacks. First, it is not clear how an engineer can draw quality inferences about a system based simply on knowing that a given component has been certified according to a specific level. Second, if a component is modified, it is not clear how much effort should put into re-certifying the component, because there is no way to trace the specific properties that might have been altered in the course of the adaptation.

In a ranked scheme, scores are assigned to a component based on how well it adheres to certain quality criteria. Different criteria are assigned different weights, according to each criterion's perceived importance. A component's rank is a composite of its scores in each quality area. The goal is to enable the user, if he is presented with a set of similar assets, to choose the asset with the highest overall rank.

Ranked schemes suffer from the same problems as levelled schemes, along with an additional complication:- inconsistency of scoring and weighting. For example, a component might be assessed according to such aspects as memory usage, performance, and mean time to failure, each aspect of which would be scored separately, weighted, and then composed into an overall score. Another component might be assessed according to different attributes with different scores, or to the same attributes with different weights. Unless there is consistency in scoring, there is no way for a library user to know, based on looking at a component rank, whether one component is "better" than another.

Enumerative schemes are intended to address the short-comings of levelled and ranked methods. In an enumerative scheme, specific quality properties are verified by either formal or informal methods, and explicitly stated to the engineer in the form of a list. Quality properties might be trivial, such as "Conforms to organizational formatting standards," or complex such as "Free from memory leakage". Enumerative schemes allow the programmer to understand the specific quality characteristic possessed by a component and to draw inferences about the systems that will use these components.

The main drawback of enumerative schemes is the logistics of organizing and presenting properties to the engineer. A component can possess an arbitrarily large number of properties and it is difficult for the engineer to discern whether all of the properties critical to his domain have been taken into consideration.

Research Issues

Metrics

The term "metrics," in the context of certification, refers to the means by which certification criteria are established and assessed. The main research topics in this area are three-fold. First, there is no agreed-upon set of standards for measuring a component's quality. Second, there is little experience in applying certification standards to existing libraries. Third, it is not clear what set of mechanisms can be used to derive these metrics.

With respect to lack of agreed upon quality measurements, an example is the situation described earlier with ranked certification schemes. If the providers of a set of components do not agree on which quality criteria should be certified, or on the importance of a given certification property, the result is confusion to the component's user. Thus, emphasis is needed on determining which quality attributes of a component are important for specific domains, and determining appropriate measurements for these attributes.

With respect to lack of experience, more work is needed in documenting which schemes are most effective in a given situation. Specifically, work is needed to determine which schemes best enable engineers to make decisions about the components to use and how the component will impact the overall quality of systems built using the components. A framework for assessing the different certification schemes would prove quite useful at this stage of the technology.

With respect to mechanism, there are three different ways of determining metrics: formal, informal, and empirical. Formal methods involve using program proof techniques to assure that specific properties hold. Informal methods include inspection. Empirical methods include testing and statistical modeling. Guidelines are needed on which methods are best used to establish specific classes of properties. Furthermore, practitioners would welcome assessments of available tools to help establish certification properties.

Local Certifiability, Modularity, and Scalability

An important issue in reuse technology is the benefit that accrues when a component's certification is local. The working group concentrated on certification of correctness, but the idea applies to any property. Local certification of a component property means establishing that the property holds out of the context of any particular client of the component. There may be constraints on the client contexts in which a certification applies, but if these are stated explicitly as part of the component specification then a certification can (in principle) still be local. Unfortunately, just stating these constraints is not sufficient for local certifiability. Some of the position papers from WISR '93 and WISR '92 illustrate the technical problems with poorly-designed components and programming languages.

There are two major reasons for wanting to certify component properties locally. Both are directly related to reuse. The first is that local certifiability is tantamount to the ability to reason modularly, on a component-wise basis, about software systems. The intrinsic importance of local certifiability is based on the intractability of reasoning about the monolithic programs that would result from recursive source-code expansion in large software systems. It is simply a practical reality that if one cannot reason modularly about a large system, one cannot really hope to reason about it at all. Any scalable reuse technology, then, must be based on foundations that support local certifiability of component properties.

The second reason has to do with cost. Although there is evidence that formal methods can reduce life-cycle costs, there is still a potentially significant short-term expense that tends to limit their use in practical certification efforts. If a certification can be done locally, then even if the process is costly—as might be the case for, say, a formal proof of correctness—this cost can be amortized over the many future uses of the component. In this case the marginal cost of even an expensive certification becomes negligible for a heavily used component. But if a certification has to be redone for every use of a component, then incremental cost becomes a significant issue and one of the main potential benefits of reuse (i.e., amortized costs) is reduced or lost.

Reuse after modification vs. black box reuse

The group was divided on the issue of whether or not a reusable component or system could be modified once it certification was complete. The two positions were discussed as follows.

Reuse after modification

Research on local certifiability and modularity is an ongoing topic to enable such modifications without requiring the reuser to recertify or reverify the whole system or component. This is obviously an important issue in terms of cost. If the recertification could be done locally, then user modifications would not destroy the integrity of the system or component.

Black Box Reuse

Software components that are stored in a repository for software reuse must be used exactly as is, and not modified before they are integrated into their host system. There are two arguments for this stance, a conceptual argument and a pragmatic argument.

- **Conceptual argument:** We consider that components are represented by their functional specifications, and that their internal structural details are not available to the "reuser." If the reuser feels the need to modify a software component, we view this as a reuse, not of the overall component, but rather of its building blocks. Then these building blocks should have been previously included in the repository, following proper certification and baselining procedures.

- **Pragmatic argument:** The argument that was invoked here is the syndrome of the programming student who protests "my program was working fine –I just changed a line." The fact of the matter is, software reuse must be applied along with certification-based baselining procedures; and we find it unquestionable that the certification of a component is totally invalidated the minute any modification is done on the component.

More work needs to be done on connections between certification, verification, reuse, programming languages

We view the techniques from testing, reviews and inspections, software metrics, certification, and formal verification as contributing to verifying that a given software artifact has desirable properties such as correctness, functionality, performance, structure, and conformance to standards (of documentation, of layout, of use of language, etc). It is not clear which properties correlate best with reusability of the artifacts; more experimental evidence is required. Furthermore, it is not clear that formal methods are the most appropriate way to verify these properties, despite an intuition that "correct" software is more reusable than "incorrect" software.

The application of some techniques leads to undecidable, intractable, or impractical problems for automatic tools. Aliasing and pointers are common examples. These problems are sometimes inherent in the logic and programming language constructs used, rather than reflections of our ignorance of better techniques. Can, and should, we remove these difficulties by designing languages and methodologies which avoid the use of problematic constructs and properties?

Our discussion needs to go beyond code reuse and address the connections between formal methods and reusable design, and reusable specification.

Conclusions

The application of formal methods and certification techniques and tools to reuse is critical from the perspectives of the producer, broker and consumer to ensure quality software. More work needs to be done to improve these tools and techniques as outlined above. Increased consumer demand for quality assurance of reusable components will alleviate the problem of trust and further encourage consumers to practice reuse.

Management Issues

Patricia Stump, IBM Endicott
stump@vnet.ibm.com
Terry Huber, DSD Laboratories, Inc.
huber@dsd.paramax.com

The Management Issues Group was a diverse group made up of representatives from the Department of Defense, Defense Contractors, commercial organizations and universities. During initial discussion, the group discussed the differences among the impacts of reuse on business practices and business decisions on each of these enterprises. In particular, the group discussed how each of these enterprises approach incentives and how customer involvement is different.

In commercial enterprises, the practice of software reuse is internal to the developer. Although a user may indirectly benefit from an organization developing software using software reuse processes, (faster time to market, lower product price, etc.), the process is unknown to them. This is true due to a "off-the-shelf" as opposed to "made to order" environment. The customer is interested in the final product and the product sold is usually mass-produced. On the other hand, in the DoD (and Government) environments, the developer (contractor) is in a "make to order" environment and the Government customer is involved in the process. The involvement of the Government in this environment includes control over the software development process. Commercial enterprises, as well as Government and DoD organizations are interested in developing an incentive program for its employees. However, the Government is also interested in developing an incentive program for its contractors, which would be tied into the acquisition process.

The remaining discussion during the group session primarily focused on the "commercial" enterprise.

Approach

The original intent of the group was to focus on topics that should be the crux of a business plan/business model for implementing software reuse and to develop an outline(s) to be used as a tool in making business decisions. As we were taking turns defining "business model," we learned about Hewlett Packard's Flexible Software Factory (FSF) [Navaro93]. We used the FSF model's structure and concepts for most of our remaining discussion on management issues.

Two of the main components of the FSF are the Elements and the Adoption Strategy. The elements, of which there are five, are the theories and concepts behind a reuse program which address where an organization is, and where it wants to go with respect to reuse. The adoption strategy is built on the elements and represents the implementation of how to get from an initial state to the state of fully institutionalized software reuse.

The FSF elements are:

- Customer and Values Elements
- Business System Elements
- Structural System Elements
- Support System Elements
- People System Elements

Customer and Values Elements address market stability, market history, market predictability and the culture of the organization. Business System Elements represent the support structure within an organization, for example its purpose, objectives, group norms, and resource planning. Structural System Elements are the structures and processes that are put in place to meet the organization's objectives, for example a software development process, roles, organizational boundaries, communication, and problem solving and decision making processes. Support System Elements are the infrastructure that supports the structural system, for example tools, technologies, funding models, and measurement and feedback systems. People System Elements are the human resource structure, for example staffing, evaluation and reward, and education and training.

Our approach was to create a few questions about the FSF Elements and about the Adoption Strategy which would allow us to share our professional experiences and insights into the problems and solutions of successful software reuse.

- Questions to answer for each Element:
  - What are the impacts of a reuse program on this element?
  - What are the software engineering methods that impact or help control these elements?
- Questions to answer for Adoption Strategy:
  - What mechanisms do you use to change the elements?
  - How do you change the behaviors?

Though we agreed we would not have time to address it, we felt another question which needs to be addressed is: How do the elements interact with each other?

Our discussion and results focused on Customer and Values Elements and the Structural Elements. We included ideas, thoughts, and discussions about Reuse Adoption Strategy as they arose. Many of our concerns were included and answered by Ted Davis using the Reuse Adoption Guidebook [SPC93]. Of value to our discussion was a table of critical success factors which is used to set reuse goals focusing on "what" is to be accomplished. It lists all the critical success factors in each of four groups: application development, asset development, management, and process and technology. We also used the Domain Assessment Profile from the Reuse Adoption Guidebook. This figure is used to help assess reuse potential by assessing the degree to which five factors are exhibited in the organization.

- Market Potential
- Commonalities and Variables
- Standardization in the Domain
- Domain Stability and Maturity
- Existing Domain Assets

Results

Customer and Values Elements

In discussing aspects of the Customer and Values Elements (which addresses market stability, market history, market predictability and culture of the organization), the group separated the discussion into two groups: (1) short-term benefits under initial software reuse and (2) long-term benefits under institutionalized software reuse. Furthermore, the group defined "customer" as either the end-product user or the software developer.

During the short-term, or when software reuse is being initiated into an enterprise, the benefits gained by the customer is dependent upon the availability of reusable assets, the understanding of the customer, and the product family size. If there are no available reusable assets and the product family size is small, then the product will take much longer to go to market. If there are no available reusable assets, but the product family size is large, developing the product suite will be faster, but the first products will have a longer time to market. If reusable assets are available and the customer's needs and wants are understood, then, the benefits to the customer would be similar to that of the long-term or institutionalized reuse.

In the long-term, when software reuse is institutionalized, the benefits to the customer include: consistency across products, lower product price, lower service/maintenance costs are lower, the product is available sooner and a more reliable, higher quality product.

Structural System Elements

In discussing the various aspects of the Structural System Elements, such as software development process, roles, organizational boundaries, communication, and problem solving and decision making processes, several problems or barriers were raised. The group identified six problems/barriers. They are listed here, with discussion of solution following.

1. <u>Product manager loses control over assets going into his/her product.</u>

   Solution: Structural systems should be put into place that will result in managers losing control over system development slowly. Adoption strategies should be incremental which have both short term and long term goals. Allowing managers to have some influence over or input into asset development could be substituted for control. Begin with low-granularity of reuse, then move towards a higher granularity of reuse.
2. <u>Technical staff experiences a loss of independence.</u>

   Solution: All members of the team need to understand the business/technical objectives of the product family (i.e., both the asset development and product development). The technical staff needs to be educated on the business objectives for doing software reuse, so that they can understand the reasons behind the move to reuse, as well as the organization's implementation plans (See discussion in #4 below).
3. <u>Structural elements need to support:</u> (a) The asset life cycle and the product life cycle, since they are treated separately; and (b) Communications, problem solving and decision making across asset and product organizational boundaries must be addressed.

   Solution: (a) Development of an asset needs to be treated similarly as a product. An asset has a life cycle, just as a software product does. Thus, support must be provided for the asset through out its full life cycle, including maintenance. However, training and education is important to reach this goal. Also, asset life cycles need to be incorporated into product planning from the beginning and tailored to the particular environment.

   (b) By establishing working groups at all levels of the organization, communications links can be established. Not only do new roles need to be introduced, the organizational boundaries may have to change (i.e., involve marketing representatives in process). So that organizational boundaries support communication, problem solving and decision making across an organization, new boundaries need to be established. These boundaries should support complete units of work, reconciliation of processes between units of work and a shared understanding among all personnel involved. Structures/Boundaries need to support a single view of the end customer and a feed back loop needs to be put in place from the system developers to the asset developers.

4. Lack of understanding of the process, system, and planning.

   Solution: The extent and content of understanding for reuse processes and organizational objectives must be the same for all personnel involved. First, the organization should perform an assessment to get a profile of the organization. This of course should be facilitated by an experienced person who is an objective party. As a result of such an assessment, organizational goals can be developed and then a baseline against which to measure those goals will be in place.

5. Potential productivity loss trying to debug using components not owned by the product line (may not have source code for the component, may not be allowed to contact the component developer, etc.).

   Solution: Select components that are "well specified, correct, robust and efficient."

6. If you are producing and consuming simultaneously, how do you trade off between producing a product and generating new assets.

   Solutions: Need measurable goals for reuse cascading at all levels of management that support the production and maintenance of assets. Rewards need to be tied to successful reviews.

   Through re-organization, asset development can be kept separately from system/product development, with a liaison working as a domain manager. The domain manager would be able to see the goals of both asset development and system development.

   A reuse advocate could be inserted in the project and be responsible for making sure product engineers understand what assets exist and what new assets can be developed/generated.

References

[Navarro93] J. J. Navarro, "Organization Design-Based Software Reuse Adoption Strategy," 6th Annual Workshop on Software Reuse, Owego, NY, 1993.

[SPC93] Instituting Reuse Practices, A Reuse Adoption Guidebook, Software Productivity Consortium Services Corporation, available November 15 from (703) 742-7211.

Additional Information

You may obtain soft copies of the Proceedings of the past two WISRs (WISR'4 and WISR'5) and this year's workshop (WISR'6) via anonymous ftp to

```
gandalf.umcs.maine.edu  (/pub/WISR/wisrN directory)
```

where N is the workshop number.

For additional information contact:

Larry Latour
University of Maine
Department of Computer Science
222 Neville Hall
Orono, Maine 04469

office: 207-581-3523 fax-1604
email: larry@gandalf.umcs.maine.edu

gandalf.umcs.maine.edu

where N represents the workshop number.

A summary of last year's WISR'5 working group reports appears in the April 1993 issue of SEN.

Paper copies of the workshop proceedings are available at a cost of $30.