| UNAS     | Universal Network Architecture Services |
| USC      | University of Southern California      |
## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>Assessment and Assimilation</td>
</tr>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
</tr>
<tr>
<td>CASE</td>
<td>Computer Aided Software Engineering</td>
</tr>
<tr>
<td>CM</td>
<td>Code Modified</td>
</tr>
<tr>
<td>COCOMO</td>
<td>Constructive Cost Model</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off The Shelf</td>
</tr>
<tr>
<td>CSE</td>
<td>Center for Software Engineering (USC)</td>
</tr>
<tr>
<td>DM</td>
<td>Design Modified</td>
</tr>
<tr>
<td>DOD</td>
<td>Department of Defense</td>
</tr>
<tr>
<td>DSSA</td>
<td>Domain Specific Software Architecture</td>
</tr>
<tr>
<td>FCC</td>
<td>Fire Control Center</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>H&amp;S</td>
<td>Satellite Health and Status</td>
</tr>
<tr>
<td>IM</td>
<td>Integration and Test modified</td>
</tr>
<tr>
<td>IOC</td>
<td>Initial Operational Capability</td>
</tr>
<tr>
<td>ISI</td>
<td>Information Sciences Institute (USC)</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>OCC</td>
<td>Operations Control Center</td>
</tr>
<tr>
<td>OTS</td>
<td>Off The Shelf</td>
</tr>
<tr>
<td>SGS</td>
<td>Satellite Ground Station</td>
</tr>
<tr>
<td>SPC</td>
<td>Software Productivity Consortium</td>
</tr>
<tr>
<td>SU</td>
<td>Software Understanding</td>
</tr>
<tr>
<td>TBD</td>
<td>To Be Determined</td>
</tr>
<tr>
<td>TQM</td>
<td>Total Quality Management</td>
</tr>
<tr>
<td>TT&amp;C</td>
<td>Tracking, Telemetry and Commanding</td>
</tr>
</tbody>
</table>


Software Productivity Consortium, Reuse Adoption Guidebook, SPC-92051-CMC, version 02.00.05, November 1993.


Others:

Public Ada Library (PAL), contact Richard Conn for PAL Frequently Asked Questions at conn@wuarchive.wustl.edu

ASSET, contact ASSET Accounts Administrator at info@source.asset.com


Pointers to Useful Resources

Literature:


C. Gacek, A. Abd-Allah, B. Clark, and B. W. Boehm, “Focused Workshop
Participants

(to include)
Agenda

(to include)
Appendix
c) Asset utilization

3. (D.3) Additional metrics
   a) Cycle time view (must hit market window)
   b) Complexity view
   c) Time of opportunity
   d) New knowledge
   e) ROI
   f) Leverage

E. (D.4) Common definitions of reuse terms to facilitate economic analysis.
6. (A.6) Tax law treatment on reuse  
   a) Expense vs. capital investment
7. (A.7) Reuse incentives  
   a) Internal  
   b) External  
   c) DOD
8. (A.8) Need a “break-through mechanism” for COTS and reuse  
   a) e.g. Mosaic, IBM PC
9. (A.9) Risk management diversification  
   a) Portfolio concept

B. Critical Issues: Factors in Asset Management  
1. (B.1) Extra effort required to make components general
2. (B.2) Repository  
   a) Versioning  
   b) Operations and maintenance  
   c) Evolution
3. (B.3) Product family: Software products built from assets as opposed to a family of assets.  
   a) Versioning  
   b) Operations and maintenance

C. Critical Issue: Factors in COTS  
1. (C.1) Interface to reuse components
2. (C.2) Performance
3. (C.3) Reliability
4. (C.4) Incompatible domain constraints

D. Critical Factors: Factors common to all (economics, assets, COTS)  
1. (D.1) Supplier relationships significantly impact economic of reuse  
   a) Licensing of acquired assets  
   b) Certification (economics of analysis vs. level of certification)  
   c) Maintenance and support
2. (D.2) Training. The economic impact on stakeholders.  
   a) Reuse process  
   b) Initial / recurring costs
The following items identify research issues that are ordered in the diagram above by importance and difficulty. Each letter in the bubble in the diagram is identified below.

A. Critical Issues: Theory, models, etc. of reuse economics

1. (A.1) Communicating reuse progress to management
   a) Need visualization models

2. (A.2) Economic view of reuse should be: Profit = Income - Cost
   a) TQM economics model
   b) Economic analysis should not focus on cost

3. (A.3) Express economic model in terms of Domain Analysis
   a) More expressive than % code reused
   b) How much of reuse effort exploited

4. (A.4) Economic analysis
   a) Need empirical data
   b) Analysis used sometimes to kill reuse
   c) Incremental investment / benefit strategy

5. (A.5) What are the decisions a product line manager must make regarding reuse? Economic models should address these.
b) Operations and maintenance

C. Factors in COTS that affect reuse economics

1. Interface to reuse components
2. Performance
3. Reliability
4. Incompatible domain constraints, e.g. vendor tells you to just ‘reboot’ the system to fix the current problem.

D. Factors common to reuse economics, assets, and COTS

1. Supplier relationships significantly impact economic of reuse
   a) Licensing of acquired assets
   b) Certification (economics of analysis vs. level of certification)
   c) Maintenance and support
2. Training. The economic impact on stakeholders.
   a) Reuse process
   b) Initial / recurring costs
   c) Asset utilization
3. Additional metrics
   a) Cycle time view
   b) Complexity view
   c) Time of opportunity (must hit market window)
   d) New knowledge - COTS can bring in whole new concepts / technology
   e) ROI
   f) Leverage
4. Common definitions of reuse terms to facilitate economic analysis.
a) Need visualization models

2. Economic view of reuse should be: Profit = Income - Cost
   a) TQM economics model
   b) Economic analysis should not focus on cost

3. Express economic model in terms of Domain Analysis, i.e. determine commonality versus variability.
   a) More expressive than percentage of code reused
   b) How much of reuse effort exploited

4. Economic analysis
   a) Need empirical data
   b) Analysis used sometimes to kill reuse
   c) Incremental investment / benefit strategy

5. What are the decisions a product line manager must make regarding reuse? Economic models should address these.

6. Tax law treatment on reuse
   a) Expense vs. capital investment

7. Reuse incentives
   a) Internal company reuse groups bid for projects the company is producing.
   b) External
   c) DOD

8. Need a “break-through mechanism” for COTS and reuse e.g. Mosaic, IBM PC, GUI Builders

9. Risk management diversification
   a) Portfolio concept: Diversify the types of components to increase the possibility of reuse.

B. Factors in asset management that affect reuse economics

1. Extra effort required to make components general

2. Repository: dealing with the phase-out / removal of components
   a) Versioning
   b) Operations and maintenance
   c) Evolution

3. Product family: Software products built from assets as opposed to a family of assets.
   a) Versioning
b) Internal

3. User
   a) Component consumer
   b) Application creators
   c) Economic impact of reuse on end-user (managers, technical specialists, operator, etc.)

4. Supplier
   a) Component creator
   b) Marketing (new market shares)
   c) Tool developers

C. State of the practice
   There is a gap in research practice and business directions. The result is that research products are having limited impact on business decisions.
   1. Research practice
      a) Drivers - Cost metrics
      b) Code-focused
      c) Limited availability of case studies
      d) Models and measures
   2. Business directions
      The business direction does not preclude the use of cost models. The feeling is that companies make decisions based on other criteria listed below in determining whether it is worthwhile to change their software practices.
      a) New measures of S/W progress:
         (1) Productivity & quality
         (2) Customer satisfaction
         (3) Cycle time
         (4) Technical roadmap
         (5) Process maturity
      b) Capabilities and services
      c) Focus on success factors, CASE studies, best practices

Most Critical Issues

A. Theory, models, etc. of reuse economics
   1. Communicating reuse progress to management
Summary of the Reuse Economics Discussion

Leader: Gary Thomas (E-Systems)
Participants: Max Brown (Lockheed)
Deborah Carino (Rome Labs)
Ted Davis (SPC)
Charles Little (Asset/SAIC)
Helmut Korp (Motorola)
Dana Ulery (ARL)
Scribe: Brad Clark (USC/CSE)

Introduction

A. Scope

- A: Theory, models, etc. of reuse economics
- B: Factors in asset management that affect reuse economics
- C: Factors in COTS that affect reuse economics
- D: Factors common to reuse economics, assets, and COTS

B. Identify stakeholders

1. Business decision makers
   a) Executives
   b) Technical
   c) Financial
   d) Managers
   e) Policy drivers

2. Library
   a) External
• How do we manage the impact of evolving technologies? (OO, integration) - 3
• Tools to support reuse - 2
• What are mechanisms to facilitate and encourage reuse? - 2
• Reuse Adoption Process - 2
• What are the Off-The-Shelf (OTS) evaluation criteria to enable their reuse? - 1
• How do we identify and scope domains? - 1
• How does reuse fit into the software development process? - 0
B. Reuse Adoption Process

One of the major issues here is in determining the steps for moving from opportunistic to systematics reuse, accounting for also providing some near term benefits. Clearly, entry costs for DSSA, systematic reuse, and reuse driven development are too high. The steps involved in the process must then include establishing benefits and managing risks in increments.

It is also very important that we learn how to build business cases for reuse. In order to do so, we have to learn how to estimate investments and their return. This will only be doable once there is a set of cost data available for analysis.

The short term view of managing human resources is to obtain management buy-in, thus providing investment. For longer term results it is necessary to motivate people to do creative reuse, building systems from existing components, while combining them in creative ways. This motivation is achievable by means of rewards for individuals that do adopt the new behavior, and obtain customer satisfaction.

C. How does reuse fit into the software development process?

Accepted processes, tools, and methods are inconsistent with domain centered reuse. We need processes which facilitate/ensure appropriate reuse practices or strategies to occur, and prevent divergence of multiple versions of the reusable assets. Thus, not only do we need a better defined reuse process, but we also must integrate it with the software development process.

Mechanisms that inform and promote development and use of reusable components must be implemented. Additionally, a reuse training process must be put in place along with a WINWIN process for engineering the reusable components.

Most Critical Issues

The voting results were as follows:

- Identify, modify, use and track existing reusable assets - 8
- Existing models and how to use them - 5
- Need better technologies to define/represent architectures - 5
- How do existing and evolving assets/technologies drive the design of an architecture? - 4
interaction patterns among them. This could be obtained by a powerful, currently inexistent, architecture description language (ADL).

Current ADLs are considered inadequate to describe multiple architecture styles, support reasoning about the architecture, represent dynamic behavior, and to provide multiple levels of abstraction.

Architecture description technologies need to be able to scale, i.e., be able to support both small and large scale systems. Methodology for describing architecture of large-scale systems need to include multiple levels of detail, multiple perspectives (e.g. software, hardware, dataflow), and provide for easy derivation of relationships to subsystems and actual code. In the long run, software should be generated from architecture descriptions rather than be built at line code level.

It was also pointed out that we need a large grain module description ADL to help manage evolution and extensibility based on architectural style, object types for locating variation, and processing/computational engine.

There is also a need for research on architectural classification. A method for architectural classification should include the various views involved, as well as support optimal search on the classified objects. This leads to the question of how to determine potential areas for reuse given that there are a number of ways to view a system.

Once we are able to describe architectures, we will also need to be able to describe domains so commonalities between them become apparent. Thus, we need to determine how much supplementary data, information, knowledge is needed for a DSSA, and how to represent it.

Adoption Management Process

A. What are mechanisms to facilitate and encourage reuse?

There is a need to determine what environments, interface standards and conventions are the most relevant, and best support software reuse. These should include interaction patterns for reusable components, and open system standards.

Clearly, simplification of modeling notation is required for artifact understandability. On the other hand, over simplification may omit some details that will later on become problems.

In the object-oriented world, inheritance and polymorphism push good design concepts to the programming community in initial development. They encourage well-defined “kluges” during...
e.g.: front-ends, architecture building, end-product generation. When doing so, we must keep in mind that tools do not capture variability, nor the sources of variability.

C. How do we manage the impact of evolving technologies? (OO, integration)

Technology is always evolving, and we are constantly drawn to the same migration problems when trying to move forward.

Currently the major problem in this area seems to be the actual move towards object-oriented technology and methods, and how to integrate them with other architecture description paradigms.

Another issue that should be looked at is the impact of framework integration standards, such as PCTE and CORBA, on reuse.

Persistent object bases are just emerging, and are expected to meld with relational data bases. How do we plan for this?

Is there some means of always being prepared for evolution even though we do not know what might happen next?

D. How do we identify and scope domains?

There is a need for identification of domains and standard inter-domain semantics. It is clear that domains are not mutually exclusive, they may contain overlaps or a domain may be composed of other domains. This characteristic requires that we not only study reuse throughout a product-line, but also across domains.

In order to obtain maximum leverage from scoping reuse domains, we should identify tools to support domain analysis and architecture specification, as well as derive methods for determining relationships of reuse potential to scope of domain.

E. How do existing and evolving assets/technologies drive the design of an architecture?

The issues faced here are the lack of reuse standards, as well as the lack of tools to support them. It is known that open system standards and integration standards affect decisions about domain analysis, but further study is needed to determine the actual extent and characteristics of these influences.

Further study on interface specifications is also badly needed.

F. Need better technologies to define/represent architectures

There is a need for a common taxonomy and notation for describing the behavior of components/subsystems/services, as well as the
Technology

A. Existing models and how to use them

A.1 What models are there?

Different models exist, such as logical design models, best product line design, physical model, architecture style, resources model, and performance model. What this group would like to see is a study determining the models that are generally needed, as well as a description of some domains which have some variability over the common set of models.

A handbook with guidelines about the various architecture models, how to use them, and what their specific characteristics are, is also being badly needed.

A.2 How to determine applicability of different models?

Once again we discussed whether some models best work on specific domains. From this discussion it became apparent that what we need is a methodology for choosing available models (e.g. set of standards or communication protocols) that best suit specific occasions.

Currently, organization are not using models to their full benefit. Models are being used to define persistent objects, subsystems, or architectures, and then are left aside. Models could be used to elicit customer needs and match to capabilities to produce requirements, or to plan for program evolution.

In order to facilitate an appropriate selection, architecture models handbook/guidelines need to include advantages/disadvantages, strengths/weaknesses of each model; correlate with attributes of domains they work best in; and correlate with anticipated evolution direction they support best.

B. Tools to support reuse

The integration of tools and OTS\(^1\) in general have already proven to be a problematic issue. It is very important to not only to pin-point the tools that best support the reuse process (e.g. GUI builder with OOA tool), but also to look into their integration issues, like the integration of tools with OTS and internally developed assets.

It is imperative that we define where OTS should be used or not,

---

1. Note: This group opted to use the term OTS instead of COTS, in order to highlight that Off-The-Shelf products may also be internally developed, not necessarily being commercialized.
level of reusability of existing assets, as well as the factors that do influence each assets’ life span. This has an immediate effect on being able to project the number of times an artifact will be reused, and thus on how it would be payed for.

A.2 How to re-engineer legacy software?

There exists a huge platform of legacy software currently available which presents an enormous potential for reuse. In order to be able to leverage from it, we must spend efforts in re-engineering and standards/technology understanding.

One approach to reusing legacy systems is to simply reuse them as they are. This may require mixing subsystems that have different underlying architectural styles. Consequently, there is a need for research in this topic.

A different approach to reusing legacy systems would be to map or convert them from their current state to a form compatible to its intended use. This new form could either be simply to include some modern standards, or actually to transform from one architectural style into another.

A.3 What are the issues for managing reusable assets?

The management of software assets involves categorization, retrieval, control (configuration management), and certification of assets. The group was concerned the most with configuration management issues. After some discussion, it became clear that there should be some tracking mechanism to support configuration management of assets that were either modified internally or externally.

B. What are the Off-The-Shelf (OTS) evaluation criteria to enable their reuse?

There is a need to define what information affects OTS selection and use, e.g.: behavior, and interface standards.

A survey needs to be done to identify what OTS are out there, along with their main characteristics, and then a pros and cons evaluation of each one.

It is also important to define how OTS fit into architectures. That is, understand how integration with other composing items is to be done, which includes mixing various architectural styles, and allowing for future software evolution.
• **Asset** - something of value; work product used in application engineering, software development, maintenance; anything in library

• **Methodology** - set of procedures and guidelines for building products

• **Model** - representation; approximate simplification; used to describe, evaluate, and reason about certain attribute/function

• **Reuse** - includes adaptation, customization, enhancement, instantiation (specialization), library utilities (sort routines), assembly, integration of components, generation, sharing of knowledge and processes, leveraging something that exists in a new way, applies to requirements, design, standards, processes (all assets that can be stored in a repository)

• **Technology** - embodied in an asset; state machines, engine that enables the production of assets, algebraic specification

**Affinity Groups**

The affinity groups were distributed in three different categories according to their overall focus. These categories are of issues related to assets, technology, and to the adoption management process.

**Assets**

A. **Identify, modify, use and track existing reusable assets**

A.1 How to identify and use reusable assets?

The identification of software for reuse and successfully reusing them is extremely relevant, and directly depends on the appropriate description and categorization of reusable assets.

This group believes that among the various things that need to be included in the assets’ descriptions are the functionality, the intent, the constraints, the interfaces and protocols, and the domain of applicability. Clearly, functionality and constraints should also be included in the requirements thus facilitating the search for reusable assets.

In order to be able to deal with the assets’ descriptions described above and their search, it is required that some new techniques, methodologies, tools be devised to guide in such process. The intent is to see if some day we will be able to have a catalog of “computes components” that is somewhat comparable to an integrated circuit catalog with “fact sheets”.

Another issue included under this affinity group was assessing the
Summary of the Domain Architecture Based Reuse Discussion

Leader: Jim Withey (SEI)
Participants: Elaine Abrams (Motorola)
Prasanta Bose (USC/CSE)
Clinton Carr (EDS)
Patricia Dousette (Litton Data Systems)
Henson Graves (Lockheed)
Hugh Heide (Rockwell)
Morton Hirschberg (US - ARL)
Ron Hughes (EDS)
Judy Kerner (Aerospace)
Elmain Martinez (JPL)
Walt Scacchi (USC/CSE)
Peter Shames (JPL)
Scribe: Cristina Gacek (USC/CSE)

Introduction

The overall discussion format was to gather the issues that were considered the three most relevant ones by each of the participants. Then group these issues by affinity based on their underlying rationale and group discussion. Subsequently, each participant distributed three votes among the affinity groups that she/he thought most relevant. This voting facilitated the generation of an overall ranking of the issues discussed.

The group also spent an extra effort towards reaching consensus definitions for several terms.

Definitions

• Architectural style - one level above each architecture, pre-determining some of its basic characteristics

• Architecture - identifies the system’s components, their static inter-relationship, their dynamic interactions, properties and characteristics, and constraints on these [M. Penedo and W. Riddle, “Process-sensitive SEE Architecture (PSEEA) - Workshop Summary”, Software Engineering Notes, ACM SIGSOFT, vol. 18, no. 3, July 1993, pp. A78-A94]
Discussion Group Results

This section presents the results of the two discussion groups:

- Domain Architecture Based Reuse Group (led by Jim Withey, SEI)
- Reuse Economics Group (led by Gary Thomas, E-Systems)
Affiliates’ Presentations

Deborah Cerino

Deborah Cerino from Rome Lab talked about their efforts towards the certification of reusable software assets. Their underlying rationale was to accelerate reuse within the Air Force and throughout DOD, by making software certification usable, practical, cost-effective, and measurably beneficial. They have several participating user sites providing inputs to their program, which act as interviewees, concept contributors, pilots sites, or distinguished reviewers. Their deliverables will be a certification framework and corresponding process, and an automated certification system, and should all be available by June 96. Their certification framework is going to consider several levels of certification, kinds of artifacts, and concerns.
Affiliates’ Presentations

Lewis Johnson

Dr. W. Lewis Johnson from USC/ISI presentation was on Software Understanding and Reuse. The understanding of a software component is critical through all steps performed while trying to reuse it, i.e., while trying to locate it, adapt it, or integrate it. Dr. Johnson’s approach is to generate software documentation on demand, composing the presentation dynamically while using a Mosaic hypertext to view it. His tool uses HTTPD and REFINE, and automatically maintains consistency between the components’ code and their documentation.
Affiliates’ Presentations

Charles Lillie

ASSET’s Dr. Charles W. Lillie described the ASSET source for software engineering technology. The ASSET’s mission is to facilitate software reuse in the US by means of a distributed support system. Dr. Lillie described the ASSET libraries contents, and also showed some statistics on number of subscribers, type of project of subscribers, kind of services requested, and search criteria used. He also described the evaluation/certification criteria used for the assets, and mentioned how to access things through the world wide web (http://source.asset.com/).
Affiliates’ Presentations

Henson Graves

Lockheed’s Henson Graves presentation was on Software Mining, and building a software mining system. The approach he described is based on linguistic processing of semi-structured data. It includes locating code units and preforming a descriptive classification; indexing code units by phrases; associating functional topics with phrases; and organizing code units by functional taxonomies. Once that is done, code retrieval can be done by searching the appropriate functional segment.
Affiliates’ Presentations

Ted Davis

Ted Davis from SPC gave a presentation on Adopting Reuse. He started by describing the motivation behind reuse: increase in productivity and quality, and decrease in integration time and training costs among others. He then talked about some reuse success stories, and why some companies fail to achieve good results. With this motivation, he went on to present SPC’s reuse adoption process and strategy, their domain assessment model, and their reuse capability model.
Affiliates’ Presentations

Don Reifer

DISA’s Don Reifer began the Affiliates’ presentations by discussing a preliminary version of the Reuse Adoption Plan of the DOD Reuse Initiative\(^1\). He said that the DOD vision is that we should be moving towards a reuse that is domain-specific, architecture-centric, process driven, consumer-oriented, easy, and has an existing technology base. He thinks that the strategy ought to focus now on adoption, and that in order to implement a specific reuse plan we must first take into account all the issues involved in the DOD reuse vision, then think about the strategy and short term objectives, and then devise a plan that makes everyone a winner. As an overall strategy he suggested that first the product line approach be implemented, then develop reuse based systems and software engineering paradigms, afterwards remove contract barriers to reuse and quicken technology transfer, and then make success apparent by marketing outreach and benchmarking.

\(^1\) Since this was a preliminary version, Don Reifer preferred not to distribute copies of his charts, but rather let the community receive the definitive version when it becomes available.
The postscript version of the document is the only artifact available.
State-of-the-Art Presentations

Reuse Economics

Bradford Clark of USC presented the current state of the practice in calculating the cost of reusing components. The presentation focused on a general model that net profit is based on the investment in making reuse possible and recovering that cost by reusing the components a number of times, see model below. Reuse programs range from scavenging components from existing software to translating components into a form compatible with the new system to building reuse components from scratch. The reuse investment rises with each method respectively. Any reuse program carries with it an overhead that is due to the need to understand the software, assess whether it is suitable for the new system, and assimilating the component’s documentation into the new system. The profit realized from reuse comes from keeping the investment in reuse low and maximizing the number of times a component is reused. The model below shows savings as the difference in cost between developing a component from scratch and developing a component by adoption from a reuse repository. The number of times a component is reused needs to exceed the investment in creating that component.

\[
NetProfit = \left[ \sum_{i=1}^{n} (NewDevelopment_i - Adaptation_i) \right] - Investment
\]

where

\[
n = \frac{MarketSize}{AvgContractValue} \times MarketShare \times Lifecycle
\]

The obstacles to the adoption of reuse are economic and cultural. Reuse programs need to be market sensitive as well as anticipate requirements for future products. Only components that are going to be reused many times should be developed. Cultural resistance could be overcome with training, incentives, measurement, and management.
Notes:

- Perry and Wolf: excellent general architectural considerations
- Prieto-Diaz and Freeman: faceted approach to software artifacts classification
- Software Productivity Consortium: SPC’s reuse adoption process, and specially their domain assessment model
Some Important References (cont.)


• R. Prieto-Diaz and P. Freeman, “Classifying Software for Reusability”, *IEEE Software*, vol. 4, no. 1, January 1987, pp. 6-16. (Faceted approach)

• Software Productivity Consortium, *Reuse Adoption Guidebook*, SPC-92051-CMC, version 02.00.05, November 1993. (Domain assessment)
Notes:

- B. W. Boehm and W. L. Scherlis: Megaprogramming
- Garlan and Shaw: strong architectural styles discussion
- Krueger: very encompassing survey on approaches to software reuse
Some Important References


Notes:

- By being very specific about the domain, and using a reference architecture, the potential for reuse is extremely high.

- By having the reference architecture providing an integrating general framework for the reusable components, we avoid all problems involved on the actual composition problem usually seen when trying to integrate reusable lower levels of artifacts.

- As we have already seen, the use of domain specific facets facilitates enormously the repository selection stage. Since here we are dealing with a specific domain, the use of domain specific facets becomes readily executable.

- The domain model and reference architecture clearly define what is considered common in the domain, thus inducing an optimal population of the repository.

- Since without the domain model and reference architecture defined one cannot start populating the repository, it becomes clear that domain engineering costs must come very early in the DSSA life-cycle. This causes a very large up front investment, which will require some time to be payed off.
DSSA-Based Reuse (cont.)

• Some benefits
  • Large amount of reuse
  • Provides a general framework for composing reusable components
  • Supports the use of domain specific facets
  • Optimal repository population

• Some drawbacks
  • Very large up front investment
Notes:

- This is the specialization of the SGS reference architecture that we derived for the Fire Satellite Ground Station.

- Here we not only dropped some parts of the reference architecture, such as “Encrypter/Decrypter”, but also added something specific to this application which is the “Mapper”, that is responsible for mapping the fire points detected into a US map.

- Things like “Telemetry History”, “Telemetry Limits”, and “Events” that existed in the reference architecture were kept in this instantiation.

- If we had a domain specific repository available, based on the reference architecture and on the fire SGS requirements we would be able to locate and reuse some domain specific reusable components.
DSSA-Based Reuse (cont.)

• Example
  • Fire Sat SGS Architecture
Notes:

• This is a reference architecture for a Satellite Ground Station (SGS) that was developed by Ahmed Abd-Allah, Bradford Clark, and myself, in the course of our software architecture research. It was based on a study done by SPC, some literature furnished to us by the Aerospace Corporation, and several discussions with Dr. John Reeves from the Aerospace Corporation.

• After having done that, we started concentrating on a specific instantiation of it. We started working towards the development of a satellite ground station that supports a satellite dedicated to forest fire detection.

• For this instantiation of the reference architecture, we were able to drop some parts of the reference architecture, such as “Encrypter/Decrypter”.

We have used UNAS in both cases.
DSSA-Based Reuse (cont.)

• Example
  • SGS Reference Architecture
Notes:

- The reference architecture is always reused, thus by simply working in the domain, one should have access to it.
- Since one already knows what the domain is, and its general characteristics before starting to build the repository, the repository can be built already contemplating just the facets that are relevant. Thus facilitating the selection process.
- The reference architecture has to be adapted to the specific application, that is unwanted parts removed and specific ones added.
- All changes made to the reference architecture in order to generate the specific architecture should be reflected in the corresponding lower level components.
- Since components’ interfaces are well defined by the reference architecture, integration becomes straightforward.
DSSA-Based Reuse (cont.)

• Artifact reuse process:
  • Abstraction- reference architecture representation (ADL)
    • Some representation of the other artifacts facets in the repository
  • Selection- set of domain specific facets for items in the repository
    • Immediate for the reference architecture
  • Specialization- adapt reference architecture
    • Reflect changes made to the reference architecture to the lower levels
  • Integration- use the well defined interfaces
Notes:

- As seen on the previous slide, the reuse of the reference architecture drives reuse of common domain design. Based on the current common domain design, one is able to reuse existing components, as well as generate new ones.

- The domain model and reference architecture clearly define what is considered common in the domain, thus inducing an optimal utilization of the reuse process and its repository.

- A thorough domain understanding is needed in order to determine components of all granularity levels that would constitute the optimal reusable components set. This domain understanding is captured by the domain model.

- The existence of a domain specific repository reduces several asset management problems, such as defining the various facets that ought to be available for querying.

- Domain model, reference architecture and repository are continuously evolving based on the domain evolution observed through the requirements evolution of applications in the domain.
DSSA-Based Reuse (cont.)

- Artifact involved:
  - Reference architecture, design, requirements, and code

- General characteristics:
  - Requires thorough domain understanding
  - Clear definition of what should be included in the repository
  - Domain specific repository
  - Domain model, reference architecture and repository evolution
Notes:

- With the domain requirements, domain modeling goes on, generating the domain model which describes other entities and the environment in which the application will operate.
- The domain model and the current application system requirements are used to define the current applications’ requirements.
- The domain model is also used to generate the domain reference architecture -- the software architecture for the family of application systems contained in the domain, which consists of generic components and their interconnections, and is used as a starting point for describing the software architecture of the various applications.
- Based on the reference architecture, a set of reusable components is acquired and/or developed.
- Using the specific application system specification and constraints, plus the reference architecture, one is able to generate the specific life-cycle architecture to be used. From that, one can also determine what parts of the reusable domain design currently apply, and then reusing those along with generating the missing ones, the current application system design is devised.
- The system implementation is done based on the current application system design, which serves as a guideline for selecting and reusing the applicable reusable components from the library.
- Based on the domain requirements evolution observed on the generation of new systems, the domain model is evolved, causing evolution of the reference architecture, and acquisition and/or generation of new reusable components.
DSSA-Based Reuse

- Overview of the DSSA Lifecycle

Notes:

- Based on the very high degree of automation of the development process, great leverage can be drawn from this approach.
- The fixed code part generated can capture standards that are relevant to the application domain.
- The artifacts involved have been developed for reuse, and have already been tested.
- Complex concepts and optimization can be developed just once, and then used over and over again. This takes care of many of the critical issues involved in a product line, such as high reduction of risks, cycle time and costs involved.
- On the cases when a complete application system is output, the developer needs not understand any of the generated output. He/she may simply use it as a black box, not even needing to worry about integration issues.
- Since the domain coverage has to be very specific, it is extremely hard to find application generators for most domains.
- It is very hard to build application generators that contain the necessary functionality and performance to cover some desired range of applications.
Application Generators (cont.)

• Some benefits
  • Very high degree of automation
  • Can capture standards
  • Artifacts have been developed for reuse, and have already been tested
  • Provides users with implementations of complex concepts and optimizations
  • Black box output

• Some drawbacks
  • Hard to find appropriate application generators
  • Difficult of building application generators
Notes:

- The output code looks very complex, but the user need not understand it, just compile it and run it.
**Application Generators (cont.)**

**output:**

```c
#include "stdio.h"
#define U(x) x
#define NLSTATE yyprevious=YYNEWLINE
#define BEGIN yybgin = yysvec + 1 +
...
#include "globals.h"
```

```c
struct node *hashtab[HASHSIZE]; /* ptr table */
char *str_seg[MAX_STR];
...
#define YYNEWLINE 10
```

```c
extern int yyprevious;
yyin = fp_t;
while((nstr = yylook()) >= 0)
yyfussy: switch(nstr){
  case 0:
```
Notes:

- Lex is an application generator that given inputs in a very specific format actually generates a C code scanner for the corresponding set of rules entered.
Application Generators (cont.)

- **Example**

- **Lex**

  input:

  ```
  %{  
  #include "globals.h"

  struct node *hashtab[HASHSIZE]; /* ptr table */
  char *str_seg[MAX_STR];
  int str_num = 1;
  %{  

  extern FILE *fp_t;
  void ignore_comment();
  yyin = fp_t;

  \[ \t\]  
  \n  linenum++;  
  /*"*/
  ignore_comment();

  `&` 
  return AND;
  return UNKNOWN;

  %}

  /* Ignores the comment maintaining linenum correct. */
  void ignore_comment()  
  ```


Notes:
- Since application generators need only receive information about some of the details that must be filled out, while already knowing all of the common parts, the format of abstraction used depends heavily on the specific domain.
- Even though the formats vary depending on the domain, abstractions are generally some form of high level description of the variable part.
- The selection process involves choosing the application generator that does cover the specific application domain, and includes the current requirements.
- If more than one application generator exists that does meet the coverage requirements, then the choice should be done based on familiarity with the input format, reliability history, and so on.
- Typically application generators will always reuse the same piece of code to implement the common parts, while the variable ones might be done by filling out the appropriate templates based on the specific input.
- Specialization by the software developer is done by providing the input specification, since that is all that is needed to generate the variable part of the application, and the fixed part is already known by the application generator. As it has been previously said, this input format and contents vary according to the domain.
- Integration is only an issue when the output of the application generator is a subsystem. In these cases, the application generator is supposed to provide a clear description on how the generate subsystem is supposed to interface with the overall context. It could be by function call and parameter passing, or message passing for example.
Application Generators (cont.)

- Artifact reuse process:
  - Abstraction- format vary by application domain
    - High level description of the variable part
  - Selection of application generators- based on application domain and current requirements coverage
    - Additional criteria may be used
  - Selection of reusable parts by the application generator- some constant common part, and various possible templates to fill out
  - Specialization- provide the input specification
  - Integration- not always an issue
    - Varies by application generator
Notes:

- The reusable artifact involved with this approach is a complete software system design, this includes the global software architecture, major subsystems within the architecture, specific data structures and algorithms.

- Application generators take as input high-level domain specific specifications and automatically generate complete executable systems. The workings of these generated systems are totally transparent to the user.

- Appropriate domains for application generators are those where many similar systems are built.

- When constructing an application generator, the common part is implemented only once and reused every time a new software system is generated. All that is missing are some of the details, that can actually be filled out based on the specification that is input.

- Trying to build an application generator for a domain that is not to mature would not be very wise, since one would have to be constantly adjusting the internal works of the generator. Another cause for this problem is that when domains are not to mature, people might not have a very clear understanding of what the best practice would be.
Application Generators

• Artifact involved:
  • Global software architecture, major subsystems within the architecture, specific data structures and algorithms.

• General characteristics:
  • From domain specific specifications generate complete executable systems
  • Appropriate domains must be able to leverage from their use
  • Allows for single implementation of the commonalities, and fills out the missing details
  • Tends to happen as domain gets very mature
Notes:
Outline

• Software Architecture Overview
• General Characteristics of Software Reuse
• Source Code Reuse
• Design and Software Architecture Reuse
• Application Generators
• DSSA Based Reuse
• Some Important References
Notes:

• High flexibility is provided by allowing for different instantiations of various components, as well as the addition and/or removal of components.

• By supporting the reuse of large-grain artifacts it permits great reuse leverage, positively affecting most of the product line critical success factors.

• Does facilitate black box reuse, since the reuser does not have to understand the internal workings of components that they determine at a high abstraction level, need not be changed.

• By providing an integrating general framework for reusable components, reuse of software architectures avoids all problems involved on the actual composition problem usually seen when trying to integrate reusable lower levels of artifacts.

• By means of a layered architectural representation, the user can analyze each layer. Whenever there is a mismatch (even if only partial) between the current architectural part and the users needs, then the user may recurse one more level. Consequently, there is a clear distinction between what has to be changed and what can be reused as a black box.

• Being unable to use tractable abstraction representations makes the repository management extremely hard.

• Inexistent methods for analyzing architectures in order to be able to determine some of the possibly desired facets, such as performance and reliability, make it very hard to characterize some of the facets that might be of interest to some domains.
Design and Software Architecture Reuse (cont.)

• Some benefits
  • High flexibility
  • Possibility of reusing large-grain artifacts
  • Black box reuse
  • Software architectures provide a general framework for composing reusable components
  • Locality of change

• Some drawbacks
  • Difficulty on abstracting a tractable representation
  • Lack of analytical methods to determine some facets
Notes:

• In the case of a compiler software architecture with the topological view above, we can plug in different scanners, parsers, semantic analyzer, and so on.

• In case we are actually copying this from a previously built compiler, we may keep in all or at least some of the previously underlying parts.
Design and Software Architecture Reuse (cont.)

• Example
  • Compiler Software Architecture

![Diagram showing the components of a compiler software architecture]

- Scanner
- Name Table
- Parser
- Symbol Table
- Semantic Analyzer
- Abstract Syntax Tree
Notes:

- Layered architectural representation allows the user to analyze and try to understand to a level of detail that he/she wants. Thus being an optimal representation choice. However, non-layered or “flat” representations may be used, but these make it harder to grasp the overall picture.
- The user must be able to search for components using the abstractions available in the repository.
- Some domain specific facets may apply, such as style used for the architectural reuse case -- e.g. layered architectures usually have a negative impact on performance.
- Independent of the style used, locality of change in an architecture is usually very well defined.
- Although some of the internal details of some components might need to be changed, it does not necessarily have to reflect on the architecture (or design).
- Since components’ interfaces are very well defined in the architecture being reused, integrating any modified piece should not cause much problem, as long as we are dealing with the same architectural style.
- When just some intra-component fragments need to be changed, the overall integration scheme remains unchanged.
- On the other hand, in case one is trying to reuse some architectural piece that was built using a style that is different from the current context, it becomes extremely hard to integrate them. Study is needed on mixing styles in order to be used as a guideline.
Design and Software Architecture Reuse (cont.)

- Artifact reuse process:
  - Abstraction- succinct description kept in repository
    - Layered vs. “flat” architecture representation
  - Selection- repository must support efficient component search
    - Allow use of domain specific facets
  - Specialization- locality of change well defined
    - Possibility of modifications while keeping the overall architecture (or design)
  - Integration- take advantage of well defined interfaces
    - Not clear when mixing architectural styles
Notes:

- Reuse of software architectures represents a significant design and implementation effort that can be reused as a whole, or partially. Whereas the reuse of designs supports for the reuse of the corresponding code.
- Can be reused stand-alone, as a lower level instantiation of a component or as new integrating part of a new overall context architecture and/or design accordingly.
- By reusing a software architecture one may also reuse its associated design and code, or keep the design making simple changes to the code. One can also keep the architecture while making minor changes to the design and its corresponding code, or make minor changes to the architecture, and let that be reflected all the way down to the code. Similarly, when reusing design one may reuse the design and its corresponding code on their entirety, reuse the design integrally and perform minor adaptations on the code, or reuse the design partially and reflect that on the corresponding code.
- The designs and software architectures being reused may be scavenged or stored in a repository. When dealing with a repository implementation it is extremely complex to come up with a succinct representation of the either design or architectural facets, as wells as to come up with a good repository management scheme.
- It is possible to have several different lower level implementations of the components that are part of the design (or architecture), each one of them having its own pros and cons. Different users may choose different kinds of implementations based on their current needs. Consequently, actually populating a repository with most of the desired lower level implementations consumes a lot of effort.
Design and Software Architecture Reuse

- Artifact involved:
  - Fragments of existing software architectures and/or designs

- General characteristics:
  - Various ways of reuse context fitting
  - Supports for reuse of several lower level artifacts
  - Designs and software architectures may be scavenged or stored in a repository
  - Considerable amount of effort involved in providing various lower level implementations
Notes:

- As artifacts are written specifically for reuse, they tend to be very parameterized, thus reducing drastically the need for manually editing (and mandatory code details understanding) by the reuser.

- By having previously tested artifacts with information on their certification levels, the reuser has a greater insight on what to expect as far as bugs are involved.

- The fact that most of the times artifacts descriptions are written in natural language, makes it sometimes hard to do a first level cut on the reusable options available, as well as makes it often hard to understand what the artifact actually does. For one approach to solving this problem, see A. M. Zaremski and J. M. Wing, “Signature Matching: A Key to Reuse”, Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering, December 1993, pp. 182-190.

- The huge size of general-purpose component libraries makes it very hard to build and search.

- As components are developed for reuse, developers try to make them as general as possible. Clearly this requires extra effort, which wouldn’t occur in case of tailor building. The same thing is true for component certification.
Source Code Reuse (cont.)

• Some benefits
  • Artifacts written specifically for reuse
  • Artifacts have already been tested, and have some certification level given to them.

• Some drawbacks
  • Artifacts descriptions are usually written in natural language
  • General-purpose component libraries are extremely large
  • Extra investment for generalization and certification.
Notes:

- In this case we are doing black box reuse of a C code for string comparison. There is no need of actually copying the code because it is stored in a library that is visible in the context.
Example

C string comparison

```c
#include <string.h>

/* Checks if a particular token is any of the keywords by doing binary search on the corresponding table. */
int check_keywords(char *checking)
{
    ...  
    if (strcmp(checking, keywords[0][i]) < 0)
        high = i - 1;
    else {
        ...  
    }
    return RESERVESIZE;  /* did not find token in the table */
}
```
Notes:

- The artifact’s abstraction is a succinct description placed in the repository by the repository implementor.

- The information included in the abstraction is domain specific. The various facets that are relevant to the domain in question ought to be represented somehow. On general-purpose repositories it becomes harder to determine the facets (other than functionality) that must be included.

- Along with the components repository there must be a method or technique that supports efficient component search. Domain specific repositories tend be a lot more efficient, since the search space is a lot smaller, the actual functionality abstraction description can be much more succinct, and more of the relevant facets can be also used.

- This kind of reusable artifacts are generally extremely parameterized, trying to prevent the reuser from having to manually edit the code. In cases the reuser must manually edit the code, this reuse technique is not as effective, since it does force the reuser to actually understand the underlying details related to the artifact that he/she is trying to reuse.

- As usually one is reusing a complete module, interfacing is done by issuing procedure calls and so on. All the issues relevant to a component’s interface should be clearly stated in the repository, to be used as a guideline for the reuser.

- Since some of these module may actually call some internal procedures and so on, some name resolution might be necessary.
Source Code Reuse (cont.)

- Artifact reuse process:
  - Abstraction- succinct description of the artifact kept in the repository
    - Information included in the abstraction (facets) is domain specific
  - Selection- the repository must support efficient component search
    - Domain specific repositories are more efficient
  - Specialization- done by means of setting parameters
  - Integration- procedure calls
    - May require name resolution
Notes:

• This involves the reuse of off the shelf source code components. These have been typically written in some high level programming language.

• The idea here was to go one level up from simple high level language reuse, and actually reuse complete high level language patterns that occur often.

• Artifacts are in general developed for reuse, thus they include general desirable reuse related characteristics.

• Artifacts are then placed over in a repository that is accessible to everyone.

• Contrasting with code and design scavenging, the reusable artifacts should require very little (if any) effort to adapt.

• Component certification requires some investments to provide for asset evaluation with respect to the certification levels used. Once this is done, certification level becomes a facet that can also be queried.
Source Code Reuse

- Artifact involved:
  - Off the shelf source code components
- General characteristics:
  - Reuse of high level language patterns
  - Reusable artifacts were developed for reuse
  - Available artifacts are stored in a repository
  - Very little or no adaptation required
  - Investments on certification
Notes:

- In ideal cases, the reuser is still very familiar with the scavenged piece, and is able to adapt it promptly.
- There is very little cycle time reduction, since locating, understanding, modifying and debugging the scavenged artifact may end up requiring more time than developing the fragment from scratch.
- This approach heavily relies on the developer’s memory capacity, and his/her ability to recognize possible matches based on whatever abstraction they remember.
- There is always a risk of the reuser trying to force fit some scavenged fragment into a current situation, simply because him/her is very familiar with the scavenged fragment. This might end up introducing a lot of new errors as well as jeopardize the final products quality.
- As no common repository exists, different developers will be able to scavenge different fragments, thus even though there may exist fragments with better quality and better fit some developers may not be able to use them.
- Success possibilities heavily depend on the scaling factor, that is a very small company that concentrates on building some kind of small packages, say accounting packages, may be very successful at reuse by scavenging. On the other hand, extremely large organizations, dealing with somewhat more complex systems (e.g. Aerospace developing satellite ground stations), scavenging becomes very inefficient.
Design And Code Scavenging (cont.)

• Some benefits
  • May require little effort on understanding and adapting
  • Very effective on small companies working on simple domains

• Some drawbacks
  • Very little cycle time reduction
  • Heavily relies on developers memory and matching capacity
  • Force fit caused by familiarity
  • Lack of common repository
  • Low success possibilities for large companies working on somewhat more complex domains
Notes:

- The abstraction of the reusable artifacts is some level of description that is remembered by the software developer. This is very informal and imprecise.
- The selection process involves the software developer noticing similarities between his current needs, and some previous experience of his/her. He/she may then proceed to simply reuse whatever he/she remembers, or may actually refer back to the pre-existing artifacts.
- The reuser specializes the code fragment being reused by manually editing it. He/she edits the code to resolve differences between what was used then, and what is needed now.
- In order to integrate scavenged code into a new context the reuser has to modify the scavenged code in order to do name resolution.
Design and Code Scavenging (cont.)

• Artifact reuse process:
  • Abstraction- some description of the artifact which is remembered by the software developer
  • Selection- mentally matching of past experience and current needs
  • Specialization- manually editing of the code fragment being reused
  • Integration- modifying the fragment for name resolution
Notes:

- This involves the reuse of fragments of existing software systems as part of a new software development.

- The idea behind code scavenging is to go one level up from simple high level language reuse, and actually reuse complete high level language patterns that they have previous knowledge of. That is, they copy over contiguous blocks of source code from existing systems that they are familiar with.

- The idea behind design scavenging is to retain the overall original design, while reusing parts of the original code. That is, a large block of code is copied, many of the internal details are deleted while the global template of the design is kept.

- Reusers copy as much as possible from analogous systems that have already been designed, implemented, and debugged, and which they are familiar with.
Design And Code Scavenging

• Artifact involved:
  • Fragments of existing software systems

• General characteristics:
  • Reuse of high level language patterns (code)
  • Reuse of code fragments and overall design characteristics (design)
  • Reusers copy from analogous systems
Notes:

• For the following analysis of several reuse categories I will be using the same outline.

• I will first describe what exactly is the artifact being reused, and some general characteristics involved.

• Then I will describe what the reuse process looks like for that particular kind of artifact. This includes a description of what the abstractions look like, how the artifact selection is performed, what is involved on the specialization, and how to integrate the artifact in the surrounding context.

• Afterwards I will show an example, and talk about some of the benefits and drawbacks of this particular reuse approach.
Outline of Reuse Assortments Presented

• Artifact involved
• General characteristics
• Artifact reuse process
  • Abstraction
  • Selection
  • Specialization
  • Integration
• Example
• Some benefits
• Some drawbacks
Notes:
Outline

• Software Architecture Overview
• General Characteristics of Software Reuse
• Source Code Reuse
• Design and Software Architecture Reuse
• Application Generators
• DSSA Based Reuse
• Some Important References
Notes:

- The algorithm above is an overall description of what goes on when an artifact is to be reused while considering the different facets involved, as well as the user’s priorities on those.
- If more than one identical match is found then these are ranked according to the user priorities on the different facets, thus allowing an optimal selection of artifacts.
Faceted Approach (cont.)

• General artifact reuse algorithm: based on various facets, considering different priorities

  search the library

  if identical matches were found then
    make a set of the identical matches
  else
    make a set of similar components

  request the user to prioritize the various facets

  for each component in the set
    compute degree of match based on facets priorities

  rank the set based on the degree of match

  select the best match from the ranked set

  modify the selected component (if needed)

  and use it
Notes:

- The algorithm above is an overall description of what goes on when an artifact is to be reused. Although it can take into account different facets, it does not deal with the possible facets priorities by the user.
- If more than one identical match is found then there is no extra guidance for the user as to which one would best suit his/her needs.
Faceted Approach (cont.)

- General artifact reuse algorithm: based on facet matching only, no priorities included

search the library
if identical match is found then
  use it
else
  make a set of similar components
  for each component in the set
  compute degree of match
  rank the set based on the degree of match
  select the best match from the ranked set
  modify the selected component and use it
Notes:

- Software artifacts are classified according to various facets. That is, several characteristics of the artifacts must be considered in order to facilitate the comparison between different artifacts, thus facilitating an optimal selection.

- Prieto-Diaz and Freeman suggested the use of functionality and environment. Functionality should be described in terms of function (operation performed), object (type of data object on which the operation is performed) and medium (larger data structure in which the data object is located). While environment is composed of system type (type of subsystem for which the component was designed), functional area (application dependent activities) and setting (application domain).

- Gacek suggested the inclusion of some few other facets of artifacts like execution time, memory space usage, and level of certification. The relevance of each of these facets on a particular selection is totally domain dependent. For example, when dealing with a student registration by phone system, execution time is certainly not a great concern, while dealing with sensor data processing this can be critical.

- Both works were developed in the code reuse context, thus they would be referring to facets of reusable code. Clearly, they can be expanded to cover other types of reusable artifacts.

- Different domains may include specific facets, and attribute them varying degrees of relevance.
Faceted Approach to Code Reuse

- Different facets used to classify artifacts
- Prieto-Diaz and Freeman:
  - Functionality: function, object and medium
  - Environment: system type, functional area, and setting
- Gacek:
  - Execution time
  - Memory space requirement
  - Level of certification
- Both were thought of in the code reuse context, but could be abstracted to other types of artifacts
Notes:

- Reuse in various categories, such as specifications, system and software architecture, design, code fragments, and so on.

- The leverage obtained through reuse clearly depends on the reuse category applied (level of granularity involved), as well as the quality of the items being reused.

- Code fragments reuse involves reusing bundles of high level programming languages constructs.

- When reusing a design, the reuser may want simply to reuse the design, nevertheless s/he may also reuse the code that was derived from the original design, or just pieces of it.

- When reusing a software architecture, the reuser may want simply to reuse the architecture, nevertheless s/he may also reuse the design and/or code that were derived from the original architecture, or just pieces of them.

- Reuse obtained through application generators is somewhat similar to the reuse achieved by high level programming languages. They translate the specifications that they receive as inputs into high level programming language code.

- As we will see later on, DSSA s support the reuse of a common architectural framework existent for the specific domain, and from that the reuse of domain specific components.
General Characteristics of Software Reuse (cont.)

- Leverage obtained through reuse is impacted by the quality and granularity of items being reused impact

- Reuse categories covered
  - Code
  - Design
  - Software architecture
  - Application generators
  - DSSA based
Notes:

• A successful example of domain scoping is the one used by UNAS, a product of TRW/Rational to support architectures based on message-passing

• Provides a set of functions that is common to most distributed systems, which are usually very complex to develop and test:
  • Interprocess communication
  • Error logging
  • Performance monitoring
  • Initialization
  • Fault isolation
  • Fault recovery
  • Reconfiguration
  • Health/status monitoring

• Runs on various hardware platforms, like SUN Sparc, HP 9000, and IBM RS600
• The components made available are written in Ada, and so is the code produced, but supports development of components in either Ada or C
Scope Of A Reuse Program (cont.)

• A successful example
Notes:

• One of the ways of evaluating the set of factors at hand is by plotting them in a Kiviat diagram as above.

• The optimal business opportunity lies on being able to score the highest on each and every one of the axis.

• If the scores are very high on some of these axis, but very low on at least one, then the risks are highly increased, thus requiring some serious risk analysis.
Scope Of A Reuse Program (cont.)

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

1- Not exhibited
2- Slightly exhibited
3- Moderately exhibited
4- Mostly exhibited
5- Fully exhibited
Notes:

- The scope of a reuse program should be defined targeting the optimum exploration of the reuse opportunities existing in the domain being explored.

- Factors to consider while determining the scope to be used:
  - Market potential: study the size of the market and the business’ potential share of it, and identify the factors relevant to the involved product line.
    - Justification: The market potential determines where good business opportunities are.
  - Commonalities and variabilities: assess what features are common across the product family, as well as the extent of variability.
    - Justification: Helps determining whether basing the product family on reuse is feasible or not.
  - Standardization in the domain: identify existing standards in the domain, features of the product family that are or could be the subject of future standards, and features that can be profitably standardized within the product family.
    - Justification: Standards can be the basis for reusable components.
  - Domain stability and maturity: identification of the system features that are likely to change over time, and how soon these modifications will be required.
    - Justification: Time is needed to recover the investment.
  - Existing domain assets: identification of existing domain assets that can be used.
    - Justification: Use of existing assets cause a higher return potential. Assets can include people, documentation and software.
Scope Of A Reuse Program

• Optimally explore the reuse opportunities of the domain at hand

• Factors to consider:
  • Market potential
  • Commonalities and variabilities
  • Standardization in the domain
  • Domain stability and maturity
  • Existing domain assets

---

4. Based on the SPC Domain Assessment Model part of the Reuse Adoption Guidebook and Course
Notes:

On the previous chart we talked about cycle time reduction as a critical success factor. Here are some others to be considered:

On the side of consumers of reusable parts:
- **Cost avoidance**: the more products in the product line, the greater the savings are going to be down the line.
- **Quality improvement**: the best the reusable assets are, the better quality we get by simply reusing them.
- **Risk reduction**: trying to reuse an artifact should try to reduce currently existing risks, never to increase them.
- **Sales income**: investments on reuse should only go as far as there is a foreseen return on investment. That is, one should always think about the market share available.
- **Component maintenance**: must be able to fix, and evolve the existing reusable artifacts.
- **Process definition**: reuse must be an integral part of the software development process, otherwise it may actually be a disincentive to reuse.
- **Training and incentives**: it is imperative that potential reusers be trained as to make optimal use of the reusable artifacts available. Cash incentives can be useful when initiating a reuse program. Research done by SPC indicates that some time after the program has been initiated, people will reuse or not independent of incentives, they have either bought in or not.

On the side of producers of reusable parts:
- **Correctly choosing domain**: when developing a reuse repository to support outside users, instead of in house, careful consideration must given to the appropriate domain choice.
- **Correctly choosing scope**: it is critical to choose a suitable scope within the domain, in order to make sure there is at least potential for receiving the appropriate return on the investment incurred by producing the reusable parts.
General Characteristics of Software Reuse (cont.)

- Other critical success factors:
  - Cost avoidance
  - Quality improvement
  - Risk reduction
  - Sales income
  - Component maintenance
  - Process definition
  - Training and incentives
  - Correctly choosing domain and scope
Some of the requirements pertaining to an effective reuse technique are:

- If reuse helps on a specific development stage, but has a negative overall effect, then it should not be applied. For instance, if by reusing a particular software architecture the development group is drawn towards a concept that they are not familiar with (for example the use of OO techniques), then the effects on the overall development may be extremely harmful.

- It is a huge draw back if one has to spend a lot of effort in order to be able to reuse a specific artifact. This extra effort might be something like trying to figure out the interface format, or trying to understand what side effects a particular artifact might have. “It must be easier to reuse artifacts then to develop them from scratch.” Otherwise reuse is just not worth it, one would be better off by simply building custom code.

- If locating an appropriate reusable artifact takes longer than building another one with the same quality level, then one should just build it from scratch.

- A basic selection criteria for reusable components is its function. Depending on the domain some other necessary characteristics may arise, such as reliability, usage environment required, timing and space concerns. For example, if we are talking about a student phone registration system, its reliability requirements are a lot less stringent than those related to a satellite ground system or a patient health monitoring system.
General Characteristics of Software Reuse (cont.)

- Requirements for an effective reuse technique:
  - Reusing artifacts must reduce the overall effort required to implement a system
  - It must be easier to reuse artifacts than to develop them from scratch
  - One must be able to locate reusable artifacts faster than build them from scratch
  - To select an artifact to reuse, one must know what it does, as well as some other domain related characteristics
Notes:

- Software reuse is using existing software artifacts when building a new software system. The artifacts involved include not only source code, but also design, architecture (system and software), test cases, documents, life cycle processes, and specifications among others.

- By reusing software artifacts not only does one have a potential of reducing the time and effort involved in building a new system, but also the quality of the system being created can be enhanced depending on the quality of the reused parts. This quality improvement has as a side effect the reduction of time and effort to maintain the software system.

- All reuse techniques involve:
  - Abstraction of artifacts: Brief description of the artifacts, highlighting the important information about the artifact, while suppressing unimportant details.
  - Selection of artifacts: Based on the artifacts abstractions users should be able to find, understand, compare and select the one that closest fulfills his/her needs.
  - Specialization of artifacts: May involve the modification of a pre-existing artifact, or simply its instantiation.
  - Integration of artifacts: Involves the understanding of the artifact's interfaces, and properly connecting it to the other artifacts.
General Characteristics of Software Reuse

- What is meant by software reuse?
  - Use of existing software artifacts when building a new software system
- Why reuse software artifacts?
  - Reduce time and effort required to build new software systems, and improve their quality
- What is involved in the reuse process?
  - Abstraction, selection, specialization, and integration of artifacts
Notes:
Outline

- Software Architecture Overview
- General Characteristics of Software Reuse
- Source Code Reuse
- Design and Software Architecture Reuse
- Application Generators
- DSSA Based Reuse
- Some Important References
Notes:

- Spiral model:
  - Concept of operations is growing, requirements are growing, and so is the software architecture. This growth should be done in compatible ways.
  - Cycle 3 - TBD: architecture is not completely defined, if something is not of high-risk, then decisions concerning it may be deferred, keeping an open option. For example, choice of the particular DBMS to be used.
  - Life-cycle architecture is the baselined architecture that is going to be used throughout the development and maintenance of the software system. Consequently, it will evolve with time, but it shouldn’t be “twisted” too much.
Software Architecture and the Development Process (cont.)

**Spiral model:**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Determination of top-level concept of operations</td>
<td>Determination of detailed concept of operations</td>
<td>Determination of IOC requirements, growth vector</td>
</tr>
<tr>
<td>System scope/ boundaries/interfaces</td>
<td>Top-level HW, SW, human requirements</td>
<td>Choice of life-cycle architecture</td>
</tr>
<tr>
<td>Small number of candidate architectures</td>
<td>Provisional choice of top-level information architecture</td>
<td>Some components of above TBD (low-risk and/or deferrable)</td>
</tr>
<tr>
<td>Top-level analysis supporting win-win satisfaction</td>
<td>More detailed analysis supporting win-win satisfaction</td>
<td>Thorough analysis supporting win-win satisfaction</td>
</tr>
</tbody>
</table>

Tasks and decisions made in the first three cycles of the spiral model.
Notes:

• Waterfall model:
  • Requirements are assumed fixed
  • Architecting stage focuses on appropriately allocating requirements to architectural components
  • Design focuses on translating requirements and architecture into low level design
  • The architecting stage thus fits between requirements and design
  • The created architecture is then used throughout design, testing/acceptance, and maintenance
  • It should be clear that the architecture is key during the maintenance stage, for the overall software system structure may evolve, thus requiring very careful considerations as not to “break” it.
Software Architecture and the Development Process (cont.)

- Waterfall model: architecting stage fits between requirements and design
Notes:

- A software architecture provides a framework for the big picture. It incorporates some of the most important decisions that should be reflected on the software system.
- Software architectures facilitate early identification of reuse opportunities, providing a great leverage.
- The baseline architecture should satisfy all functional and attributes requirements, as well as be able to accommodate anticipated directions of requirements change and growth.
- Software architectures should be considered as a basis for software maintenance in order to prevent architectural erosion and drift, two of the major causes of increasing brittleness upon system evolution. Note: Architectural erosion is due to violations of the architecture, and architectural drift is due to insensitivity about the architecture [Perry and Wolf].
Software Architecture and the Development Process

- Software architectures as a basis for design

- Detecting reuse opportunities at the architecting stage

- Software architectures as a framework for accommodating life-cycle requirements

- Software architectures as a basis for software maintenance
Notes:

Advantages:

- **Reuse support and ease of maintenance:** filters may be viewed as black boxes and hooked together at will, provided that they agree on the data to be transmitted between them. This also allows the addition of new filters, and the replacement of previously existing ones.

- **Behavior visualization:** allows the visualization of the system's behavior as a composition of the behaviors of the individual filters.

- **Concurrent and pipelined execution:** depending on the system topology, filters may be executed in parallel and/or in dataflow pipelines for high performance.

Disadvantages:

- **Components are not easily reusable in architectures built using different styles:** if something was built in a non-pipe and filter architecture one would probably have a lot of difficulty of reusing it in a pipe and filter architecture, since it could for example be expecting to receive the inputs in a single bundle, rather than incrementally.

- **Encourages batch processing:** by making the designer think of each filter as sequentially providing a complete transformation of input to output data.

- **Not good at handling interactive applications:** since there is filter independence, and possibly parallel execution, it becomes very hard to implement a plausible interaction interface.

- **Forces lowest common denominator on data transmission:** this happens in order to provide for arbitrary plugging and unplugging of filters. This can also be interpreted as the lowest performance filter may dictate the overall system performance.
Pipes and Filters (cont.)

• Advantages:
  • Reuse support and ease of maintenance
  • Behavior visualization
  • Concurrent and pipelined execution

• Disadvantages:
  • Components are not easily reusable in architectures that use different styles
  • Encourages batch processing
  • Not good at handling interactive applications
  • Forces lowest common denominator on data transmission
Notes:

• A filter reads streams of data on its inputs, processes them, and produces streams of data on its outputs.
• A pipe transmits outputs of one filter to inputs of another.
• Filters are independent entities, and have no knowledge of which filters precede or follow them.
• Filters may produce outputs before their inputs are fully consumed
• Examples:
  • Programs written in Unix shell, using Unix pipes
  • Sensor processing transformations
• Not good for:
  • Database management systems
Pipes and Filters

Diagram showing the relationship between pipes and filters in a system.
Notes:

• Even though I am only mentioning these, one should be aware of the existence of other styles, such as table driven interpreters, data abstraction and object oriented and some special cases of the styles above, e.g. client-server which is a special case of distributed processes.

• I am only going to be shortly talking about the pipes and filters style.

• For a more complete discussion on architectural styles see Garlan and Shaw.
Most Common Architectural Styles\(^3\)

- Pipes and Filters
- Layered Organization
- Distributed Processes
- Repositories
- Event-based, Implicit Invocation
- Main Program/Subroutine, Explicit Invocation

---

3. Material adapted from Garlan and Shaw
Notes:

- An architectural style defines a family of systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. [Garlan and Shaw]

- The important thing about an architectural style is that it encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their relationships. [Perry and Wolf]
Architectural Style\textsuperscript{1,2}

- Defines a family of architectures constrained by:
  - Component/connector vocabulary
  - Topology
  - Semantic constraints
- Encapsulates important decisions about architectural elements
- Emphasizes important constraints on the elements and their relationships

\textsuperscript{1} From “Architectures for Software Systems” tutorial on ACM SIGSOFT’93, by David Garlan and Mary Shaw
\textsuperscript{2} From Perry and Wolf
Notes:

In order to accommodate the different expectations of the various stakeholders, a software architecture must incorporate different, multiple views.

1. Static Topological
   - Organizes code in subsystems and subsystems decomposed into layers of abstraction.

2. Behavioral / Operational
   - Describes the state changes and scenarios in the architecture.
   - Takes care of attribute requirements: performance, availability, etc.

3. Dataflow
   - The source, transformation and destination of data within the system.

4. Computing Environment
   - Computer hardware that will host the software.
   - Implementation language, O/S interfaces, protocols and standards.

5. Process Environment
   - Is it feasible?
   - Is it affordable?
   - Can it be done in time?
   - Are the resources available? (people, offices, etc.)

Although literature usually discusses only the topological view, we believe that all of the ones above are necessary (or at least must be somehow represented).
Views of a Software Architecture

10. Static Topological

11. Behavioral / Operational

12. Dataflow

13. Computing Environment

14. Process Environment
Notes:

- A software architecture has a different meaning and use for different stakeholders, based on their concerns. These are summarized on the table above.
- The goal of a software architecture is to address each and every one of the concerns above, as to satisfy the win conditions of all the stakeholders involved.
- Actually, one can draw an analogy to building architecture. In this context we may have the following:

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Concern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>• Cost</td>
</tr>
<tr>
<td>User</td>
<td>• Where is the kitchen with respect to the dining room?</td>
</tr>
<tr>
<td>Architect / System Eng.</td>
<td>• Completeness, consistency of the architecture</td>
</tr>
<tr>
<td>Developer (building contractor)</td>
<td>• Detailed enough to support decisions as to determine what sort of foundation to use.</td>
</tr>
<tr>
<td>Maintainer</td>
<td>• What is the best place to add an extra bedroom?</td>
</tr>
</tbody>
</table>

Table 5: Building Architecture Stakeholder Concerns
Goals of a Software Architecture

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Concern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>• Schedule and budget estimation</td>
</tr>
<tr>
<td></td>
<td>• Feasibility and risk assessment</td>
</tr>
<tr>
<td></td>
<td>• Requirements traceability</td>
</tr>
<tr>
<td></td>
<td>• Progress tracking</td>
</tr>
<tr>
<td>User</td>
<td>• Consistency with requirements and usage scenarios</td>
</tr>
<tr>
<td></td>
<td>• Future requirement growth accommodation</td>
</tr>
<tr>
<td></td>
<td>• Performance, reliability, interoperability, etc.</td>
</tr>
<tr>
<td>Architect / System Eng.</td>
<td>• Requirements traceability</td>
</tr>
<tr>
<td></td>
<td>• Support of trade-off analyses</td>
</tr>
<tr>
<td></td>
<td>• Completeness, consistency of architecture</td>
</tr>
<tr>
<td>Developer</td>
<td>• Sufficient detail for design</td>
</tr>
<tr>
<td></td>
<td>• Reference for selecting / assembling components</td>
</tr>
<tr>
<td></td>
<td>• Maintain interoperability with existing systems</td>
</tr>
<tr>
<td>Maintainer</td>
<td>• Guidance on software modification</td>
</tr>
<tr>
<td></td>
<td>• Guidance on architecture evolution</td>
</tr>
<tr>
<td></td>
<td>• Maintain interoperability with existing systems</td>
</tr>
</tbody>
</table>

Table 4: Stakeholder Concerns
Notes:

Before we go into the overview of software architectures, it is very important to understand that why we are going to be talking about this. Software architecture come very early on the life-cycle of a system, thus if reuse is applied at that level it produces a great leverage.

One can find several different definitions of software architecture in the literature, all of them agree that it describes the overall system organization. The most commonly used ones are this two.

- Garlan and Shaw define software architectures as including components, connectors and configurations. Where:
  - components define the locus of computation,
  - connectors define the interactions between components, and
  - configurations define the topology of the components and connectors.

- Perry and Wolf say that a software architecture is a set of architectural elements that have a particular form, and an underlying rationale. Where:
  - elements are the processing elements, data elements, and connecting elements
  - form consists of weighted properties constraining the choice of architectural elements, and weighted relationships constraining how the different elements interact and how they are organized with respect to each other in the architecture
  - rationale captures the motivation for the choice of architectural style, the choice of elements, and the form.

- Note that these two definitions concentrate on the topological view
What is a Software Architecture?

- Describes the organization of the overall system

- Consists of *components, connectors, and configurations* [Garlan and Shaw]

- Set of architectural *elements* that have a particular *form*, and an underlying *rationale* [Perry and Wolf]

Note: Upcoming software architecture IEEE issue edited by Garlan and Perry might bring in some new ideas
Notes:

- Software Architecture Overview
  - Definition
  - Architectural Views & Styles
- General Characteristics of Reuse
  - At what stage of the development cycle should reuse be applied?
  - Defining the scope for a reuse program
  - Quick description of the faceted approach
- Various Reuse Categories
  - What is involved, and what are the benefits?
  - Constraints and facets used
  - Some drawbacks
  - Example
Outline

• Software Architecture Overview
• General Characteristics of Software Reuse
• Source Code Reuse
• Design and Software Architecture Reuse
• Application Generators
• DSSA Based Reuse
• Some Important References
Notes:
Domain Specific Software Architecture Based Reuse

Cristina Gacek

October 17, 1994
State-of-the-Art Presentations

Domain Architecture-Based Reuse

Cristina Gacek of USC presented the underlying motivation to invest on reuse based on Domain Specific Software Architectures (DSSA). She first discussed the definition of software architecture that can usually be found in the literature, also discussing what its goals are, as well as how it should be viewed. This was followed by a discussion on some of the currently accepted architectural styles, including a more detailed discussion of one of them. She then moved on to introduce what could be thought of as an initial approach to the architecting process, and how this is envisioned to fit in the software development process. This was followed by a discussion on the general characteristics of software reuse, including scoping of a reuse program, and the faceted approach. This introduced an overall description of reuse being applied at several granularity levels: source code, design and software architecture, application generators, and finally DSSA based.
State-of-the-Art Presentations

Software Reuse and Megaprogramming

Barry Boehm of USC-CSE presented an introduction to megaprogramming and some of its associated risks and pitfalls. He then discussed the megaprogramming enterprise model, and how a product line is to work. Some of the critical success factors here discussed include process models, information and software architectures, and asset library support. The PRISM and DSSA programs were used to demonstrate how information and software architectures can be used to support reuse. As an example the reference architecture for satellite ground stations, and one of its possible instances (the architecture of a fire satellite ground station) developed at the USC-CSE were shown. In the context of environment and asset library support Dr. Boehm briefly described the STARS project, and the HP software factory. He then concluded his talk by discussing what he thinks vital for achieving optimal software product lines.


Bibliography


is able to solve these conflicts.

3. Examine the cost of reusing components with respect to the [Selby88] data.

4. Develop a reuse return-on-investment model based on Tables 1 and 2.

5. Explore DSSA-based reuse using as a basis the on going USC CSE satellite ground station research.

6. Explore generalizations of reuse across satellite ground stations to reuse across other operations control centers.

7. Investigate the various infrastructure characteristics desired for reuse to be implemented at the different lifecycle stages.

8. Elaborate on the issues involved with asset management, including its interaction with DSSA-based reuse.
8. Object-Oriented Reuse

For many application areas, people find it advantageous to think in terms of objects. The success of the object-oriented approach to reuse is attributed to the ease of association between real world objects and software objects. Object-oriented languages provide a means to construct software components that are flexible and easily customizable.

Object-oriented languages use encapsulation, inheritance, and polymorphism to simplify the reuse of components in the production of software. Encapsulation is the hiding of implementation details. Inheritance is the ability to access the methods and attributes of a super-class. Polymorphism is when the same operation takes on different forms in different classes. However, as classes are reused to derive sub-classes, the class lattice grows in path length and complexity. Multiple inheritance makes resolution of methods unpredictable. It becomes progressively more difficult to understand individual components, since with inheritance, the code of a component is spread through the class lattice forcing the reuser to examine a number of classes before a component can be completely understood. Consequently, better results can be obtained if the inheritance hierarchy is kept shallow.

Focused Workshop Issues

- For what classes of applications an solutions to real world problems be expressed solely in terms of objects?
- Do inheritance and polymorphism effectively contribute to making a component reusable?
- What has the Affiliates’ experience been with complex inheritance, deep class hierarchies, and proliferation control?

9. Future work

The following research topics are current candidates for USC-CSE research. All of them are undergoing some exploration and experimentation. We would like your feedback on their relative importance to your Affiliate organization.

Please identify the three topics on this list of most importance to your organization in your issue paper. Feel free to add topics to this list.

1. Examine the architecting process and develop guidelines for reuse at the architectural level, both architecting for reuse and architecting with reuse.

2. Study how different component architectural style assumptions interface with other architectures. I.e., study how the characteristics of different architectural styles conflict, and what if any adaptation
some facets of general interest--like function, performance and certification level--most of the facets that drive the reuse decision are domain specific, while some others are application specific. Consequently, there should be one repository per domain area which contains information on the desired facets, or at least each domain should be differentiated from the others within the repository. To support reuse across several networked domain repositories, it is therefore necessary to provide users outside the domain with information about the domain’s shared assumptions.

The definition of the component certification criteria involves determining the general certification levels to be used, and means of evaluating the assets with respect to these levels. Another thing to be considered is the minimum level of certification required for the inclusion of an asset in the library.

Another issue arises during library maintenance: not only do problems in assets need to be corrected, but improvements are often demanded. One approach for handling this particular problem is to issue a notification to the affected reusers, fix the component, recertify it, and then return it to the library. However this may cause conflicts with reusers that do not want the new version, and expect to receive the promised support to the use of the previous one. USC-CSE is encountering this issue as we try to export our prototype tools to Affiliates with different versions of COTS infrastructure software.

Something else to consider is the continuous evolution of the reference architecture for the domain in question. Once changes are made to the reference architecture, some of the previously existing library components may no longer properly fit, hence should be removed from the current repository. This raises the question of how to deal with the fact that these components are still in use in some applications, and clearly might require some maintenance work. Furthermore, applications that were developed using an earlier version of the reference architecture undergo maintenance, and consequently could benefit from using assets related to that architectural version.

**Focused Workshop Issues**

- How does one deal with the control of component proliferation: version control, catalog and querying schemas, change propagation? What are good strategies for dealing with COTS proliferation?

- What measures should be taken in order to allow repository adaptation to evolution of the reference architecture?

- What is the optimal component certification schema? How about component recertification, should it include some special treatment?

- Which asset facets tend to be reuse decision drivers? Is this set of facets domain independent?
where $A$ is the life-cycle cost involved in developing a component from scratch, $B$ is the life-cycle cost involved in reusing an already existing component and $n$ is the number of times a component has been reused [Bollinger90]. It can be deduced from this equation that initially there is no cost benefit because the investment is high. But with more reuse there will be a return on investment.

**Focused Workshop Issues**

- What has been the Affiliate experience on life-cycle costs and benefits of reuse investments?
- Are the candidate cost drivers and benefit drivers list given in Tables 1 and 2 valid?
- What incentives can organizations use to stimulate reuse?
- How are the effects of reuse accounted for when estimating effort required to produce a software application?

**7. Asset Management**

There are several management issues involved with the creation and evolution of the assets reuse library. These include the management of component catalogues that include component’s relevant information, catalogue querying infrastructure support, component certification criteria, and configuration management which encompasses configuration integrity between assets and version control.

A set of component facets can be used to organize the search space for a specific component [Prieto-Diaz87]. In order to provide reusers with the best possible search mechanism, the component facets that are most important to the particular reuser should be represented somehow. We believe that whereas there are
• How does domain knowledge aid asset production, location, evaluation, and adaptation?
• How does one determine the appropriate breadth of domain across which to reuse software components?

6. Reuse Economics

Reusing software reduces software development costs. As in any system that reuses components, there are start-up costs and operating costs associated with maintaining the components. Some of these costs are in the Table 2.

Start-up costs are domain engineering, architecture determination, development of reusable components, and repository development. Domain engineering requires that expertise in the area of application be available. This should already exist within the organization otherwise thought should be given to using someone else’s components. Costs associated with domain engineering are in the capturing of domain knowledge. Part of domain engineering will identify software architectures that work well for a particular application domain. This is an important step because it will influence how reusable components will be constructed. Components have to be made available either through a component library or component generators. Each component needs to be documented to the degree that a developer can determine if it will be usable in the application under development. A component library has to be developed that efficiently serves the application builder.

Operating costs include component library operations, and component maintenance. The library will evolve in storing families of components as well as store new capability components. Maintenance of components is another cost. This involves certification of components and fixing errors in components.

Another cost of reuse is making use of the reusable assets that are available. This is more than just knowing what is available in the component library. There is a need to determine whether a fully-reused software module is appropriate to the application, and to integrate its description into the overall product description. Analysis in [Selby 1988] indicates that the reuse cost function is nonlinear in two significant ways (see Figure 2):

• It does not go through the origin. There is generally a cost of about 5% for assessing, selecting, and assimilating the reusable component.
• Small modifications generate disproportionately large costs. This is primarily due to two factors: the cost of understanding the software to be modified, and the relative cost of interface checking.

For reuse to be cost effective, the benefits must outweigh the costs. This can be expressed as

\[
Benefits = \sum_{i=1}^{n} (A_i - B_i) - Investment
\]
Focused Workshop Issues

- What are reasonable levels of reuse granularity, e.g. requirements, architectural, code?
- How does infrastructure evolution drive DSSA evolution and reuse?
5. DSSA-Based Reuse

Domain Specific Software Architectures (DSSA) can help identify components to be developed for reuse. The selection criteria for a DSSA involve identifying the expertise and technology required to solve a problem and the degree of commonality that exists between different solutions to that problem. An architecture in a specific domain will identify the domain’s conventions and shared assumptions (about timing, synchronization, data forms, coordinate systems, environment models, etc.), the required degree of generality (number of parameters that can change) and integrability (how a component fits within the context of the application). Building reusable components based on DSSA helps avoid the miscalculation of the cost-benefit relationships associated with broad-based general reuse. This is because a DSSA is focused on a narrow, well-defined class of problems. Difficulties with past reuse attempts have been in determining the amount of effort it takes to produce reusable components, estimating the number of components to reuse, and adapting components originally thought completely reusable to subtle differences in domain assumptions.

Software architectures act as an integration mechanism for assembling software reuse components. In the case of adaptive reuse, an existing architecture shows where the software can be modified. This is particularly important for software maintenance. In the case of compositional reuse, an architecture constrains the arrangement of reuse components.

Figure 1 shows an overview of the DSSA process [Balzer93]. This figure shows the influence and reliance a DSSA has on components. It shows that a thorough understanding of the problem domain, captured by the domain model, is required to determine the components to specify for reuse. Components can be generated, imported or composed. The repository of components continually evolves and new applications are constructed.

There are several outstanding issues regarding how domain knowledge can best assist in asset production, component location and evaluation, customization, and composition. Production questions include how a component is represented and then viewed. A program component could be represented as machine code, source code or as problem-oriented language statements. Several authors mention the need for various views to represent a software architecture that will contain these components [Kruchten94] [Gacek94]. Components can be stored under different schemas which need to support evaluation of the component for use in the software architecture. Components can be customized by parameterization that is either manually or automatically performed. Glue code may be required to get components to integrate with the rest of the system.

As we make use of supporting infrastructure technology to represent and deal with software architectures, it is of great relevance that we be able to detect how infrastructure evolution affects DSSA’s and reuse.
Table 3.

<table>
<thead>
<tr>
<th>Critical Success Factors (cost)</th>
<th>Major Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Domain engineering</td>
<td>• Breadth and maturity of domain</td>
</tr>
<tr>
<td>• Architecture determination</td>
<td>• Architecture maturity; diversity of legacy or COTS software</td>
</tr>
<tr>
<td>• Reusable component development</td>
<td>• Breadth of reuse; other cost model factors</td>
</tr>
<tr>
<td>• Legacy software reengineering</td>
<td>• Legacy software structure, understandability</td>
</tr>
<tr>
<td>• Process redefinition</td>
<td>• Process maturity; process diversity</td>
</tr>
<tr>
<td>• Training, teambuilding</td>
<td>• Novelty of architecture approach; number of trainees</td>
</tr>
<tr>
<td>• Repository development, operations</td>
<td>• Number of components, sites, users</td>
</tr>
<tr>
<td>• Component certification</td>
<td>• Number of components; degree of certification</td>
</tr>
<tr>
<td>• Component maintenance</td>
<td>• Stability of application, technology, environment</td>
</tr>
<tr>
<td>• Marketed product costs</td>
<td>• Type of product (package, service); market size and maturity</td>
</tr>
</tbody>
</table>

Table 2: Product Line Management Critical Success Factors for Costs

<table>
<thead>
<tr>
<th>Critical Success Factors (benefits)</th>
<th>Major Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Cost avoidance</td>
<td>• Number of products in product line; degree of reuse</td>
</tr>
<tr>
<td>• Cycle time reduction</td>
<td>• Completeness of architecture solution; staff experience</td>
</tr>
<tr>
<td>• Quality</td>
<td>• Completeness of architecture solution; certification levels</td>
</tr>
<tr>
<td>• Risk reduction</td>
<td>• Completeness of architecture solution; process used</td>
</tr>
<tr>
<td>• Marketed product income</td>
<td>• Market size and share; price structure</td>
</tr>
</tbody>
</table>

Table 3: Product Line Management Critical Success Factors for Benefits

This paper is organized around the general topics that will be covered by the workshop discussion groups, therefore serving as a thought triggering mechanism.
4. Introduction

The benefits of proper software reuse are well known. Reuse can reduce the risk of project failure, shorten development time, improve software quality, and greatly increase the productivity of the individual software developer. The application of software reuse occurs at many different levels, such as specifications, system and program architectures, designs, data models, program code fragments, software packages, test cases, plans, data, documentation, and software life cycle processes.

One of the major problems involved in properly implementing reuse is the assembly of numerous software components for potential reuse without an integrating architecture. Another problem is misjudging cost-benefit relationships in choosing an applications domain. For example, by investing in a rapidly evolving domain, the reusable components that are developed for the current domain state may become obsolete. Some further problems associated with reuse are [Boehm94]:

“Additional pitfalls have involved overfocus on technical solutions, without a complementary focus on processes, incentives, education, and culture changes. Inadequate budgets for component certification and maintenance have also undermined reuse due to component usage risk and obsolescence.”

In order to avoid these problems one must consider the product line management critical success factors and their major drivers presented in Table 2 and
Copies of Issue Papers

This section presents:

• A list of the issue papers contributed by the workshop participants
• Copies of the contributed issue papers

Software Reuse Workshop


Hugh A. Heide, “Position Paper on Domain Models and Domain Analysis”

Morton A. Hirschberg, “Focused Workshop on Software Reuse -- Issue Paper”
Workshop Objectives and Context

The set of charts that follow were presented by Barry Boehm on October 18, 1994 before the Affiliates’ presentations. This presentation by Dr. Boehm defined the context and objectives of the workshop.

In this October 18 presentation, an overview of the workshop was presented including its objectives. The first objective of the workshop was to assess new directions in software reuse technology. The second objective was to provide guidelines for USC-CSE research as well as Affiliate’s activities. The last objective was to stimulate the collaboration between the Affiliates and USC-CSE in the software reuse area.

Several reuse issues were raised in the areas of DSSA-based reuse, reuse economics, asset management, and object-oriented reuse. These included determining the breadth of a reuse domain, discussing critical cost-benefit drivers, cost/risk assessment for COTS reuse, adapting assets to architecture evolution, and how inheritance and polymorphism affect reuse.
General Insights

• Stakeholder model good: need to consider asset library manager, product line manager

• Need to evaluate methods, CASE tools re ability to accommodate reuse
  • e.g. ADARTS: binding, parametrization

• Mining: role of natural language processing

• Valuable pointers to information: ASSET

• Confirmation of some current research directions
  • Re-engineering into object-oriented domain architecture
  • Architecture definition languages and modeling
    • COTS as good test case for ADL’s

• Value of Affiliates’ experience
Megaprogramming Process:

Next level

1. Determine overall system needs, priorities
2. Evaluate candidate DSSAs wrt needs, priorities
3. Resolve risks
   - Risky
   - Partly
   - Yes
4. Will existing DSSA support life-cycle needs envelope?
   - No
   - Partly
   - Yes
5. Develop custom Life-Cycle based on domain engineering opportunities
6. Extend DSSA to serve as Life-Cycle architecture
7. Use DSSA as Life-Cycle arch.
8. Evaluate / select reusable components for use in product
9. Determine incremental / evolutionary strategy
   - Develop Life-Cycle plan
10. Execute next steps in plan
    - Update future steps
11. Product obsolete wrt needs / technology?
   - Yes
   - No
Megaprogramming Process: Application Sw Engineering

Short Form
Reuse-Oriented Processes

- Architecture-driven process
- Spiral variants
  - Domain engineering
  - Business process reengineering
  - Business case analysis
- Screening matrix approach
COTS Integration Costs and COCOMO 2.0

Need to accommodate 3 types of COTS use

- COTS as part of application platform
  - use Platform Volatility, Platform Experience cost drivers
- COTS as part of tool suite
  - use Tools, Language/Tool Experience cost drivers
- COTS as a component to integrate
  - Sizing: maybe function points
  - Assessment & Assimilation: current reuse model
  - Cost Drivers
Reuse Economics

• Importance of cycle time to Affiliates
  • Adjust COCOMO 2.0 priorities
• Reuse business-case models:
  • Emphasize benefits as well as costs
• COTS cost/risk analysis
Domain Engineering: Importance of Definitions

Example: cost to develop a “domain model”

- Integrated Knowledge Base
- Performance Models
- Environment Definition
- Domain Architecture (interfaces, fixed/variable)
- Domain Taxonomy

Graph showing:
- Domain Model Elaboration
- Small, Precedented Domain
- Large, Unprecedented Domain

Cost (person-years)
1 10
Outline

• Domain Engineering
• Reuse Economics
• COTS Cost/Risk Analysis
• Reuse-Oriented Processes
• General Insights
Software Reuse Focused Workshop:

USC-CSE Highlights and New Research Directions

Barry Boehm

Oct 20, 1994
stakeholder win conditions.

The major key is in the identification of alternatives. The recognition that “continue with the present system” is always an alternative ensures that characterization of one’s current business approach is always an important step. Another important step is the identification of COTS and reuse opportunities as alternatives to be evaluated with respect to objectives and constraints. This provides the framework for the COTS integration risk assessment and management approach discussed above.

In many situations involving the evaluation of reuse-oriented alternatives, a method is needed for rapidly screening a number of reuse-oriented and other alternatives with respect to a number of objectives and constraints. The Feasibility Phase Screening Matrix, from Software Engineering Economics [Boehm, 1981; page 268], provides such a method. It has proved useful in a number of COTS and reuse option evaluation situations.

General Insights from Focused Workshop

We found the stakeholder model helpful in characterizing reuse-oriented objectives and constraints. Two reuse-intensive stakeholder viewpoints needing further emphasis are those of an asset library manager and of a software product line manager concerned with portfolio-management of his or her investments in domain engineering and reusable-asset creation, use, maintenance, and phase out.

Another important insight is that CASE tools frequently have built-in assumptions that software components are going to be developed rather than reused. ADARTS was an example that was given at the Workshop. The other Affiliate presentations at the Workshop (included in later charts) identified some additional promising research directions: software mining via natural language processing (Henson Graves) and reusable component certification aids (Deborah Cerino). They provided valuable pointers to sources of information such as ASSET, CARDS, and COSMIC (Chuck Lillie), and insights on large-scale enterprise reuse strategies (Don Reifer and Ted Davis).

We received positive feedback on some of our current research directions, such as using domain knowledge for software re-engineering (Cristina Gacek), and use of architecture definition language modeling and analysis of such issues as heterogeneous architectural style composition (Ahmed Abd-Allah). As bottom line, our Affiliates’ experience base continues to be extremely valuable in our attempts to improve the content and relevance of our research results.
above would be applied as effort adjustment factors to estimate the cost of such COTS integration.

We will be building on this baseline technical approach to incorporate related results in the literature (e.g., the NYU analyses of the cost of applications composition) and in our Affiliates’ experience (e.g., the Loral and SAIC COTS cost models; the Loral model similarly uses function points and a comparable set of cost drivers). We will also iterate the model in our upcoming 3-day Affiliates’ workshop in May on COCOMO 2.0 issues.

Once a baseline model is established, we will extend our COCOMO 2.0 database effort to collect and analyze Affiliates’ data on COTS integration costs and cost drivers. These will then be used to refine and calibrate the COCOMO 2.0 COTS integration cost estimation capability.

USC-CSE and SEI have been collaborating in such areas as software risk taxonomy definition and evaluation of software architecture definition languages. Under this project, we plan to extend this collaboration to address COTS risk and architecture issues. We would help extend the SEI Software Risk Taxonomy to cover COTS integration risks, using Table 1 as a starting point. We would use current USC-CSE, CMU (David Garlan), and SRI (Mark Moriconi) work on characterizing interactions among heterogeneous architectural styles to analyze architectural style issues in COTS integration. We will then use the results to analyze case studies, and to formulate guidelines for architecture-based COTS integration and future COTS development.

Reuse-Oriented Processes

Very few current software process models explicitly address software reuse. Some exceptions are the process models embodied in the ARPA STARS environments, and the SPC Synthesis model. Charts 19 and 20 provide a top-level and next-level Megaprogramming process model related to the STARS and SPC models. The model in chart 20 explicitly indicates stages for relevant DSSA identification, evaluation, and tailoring; reuse risk assessment and resolution; and reusable component evaluation and selection.

We plan to elaborate on charts 19 and 20 in the context of the WinWin Spiral Process. The next few charts indicate how spiral processes help address reuse-related issues. Chart 21, from the Air Force Scientific Advisory Board’s Information Architectures study, shows the spiral process used by the Internet community for evolving the Internet architecture to adapt to changing technology.

Charts 22, 23, and 24, prepared for one of our Affiliate professor-visits, show how the spiral model’s concepts provide a uniform framework for addressing such reuse-related processes as Domain Engineering, Business Process Re-engineering, and Business Case Analysis. In each case, a major step is to define system objectives and constraints -- with respect to system
COCOMO 2.0 cost estimation model to cover COTS integration cost estimation, including data collection and analysis to calibrate the model. CSE will also collaborate with SEI to analyze heterogeneous COTS architecture issues, and to extend the SEI risk taxonomy to cover the risks associates with COTS integration.

The top-level framework for the cost/risk modeling is the workshop chart 17, also summarized in Table 1 below:

**Table 1: Top-level COTS Cost and Risk Factors**

<table>
<thead>
<tr>
<th>Cost</th>
<th>Risk</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>COTS immaturity and lack of support</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>Staff inexperience with COTS and COTS integration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Incompatibility of COTS with:</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>- Application</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>- Other COTS and other reused software</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>- Computing platform, infrastructure, programming language</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>- Performance and scalability requirements</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>- Other mission attribute requirements: portability, availability, etc.</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td>Licenses, tailoring, training</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>Lack of COTS controllability and uncertain evolution compatibility</td>
</tr>
</tbody>
</table>

For the COCOMO 2.0 extension, we have identified three types of COTS usage, and COCOMO-based cost modeling methods for each.

a. If COTS packages are part of the platform on which the application is developed (e.g., network and database infrastructure), the COCOMO 2.0 Platform Volatility and Platform Experience cost drivers would be used to model the effects of these factors.

b. If COTS packages are part of the tool suite involved in developing the application (e.g., GUI builders or applications networking tools such as UNAS/SALE), the COCOMO 2.0 Language/Tool Experience and Use of Software Tools cost drivers would be used to model the effects of these factors.

c. If COTS packages are components of the delivered applications product, the components' equivalent size would be expressed in terms of function points; the reuse model would be used to estimate COTS assessment and assimilation effort, and the factors in Table 1
by defining an agreed-on set of conventions for module interface definitions. As indicated in chart 12, taken from the ARPA Domain Specific Software Architecture (DSSA) program, conventions on units, ranges, coordinate systems, etc. enable compact and consistent module interface definitions. Of course, if one wants to use one of these modules in another domain, the overall set of domain conventions needs to be available, and reconciled wherever necessary with differing conventions in the other domain. This type of reconciliation is a major component of current COTS and reusable-component integration difficulties.

Reuse Economics

Our current focus on reuse economics has been on upgrading the COCOMO model to better estimate the cost impact of reusing previous software on a future project. Feedback from Affiliates at the Workshop indicates that three other topics are at least as important (chart 13):

- Cycle time or development schedule estimation;
- Reuse business case models, covering benefits as well as costs;
- Commercial-off-the-shelf (COTS) software integration cost and risk analysis.

Chart 14 shows the long-range plan for the USC-UC Irvine COCOMO 2.0 project. We have adjusted our priorities to focus more on modeling the effects of reuse on project schedules. An initial step has been to clean up some anomalies in the COCOMO schedule compression cost driver. We also plan to investigate the effects of other cost drivers on project schedules, particularly the new COCOMO 2.0 reuse and reengineering cost drivers. Ray Madachy’s recent Ph.D. thesis, on a system dynamics model to investigate software cost-schedule-quality tradeoffs involving the use of inspections, provides another avenue for investigating cycle time effects.

Chart 15, taken from an ACM Computing Survey article, indicates the complexity of modeling information system benefits in a software reuse economics model or elsewhere. Two approaches look promising for addressing this complexity: the domain-specific approach discussed above, and a business case analysis approach discussed later (chart 18) under Reuse-Oriented Processes.

COTS Cost/Risk Analysis

We were surprised to see how rapidly the issue of COTS software integration has become a top concern almost uniformly across our commercial, aerospace, and government Affiliates. As a result of the Workshop and some previous discussions with the SEI (which has subsequently sponsored a COTS Integration workshop with MCC), we have initiated a joint effort with SEI to address COTS cost, risk, and architecture issues.

This joint project will analyze sources of COTS software composition cost and risk. CSE will then use the results to extend the USC-CSE
As with previous Focused Workshops, the interaction with our Affiliates at the Reuse Workshop has provided USC-CSE with a number of insights which have enabled us to reorient our reuse-related research in more promising and significant directions. The four primary insight areas, in which we plan to add and reorient emphasis, are:

- Domain Engineering
- Reuse Economics
- COTS Cost/Risk Analysis
- Reuse-Oriented Processes

This summary addresses each of these areas, and concludes with a general set of insights gained from the workshop.

Domain Engineering

Discussions and Affiliate experience reports at the Workshop indicated that domain-oriented approaches to reuse add significant value as compared to domain-independent approaches. They also help avoid overly complex reuse solutions.

One area we plan to work on is clarification of definitions related to domain engineering. An example issue that came up in the Workshop is illustrated by the chart 9. The cost of developing a domain model was sometimes reported as relatively small, and sometimes as quite large. The chart shows that both can be true, depending on the definitions of “domain” and “domain model”. Developing a domain taxonomy or even a domain architecture for a small, preceded domain can be done in less than a person-year. However, developing high-fidelity performance models (including the necessary environment models) or an integrated knowledge base for a large, unprecedented domain involves investments of well over 10 person-years. Thus, it is important to provide lower-level breakdowns of such general terms as “domain” and “domain model” in order to reason about their properties.

Another area involves the definition of domain architecture interfaces. As indicated on chart 10, a Module Interface Formalism ensuring the interoperability of domain software modules must represent not only the identity of interface data items and control conventions, but also a number of their attributes. Chart 11, taken from the USAF-ASD/TRW 1978 Acquisition Guidebook on Software Requirements, indicates that there are a large number of such attributes just for data interfaces.

Attaching all of these attributes to a module’s definition is important for confident reuse, but expensive. A domain-oriented approach can simplify the problem.
Background and Overview

The USC Center for Software Engineering’s Focused Workshop on Software Reuse was held on October 17-20, 1994 as part of the USC-CSE Affiliates Program. Over 30 attendees participated in this workshop, including USC faculty members and students as well as top research staff from CSE’s 19 Affiliates in academia, government, and industry.

Focused Workshop Timetable

- 8/15/94  -  CSE issue paper sent to Affiliates
- 9/94     -  Affiliates send in their issue papers
- 10/17/94 -  State of the art survey presentations
- 10/18/94 -  Affiliates’ presentations
- 10/19/94 -  Breakout Group Discussions
- 10/20/94 -  Wrapup and Closing Remarks

Knowledge Summary

The next section presents the highlights of the workshop in terms of new insights and new research directions for the CSE software reuse program. Section 3 presents the context of the workshop and its objectives based on the original issue paper sent out by CSE. Section 4 gives copies of the issue papers received.

Section 5 presents the slides of the state of the art surveys given on the first day of the workshop. Section 6 presents summaries of the Affiliate’s presentations along with complete sets of their slides. Section 7 contains summaries of the breakout group discussions on reuse economics and reusable asset management, and domain architecture-based reuse and object-oriented reuse. Finally, the Appendix contains the workshop agenda, a list of workshop participants, and a list of acronyms.
# Table of Contents

**Background and Overview**
- Focused Workshop Timetable ................................................................. 1
- Knowledge Summary ........................................................................... 1

**USC-CSE Highlights and New Research Directions**
- Domain Engineering ........................................................................... 2
- Reuse Economics ................................................................................. 3
- COTS Cost/Risk Analysis ................................................................... 3
- Reuse-Oriented Processes ................................................................ 5
- General Insights from Focused Workshop ........................................... 6

**Workshop Objectives and Context** ......................................................... 27

**Copies of Issue Papers** ........................................................................ 45

**State-of-the-Art Presentations**
- Software Reuse and Megaprogramming ............................................. 65
- Domain Architecture-Based Reuse ....................................................... 114
- Reuse Economics ............................................................................... 165

**Affiliates’ Presentations**
- Don Reifer ......................................................................................... 188
- Ted Davis ........................................................................................... 189
- Henson Graves ................................................................................... 212
- Charles Lillie ..................................................................................... 221
- Lewis Johnson ..................................................................................... 233
- Deborah Cerino .................................................................................. 254

**Discussion Group Results** ................................................................. 279

**Summary of the Domain Architecture Based Reuse Discussion**
- Introduction ....................................................................................... 280
- Definitions ......................................................................................... 280
- Affinity Groups .................................................................................. 281
- Most Critical Issues .......................................................................... 286

**Summary of the Reuse Economics Discussion**
- Introduction ....................................................................................... 288
- Most Critical Issues .......................................................................... 289
- Most Promising Opportunity and Research ....................................... 292

**Appendix**
- Agenda .............................................................................................. 296
- Participants ......................................................................................... 297

**Pointers to Useful Resources**
- Literature: ........................................................................................ 300
- Others: .............................................................................................. 302
- List of Acronyms .............................................................................. 303
KNOWLEDGE SUMMARY

USC Center for Software Engineering

Focused Workshop on

Software Reuse

Bradford Clark, Cristina Gacek and Barry Boehm, Editors
USC (Los Angeles, CA)
October 17-20, 1994