RMISE Workshop on Software Reuse Meeting Summary

Will Tracz

Stanford University

Summary

A three day workshop focused on software reuse brought together a small group of leading researchers and practitioners. The goal of the meeting was to identify the current state-of-the-art and state-of-the-practice in software reuse and to suggest ways of closing the gap between the two. The meeting consisted of ten invited presentations addressing a broad range of technical, economic, legal and social issues The results of the workshop included six working group reports and a list of software reuse issues that the attendees reached consensus on. The most significant conclusion of the workshop was that tools and technology are emerging today that show great promise in capturing the design and implementation tradeoff knowledge that can be reused to gain an order of magnitude more productivity over simple forms of reuse (subroutine libraries and code templates).

1. Overview

The first jointly planned and sponsored workshop by SPC (Software Productivity Consortium), MCC (Microelectorics and Computer Technology Corporation), and SEI (Software Engineering Institute) in conjunction with the RMISE (Rocky Mountain Institute on Software Engineering) was held October 14-16 in Boulder, Colorado. The goal of the meeting was to identify the current state-of-the-art and state-of-the-practice in software reuse and suggest ways of closing the gap between the two. The meeting consisted of ten invited presentations:

1. **Issues and Overview** -- *J. Goodenough (SEI)*
2. **Methods** -- *G. Booch (Rational)*
3. **Representations** -- *C. Richter (MCC)*
4. **Report on Minnowbrook** -- *S. Bailin (Computer Technology Associates, Inc.)*
5. **Tools and Environments** -- *W. Riddle (SPC)*
6. **Constructive vs Generative Approaches** -- *T. Biggerstaff (MCC)*
7. **Domain Analysis** -- *J. Neighbors (Systems Analysis, Design and Assessment)*
8. **Social Issues** -- *W. Tracz (IBM)*
9. **Economics** -- *J. Gaffney (SPC)*
10. **Legal Issues** -- *J. Goodenough (SEI)*

The rest of the program was allocated to working groups and general discussion on the following topics:

- **Software methods which support reuse** -- *Jones (GTE)*,
- **Design representations which enhance reuse** -- *Baxter (MCC)*,
- **Code representations which support reuse** -- *Goodenough*,
- **Tools and environments which support reuse** -- *Williams (RMISE)*,
- **Constructive and generative approaches to reuse** -- *Bailin*, and
- **Domain analysis** -- *Neighbors.*

This report summarizes each presentation and the results of each working group along with the issues which achieved consensus. The final section includes a summary of the key statements made by the attendees.

1.1. Demographics/Background

Of the 30 attendees, four were from SEI, five from MCC, four from SPC and two from RMISE. The remaining invited attendees were split 9:4, industry to academia. (Of these attendees, only 2-3 were what I would call *"window shoppers"*, i.e., they came to observe rather than participate. This left 95% of the attendees who were researchers or software developers actively pursuing reuse technology and tools.) The mix of industry to academia resulted in a "down in the trenches" rather than ivory tower atmosphere.

EH0278-2/88/0000/0041$01.00 ©1988

MCC was addressing software reuse technology, looking 5-10 years out. SPC was looking for tools and methodology in the 3-5 year time frame. SEI, besides addressing the technology transfer issue, was evaluating current technology. The academians offered empirical evidence, and, in general, there was no one "political or religious" view that dominated. Because roughly a third of the attendees also were at the Minnowbrook workshop this past July 28-31, there was strong guidance to avoid some of the thrashing and biased views that had occurred during some of the sessions there. As Lloyd Williams, the director of the RMISE and chairman of the workshop committee, put it, "The workshop was the right people, at the right place at the right time."

2. Consensus Items

The following conclusions were agreed upon by those present at the workshop:

2.1. Domain analysis is at the heart of reuse.

This is the Catch-22 in any engineering discipline: "Before you can reuse something, you have to have something to reuse." The more effort spent in analyzing what is reusable, and how it can be reused, the more likely it is to be reused.

2.2. The largest payback is on "vertical" domains.

The message here reflects something that the Japanese software factories as well as small PC software houses have capitalized on well, that is, if you go after a small, well defined market, you can make a good business case to justify the increased expenses involved in developing a good set of parameterized components or templates that address that problem domain and the tools for generating applications from them.

2.3. There exists a small number of abstractions for each problem domain.

This conclusion was reluctantly accepted by some attendees. It was felt by some that a large number of abstractions (>1000) would be necessary for programmers to effectively construct new applications. From the data presented, and the experience of those in attendance, it was observed that somewhere under 100 abstractions are all that exist in most problem domains. Furthermore, if a larger number of abstractions did exists, then there was probably just cause for further analysis and parameterization to cut down the total number.

2.4. Object-oriented approaches are the best for reuse.

Object-oriented software design and development appeared to be the best "yet" known method of software development for reuse. It was agreed that the data encapsulation and information hiding, along with inheritance were the features that promoted reuse.

2.5. Non-technical reasons inhibit widespread reuse.

There was unanimous consensus on this issue. The general feeling was that management had a short sighted view on software development and wasn't about to commit the resources for tools and training in reuse technology. The "not-invented-here" problem was felt to be easily overcome with proper management incentive. The GTE experience appears to be a good case in point.

Gerry Jones and Ruben Prieto-Diaz from GTE Laboratories described how their research center helped set up a small library for reusable software (ORACLE on a PC) for one division in GTE. Management offered $25.00 for each component accepted into the library. It set up a certification group to screen the contributions, and contracted SofTech to adapt their Ada Reuse Guidelines [5] for COBOL and C. A separate Software Asset Group maintained the library and monitored its usage. Initially 300 components were accepted. Currently (one year later), the number is down to 150 due to redundancy and lack of use.

Furthermore the Software Asset Group is soliciting (build or buy) for strategic contributions to further populate the library. Management was directed to plan for reuse. Currently 25% of each delivered software product is reused code. The goal is 50%. Two other divisions in GTE are setting up similar reuse libraries and their corporate research center is developing domain analysis tools and techniques to support populating them.

2.6. Recording design/code decisions helps maintenance and reuse.

Software maintenance and software reuse are very similar in nature. Tools that support one, often support the other. It was widely recognized that in order to move software reuse out of programming-in-a-small-group to programming-at-large, there is a need for tools to capture the loose, unstructured process of evolving design from specifications and into code. This "traceability" (see *"Preventing the most-probable errors in design"* by Bob Poston in the November issue of **IEEE Software**, page 87) of what part of the requirements is satisfied by what part of the design and implemented by what part of the program (and what the design or implementation tradeoffs that were made to arrive at this design) have eluded most developers, partially because it is time consuming, but also because there are no tools to support the process -- *until now.* Ted Biggerstaff (MCC) described the gIBIS (Graphical Issue Based Information System) based on HyperText technology and a design theory based approach by Horst Rittel of Berkeley. Hypertex allows you to create a web of different types of information. This web can connect documentation with design, requirements specs and code, etc.. The power of gIBIS lies in the email system placed on top of the web. The mail system supports programming-in-the-large. Large systems are composed of several subsystems. Each subsystem may have certain hardware and software dependencies on other components. The web provides a mechanism for organizing decision/tradeoff points. At each point several possible positions (solutions) are stated along with the arguments for and against. Individuals concerned about certain issues can subscribe to these decision points. Whenever a new position is stated, a new issue raised, a new argument made for or against an existing issue, or decision made that may affect others, the subscribers are automatically notified. Underlying this scheme is a mechanism for capturing design decisions -- answering the question "Why was it done this way?", or "Why wasn't it done this other way?"

3. Invited Presentations

3.1. Issues and Overview

As lead presenter John Goodenough gave a "provocative rather than definitive" statement on software reuse. The issues that were raised served to contrast the differences between undisciplined program development and reuse-oriented programming. Also, no definition of software reuse, or classification of what can be reused was given. This was strategic in that some of the attendees felt that an unfortunately large amount of time was spent at the Minnowbrook conference arguing over these points. For purposes of the workshop, reuse consisted of reusing any knowledge that was codified.

The following is a summary of some key points in this presentation

- **Reuse Goals**: The goals of reuse are to improve productivity, reliability, schedule and reduce maintenance cost -- "to exploit what has been learned and used for similar programs in the past."
- **Capturing and Applying Programming Knowledge**: There are many ways to reuse code: parameterization, program generators, fourth generation languages, transformation systems. A new paradigm should be observed: concentrating on the difference between existing software and a new system.
- **Technology Readiness/Impediments**: Knowledge-base tools, program transformation system and formal specifications are research areas which show promise. Program generators and generic software designs/architectures are technologies that work in specific domains. Component libraries and retrieval systems raise legal issues. Training, tools and techniques need to be developed. Finally, an unanswered issue concerns whether these techniques scale.

- **Non-technical Impediments**: Government procurement regulations and other legal issues as well as the momentum of traditional programming development styles inhibit reuse. The costs of capturing, developing, maintaining and learning to design with reusable software also need to be addressed.

3.2. Methods

Grady Booch from Rational cited the importance of methodology in handling complexity and communication. He cited the trend to build reusable software components as risk reduction (error reduction) technique and a way to justify their costs. The following domains of reuse were observed in regards to targets of opportunity:

- inside the head,
- inside a program,
- between programs,
- between programmers,
- between projects,
- between divisions, and
- between companies.

As stated in his book **Software Components with Ada** [4], a package does not capture large enough abstractions, and so the concept of reusable subsystems needs to be addressed.

3.3. Representations

Charlie Richter (MCC) presentation gave an AI slant to reuse. His view of reuse was broken down into two sides: *Producers* and *Consumers*. The producers

- identify,
- capture,
- organize, and
- describe

software for reuse. The consumers

- *find,*
- *browse,*
- *understand, and*
- *compose*

reusable software. If one analyzes the operations cited from these two perspectives, one needs to ask the question, who are the producers and consumers, what are their backgrounds, and what tools can assist in each step? The more fundamental issues are how can the knowledge/information be represented to facilitate each of these operations and what limitations does a representation pose?

Domain analysis serves to identify software for reuse. Frame-based knowledge representation systems (CAPS-Netron) and object-oriented programming systems (Smalltalk) with inheritance and classes offer representational support for taxonomies. Abstract data types, Ada generics and software templates offer a lower level of representational support. HyperText systems help organize documentation and facilitate describing for reuse.

On the consumer side, pattern matching serves as underlying approach to locating software for reuse, but, pattern matching is driven by the representation, or vice versa. MCC has two tools to assist in finding software: PARIS is based on matching pre- and post-conditions on components, and ROSE, matches data flow diagrams. In each case the representation affects the domain. Composing new systems from reusable artifacts is still an open problem. Superimposition of code or designs was one technique proposed (automatically combining and optimizing two algorithms).

3.4. Report on Minnowbrook

Sid Bailin from Computer Technology Associates, Inc. gave a summary of results generated by four working groups at the Minnowbrook Workshop on Software Reuse, which was held July 28-31, 1987 and sponsored by Syracuse University and the University of Maryland in cooperation with RADC. There were 12 presentations, and 11 position papers. The four working groups were:

1. **Scope of Software Reuse** -- Basili(UM) and Shaw(SEI)
2. **State-of-the-Art/State-of-the-Practice** -- Duvall (Duvall Computer Technologies) and Evans (Expertware)
3. **Tools and Environments** -- Norcio(NRL) and Bailin (CTA)
4. **Future Directions/Foundations** -- Zelkowitz(UM) and Musser(GE)

The **Scope of Software Reuse** working group generated the following (broad) definition: **Reuse** *is a matter of employing knowledge that has been previously compiled. This information may take the form of information passed between people, knowledge recorded in plans or knowledge embodied in code, tools and products.* They also derived a classification system to document forms of reuse. Information gathered include: activity (e.g., specification, design, code), type of artifact (e.g., document, processable document), medium availability (e.g., institutional, department, personal), maturity of use (e.g., used by single person, a department, etc.), mechanism/representation (e.g., parameterize, template, black box), extent of modification, and granularity (e.g., small fragment). They further identified three forms of knowledge that is commonly reused: people, methods and tools. Defining the relationship between these three areas and capturing the knowledge about them formed the framework for future research.

The **State-of-the-Art/State-of-the-Practice** working group arrived at two related conclusions:

1. Reusable components are really programming language or ISA extensions.
2. "We are focussed on the easy problem - not dealing with the conceptual level."

These statements emphasize the differences between the SOA (State-Of-the-Art) (reusing abstractions) versus the SOP (State-Of-the-Practice) (reusing code).

Other comparisons made were:

- Libraries
  - **SOA** - commercial libraries of requirements, design, code and tests.
  - **SOP** - low level repositories of math, and IO routines.
  - **Obstacles** - not a standard language, no methodology, no standard interfaces, no useful abstractions.
- Tools
  - **SOA** - mostly domain specific
  - **SOP** - collections of small unintegrated tools
- Development methods and practices
  - **SOA** - Ada generics, reuse of architectures, recognized need for standards.
  - **SOP** - code not designed for reuse, unorganized reuse of specs and designs, no standards or guidelines.

The **Tools and Environments** working divided the basic activities of reuse into two categories: *Creation* and *Use*. The creation activity focused on

- **Domain Analysis**,
- **Development/Refinement**, and
- **Classification/Storage**.

The use activity focused on

- **Search/Retrieval**
- **Assessment**, and
- **Incorporation**.

The proposed approach to **Domain Analysis** was modeled after DRACO [6] as modified by

Prieto-Diaz [7]. No mature tools were assessed to be currently available to assist in this process. **Development/Refinement** tools include object-oriented languages, configuration management tools, structure analyzers and quality metric or standards checkers. Various controlled and uncontrolled vocabulary as well as knowledge based tools are available to support software **Classification/Storage**.

Search/Retrieval is supported by some natural language and structured query tools along with some semantic search capabilities. Browsing can be found in HyperText systems and programming environments such as Smalltalk. No tools are available to help the user evaluate/**assess** the components once they are retrieved, but several language processors and application generators help **incorporate** the components in applications (assuming reuse is at the code level).

3.5. Tools and Environments

Bill Riddle (SPC) made several points on the SOA versus SOP of tools and environments. There are currently no high level stand-alone, broad-scope, integrated environments. People are looking into tools that support the selection, adaptation, and assembly of systems at a high level of granularity. Reuse at a high level results in large improvements in productivity at the cost of flexibility, therefore tools are needed to assist in the adaptation (extensibility, customization, tuning and portability) of components. The mechanisms for adaptation identified were *substitution*, *transformation*, and *generation*. Tools play a key role in affecting the way programmers think about solving problems, and the methods that they use in arriving at implementations.

3.6. Constructive vs Generative

Ted Biggerstaff's presentation was split between a review of his seminal framework [2] for reusability technologies in which the composition and generation approaches to reuse were originally stated, and an overview of the reuse projects currently under development at MCC. Most of the points covered in the first part of the presentation can be found in his most recent article in the March issue of IEEE Software [3]. In particular he emphasized the representational (components and composition), operational (tools to find, understand, modify, and compose), and organizational (retrieval and inheritance) aspects of reuse (initially described by Charlie Richter in his presentation), then cited the tradeoff dilemmas of reusability -- generality versus payoff, component size versus reuse potential, implementation effort versus maximum payoff, and capital investment for library population versus payoff. While reuse of code has given some limited success (25% payoff threshold) in narrow, well defined and understood domains, the reuse of designs needs a representational breakthrough to capture and manipulate domain knowledge and achieve an order of magnitude improvement in productivity.

The second part of his presentation focused on moving reuse upstream in the software development life cycle. All the systems described (Prep, ROSE and gIBIS) focus on the reuse of requirements, specifications and designs by having a library of design schemas and abstract algorithms with domain types and constraints to help synthesis new applications (or to assist the user in the construction of new applications). Several applications of HyperText systems were describe. The most notable was gIBIS. Other projects include support for "vivisection" or reverse-engineering a design or code. The key point was the capture of knowledge to assist in the "understanding problem".

3.7. Domain Analysis

Jim Neighbors presented a summary of the DRACO approach to domain analysis. Draco is a generative approach to reuse. It can be viewed as an application generator generator. Basically, a domain is analyzed, and vocabulary defined consisting of objects and operations. A grammer is then defined from the vocabulary and various components associated with the vocabulary identified. A generator is created that takes "specifications" written in the grammer and generates code. The code generation may require more sophisticated domain analysis such as scripts, plans and goals, in order to refine and parameterize the code fragments and components that will be reused.

3.8. Social Issues

Will Tracz presented a summary of the most often cited reasons why software isn't reused: lack of tools, training, education, methodology, motivation -- both financial and psychological (see COMPCON Paper [8] for more details). The Japanese software factory approach was then analysed, and their training methods, management style and tools contrasted with current US software development practices. The conclusion was that, while the Japanese may not be using the most current state-of-the-art methods and facilities, they have developed expertise in using the tools they have and understanding the marketplace they address and are willing to standardize their process and invest the capital necessary to develop reusable software components and code templates, along with component development and certification groups necessary to support them.

3.9. Economics

John Gaffney presented a simple cost model for calculating the cost (C) of developing software given that R% of the code was reused, and that the cost of reusing a line of code is b% of the cost to develop new code. The simple cost model is as follows:

$C = (1-R)*1 + b*R$ -- cost of new SW plus cost of reused software

A second model was introduced that take into account the cost of developing the reusable software originally (E). Given this factor, it the minimum number of times that software needs to be reused in order to break even (N_0) can be determined.

$C = (1-R)*1 + R(b+E/n)$ -- where n is number of times reused

$N_0 = E/[1-b]$ -- Break even point = Cost to generate/cost savings per reuse

The final model reflects the cost of developing software on a given project for reuse (Rc% of total) over "m" projects along with the R% reused and (1-Rc-R)% new.

$C = (1-Rc-R) + (Rc*(E/m)) + R*((E/n)+b)$

Possible values for b (the cost savings of reuse) are:

- For reusing code only: *0.85*
- For reusing design and code: *0.35*
- For reusing requirements, design and code: *0.08* (testing = 8% remaining cost)

3.10. Legal Issues

John Goodenough summarized some of work being done at SEI by Pamela Samuleson along with the results of the SEI sponsored workshop on legal issues in software. Two topics were covered: the role of government rights, and the current judicial view of protecting software. Ideally the law should protect the vendor by preventing competitors from selling identical, improved or corrupted versions of their software, and establish who is liabile for errors. Similarly, the reuser should have the ability to use, modify, extend and correct purchased software, and in turn provide software to others. There is currently disagreement about the best legal basis for protecting and rewarding each party and no clear cut incentives to the developer or the end user and maintainer of the end user system.

Part of the problem is the unlimited government rights to software and its documentation. The government can use, duplicate and disclose any software developed under contract. Restricted government rights have been proposed that limit revealing the software only for such purposes as maintenance, thus protecting the investment of the original contractor.

The US justice system is also sending mixed signals to software developers. Existing forms of software "protection" have proven inadequate. While it is legal to reverse engineer hardware and even analyze and reproduce a chip mask. The Whelan case demonstrated that even if a program is rewritten in a different language, using different algorithms, if the logic and structure are the same, then the original

programs copyright was infringed. Yet a copyright is traditionally interpreted to only protect "expression" (i.e., source) not "ideas". Numerous other cases were cited and reference documents listed.

4. Working Groups Results

Six working groups were planned originally for the workshop but Domain Analysis became such a hot topic that a new working group was formed and the Construction and Generative working groups combined.

4.1. Software Methods Which Support Reuse

The goal of the software methods working group was to identify the characteristics of the methods that support software reuse. It was recognized that "methods" encompass the entire software life cycle, not just software design. Certain "brand name" (JSD, SADT, and OOD) methods were discussed and it was agreed that reuse could be mapped on to them, but that some supported reuse better than others. The reasons that some methods support reuse better than others is that systems developed from these methodologies tend to be based on concepts that are more stable than functional decomposition (i.e., JSD is takes an environmental view, SADT looks at data flow, and OOD focuses on objects).

In general, a method needs to help develop a conceptual model, represent it and canonicalize it. The method should support suitable levels of abstraction and facilitate synthesis of new applications through clean adaptable interfaces. It was pointed out that there are no physical rules (yet) that impose restrictions on specifying software abstractions and the multiplicity of representations inhibits reuse. (Note: part of the problem with the lack of significant results of this group was the fear of overlapping the design group.)

4.2. Design Representations Which Enhance Reuse

The design group recognized that the design process applies to not only code, but specifications, documents, tests and even designs. Even though design spans several concepts: domain analysis, constraints, architectures, plans and goals, process programs, and specifications/requirements, one tool/methodology/representation supports them all -- gIBIS (see section 2.6).

4.3. Code Representations Which Support Reuse

The code reuse working group addressed two separate issues:

1. What information is needed to reuse code?
2. What are the characteristics of language support for reuse?

A potential reuser of software is interested in a number of characteristics of the code. Its functionality, performance, dependence on other units, and overall correctness are of primary importance. This information can be represented as textual documentation, diagrams, benchmark reports, call graphs, usage history and validation certification.

A programming language affects the granularity and style of reuse. It affects the modularization and interface design as well as the ability to add or remove operations from an existing module.

4.4. Tools And Environments Which Support Reuse

The tools and environments group built upon the results of the Minnowbrook working group (see section 3.4). They concluded that future reuse systems would depend upon an information repository consisting of project (application specific), domain specific, and general knowledge/components. Current SOA provides some tool support for code reuse, but SOP tool support was projected for 1995. Tool support for reuse of design was projected to reach SOA by 1995, with no guesses made on SOP of design tool support. The group felt that transformational approaches would make the most significant impact in the future. For example, if a portion of a parts library became crowded with several versions of the same

component, then the copies might be replaced by a parts generator. The importance of traceability was emphasized to determine what tools/components are actually being used[1].

Integration was the basis of an software development environment. The environment consists of a notation that drives the tools that support the methodology that uses the notation. Two forms of integration are important: *internal integration*: tool-to-tool and tool-to-information-repository through an object management system, and *external integration*: a uniform user interface to the tools and a process management system.

4.5. Constructive And Generative Approaches To Reuse

It was fortuitous that the constructive and generative working groups were combined because, as a result of considering both, certain dualities surfaced. A generative approach is the evolution of a constructive approach. Black boxes may be thought of as the purest form in a constructive approach, but macros are both constructive and generative. An application generator uses templates and building blocks to create a new program. With black boxes there is more flexibility to glue them together in different combinations, while with application generators and 4GLs, the combinations are more restricted because the system supplies the glue and it only knows how to glue things together in certain patterns. (A constructive approach requires the domain knowledge be supplied by the user while a generative approach has the domain knowledge built in.) The output of an application generator is harder to extend or modify than a system constructed from building blocks. (Note the CAPS system from Netron [1] handles all these situations nicely.) Finally, a constructive approach is more imperative or procedural, while a generative approach is declarative.

Different forms of "glue" or component composition mechanisms include:

- inheritance,
- instantiation,
- sequence,
- pipes,
- functional composition,
- superimposition,
- binding, and
- module interconnect languages.

Many of the remarks in section 5.2 focus on the applicability of using a constructive versus a generative approach. In general one starts off with identifying the building blocks for a constructive approach, then, through recognizing patterns for putting them together, a parameterized framework is constructed in the form of an application generator.

4.6. Domain Analysis

Domain analysis can be used to match requirements to specifications and to create a model that can be used to educate new programmers. The domain analysis working group selected a sample domain to analyze (The library problem as posed for the ICSE-87). Domain analysis proceeded from several approaches (DRACO, SADT, OOD, Flow-oriented, semantic net, and GTE [7]) As a result, a richer analysis of the domain was achieved. While this result is not surprising, it highlighted the fact that certain approaches may bias the overall analysis of the system. In order to develop a truly robust abstraction, the problem domain should be analyzed from several perspectives.

Two domain analysis procedures were proposed:

1. Analysis by Domain Experts Approach

---

[1] As a sidepoint in the discussion, the Ada Software Repository was cited as being the greatest counter collection of examples of how not to write good software.

- Establish domain subject area
- Collect experts in one room
- Free associate on what we might account for in the domain
- Distinguish significant things; constrain the domain
- Elaborate foundational things
- Define objects, relationships and constraints
- Clean up the diagrams
- Re-express for reusability

2. Analysis of Implementations Approach
   - Select a set of examples - existing implementations.
   - Identify similar functions across examples, collect names and terminology.
   - For identified routines, map between them using parameters.
   - Identify levels of abstraction used to implement the functions.
   - Look for layering among functions.
   - Graph distribution of instances of functions against possible reuse factors.
   - Depending on the distribution, use a generative approach (even distribution) or a constructive approach (clusters).

5. Key Statements

The following quotes are excerpts of presentations and discussions that took place at the workshop. They reflect the personal philosophies of some of the attendees, as well as general statements about software reuse. They have been organized by topic.

5.1. Comments on Non-Technical Issues of Software Reuse

- **"Code reuse in the small happens all the time with a small group."** -- *Unattributed*
- **"The problem is to make reuse fun!"** -- *Bailin*
- **"There is a fear reuse doesn't allow fresh ideas."** -- *Jones*
- **"US management reward for working hard, the Japanese reward for productivity"** -- *Pyster*
- **"Managers should note: there is no silver bullet, no free lunch. Reuse is like a savings account, you have to put a lot in before you get anything out."** -- *Biggerstaff*
- **"Management understands the tradeoff between less labor and more tools."** -- *Booch*
- **"Software is viewed as an expense not as a capital investment"** -- *Pyster*
- **"Software gives a false sense of economy because it is easy to change"** -- *Woodfield*
- **"If we only solve the technical problems (of reuse), we won't solve the problem."** -- *Goodenough*
- **"Current government contract clauses disincentivecise reuse."** -- *Goodenough*
- **"The Japanese do not have any non-technical inhibitors to reuse"** -- *Tracz*
- **"This graph shows you need a lot of reuse before you get fantastic cost savings"** -- *Gaffney*
- **"Reuse provides savings in cost, quality and schedule"** -- *Gaffney*
- **"People won't do it (reuse) to be altruistic"** -- *Gaffney*
- **"The greatest barriers to effective reuse (i.e., > 25%) are not technical; they are cultural, organizational, legal, and economic."** -- *Pyster*

5.2. Comments on Software Reuse Tools and Methodology

- "Code reuse is here today, the question is how to do it better" -- *Williams*
- "Reuse of code is a good way to get started -- not a good way to end up." -- *Riddle*
- "Reuse follows abstraction -- abstraction follows practice" -- *Booch*
- "Before tools, you need to understand the process, you need to codify." -- *Booch*
- "There is nothing worse than a dumb tool that tries to be smart" -- *Simos*
- "Some methods put impediments in the way of reuse." -- *Jones*
- "Using a standard method is more important than using the best method" -- *Pyster*
- "A simple mechanism seems sufficient for construction new systems. " -- *Neighbors*
- "A generative approach .. only a matter of time until you run into the brick walls" -- *Goodenough*
- "The ability to produce garbage is a sign of power (in generative approaches)." -- *Goodenough*
- "A constructive approach is more easily completed or extended than a generative approach when the requirements are not immediately known" -- *Perry*
- "If you only have a problem, then use a constructive approach; if you have solutions, use a generative approach." -- *Perry*
- "The only problem with parameterization is you never have enough parameters." -- *Goodenough*
- "Think of a code generator as a pre-processor, then is your program code with pre-processor calls, or pre-processor calls with code." -- *Simos*
- "You establish standards within a technology window. As technology changes, standards change, and reuse falls off." *Biggerstaff*
- "The (reuse) tools which are best to create in the next 6 years are: a repository, a configuration management tool, and an integration tool." -- *Booch*

5.3. Comments on Domain Analysis

- "There are two kinds of knowledge: Domain Knowledge and Implementation Knowledge" -- *Embley*
- "Not all domains are the same, some are broad and expanding. Reuse is better in restricted domains." -- *Biggerstaff*
- "What needs to be done is the hard work (Domain Analysis") -- *Baxter*
- "We want a method to find canonical representations" -- *Durek*
- "Reverse engineering is one way of mining components, ... but modified systems become a blur." -- *Neighbors*
- "Reverse Engineering is a rats nest if you have to analyze code." -- *Iscoe*
- "99% of the effort is taking domains and extracting objects." -- *Biggerstaff*
- "I don't care how (domain analysis is performed), just get these results (objects and operations)." -- *Neighbors*
- "You benefit from different models of the same domain." -- *Neighbors*
- "Scenarios are the glue that hold objects together" -- *Potts*

- **"Methods for capturing and representing domain knowledge are the key."** -- *Baxter*
- **"Biggerstaff's 3-system rule: If you have not built three real systems in a particular domain, you are unlikely to be able to derive the necessary details of the domain required for successful reuse in that domain. In other words, the expertise needed for reuse arises out of 'doing' and the 'doing' must come first."** -- *Biggerstaff*
- **"The unifying concept is knowledge capture."** -- *Goodenough*
- **"It all comes back to Domain Analysis"** -- *Baxter*

5.4. Miscellaneous Remarks

- **"We successfully built on the results of Minnowbrook rather than replaying them."** -- *Simos*
- **"I measure how good a workshop is by the amount of other things I get done."** -- *Williams*

6. Summary/Personal Observations

The workshop reinforced the basic conclusions I had arrived at in my observations of the state-of-the-practice. It was disappointing, and at the same time encouraging, that the emerging technology is based on fairly simple (but not cheap) technology: high resolution graphics workstations, object-oriented programming languages and formal methods of specifying pre- and post-conditions. I believe momentum is gathering in industry and government to foster the proper environment for software reuse (i.e., create the proper financial incentives). Domain analysis is essential in populating reusable component libraries (constructive approach), or developing application generators (generative approach). Expert system technology is assisting in developing approaches analyzing application domains and representing and manipulating such knowledge to assist in the development or synthesis of new applications.

One personal revelation was stated by John Goodenough in one of the working group sessions. The topic of conversation was centered on what is the glue that holds things together in a constructive approach versus a generative approach. His observation was that as one moves more and more to a generative approach, the programming style becomes *declarative*. This reflects back on a statement he made in his opening remarks to the attendees that we should focus on what is different between one application and previous applications. Again, the importance of *scenarios* in domain analysis; if one can foresee how the components will be used, one can create the architecture and mechanism to instantiate it given certain parameters (e.g., an application generator).

The second significant outcome of the workshop was the overall excitement generated by gIBIS. This simple tool supports the capture of transformation knowledge and design decisions that, before this time, was unmanageable.

In closing, I have concluded that reuse is not an end in itself, but it is a means to an end. Furthermore, reuse is a side effect of the methodology as supported by the environment. Reuse is intuitive when the proper mechanisms are provided. It is part of a more nebulous concept called software engineering, which is another form of general problem solving. It is time the software community agree upon some basic abstractions and build upon them rather than reinvent them each time we build a new system.

References

1. Bassett, P.G. "Frame-Based Software Engineering". *IEEE Software 4*, 4 (July 1987), 9-16.

2. Biggerstaff, T.J. and Perlis, A.J. "Forward: Special Issue on Software Reusability". *IEEE Transactions on Software Engineering SE-10*, 5 (September 1984), 474-476.

3. Biggerstaff, T., and Richter, C. "Reusability Framework, Assessment and Directions". *IEEE Software 4*, 2 (March 1987), 41-49.

4. Booch, G.. *Software Components with Ada.* Benjamin/Cumming, 1987.

5. Braun, C.L., Goodenough, J.B., Eanes, R.S. Ada Reusability Guidelines. Tech. Rept. 3285-2-208/2, SofTech, Inc., April, 1985.

6. Neighbors, J.M. "The Draco Approach to Constructing Software from Reusable Components". *IEEE Transactions on Software Engineering SE-10*, 5 (September 1984), 564-573.

7. Prieto-Diaz, R. Domain Analysis for Reusability. Proceedings of COMPSAC 87, 1987.

8. Tracz, W.J. Software Reuse: Motivators and Inhibitors. Proceedings of COMPCON87, February, 1987.