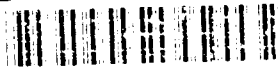


AD-A240 109



TASK: US40
CDRL: 03725
27 Aug 1991

US40 - STARS Reuse Concept of Operations, Volume I Version 0.5 - *DRAFT*

Informal Technical Data

INTO
SEP 03 1991



STARS-SC-03725/001/00

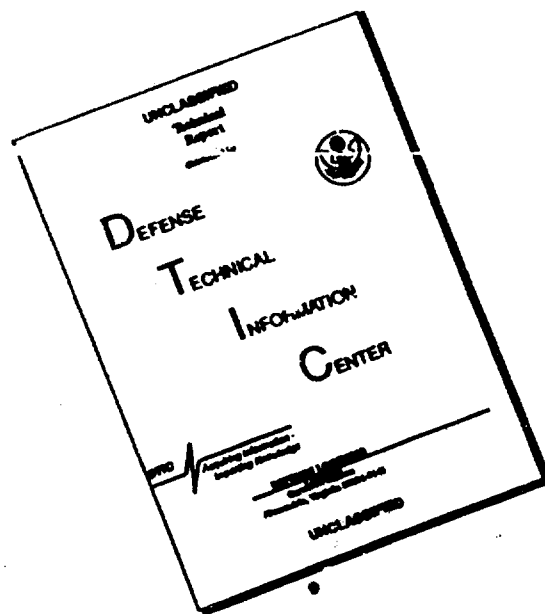
27 August 1991

[Faint, illegible text]

91-10080



DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

TASK: US40
CDRL: 03725
27 August 1991

INFORMAL TECHNICAL REPORT
For The
SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS
(STARS)

STARS Reuse Concept of Operations
Volume I
Version 0.5 DRAFT

STARS-SC-03725/001/00
Publication No. GR-7670-1259(NP)
27 August 1991

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0003

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

The Boeing Company,
Defense & Space Group,
P.O. Box 3999, MS 87-37
Seattle, WA 98124-2499

IBM,
Federal Sector Division,
800 N. Frederick Pike,
Gaithersburg, MD 20879

Unisys Defense Systems, Inc.,
Tactical Systems Division,
12010 Sunrise Valley Drive,
Reston, VA 22091



Accession For	
NIS ORANI	J
DND TAB	
Unannounced	
Justification	
By <i>per ltr</i>	
Distribution/	
Availability Codes	
Dist	Availability Special
A-1	

Distribution Statement "C"
per DoD Directive 5230.24
Distribution Authorized to U.S. Government Agencies and their Contractors:
Administrative (27 August 1991)

Data ID: STARS-SC-03725/001/00

Distribution Statement "C"
per DoD Directive 5230.24
Distribution Authorized to U.S. Government Agencies and their Contractors:
Administrative (27 August 1991)

Copyright is assigned to the U.S. Government, upon delivery thereto, in accordance with the DFAR Special Works Clause.

This document, developed under the Software Technology for Adaptable, Reliable Systems (STARS) program, is approved for release under Distribution "C" of the Scientific and Technical Information Program Classification Scheme (DoD Directive 5230.24) unless otherwise indicated. Sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA) under contract F19628-88-D-0031, the STARS program is supported by the military services, SEI, and MITRE, with the U.S. Air Force as the executive contracting agent.

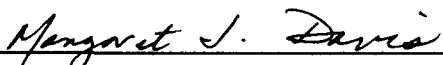
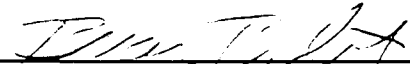
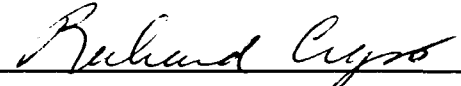
Permission to use, copy, modify, and comment on this document for purposes stated under Distribution "C" and without fee is hereby granted, provided that this notice appears in each whole or partial copy. This document retains Contractor indemnification to The Government regarding copyrights pursuant to the above referenced STARS contract. The Government disclaims all responsibility against liability, including costs and expenses for violation of proprietary rights, or copyrights arising out of the creation or use of this document.

In addition, the Government and its contractors disclaim all warranties with regard to this document, including all implied warranties of merchantability and fitness, and in no event shall the Government or its contractors be liable for any special, indirect or consequential damages or any damages whatsoever resulting from the loss of use, data, or profits, whether in action of contract, negligence or other tortious action, arising in connection with the use or performance of this document.

TASK: US40
CDRL: 03725
27 August 1991

INFORMAL TECHNICAL REPORT
STARS Reuse Concept of Operations
Volume I
Version 0.5 - **DRAFT**

Approvals:

 Boeing Reuse Technical Lead <i>Margaret Davis</i>	<i>Date</i>
 IBM Reuse Technical Lead <i>Brian Bulat</i>	<i>Date</i>
 Unisys Reuse Technical Lead <i>Richard Creps</i>	<i>Date</i>

(Signatures on File)

TASK: US40
CDRL: 03725
27 August 1991

INFORMAL TECHNICAL REPORT
For The
SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS
(STARS)

STARS Reuse Concept of Operations
Volume I
Version 0.5 DRAFT

STARS-SC-03725/001/00
Publication No. GR-7670-1259(NP)
27 August 1991

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0003

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

The Boeing Company,
Defense & Space Group,
P.O. Box 3999, MS 87-37
Seattle, WA 98124-2499

IBM,
Federal Sector Division,
800 N. Frederick Pike,
Gaithersburg, MD 20879

Unisys Defense Systems, Inc.,
Tactical Systems Division,
12010 Sunrise Valley Drive,
Reston, VA 22091

Contents

1	Introduction	1
1.1	Purpose/Context	1
1.2	Applicability	1
1.3	Scope	2
1.4	Document Context	2
1.4.1	Relationship to other STARS products	2
1.4.2	Document Organization	3
2	References	4
3	STARS Reuse Vision, Mission and Strategy	5
4	STARS Reuse Process Framework	9
4.1	Reuse Planning	12
4.1.1	Reuse Strategy Development	12
4.1.2	Incorporation of Reuse Into the Project Process	14
4.1.3	Process Measurement and Evolution	15
4.2	Asset Creation	15
4.2.1	Domain Analysis and Modeling	16
4.2.2	Software Architecture Development	19
4.2.3	Software Component Development	19
4.2.4	Application Generator Development	20
4.2.5	Asset Evolution	21
4.3	Asset Management	21
4.3.1	Asset Acquisition	21
4.3.2	Asset Acceptance	22
4.3.3	Asset Classification	23
4.3.4	Asset Cataloging	23
4.3.5	Asset Certification	24
4.3.6	Library and Asset Metrics Collection	25
4.3.7	Library Administration and Operation	26
4.3.8	Asset Maintenance and Enhancement	28
4.4	Asset Utilization	28
4.4.1	System Composition	29
4.4.2	System Generation	34
4.4.3	Feedback to Reuse Planning, Asset Creation, and Asset Management	38
5	Integrating Views of the Framework	40
5.1	Seamless Library Interoperability	40
5.1.1	Levels of Library Interoperability	41
5.1.2	Interoperability between Libraries and SEEs	42
5.2	Reuse-based Software Life Cycle Models	43
5.3	Technology Support for Reuse Processes	45
5.3.1	Reengineering Technology	45
5.3.2	Construction Technologies	46
A	Glossary	48

List of Figures

1	STARS Reuse Process Framework	10
2	Methods Supporting Reuse	44
3	Types of Domains	50

Prologue

This is version 0.5 of the STARS Reuse Concept of Operations (CONOPS). Version 1.0 of the document will consist of two volumes, the first containing the basic STARS reuse concepts and high level definitions of reuse processes. The second volume will contain more detailed definitions of the reuse processes identified in Volume I. This draft includes only Volume I.

The CONOPS is intended to be a living document. It will be revised and released on an annual basis to reflect the lessons learned in the implementation and application of the CONOPS. Revisions will also reflect the input and feedback from internal STARS and external reviewers.

The authors recognize the roughness of the transitions within the document and the need for further definition of central concepts, as well as their consistent use. In particular, we recognize the need for a thorough definition and presentation of the terms domain model and software architecture, in the reuse context.

We solicit review and comment as input to revisions leading to version 1.0, which will be delivered in January 1992.

1 Introduction

This document was jointly developed through the efforts of a STARS working group consisting of members from each of the STARS' prime contractors, the Software Engineering Institute (SEI), and the MITRE Corporation. The cooperative effort was supported by numerous meetings, conference calls, and exchange of text through electronic mail and the AFS wide-area network file system¹.

1.1 Purpose/Context

The purpose of this document is to articulate STARS concepts and expectations for reuse with respect to system and software development by:

- elaborating on the STARS reuse vision;
- stating STARS goals for reuse;
- defining a framework for definition of reuse processes;
- identifying low level reuse processes that STARS may provide as process building blocks (precise, composable process definitions) in the context of the reuse process framework and specific life cycles;
- establishing a common terminology for reuse;
- addressing the impact and opportunities for use of distributed, heterogeneous asset libraries as a reuse-enabling technology; and,
- providing a context for understanding STARS reuse plans and products.

Since we believe that there is no one "right" software development process that is applicable to all organizations, applications, projects, or methodologies, this document does NOT.

- provide a concept of operations for a total software development process;
- provide a concept of operations for a specific organization; or
- prescribe "the" way to do reuse.

1.2 Applicability

The concept of operations described in this document is generic with respect to its application by specific organizations, within specific methodologies or approaches, or as supported by a specific software engineering environment (SEE).

¹AFS is a product of Transarc Corporation.

We expect that this document will be used by those technologists who create, monitor, administer, and modify systems and software development and maintenance processes. (For clarity and consistency with concepts in the STARS Process Operational Concepts Document, we will refer to these individuals as process engineers.) Volume I of this document serves as a guide as to how reuse concepts might be applied throughout a life cycle process. Further, Volume II of this document will describe in some detail various reuse concepts and processes and the impact of alternative approaches and techniques that implement these various concepts. Volume II will be provided to assist a process engineer in selecting reuse concepts and processes that are appropriate for a particular project, application, or organization.

We also expect that this document will be of interest to:

- software program managers in understanding how reuse may affect the development process and be incorporated into project planning;
- acquisition planners who plan acquisition strategies and prepare request for proposal (RFP) packages;
- acquisition policy makers who are seeking to better understand how to foster reuse; and
- process engineers developing reuse process building blocks.

1.3 Scope

The scope of this document is limited to providing a framework for understanding the technical issues involved in integrating reuse throughout a system or software life cycle process. STARS will be providing process building blocks for some of the reuse processes described in this document.

It should also be noted that legal, business, and acquisition aspects of reuse are outside the scope of this document.

1.4 Document Context

1.4.1 Relationship to other STARS products

While this document can be read and studied independently of other STARS documents, it is closely related to the STARS Process Operational Concept Document (POCD) and the Reuse Asset Library Open Architecture Framework (ALOAF) document. The POCD addresses integrating software process with the development environment and shows how process building blocks such as reuse processes may be composed into and applied within broader process contexts. Thus, this document supplements the POCD with greater detail for reuse aspects and processes. The ALOAF provides requirements and a framework for the technical support that reuse libraries and tools may provide for seamless interoperation and data interchange as described in section 5.1.

1.4.2 Document Organization

This is Volume I of a two volume set. Section 1 provides introductory material that describes the boundaries for the remainder of the document and that gives some context with respect to the STARS program. Section 2 lists all documents that are directly referenced in this volume. Section 3 provides the STARS expectations for reuse with respect to the current state of the practice and STARS goals. Section 4 describes the reuse process framework. Section 5 discusses various views of reuse with respect to the reuse process framework. There is a glossary of terms in Appendix A.

2 References

- [BB91] J. Bladen and S. Blake. Ada Semantics Interface Specification. In *Proceedings of Tri-Ada '91*, New York, NY, October 1991. Association for Computing Machinery.
- [HCKP89] Robert R. Holibaugh, Sholom G. Cohen, Kyo C. Kang, and Spencer Peterson. Reuse: Where to Begin and Why. In *Proceedings of Tri-Ada '89*, pages 266-277, New York, NY, October 1989. Association for Computing Machinery.
- [JHD+90] A. Jaworshi, F. Hills, T. Durek, S. Faulk, and J. Gaffney. A Domain Analysis Process. Technical Report DOMAIN_ANALYSIS-90001-N, Software Productivity Consortium, SPC Building, 2214 Rock Hill Road, Herndon VA, 22070, January 1990.
- [Par79] D. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128-138, March 1979.
- [Ree91] Naval Surface Warfare Center. *Second Annual Systems Reengineering Workshop*, Silver Spring, MD, March 1991.
- [STA90] STARS. Task QM15 Phase II Lessons Learned, 1990. STARS CDRL 1520 (Separate deliveries by Boeing, IBM, Unisys).
- [STA91] STARS. STARS Vision, Version 0.1, May 1991. Internal Draft.

3 STARS Reuse Vision, Mission and Strategy

Reuse Vision

The reuse Concept of Operations (CONOPS) put forth in this document is based on the STARS vision stated in [STA91]. The CONOPS elaborates on the vision with respect to reuse. The high level STARS vision statement is the following:

Software-intensive system development will evolve to a process-driven, domain-specific reuse-based, technology-supported paradigm. The paradigm will support collaborative development across geographically dispersed project teams.

The concept of what it means to be *reuse-based* and how the process, technology, and domain-specific elements of the vision statement constrain the meaning are discussed in this section. We also provide interpretation of those terms from the perspective of the reuse capabilities within a STARS software engineering environment (SEE).

Being *process-driven* means that the software development is done in accordance with well defined processes that are enforced through management policies and for which, at a minimum, definition and guidance are provided in the SEE. In the long run the processes will be substantially automated and enforced by the environment.

Being *reuse-based* means that the standard approach to software-intensive system development and evolution is to derive new and modified systems principally from existing assets rather than to create them anew. This approach requires that relevant assets be available, as well as processes defining how to use the assets to produce the systems. The reuse vision therefore includes reusable assets as a central concept and features families of processes for asset creation, management and utilization. These three families, together with a family of reuse oriented planning processes, comprise the STARS Reuse Process Framework.

The reusable assets assumed to be in the asset libraries include not only the software components most commonly associated with reuse but also such additional kinds of information as the following:

- Reusable forms of other software products; e.g., requirements specifications, architectures, designs, test procedures
- Application domain knowledge; e.g., models, data dictionaries, algorithms
- Process definitions; e.g., for managing asset libraries, for developing application systems
- Rationale; e.g., for the inclusion of features, services, objects, and/or algorithms in a system; for the selection of one architecture or design over another.

Being *domain-specific* means that the reusable assets, the development processes, and the supporting technology are appropriate to, perhaps tailored for, the domain in which the software is being developed. STARS has selected a strategy of domain-specific reuse because we believe that is how the greatest leverage will be obtained. The domains discussed in the STARS vision document are

application domains. Application domains are generally thought of as broad, for example C³I, and as being comprised of subdomains. These subdomains may be unique to the application or common across several applications. We believe that the same reuse concepts and the same generic processes and technology apply to domains of various types and levels. The STARS vision emphasizes that a domain-specific software architecture with standard interfaces is a key aspect of the reuse paradigm.

The nature of the domain-specific assets available to be reused in the production of a new system depends in part on the maturity of the application domain and in part on the prior investment in the generation of assets. As a domain matures there is greater experience with and understanding of it and an increasing number of systems from which to draw reusable information and artifacts. If there is investment in developing assets within the domain and in maintaining, refining, and extending them based on experience with their use, then the effectiveness of the assets will increase, as will knowledge of how to use them. The STARS reuse vision includes the maintenance and improvement of assets based on feedback from their use.

Being *technology supported* means that there is substantial automated support for the reuse processes. Further, the reusable assets and the support tools are integrated in a SEE.

Doing *collaborative development* across geographically dispersed project teams means that reusable assets can be shared among libraries that are geographically distributed and hosted on heterogeneous platforms. The STARS concept is that a user will have seamless access to assets in multiple, heterogeneous libraries. The vision is that a user can use a single interface to interact with all libraries, unaware of whether or not an asset comes from a local or remote library and of the particulars of the user interface or of the data model associated with the originating library.

The principal benefit of achieving the vision is improved predictability and quality in software-intensive system development and maintenance. Realization of the vision will mean that over time the amount and quality of reusable resources will increase while the amount of new development and risk decreases.

Current Practice

Currently, software development for DoD systems is not predominantly reuse-based. Many cultural, legal, contractual, and technical reasons account for the low level of reuse on DoD systems. When reuse does occur, it is likely to be done through individual initiative, rather than in response to a deliberate plan and well defined processes. Reuse is likely to involve design and code rather than complete sets of requirements, design, code, tests, etc. The reuse occurs within a single organization, often between similar projects. Significant modification of the reused material may be needed because it was not designed for reuse.

However, in spite of the perceived barriers to reuse, there is some movement towards reuse-based development. The government has supported the research and development of reuse technology, including the Common Ada Missile Packages (CAMP) program to develop missile components and associated tools; the Reusable Ada Avionics Software Packages (RAASP) program; the Reusable Ada Packages for Information Systems Development (RAPID) and STARS library systems; the Domain Specific Software Architecture program; etc. DoD organizations such as the Joint Inte-

grated Avionics Working Group (JIAWG), the Strategic Defense Initiative Office (SDIO), and the Army Communications and Electronics Command (CECOM) and Information Systems Command have reuse initiatives. Individual companies have begun to formalize reuse and to develop reusable components for competitive advantage (e.g., TRW's Network Architecture Services) or for sale (e.g., EVB's GRACE components). These efforts, and others, indicate that some steps are being taken to move the government and industry towards reuse-based capabilities. The STARS program is building on the results of these efforts and undertaking additional initiatives to make the realization of the vision possible.

Reuse Mission and Strategy

The overall mission of STARS in the reuse area is to accelerate the shift within DOD and industry to a reuse-based domain-specific software development paradigm as described previously. The STARS strategy for effecting the acceleration of the shift to reuse-based software engineering is to:

- Demonstrate the benefits of domain-specific reuse in a familiar context for DOD applications,
- Support the transition from the current paradigm in such a way as to reduce risks in DOD's evolution to domain-specific reuse-based development, and
- Ensure that basic reuse support capabilities, both processes and technologies, are available and validated for use.

We will demonstrate the benefits of domain-specific reuse by first describing the paradigm that we envision. This document is an initial, high level description of that paradigm. The application of the paradigm will be demonstrated by actual DoD system projects in the 1993 - 1995 timeframe, and before then through trial use by STARS Affiliates.

Stars will support the transition to domain-specific reuse-based operations by providing guidance about how to implement the paradigm and lessons learned from reuse projects that have undertaken reuse in a similar manner. Examples of reuse guidance are further elaboration of this CONOPS, detailed definitions of the processes that are central to reuse, a sample software development plan using a reuse-based model, and a reuse adoption handbook.

STARS will ensure that reuse processes are available by first identifying a set of generic reuse related processes. STARS will seek and evaluate existing definitions of reuse processes; it will also develop needed process definitions. STARS will further embed the definitions in the SEE in order to facilitate their application, measurement and continuous improvement. The processes will be applied and validated internally and in the demonstration projects mentioned above.

STARS will ensure that reuse support technology is available by identifying the requirements for technology to support the reuse processes. We will then determine whether there are commercial or prototype products that meet the requirements. When there are technology products available, STARS will evaluate and integrate them into a SEE. We will determine and define how they can be used individually and together to support the reuse process. Where there are no capabilities available, STARS will attempt to stimulate the development of appropriate capabilities. This may occur through the prototyping and feasibility demonstration of needed capabilities. It may

also occur through the convening of government and industry organizations to develop proposed standards for the technology.

Implementation of these three strategies by the STARS program will accelerate the paradigm shift by:

- Showing why the DoD needs to adopt reuse-based approaches,
- Providing guidance on how to evolve to those approaches, and
- Making sure that appropriate reuse support capabilities are known and available when needed.

4 STARS Reuse Process Framework

STARS has identified functions and processes supporting reuse in the context of software-intensive system development and maintenance. Further, these reuse supporting activities have been organized into a process framework containing four families of processes. The names of these families emphasize the primary purpose of each. The reuse process families (see Figure 1) are:

- reuse planning;
- asset creation;
- asset management; and,
- asset utilization.

The families of the reuse process framework can be decomposed further to identify processes and functions focusing on different aspects of each family's purpose. Individual organizations may use different decompositions of these families to suit their goals and business strategies. However, the decomposition that is used in the remainder of this section is:

- reuse planning;
 - reuse strategy development,
 - incorporation of reuse into the project process,
 - process measurement and evolution,
- asset creation;
 - domain analysis and modeling,
 - software architecture development,
 - software component development,
 - application generator development,
 - asset evolution,
- asset management;
 - asset acquisition,
 - asset acceptance,
 - asset classification,
 - asset cataloging,
 - asset certification,
 - library and asset metrics collection,
 - library administration and operation,
 - asset maintenance and enhancement,
- asset utilization;

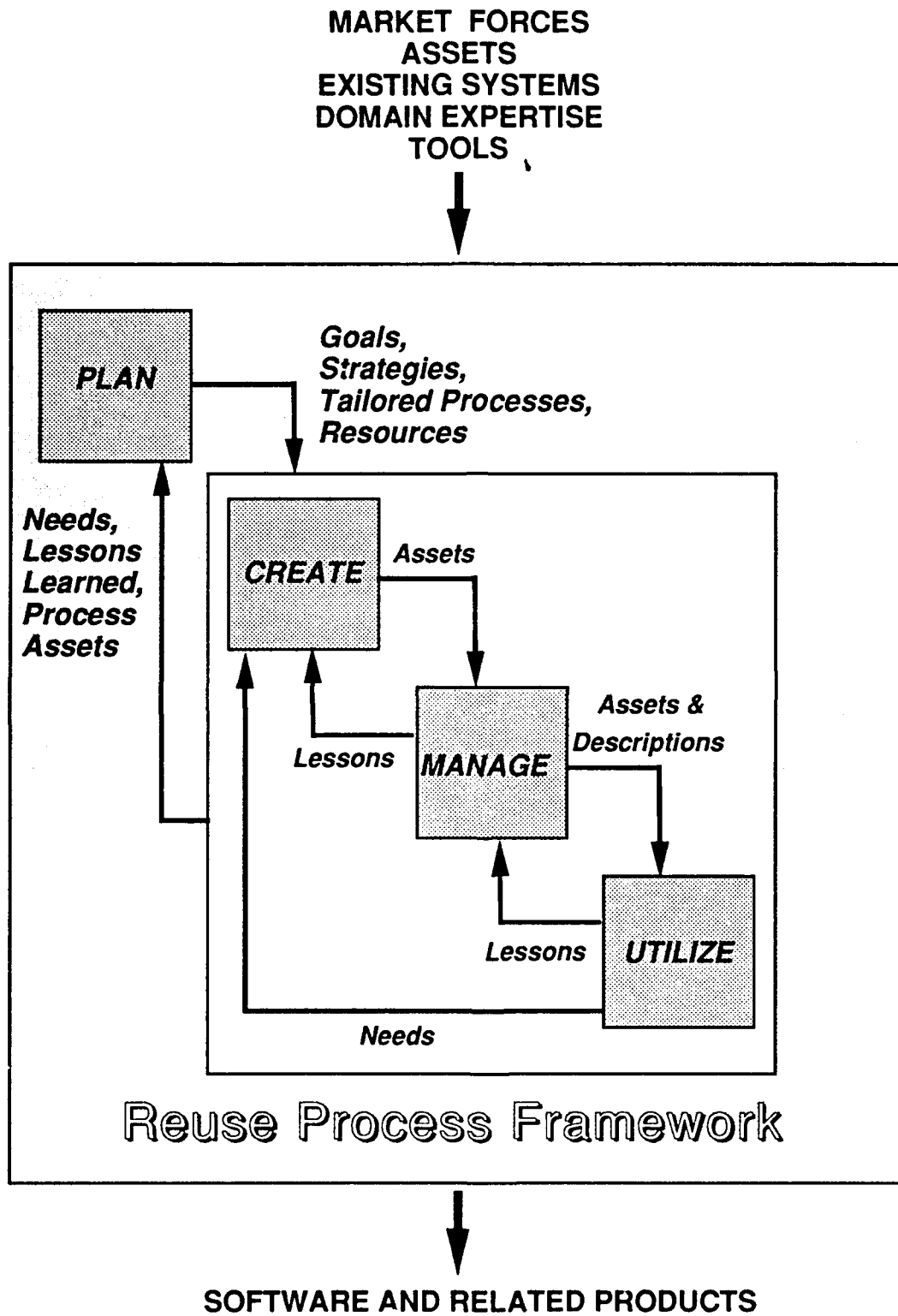


Figure 1: STARS Reuse Process Framework

- system composition,
- system generation,
- asset identification,
- asset understanding, evaluation, and selection, and
- asset tailoring and integration.

The arrows in Figure 1 represent the extensive information flow, influence, and feedback among the four process families. In general, the arrows represent the flow of decisions, constraints, experience lessons, and assets.

As the figure shows, inputs to the reuse process framework are market forces, existing assets, systems, domain expertise, and tools. A market force is defined as requirements or needs of any intended customer.

Software systems, software architectures, software components, asset libraries, experience reports, domain analysis results, and domain models are examples of the output software and related products shown in the figure.

The results of the *asset planning* processes feed separately into the *asset creation*, *asset management*, and *asset utilization* process families. Planning processes set goals and strategies, select and effect the tailoring of processes consistent with the goals and strategies, and identify and allocate existing resources. The *asset creation* process family produces software and software related assets. The *asset management* process family evaluates, describes, and organizes the assets provided by the *asset creation* process family. The *asset utilization* process family accesses the organized assets to construct software-intensive systems.

Lessons learned regarding the usage, applicability, quality, and reusability of assets are feedback from the *asset utilization* processes to the *asset management* processes. Lessons learned regarding missing assets or possible asset generalizations are feedback from the *asset utilization* processes into the *asset creation* processes. Lessons learned regarding asset quality and description are feedback from the *asset management* processes to the *asset creation* processes. Needs for new assets; lessons learned regarding process usage, applicability, and quality; and new process assets are feedback from the *asset creation*, *asset management*, and *asset utilization* processes into the *asset planning* processes.

Once an organization or project has identified the factors that constrain its planning and selection of reuse strategies and approaches, the flows shown in the diagram of the reuse process framework and the decomposition of the process families can be used to guide reuse process-related decisions.

Using the Reuse Process Framework

Historically, organizations have based their software development plans on methodology, technique, or tool selections made to implement an idealized project life cycle rather than on process building block selections. Indeed, software development has mostly been considered as one gigantic waterfall life cycle divided into major phases encompassing system conception to demise. In contrast, STARS is promoting the concept that there are multiple, valid modern software life cycle models appropriate for different organizational goals, strategies, and strengths. That is, STARS is generalizing the

concept of life cycle model from a strategy for software *system* development to strategies for software *product* development, where product includes components, interface and protocol standards, architectures, domain models, application generators, and systems.

As opposed to modeling and planning a development strategy around major activities and tools, the reuse process framework supports the notion of composing a life cycle model from process building blocks. We believe the benefits of this approach to be:

- The ability to compose multiple, reuse-based life cycle models with different goals guided by the reuse process framework.
- Easier implementation and tailoring of life cycle models in support of individual domains, organizations, and engineers.
- Easier management, measurement, monitoring, and improvement, of life cycle model implementations and improvement in life cycle models.
- Identification of the similarities in appropriate methods, techniques, and tools supporting various life cycle models and processes.

These benefits accrue because process building blocks are well-defined, may have formal representations, have definite begin and end points, have definite start and stop criteria, span a shorter time duration than life cycle phases, and can be customized to available tools and environment support.

4.1 Reuse Planning

An important function of the planning activity in Figure 1 is to define a reuse strategy and plan for its implementation within the organization that is undertaking a reuse program. A second function is to implement the strategy in plans and processes for a specific project. A related function is to measure and evolve the process for executing the plans. Many of the planning activities and products discussed here are appropriate at both the organizational and specific project levels.

4.1.1 Reuse Strategy Development

A reuse strategy is needed to guide the asset creation, management, and utilization processes. The activities required to define the strategy will depend on the nature of the organization, e.g., whether it is a company seeking to market reusable components or develop systems based on them, a DoD Program Executive Officer establishing a reuse program for a given domain, a Program Manager developing a specific system, or a maintenance organization. The strategy will be influenced by the organization's goals and top level reuse policy. The reuse strategy may define processes that identify, evaluate and select domains for reuse; define a set of methods for asset creation that are compatible with the methods for asset utilization; create plans for asset creation, management, and utilization; and define goals to measure the effectiveness of reuse. A software reuse strategy may include but is not limited to the following:

- domain selection method,

- asset creation plan,
- asset management plan,
- asset utilization plan, and
- process and product improvement plan.

Domain Selection Method

When the domain in which a reuse program is to be established is not obvious, the selection of a domain is an important early activity. Selection is a two step process where domains are identified using established criteria, and then selected after several evaluation activities are completed. Criteria for identification are that a domain:

- be well-understood and include codified experience that can predict technology and provide domain expertise [HCKP89],
- be based on predictable technology that will not make the reusable assets obsolete before the investment in their development can be recovered [HCKP89, JHD⁺90], and
- have expertise available to support domain analysis [HCKP89, JHD⁺90] and asset creation.

After candidate domains have been identified, the domain is selected after evaluating several additional factors. Evaluation of the market for systems in the domain, the readiness of the organization to pursue reuse in the domain, and economic analysis of the cost vs. the benefits are among the additional activities involved in selection of a domain [JHD⁺90].

Asset Creation Plan

The asset creation plan defines the processes, products, metrics, and technology to be used for asset creation. The technology, methods, and tools for asset creation and asset utilization are coordinated to maximize the benefits from reusable assets. Once the methods have been selected, the plan for analyzing the domain, creating a reusable software architecture, and creating reusable components is generated. This plan defines the processes and products for asset creation. These will vary as a function of the design approach, (e.g., functional or object oriented), the specific requirements of the domain (e.g., hard real-time deadlines), and the reuse approach (e.g., composition or generation). The processes may also depend on the maturity of the domain, whether there are legacy systems to be studied, and whether relevant reusable assets exist. Metrics addressing the flexibility, reliability, and modularity of the assets are also appropriate to the plan. The plan may also identify the technique that will be used to classify the assets that are created.

Asset Management Plan

An asset management plan defines the way in which assets will be acquired, made available for use, and maintained. It establishes policy and the overall technical and administrative approaches.

This plan may address the following:

- Definition of the types of assets to be stored,
- Data to be collected and stored as part of the asset description,
- Tools and taxonomy for storage and retrieval of assets,
- Criteria for accepting assets for storage and retrieval,
- Access policies and privileges by role and individual user, and
- Configuration management that addresses the evolution of assets.

The specific provisions of the plan are based on the reuse goals, the maturity of the domain, and the technology needed and available for storage and retrieval.

Asset Utilization Plan

A generic asset utilization plan identifies the reuse processes and tools suitable for utilization of the assets. A specific project asset utilization plan will define the reuse processes to be used for building a system from the reusable assets. These processes depend on the technology that is used to create the assets, the maturity of the organization, and the tools available to support reuse of assets. For example, the use of a code generator for a domain will be different from the manual composition of code components. The utilization plan will also identify the life cycle activities in which the reusable assets may be considered or employed. For example, if code is the only form of reusable asset available, the developer must "look ahead" in analysis and design to ensure that the code assets on hand are not excluded during and by those activities. The plan also defines the data that needs to be collected to measure the effectiveness of reuse. This data is dependent on the organization's goals and the project's reuse goals.

Process and Product Improvement Plan

A reuse strategy is derived from the goals for reuse. The goals for reuse may include improving the reuse process, evaluating the degree of reuse, improving the reliability of reusable assets, improving the reliability of systems, or increasing the productivity of the application programmer. The improvement plan defines how to check the process and products to determine which goals were or were not achieved. It also provides for feedback among the asset utilization, management and creation processes and to the planning processes. The plan further addresses how the measurements and feedback will be used to improve the process and products.

4.1.2 Incorporation of Reuse Into the Project Process

If the parent organization of a specific reuse project has produced generic plans for asset creation, utilization, management, and process improvement, then the project will adapt them to its partic-

ular situation. If no generic plans exist, then they will be developed. In either case detailed project specific plans will result.

These plans will be used, together with any other relevant policies and plans, to define the overall process to be followed by the project. It is assumed that a library of reusable software engineering process definitions exists, and that the plans are consistent with those defined processes. The reusable process definitions will be selected, tailored to the project's needs and to the SEE to be used by the project, and composed into the overall process. New process definitions may be developed to meet project needs. The resultant process may be tested in a dry run or simulation.

4.1.3 Process Measurement and Evolution

The reuse process measurement and evolution function is an integral part of the organizational and project specific plans for process and product improvement. This function receives input in the form of data captured about the asset creation, management, and utilization processes and products. It also receives lessons learned, asset requirements, process requirements, and any other form of relevant feedback from individuals involved in those processes. Feedback from the users of the software products is also input to this function. The process involves analysis of the input information; identification of problems and opportunities for improvement; development of solutions; identification of resources required to effect the solution; definition of changes to the process or products; modification of the plans and the process being followed; and measurement and analysis of the modified process.

4.2 Asset Creation

The goal of asset creation is to capture, organize and represent domain knowledge and produce reusable assets (e.g., specifications, designs, code, tests, documents, domain models, architectures, application generators). Asset creation may also capture knowledge about when, where, why, and how a particular asset is to be (re)used. That is, asset creation seeks to describe the variability in the application of assets as well as designing variability into the asset.

Asset creation is concerned with developing a family of software solutions that satisfy a range of constraints in a multidimensional problem space. Past and current software development practices often emphasize development of a point solution that satisfies exactly one set of constraints in the problem space. The difference between reuse-based development and current practice is analogous to the difference in solving a general quadratic equation $Ax^2 + Bx + C = 0$ and one particular instance $315x^2 + 4221x + 189 = 0$. Solving a more general problem provides flexibility in handling variation in the problem space. For example, the Graphics Kernel System (GKS), which was developed specifically to provide multi-language (Ada, C, Fortran, etc.) interaction with input and graphic devices from multiple vendors, is a more general solution than Plot10, which was designed specifically for plotting graphs by Fortran programmers using Tektronix graphic devices.

Asset creation also seeks to represent design rationale and domain experience gathered during development or maintenance activities. This representation is encoded in a form that can be used by persons other than the originators. With current development practices, important design and domain knowledge is infrequently recorded and maintained. This leads to situations of improper

usage, improper modification, or significant relearning when reuse or maintenance is attempted. Encoding and maintaining domain experience and design rationale helps reusers match current needs against the variability built into the solutions. Note that the variability may be accommodations of problem requirement variations as diverse in scope and breadth as "whether or not to always seek human confirmation before proceeding" is to technology advancements such as improved disk storage devices.

Since *asset* is a very broad term, the activities identified as asset creation include domain analysis, domain modeling, software architecture/design development, reverse engineering, design recovery, software component development, application generator development, and source code translation. Furthermore, the encoded domain expertise and design rationale guide the creation of software components, architectures, designs, and application generators. That is, there is considerable interaction and feedback among the members of the *asset creation* process family.

The remaining paragraphs of this section describe activities within the asset creation family:

- domain analysis and modeling;
- software architecture development;
- software component development;
- application generator development;
- asset evaluation response.

4.2.1 Domain Analysis and Modeling

The goal of domain analysis is to develop a domain model, reusable requirements, and domain variability description applicable to solution systems within the domain. Note that domain is being used here in its broadest sense, i.e., as an area of activity or knowledge. At a high level, domain analysis is a combination of:

- reverse engineering,
- knowledge extraction,
- technology and requirements forecasting, and
- modeling.

Most of these activities are human-intensive with few opportunities for computer aid and automation. The exceptions are mainly reverse engineering activities and assistance in representing the domain model.

Reverse Engineering

In order to extract expertise already encoded in legacy systems, existing software solutions may be analyzed using reverse engineering and design recovery techniques. These methods help identify the domain's traditional requirements and any common design and architectural features of existing solutions. Also, the information resulting from reverse engineering and design recovery activities can be used as inputs to and considerations for the processes of asset management (see section 4.3) and asset utilization (see section 4.4).

Reverse engineering methods extract low level design information from existing systems. These methods identify a software system's modularization, the relationships among the structural elements, declare/set/use patterns for variables, control flow within structural elements, and scoping information. This information can be used to analyze variation in existing solution systems and to identify essential solution concepts and their interrelationships. This information can be used to connect requirement choices with regard to their effect on performance, timing, sizing, and functionality of the resulting systems.

Design recovery methods extract high level design information from existing systems. These methods identify a software system's data structures and data management patterns and aid in separating solution systems' functionality into two categories: what functionality supports domain concepts and what functionality is fundamental to achieving a computer-based solution. Information in the first category supports application and vertical domain modeling goals to discover, validate, and represent concepts, functionality, and requirements from a user's or customer's perspective. Information in the second category supports horizontal domain modeling goals to discover concepts, functionality, and requirements from a systems/software developer's perspective.

For organizations concentrating on evolution of a domain or of a single or few systems, the information resulting from reverse engineering and design recovery activities can be used to guide subsequent modification and to provide a degree of continuity as humans enter and leave the organization. Guiding system modification means the ability to predict the impact of proposed changes and the ability to (re)structure in anticipation of adapting to new or improved technology. Providing continuity means expertise is not entirely lost as persons move on to other work and means a shorter learning period for new people. Historically, organizations effecting system evolution have high personnel turnover rates. These high rates persist and are perpetuated by management and acquisition practices despite the fact that 65 - 80% of system life cycle costs are attributable to system evolution. Thus, use of reverse engineering activities can be a significant factor in mitigating the cost of system evolution.

For organizations concentrating on long term production of systems for a specific (most likely somewhat mature) domain, these methods, by automating what is a tedious, time-consuming, error-prone manual analysis, allow humans to focus on comparisons among multiple existing systems. These comparisons aid in identifying commonalities and patterns of variation in the domain, and, where several versions of the same system are available for analysis, responses to changes in technology. This information is critical to development of domain-specific tools that use either composition and/or generation techniques to support implementation of solution systems. Increasing the number of systems analyzed and compared within a domain increases the likelihood that an individual domain model will have sufficient depth and breadth.

Knowledge Extraction

In order to capture domain knowledge held by humans and to validate results from reverse engineering, domain experts may be interviewed to define high-level domain abstractions and to verify the information obtained from the analysis of existing systems.

Processes to support knowledge extraction can be developed by adapting knowledge extraction techniques used by expert system developers, interviewing techniques used for systems analysis and requirements elicitation, and general methods used for in-depth interviewing in any discipline.

This is a software craft area ripe for innovation in both methods and tools to support completeness of the knowledge extraction and organization of the extracted information.

Technology and Requirements Forecasting

In order to forecast trends in technology, human experts and relevant literature may be consulted to ensure that the domain analysis captures pertinent technology variability information. Trends in technology and domain requirements are identified to ensure that the assets remain viable and that a return on the investment in asset creation continues.

If knowledge extraction is a craft, then technology trend forecasting is definitely an art. Short term forecasts of 9 months to two years may be developed with a reasonable amount of confidence because of the business cycle. Long term forecasts of more than two years are more difficult to develop with any confidence.

Modeling

The goal of the modeling activity is to synthesize information gathered from reverse engineering, knowledge extraction, and technology and requirements forecasting into a domain model. Besides documenting the important basic concepts of a domain and their interrelationships, the domain model also includes a set of problem space specifications and variability descriptions.

Prescriptive processes supporting model synthesis are almost non-existent, mainly because modeling is primarily a human creative activity. Processes supporting representation and documentation of a domain model include vocabulary formation, structured and systems analysis techniques, entity-relationship modeling, finite state modeling, petri net modeling, information modeling, data and control flow modeling, etc. Processes supporting validation of domain models include walkthroughs, expert review, consistency and completeness checking by tools supporting specific representations, and simulation.

Since domain analysis and modeling is an on-going area of research, neither standard domain characterizations nor matching of those characteristics to appropriate modeling methods, techniques, or processes is readily or publicly accessible. Establishment of such standard characterizations would facilitate selecting processes for a particular domain modeling project.

4.2.2 Software Architecture Development

The purpose of this activity is to produce an architecture that can be used to implement numerous systems for the domain as defined by the domain analysis. The process of architecture development seeks to identify a set of software components and their interactions that can support both the full and minimal set of domain services and objects [Par79]. Besides the domain model, common design and architectural features of existing solutions found during domain analysis may serve as the starting point for architecture development.

Very generalized, flexible architectures may provide features that permit implementation of a spectrum of systems ranging from those satisfying the minimal set of requirements to more intricate systems satisfying an elaborate set of requirements [Par79]. Such architectures are often organized in layers to permit consistent, easy addition of advanced capabilities. Note that a domain may have more than one valid architecture, depending upon design approach (functional decomposition, data-driven, abstract data type, declarative, object-oriented, etc.) or upon sets of mutually exclusive requirements and constraints.

Domain knowledge often represented in an architecture includes: tasking requirements, data allocation, user interface, the packaging of domain analysis requirements, and the rationale for selecting particular variations in the architecture. The roles of users and objects identified in the domain model will support the definition of the user interface; the triggers, events, and parallelism will support tasking definitions. Technology supporting traceability from the domain model to the appropriate architecture, detailed designs, and, possibly, software components is an essential feature of systems managing these assets.

4.2.3 Software Component Development

The goal of this activity is to develop reusable software components that implement the previously developed domain-specific architecture. Before this activity is undertaken, reuse planning has already evaluated whether component development is more appropriate than or complementary to application generator development or use. Note, however, that reuse-based system evolution or system integration life cycles may mix both software component development and application generator approaches, depending upon the complexity and breadth of the desired systems. Reuse planning activities will also have evaluated whether translation of code from legacy systems may also be appropriate.

It is assumed that the development processes for software components will follow good software engineering practices and principles such as separation of concerns and information hiding. This means coding is preceded by the development of detailed designs that guide coding and take advantage of desirable programming language features. For example, one goal of detailed Ada designs may have been to use Ada packages to hide details of the design while keeping it flexible.

It is also important that the design activity consider data, functions, and modularizations that may not be useful on all systems in order to build in flexibility and future growth. Both the design and coding processes should also support guidelines for reusability and software quality that were identified by the reuse planning processes.

It may also be efficient to create software assets by reengineering existing software code segments or components that encode information that is not readily accessible any other way. Examples include critical timing constraints, highly complex mathematics, or esoteric information only understood by a handful of human experts. It may also be cost-effective to use reengineering to improve the reusability of high quality software components since reusability implies a degree of quality but the reverse is not necessarily true.

In its most simplistic application, reengineering is reduced to translation of existing software from one computer language (e.g., CMS \Rightarrow Ada, Assembler \Rightarrow C) or standard (Fortran66 \Rightarrow Fortran77) to another without affecting data structures, modularization, or program control flow. More sophisticated reengineering may improve program control flow, lower code complexity, enforce coding standards, or improve reusability.

Along with coding or reengineering, these software component creation processes should (1) develop related information and (2) maintain traceability between the related information and the domain and design information. For example, a test driver and test cases should be developed and maintained as the means to validate the original or evolving software component.

4.2.4 Application Generator Development

The goal of application generator development is to provide a capability that allows a reuser or application developer to create software (sub)systems using the concepts and terms belonging to the domain. The point is to support the end user in stating "what" is desired rather than detailing "how" the desired effect is to be achieved. This "what" orientation can also be termed requirements-based. For example, a chemical engineer would find that the input for generators supporting chemical process control used control law concepts, symbols, and terminology. For generators supporting interactive construction of graphical user interfaces, an interface designer would specify menus, popups, buttons, etc.

Since the desire is to support statements of *what* rather than *how*, application generator development uses the results of domain analysis and modeling. Whether creating a generator that applies a series of transformations to a user-provided specification or creating a generator that works from user directed choices among parameters, domain analysis and modeling processes supply the vocabulary and relationships among requirement/constraint choices and valid software solutions.

Another factor that drives selection of processes to support application generator development is the technology that will be used to implement the generator. Depending upon the implementing technology, processes used may include textual language design, user interface design, graphical user interface design, meta-generator usage, application generator tools, graphical language design, expert system design, and knowledge-based techniques. In short, development of an application generator is very similar to development of any software-intensive system. This means that software engineering principles, reuse principles, good design processes, validation, and testing are all vital to application generator development.

4.2.5 Asset Evolution

The results of asset evaluations from the *asset management* and *asset utilization* process families are feedback into the *asset creation* processes. There should be explicit processes that receive and analyze these results. The feedback should be used to enhance the appropriate domain model, software architecture and components, and application generators. The feedback may also be used to improve or better tailor the processes of modeling, component and architecture creation, and application generator development to the needs of particular domains or organizations.

4.3 Asset Management

The goal of asset management is to acquire, evaluate, describe, and organize reusable assets to assure their availability to asset creation and asset utilization processes. Asset management is also responsible for asset library administration and operation.

Asset management activities include:

- asset acquisition,
- asset acceptance,
- asset classification,
- asset cataloging,
- asset certification,
- library and asset metrics collection,
- library administration and operation, and
- asset maintenance and enhancement.

4.3.1 Asset Acquisition

The goal of asset acquisition is to obtain assets from external asset libraries and other sources in support of asset creation and asset utilization activities.

External asset libraries, as well as other sources of potential assets (e.g., projects developing systems within a relevant domain, commercial or government off-the-shelf products, external individuals or organizations that voluntarily submit candidate assets, and so on), should be exploited as much as possible when populating asset libraries to meet particular domain needs. Such sources typically provide much of the "raw material" used during domain analysis and software architecture development; when a library is created based on the resultant domain model and architecture, these external sources may also supply many of the software components or application generators that are used to populate the library.

It may also be useful to acquire external assets during the asset understanding, evaluation, and selection processes, when no local asset meets selection criteria or existing assets are judged to be too expensive to modify.

When acquiring assets from an external asset library, differences in the data models between the local and external libraries must be resolved so that sufficient information about the asset can be collected to catalog it properly. The degree to which the process of acquiring assets from external libraries can be automated is directly related to the degree of heterogeneity of the local and external library data models and the level of seamless interoperability between asset libraries, as discussed in section 5.1.

To facilitate the location of useful external assets, library and domain cross-reference information may be of great interest to the asset acquirer. Having information available about the domains addressed by external libraries, and also possibly about the specific assets within the libraries, greatly eases the problem of directly accessing those assets remotely (when there is a high degree of library interoperability) or of acquiring them for local installation. In addition, such cross-reference information may support understanding of how assets already within the local library or assets that are candidates for acquisition are modeled within other libraries and used within other domains.

4.3.2 Asset Acceptance

The goal of asset acceptance is to ensure that an asset satisfies all legal and policy constraints and that sufficient information is available to catalog the asset.

The purpose of many of the library management policy constraints is to ensure the quality and suitability of acquired assets for use by asset creation and asset utilization activities. However, regardless of the perceived quality or suitability of an acquired asset, ensuring satisfaction of legal constraints is particularly important for assets retrieved from external sources. Although these external sources may be other system or domain developments within the same organization as well as public, government-supported, or commercial asset libraries, patents, copyrights, distribution rights, liability requirements, and other related issues may complicate or restrict the ability to reuse a particular asset.

Following are examples of asset information that may be required for asset acceptance:

- abstract,
- author/ownership information,
- author certificate of originality,
- copyrights/patents,
- distribution rights,
- distribution restrictions,
- liability statements for use/misuse,
- maintenance agreements,

- environmental dependencies, and
- dependencies on other assets.

4.3.3 Asset Classification

The goal of asset classification is to develop a scheme for categorizing assets on the basis of their domain-relevant characteristics. The classification scheme provides library users with an organizational framework for locating and understanding domain assets. This is distinct from the process (described under asset cataloging below) of determining where a particular asset belongs within the library classification scheme.

Classification knowledge can be represented in a variety of ways, including entity-relationship-attribute models, semantic networks, simple taxonomies, faceted schemes, and object-oriented class hierarchies. This knowledge, the collection of which typically begins during domain analysis, provides the conceptual basis for asset library organization. Thus, assets are organized and accessed via domain concepts or terms. If a generic domain architecture has been developed, then an alternate classification scheme can be used to relate assets to key architectural elements. The use of multiple classification schemes within a single library provides library users with alternative views of the domain and alternative strategies for locating domain assets.

It can be valuable for a classification scheme to differentiate among assets based on criteria other than domain functionality. This allows users to search for assets with desirable non-functional characteristics such as a *high degree of portability*, *high reuse potential*, *short execution time*, *low storage utilization*, and so on. Similarly, asset classification may also support an organization's reuse policy. For example, it may be useful to give users the ability to retrieve assets that are in the public domain or are accompanied by a maintenance agreement.

Additionally, it is often useful to establish relationships between heterogeneous classes of assets that are at different levels of abstraction or address different life cycle activities. For example, it may be desirable to relate generic code assets, their corresponding designs, and their test cases.

Asset classification and description is an iterative process. As legacy systems are examined and incorporated into the application domain, as new external assets are acquired, or as problems are identified with the existing classification scheme, the taxonomy of the asset library may need to evolve. The asset classification and description process should, therefore, support modifications to the classification scheme. Procedures should exist for dealing with assets that are improperly classified, either through error or because the classification scheme has changed. Other stimuli that may prompt asset reclassification are changes recommended by users during asset utilization and difficulties in finding particular classes of assets, as revealed by library metrics.

4.3.4 Asset Cataloging

Asset cataloging is broken down into three steps: asset categorization, asset description, and asset installation.

Asset categorization (as distinguished from the asset classification process discussed above) is the

process of determining where an asset belongs within the classification scheme. Once the appropriate place(s) in the scheme is/are found, the asset is said to be "instantiated". For example, categorization within a faceted classification scheme might involve identifying a term, or set of terms, for each facet.

Asset description is the process of creating, capturing, or adapting all the information that is needed to describe the asset in the context of the library's data model, once the asset has been categorized. This information may be validated against library standards. For example, if the name of an asset contributor is required, then the name might be compared to a validated list of contributors. Asset description might also involve identifying dependencies on and relationships to other assets.

Asset installation is the process of installing the categorized and described asset in the library system. This involves capturing the asset and its descriptive information in some kind of data base or other persistent store, and may also involve bringing the asset under configuration management control and performing other environment-specific operations.

4.3.5 Asset Certification

The ultimate goal of asset certification is to guarantee that software assets implement their requirements and that their execution will be error free in their intended environment. Practically, asset certification is a multi-stage process. Various levels of certification can be defined, each associated with successively more stringent sets of certification criteria. To reach a particular level, assets must satisfy the corresponding criteria. As each certification level is reached, an asset becomes more trusted in the sense that there is increased confidence that it meets its requirements without error.

The process of certification may be applied after asset acceptance and cataloging have occurred. Since asset certification is a multi-stage process to reach ever-increasing levels of trust, asset certification may be an on-going process that continues while an asset is available through the library system. In such cases, the certification level of the asset at any given time is clearly specified within the asset description.

Examples of criteria that may be appropriate for lower levels of certification include:

- Does the asset include requirements/specifications/code?
- Is the asset accompanied by test cases and/or test scaffolding?
- Does the asset adhere to some sanctioned set of reusability guidelines?
- Does the asset achieve metric standards for reusability, complexity, portability, etc.?
- Does the asset come with a maintenance agreement?
- Does the asset come with a documented usage history?

Examples of criteria that may be appropriate for higher levels of certification include:

- Was the asset developed using some sanctioned methodology?

- Is the asset accompanied by formal specification and verification artifacts,
- Does the asset come with documented evidence of frequent, successful reuse?
- Is the asset guaranteed for reuse by some organization?
- Are disclaimers of responsibility for the behavior of the reused asset either waived or omitted?

4.3.6 Library and Asset Metrics Collection

Library metrics are used to measure the effectiveness of library management processes, tools, and policies. Asset metrics are used to measure the characteristics and effectiveness of individual assets, such as their reusability. The goal of collecting such measurements is to improve the effectiveness of the library in supporting reuse processes within client organizations.

A general scheme for measuring and improving library effectiveness is:

1. Define an objective measure of library "success".
2. Take a baseline measurement.
3. Predict that a change to a process, tool, or policy will be an improvement.
4. Install the changed process, tool, or policy.
5. Take new measurements and compare them to the baseline measurements.
6. Evaluate whether to retain the change or revert to the old approach, based on the measurement comparison.

The specific library effectiveness measurement and improvement strategy is defined during the reuse planning process discussed in section 4.1. A library can provide library and asset metrics collection and storage capabilities to support this strategy. These capabilities can be provided in a number of ways, such as:

- *Automated*, wherein the library recognizes, records, and acts on the occurrence of relevant events (e.g., asset query, asset extraction, remote library access) without user intervention.
- *Imperative*, wherein a library user or administrator performs some action (e.g., a metrics tool invocation) to collect needed data, and then directs the library to store or process the data appropriately.
- *Interactive*, wherein the library explicitly requests relevant information (e.g., assessment of asset reusability) from library users.

In addition, some library effectiveness measurements (particularly those relating to the effectiveness of particular assets) may require proactive solicitation of information from users by library administration personnel during the asset utilization process.

4.3.7 Library Administration and Operation

The goal of library administration and operation is to assure the availability of the asset library for asset creation and asset utilization activities. Only those procedures that are specific to reuse libraries (as opposed to software engineering environments in general) are discussed below.

Library Access by Role

Access to asset library services should be restricted by library role. For example, an asset reuser should not be allowed to modify the classification scheme for the asset library.

Following are a few typical library roles:

- The *classifier* develops and maintains the classification scheme. The classifier should be knowledgeable in the domain and understand relevant library science concepts. If the asset library supports access to other asset libraries, then the classifier might maintain domain-specific cross references to those external libraries, in a manner that is understandable to local users.
- The *certifier* accepts, certifies, and catalogs assets.
- The *searcher/reuser* identifies, understands, and evaluates assets against specific requirements, with the goal of composing the assets into a target system within the domain supported by the library.
- The *reuse promoter* provides incentives for individuals and organizations to use and contribute to the library. Among other activities (see the discussion of reuse incentives below), the reuse promoter may monitor a variety of library and asset metrics (e.g., frequency of library use, extraction of particular assets, failed queries) in order to assess user satisfaction with library capabilities and identify ways in which the library can be improved.

Library Access by User

Access to the asset libraries, asset library subdomains, and individual assets may have to be restricted by user and groups of users. Users or user groups (e.g., companies) may have paid license fees allowing them to access specific assets. The library system needs to store such information in order to automate such license restrictions. Other access restrictions that a library system may automatically enforce include government security and company proprietary policies. If the library system can not automatically enforce needed access restrictions, library administration and operation procedures will have to be defined to enforce access policies in a more manual fashion.

Configuration Management

Configuration management is an important and potentially complex process within a domain-specific reuse-based life cycle. The domain model must be kept consistent with the generic archi-

ture and the corresponding set of components. The assets will typically be long-lived, with the potential for creation of multiple versions or variations of the same asset. These variations will need to be maintained concurrently, and the consistency of relationships between each particular variation of an asset and other library assets (and their variations) will also need to be maintained.

A related library service is asset subscription. Asset subscription allows users to be informed of all changes to an asset as it evolves, including identification of errors, changes in its classification, development of different variations, and changes to related metrics.

Asset Interchange

For some libraries, procedures will need to be defined to access remote asset libraries, either to directly access the assets within those libraries in a seamless manner or to acquire assets for local installation. Before such access can be done, the policies and data models of the remote libraries may need to be evaluated. If remote library policies and data models are similar to those of the local library, and the remote library supports standard library interfaces (such as the STARS ALOAF interfaces), access to the remote library may simply be a matter of maintaining network connectivity. However, as the local/remote library policies and data models diverge, the degree of interoperability between the libraries becomes progressively lower, ranging from the ability to automate the exchange of assets, to maintaining electronic catalogs (called "Yellow Pages") of external libraries and assets, to simply publishing paper catalogs and requesting that assets be sent on tangible media. As the latter point implies, procedures may need to be defined for non-network-based exchange of assets via a variety of media such as tapes, diskettes, and paper.

Library Support Procedures

A variety of operational processes may be defined to support the basic activities of the asset library:

- the conversion of assets between a variety of formats such as plain ASCII text, SGML, graphics, postscript, and paper,
- the conversion of internal data from cooperating tools into forms acceptable to the asset library mechanism so that tools can be integrated and, for example, so that graphics from design or architecture assets can be viewed,
- the integration of the asset library mechanism with a software engineering environment (SEE) and also with a set of cooperating tools, both in the context of the local SEE, and in the context of "subscriber" tools on remote platforms,
- the generation of a paper catalog of assets, for those remote libraries and users without network connections, and
- the definition, integration, and maintenance of test and metrics tools to support asset certification and asset evaluation processes.

Reuse Incentives

It is important to provide rationale and incentives for individuals and organizations to use the library and contribute resources to it (e.g., assets to populate the library and people to operate it). There are a variety of measures that can be undertaken to do this, and the specific set of measures employed can vary widely from one library or organization to another, depending on their characteristics.

One broad category of incentives is to “push” reuse by instituting a reward system, wherein rewards such as financial remuneration or organizational privileges are provided to asset contributors and/or reusers (either individuals or organizations). The objective of this approach is to encourage a spirit of reuse that will eventually thrive on its own, independent of the reward system.

Another broad category of incentives is to “pull” reuse via demonstration of effectiveness. This involves showing organizations the benefit of instituting a reuse program (and of using a reuse library) by directly demonstrating or documenting the successes and benefits of reuse experienced by other organizations.

On a lower level, an important method for incentivizing reuse through use of the asset library is to institute a quality management and improvement program spanning all library features and services. This involves monitoring user satisfaction with library features and services (e.g., library tools and classification schemes, asset certification and extraction procedures), and taking appropriate action to improve capabilities whenever satisfaction is not high.

4.3.8 Asset Maintenance and Enhancement

The goal of the asset maintenance and enhancement process is to iteratively improve the assets in the library relative to user and domain needs.

Problems identified during the asset utilization process, as well as suggestions for improvements to assets, are feedback to the asset maintenance and enhancement process. Problems relating to the fundamental classification and description of assets are also feedback to the asset creation process, because they may imply changes to the domain model. Execution errors, mismatches between specification and function, performance inadequacies, and documentation problems are corrected under the aegis of asset maintenance and enhancement.

Difficulty in reusing assets is identified via reports from reusers and via indicative metrics. These problems may be corrected by changes to the classification scheme or asset certification process, or by correcting deficiencies in library-related tools or processes. Alternatively, after evaluation, such problems may be fed back to the asset creation process for evaluation, leading to potential revision of assets.

4.4 Asset Utilization

In this section, we examine the asset utilization process family and its constituent processes that focus on the utilization of assets to develop software systems and related products. The processes

in this family are divided into the following categories:

- system composition,
- system generation,
- asset identification,
- asset understanding, evaluation, and selection, and
- asset tailoring and integration.

There are two primary methods of asset utilization, corresponding to the system composition and system generation processes listed above. As sections 4.2 and 5.3.2 point out, the asset utilization method(s) that a particular organization uses will be strongly determined by the asset creation methods that are employed. In general, through their reuse planning activities, organizations select, tailor, and evolve their asset utilization processes in concert with their asset creation and management processes; all these processes should be mutually compatible and comprise a consistent reuse strategy. Note that the two asset utilization methods are complementary and can both be employed within the same domain. For example, particularly well-understood subsystems within a domain may be amenable to generation, whereas a complete system may need to be composed from generated subsystems and other individual assets.

Asset identification, understanding/evaluation/selection, and tailoring/integration are processes subordinate to the two primary asset utilization methods and are approached differently within each method. This section will address each utilization method separately and discuss the subordinate processes within the context of each method. First, we examine the composition method.

4.4.1 System Composition

Asset-based system composition is a process in which the software engineer constructs new products (e.g., requirements, design, code, tests, documentation) from previously developed or newly generated parts. This is typically done by identifying, understanding, evaluating, and selecting appropriate generalized domain assets and tailoring and integrating them to meet specific system needs. The domain model supports this process by describing:

- the low-level domain assets (code modules, etc.) which form the raw material from which new system products will be created;
- the higher-level assets such as generic architectures which can be used as organizing frameworks for new systems in the domain; and
- the heuristics, rules of thumb, examples, rationale, and other information that can assist the engineer in constructing systems in the domain.

The processes of asset identification, asset understanding, evaluation and selection, and asset tailoring and integration in the context of system composition are described below.

Asset Identification

The identification of assets for system composition is driven both by the particular needs that apply during the current development activity, and by the degree to which reuse has already been employed in preceding activities. For example, early in the development effort, there will typically be a need to produce a set of system requirements, which can be done in a reuse-based environment by tailoring generic domain requirements to meet specific system needs. This tailoring can be done using various techniques, including generation, but when it is complete, there will be not only a new system requirements specification, but also traceability to the generic requirements that were reused, thus establishing an initial path to other elements of the domain model (and thus to other assets) that will apply during later development activities.

To identify and retrieve reusable assets, engineers must be able to describe their needs in terms of the domain model developed during asset creation, which defines the logical organization of the asset library. Once the needs are described, the library is searched in whatever manner is appropriate until a satisfactory set of candidate assets is identified and retrieved for evaluation.

If previously reused assets have provided sufficient traceability to other assets within the domain model that meet present needs, the asset description and searching process will be relatively trivial. One way to achieve strong traceability during asset utilization is through the use of a generic domain architecture to guide construction of the new system. The architecture essentially serves as a system template identifying the key architectural elements of the domain and sets of candidate assets that can be used to implement those elements in target systems. Ideally, a user need only visit each element of the architecture in turn and select the most appropriate asset available for that element in order to instantiate the desired system. However, in practice, the architecture may not be complete enough or the variation in the domain may not be well enough understood to allow such an automatic approach (in fact, if it is that automatic, the system is an excellent candidate for generation techniques).

More commonly, the domain model and architecture will indicate the overall structure of typical systems within the domain and provide some information about the characteristics of individual elements. However, this information may be incomplete, at too high a level to immediately indicate appropriate candidate assets, not have sufficient scope to address all the needs of the target system, define a need for assets that are not in the library, or possess a variety of other shortcomings. More problematic is the case where the domain model provides little or no focused architectural information, but rather defines a diffuse set of asset interrelationships which the user must carefully interpret to infer overall system structure. Under these kinds of circumstances, describing and searching for desired assets becomes challenging.

The initial challenge when there is little traceability information available is to determine what is needed. One approach to this problem is to browse the library (and thus the domain model) to become more familiar with the assets, their structure, and their interrelationships. It is possible that browsing alone may reveal the assets that are needed, but the domain model is typically such a rich information space that some form of query, based on the domain taxonomy, must often be employed, to focus the investigation using concise terms that are more manageable and comprehensible to the engineer. The understanding of the domain model acquired while browsing will help the engineer to formulate the queries.

At times, analysis of the domain model may indicate that other domains or other libraries need to be browsed or queried to obtain useful assets. At this point, the notion of seamless library interoperability, discussed in more detail in section 5.1, comes into play. Other domain-specific libraries, which may be locally or remotely located, will need to be browsed and/or queried in a manner similar to the initial library even though they may have substantially different underlying structures, and a high level of seamlessness between libraries will allow the engineer to perform these activities without being strongly aware that such differences exist or that the libraries may be widely distributed on a network.

In this seamless environment, the engineer browses and poses queries, possibly iteratively by gradually broadening or narrowing the search criteria, until a candidate asset set of appropriate and manageable size is achieved. Search criteria can be divided into *concept* and *context* criteria. Concept criteria define the abstract services (features) and capabilities of the desired assets, whereas context criteria define more specific asset constraints (e.g., functional limitations, operational constraints, non-functional requirements). Concept criteria are usually needed to identify the principal functional characteristics of desired assets, while context criteria are used to narrow the set of similar assets satisfying a given set of concepts.

If the search produces more assets than the engineer can evaluate, then the search criteria must be restricted. On the other hand, if the first search identifies no assets, the search criteria can be broadened until some assets are identified. If these assets are obviously not applicable and all relevant criteria have been considered, there are likely no existing assets that meet target system needs, and developing from scratch is warranted. However, if candidate assets have been identified, they should each be inspected more closely to determine which if any of them are appropriate for reuse in the target system.

Asset Understanding, Evaluation, and Selection

The engineer should attempt to understand in detail what each identified asset provides that meets system needs. Asset understanding involves a thorough analysis of an asset's description, as well as analyses of the asset itself and of related assets and other supporting data. Asset description information to be analyzed might include complete, detailed values of the attributes that can be inspected and queried during the asset identification process, as well as asset abstracts, detailed asset interface descriptions, usage histories and problem reports, metrics and quality data, and a variety of other items.

When analyzing assets themselves, an asset can be viewed in its raw source form (e.g., the source code of an Ada package specification) or can be viewed using alternative methods with the assistance of appropriate tools. Examples of such tools are hypertext systems, design diagramming tools, and word processors or more sophisticated (e.g., SGML-based) document authoring systems. Naturally, for such tools to be used, the asset management process must store with each asset the underlying representation appropriate for each tool.

Another aspect of asset understanding is the area of asset quality and assurance. This may involve the inspection or the on-line computation of a variety of metrics for an asset, and may also involve the tracing and inspection of corroborating assurance information, such as test results, formal specifications, formal or informal proofs, and the results of any formal certification or accreditation

processes which the asset (in the context of systems or subsystems of which it was a part) may have undergone.

The engineer may also want to understand the dynamic behavior of the asset. Inspection of behavioral specifications may be sufficient for this, but other approaches include the use of dynamic assessment tools to simulate the behavior of the asset under realistic conditions or the use of test harnesses to actually execute the asset with representative data to provide the engineer with live, hands-on feedback. The latter approach may be useful for better understanding both the functional and non-functional characteristics of the asset. In particular, non-functional characteristics such as performance are often addressed unconvincingly, if at all, in the static asset description, and are best understood through hands-on use. As for asset viewing above, these approaches require significant support from the asset management process to provide the appropriate tools, harnesses, and underlying representations.

Asset evaluation is the process of applying the knowledge gained about an asset through the asset understanding process to evaluate in detail how well the asset meets target systems needs. The needs may be expressed in terms of the concept and context criteria formulated during the asset identification process, possibly augmented with criteria that are not easily expressed in concept and context queries but which can be addressed through careful asset understanding. Some additional criteria may be subjective or intuitive judgements by the engineer based on his or her individual experience with the domain. As a result, aspects of the asset evaluation process may be subjective in nature. Some of the more objective approaches may include comparison of the detailed characteristics of the asset versus system needs with respect to the quantity of services provided, the specific manner in which services are provided, the time and space utilization of the asset, the variability and/or ease of modification of the asset, how well the asset and its related assets match requirements in other system development activities (e.g, maintenance documentation, test cases), and a variety of other factors.

If one or more assets meet all needs, the engineer selects the most appropriate asset for integration. If no asset meets all needs, the engineer must assess whether any particular asset is "close enough" to system needs to justify its reuse. This judgement may have some objective aspects, but may also be highly subjective. The organization may establish economic criteria for making such judgements, based on some economic model of the cost of modifying reusable assets. At the end of the evaluation process, either an asset will be selected for reuse by satisfying sufficient criteria, or a decision will be made to develop that particular aspect of the target system from scratch (preferably in the form of a reusable asset that can be incorporated into the domain model for reuse in future systems).

As an example of the asset evaluation and selection process, in a hard real-time system, asset A may meet the functional need but not the performance need, while asset B may provide only part of the functional need but meet the performance need. The engineer must evaluate the trade-off between modifying asset A to meet the performance need, adding the necessary functionality to asset B, or constructing a new asset C that provides the functionality missing in asset B. This is not a simple task because it depends on many factors, including the adaptability of assets A and B, the structure of other assets, and the data dependency between assets A and B and other assets.

Asset Tailoring and Integration

Once an asset has been selected for reuse, it will usually need to be tailored to fit the specific requirements of the target system, and then will need to be integrated into the system. These activities typically overlap to some degree, and often the distinction is blurred.

Asset tailoring comes in two forms, either or both of which may be applied to any given asset:

anticipated If the asset encapsulates anticipated variations in systems within the domain in some formal way (such as through the use of parameters), each variation must be narrowed appropriately, using the provided formal mechanisms, to meet specific target system needs.

unanticipated If target system needs lie outside the boundaries of the variations anticipated by the domain model (including, for example, new features where no variation was anticipated), the asset must be modified to meet those needs.

To perform *anticipated* tailoring, the engineer must understand what the variations are and how the mechanisms for narrowing them are used. This information should be included in the domain model in the form of "reuse instructions" for the asset, which may be augmented by examples. Parameterization (interpreted broadly) is the mechanism most commonly used for anticipated tailoring. Examples of parameterization include:

- run-time parameters that are passed procedurally to the asset during system execution,
- specifications, macros, data files, or command-line arguments that are interpreted at run-time to produce desired behavior (e.g., initialization files, document style sheets),
- compile-time parameters that are passed to the asset to produce a system-specific instantiation of the asset (e.g., Ada-style generics), and
- installation parameters that control system-specific configuration of the asset (e.g., variables controlling conditional compilation).

In addition to parameterization, another technique that can be used for anticipated tailoring is hand modification of the asset in accordance with detailed instructions. Even with a good set of reuse instructions, some experimentation may be appropriate while tailoring assets using any of the above techniques, to ensure that the tailoring is done most effectively, particularly if the relevant target system requirements are not completely understood.

Unanticipated tailoring is more of an *ad hoc* process in which the engineer assesses the asset's shortcomings relative to system needs and then employs whichever strategies are appropriate to tailor the asset to the needs. This usually involves hand modification of the asset to add desired capability or remove undesired capability. Modifications may be needed to both the *concepts* and *context* of the asset (defined in the discussion of asset identification above). Context factors that may need to be addressed include: performance, environmental considerations, and safety, reliability, and other quality factors.

Although an asset's limitations may be recognized during the asset identification and understanding processes, the full implications of those limitations may not become clear until the unanticipated

tailoring process is undertaken. At that point it may be appropriate to revisit the decision of whether to reuse the asset or develop the desired capability from scratch, depending on a variety of technical and economic considerations. If the decision is to reuse the asset, the preferred approach is to provide feedback to the asset creation process so that the asset and domain model will be modified to take into account the new variations that the target system has revealed, thus benefiting future development efforts in the domain.

Many of the tailoring activities already discussed are, in more general contexts, often considered aspects of system integration. For the purposes of this discussion, asset integration is the process of making tailored assets work with other system components in the context of a system architecture. Thus, while tailoring focuses more on the adaptation issues local to a particular asset, integration addresses adaptation and consistency issues global to an entire system or subsystem. In this view, the tailoring process is subservient to the integration process in the sense that integration may require several iterative refinements of tailoring to ensure that all system needs are being met. Integration may also involve the development of integration modules (so-called "glue code") to allow system components to interoperate when tailoring is inappropriate or insufficient for that purpose. Ideally, these integration modules will also provide feedback to the asset creation process to impact domain model evolution.

4.4.2 System Generation

System generation is a process for producing systems or subsystems that ideally incorporates all the variation in a domain into a set of parameters expressed in terms of a specification language or template. A generation tool accepts specifications from engineers that define values for the domain parameters and resolves the variation accordingly to generate components of the target system. The specifications are generally non-procedural in nature and can be expressed in a number of different forms (e.g., textual, graphical, template-based, etc.). These specifications in effect define a set of specific target system requirements that lie within a set of more generic domain requirements represented by the specification language. Thus, the target components are derived directly from a specification of system requirements, which is why generation is often referred to as requirements-based reuse.

As noted in section 4.2, the variations captured in the specification language were identified during domain analysis, but instead of building conventional reusable assets, the domain engineer codified all variation in the generator. This is generally only possible when the variation and mappings of the variation to solutions are well understood. Thus, system generation methods are usually only applicable in highly mature domains or subdomains. It is not uncommon in larger domains for there to be a number of small subdomains in which generation methods are applied. In these cases, the generation process is part of a larger system composition process. In fact, the generation process can be viewed as a very sophisticated form of the anticipated tailoring process described in section 4.4.1. However, we view generation as sufficiently distinct and important to merit consideration separate from composition methods and more conventional asset tailoring techniques.

For the purposes of this discussion, there are two key differences between generation and anticipated tailoring:

1. Generation typically involves much more sophisticated and extensive parameterization.

2. Generation employs a separate tool to generate the products that are actually integrated into the system.

With respect to (2), a key point is that, with generation, the asset that an engineer reuses is the generator tool itself, rather than some adaptable form of the eventual system product. The generator in effect provides an encapsulated black-box view of some portion of a domain, obviating the need to find and compose a set of conventional assets in that subdomain.

Since generator tools are viewed as assets within a domain model and encapsulate some portion of a domain architecture, the overall system composition process must consider them from a perspective similar to that with which it regards other assets to be composed. Section 4.4.1 did not explicitly address generation assets because they pose issues that are significantly different with respect to the asset identification, asset understanding/evaluation/selection, and asset tailoring/integration processes. These differences are addressed below.

Asset Identification

The asset identification process for generator assets is not greatly different than for other assets. A given application of an asset identification process may well identify a mix of conventional and generator assets without strongly distinguishing between the two. The primary differences revolve around the following characteristics:

- Generator assets tend to have larger scope than individual conventional assets, in that they may encompass entire subdomains and encapsulate substantial portions of architectures. This tends to simplify the asset identification process by reducing the total number of assets to be considered and reducing the complexity of their interrelationships.
- Generator assets tend to have well-defined roles within domain architectures, so the process of identifying them is generally highly amenable to architecture-based search.
- Generator assets tend to capture variation more extensively than conventional assets, since the variation in their domains of application is so well understood. This breadth of scope may yield some difficulty in describing the asset concisely yet comprehensively, which may have some negative impact on the engineer's ability to determine whether the asset meets system needs. However, this issue is somewhat more relevant to asset understanding.
- Generators typically are not readily divisible or decomposable, and their relatively large scope may remove from reuse consideration many individually useful lower-level functions that generators encapsulate. If this is foreseen as a problem within a given domain, some lower-level functions may be represented within the domain model in the form of conventional assets; in fact, some generators may utilize such assets when generating their larger-scale system components.

Asset Understanding, Evaluation, and Selection

From a high-level perspective, the understanding and evaluation of generator assets is very similar in principle to the analogous processes for more conventional assets. However, at a more detailed level,

some key aspects of the processes are significantly different, reflecting the conceptual differences between the two kinds of assets.

The process of analyzing the descriptive information provided for a generator asset within a library is generally the same as for other assets, in that it involves the inspection of asset attributes, abstracts, usage histories and problems reports, metrics and quality data, and so on. One issue here is whether the information describes characteristics of the generated products or of the generator itself; full understanding of the asset requires information about both these aspects, but depending on the generator and the role it plays in systems within the domain, an emphasis on one or the other aspect may be sufficient. A related issue is that generator assets generally distinguish sharply between two different sets of interfaces: the *tailoring* interfaces to the generator itself (e.g., the specification language, rules for invoking and interacting with the generator tool), and the *integration* interfaces to the generated products (e.g., procedural interfaces to be invoked at run-time). The engineer should work to understand both sets of interfaces.

With respect to the tailoring interfaces, the availability within the library of good documentation about the nature and usage of the generator and the specification language, supplemented with examples of specifications or tool interaction sessions, is highly valuable to the engineer during the asset understanding process. Such materials constitute the "reuse instructions" for a generator asset.

Unlike conventional assets, there is generally little or no need for engineers to inspect generator assets in their source form, since the generators themselves will not be part of the target system. The processes analogous to source code inspection for generator assets are inspection of sample specifications and inspection of generated products.

Inspection of sample specifications is done using whichever tools are available and appropriate for the particular form(s) of specification that the generator accepts. If specification is performed interactively, the generator tool itself is used, possibly in conjunction with scripts to present sample tool interaction sessions. During this activity, the entire generation process, yielding sample generated products, may be undertaken to further enhance understanding of the asset. In addition, experimentation with the specification language is also appropriate at this time, either through modification of the sample specifications or the development of small test specifications.

The inspection of generated products may be of interest to the engineer for purposes of general asset understanding, assessment of the quality of the generated products, or other similar reasons. However, such inspection is often done to assess how easy or difficult it will be to modify the generated products if the engineer suspects that some unanticipated tailoring of the products will be necessary. Sample generated products are either provided in the library or will need to be generated by the engineer via the process noted above. Issues surrounding the modification of generated products are addressed in the discussion of asset tailoring below.

Another important contributor to generator asset understanding is the actual execution of sample generated products using representative data to obtain live feedback about the functional and non-functional characteristics of the asset. This is particularly important for assessing asset performance, since it is not uncommon for assets possessing a high degree of generality to exhibit poor performance.

Some of the issues that must be addressed in evaluating whether a generator asset meets organiza-

tion and target system needs are:

- Is the scope of the asset too large or too small? It may be too small if not all concept criteria are satisfied. If the asset addresses many more concepts than are needed within the target system, or if it addresses the needed concepts in too general a fashion, it may be unable to satisfy some context criteria (e.g., performance requirements) as a result.
- If the asset is perceived as not fully addressing target system concepts and context, what alternatives are available and what are the costs and benefits of each? Specifically, should the generator be modified, the generated products be modified, additional complementary products be developed, or the generator not be utilized at all in favor of composition of lower-level assets and/or newly-developed products?
- Is the generator sufficiently versatile to meet system evolution needs? If not, again what are the alternatives and their costs and benefits? One area this addresses is the evolution of base technology, and whether the generator will need to evolve accordingly.

Asset Tailoring and Integration

Naturally, most tailoring of generator assets is of the *anticipated* variety. To tailor a generator asset to produce a desired system component, the engineer must use the asset's *tailoring* interfaces to express decisions about the system requirements within the generator's scope. This typically involves either creating a specification or interacting with the generator tool (e.g., to fill out a template interactively). The engineer may apply the knowledge gained about the asset's tailoring interfaces (through its "reuse instructions") during the asset understanding and evaluation process to tailor the asset immediately, or may engage initially in further experimentation to determine how best to reuse the asset.

Once the engineer has expressed the decisions about the system in whatever form is appropriate, the generation tool determines the validity of these decisions and ensures that the decisions are mutually consistent (in some cases, the language itself guarantees this). The decisions an engineer may make include the selection of data types, ranges, and formats, specific system services, and data and service interrelationships. The generator uses these decisions to resolve the variations it encapsulates and then generates the appropriate system components. The generator may not produce all needed parts of the system components being generated, but missing pieces should be easily identified by the engineer. For example, many existing parser generators generate code that recognizes the constructs of the language being parsed, and provides well-defined methods for passing control to user developed code that performs additional translation functions, such as semantic analysis. Another example is that generators may not provide the documentation (e.g., design documentation) required by a project, or may not provide it in the proper form. The project must either obtain an exception to the documentation requirements under these circumstances, or the documentation must be produced by project engineers. However, in the ideal case, the generator will be tailorable to produce the appropriate documentation.

Unanticipated tailoring is also an option with generator assets. This typically takes the form of modifying the generated components to meet some system need unforeseen by the domain analysts and/or generator developers. The engineer is free to add additional concepts to generated code

or to adapt it to the particular context of the system under development, but any such activity should be pursued with great caution. One problem with this is that generated source code is often not as human readable as hand-written code, and may thus be significantly less modifiable and maintainable. A potentially more serious problem is that generated components are in some sense less persistent than the specifications from which they were produced, and any time the products are regenerated, perhaps with slight variations, the modifications will have to be redone. This is inherently a risky and error-prone process, and this risk presents a substantial disincentive to perform such modifications. However, this disincentive may promote greater system integration problems, since more "glue code" or modification to other system components may be necessary to accommodate the perceived rigidity of the generated components.

Another alternative when the generator doesn't fully meet system needs is to modify the generator itself to accommodate greater variation or to satisfy some specific system requirement. Ideally, this should be done by notifying asset creators that a need exists, so that they can perform the necessary modifications and evolve the domain model in concert. Even if this is not done, the asset creators should be notified after the fact so that the modifications will eventually be reflected in the domain model. Obviously, generator modification is only possible when the generator source code is available, and this may often not be the case, particularly when the generator is a commercial product.

4.4.3 Feedback to Reuse Planning, Asset Creation, and Asset Management

The goals for each reuse-based system development effort are defined in the reuse planning process, as described in 4.1. To determine if the reuse goals have been met, the plans may stipulate that data be collected during asset utilization. The effectiveness of reuse during asset utilization is difficult to measure because different applications may have different goals, and generalization of results may be impossible without a large sample. Thus, asset utilization processes should be carefully tailored to meet each organization's particular metrics collection needs, and this aspect of the process should be followed meticulously over a series of development efforts to enable the organization to evolve and gradually improve their software development processes and capabilities.

For each reuse-based development effort, assets within the relevant domain(s) will likely need to be updated based on feedback from asset utilization. The engineering of each new system and even enhancements to existing systems will tend to identify needed changes. If changes are appropriate, the domain model must be updated, new assets may be constructed, and other assets may be changed or deleted. One possibility is that the tailoring and composition of certain assets during system construction may reveal ways in which existing assets can be further generalized or combined into larger-scale assets, or, alternatively, may indicate that what was previously considered a single domain is best viewed as a collection of subdomains, each somewhat more specialized for particular needs than its parent. In addition, unanticipated shortcomings or bugs may be revealed during utilization, and these will need to be fixed. Utilization may also identify areas in the domain where expanded capabilities are needed to meet evolving requirements; these can be identified through new feature requests that are submitted by engineers, but can also be automated to some degree by analyzing failed asset queries to identify perceived engineering needs that the library and domain model are not satisfying.

Similarly, each reuse-based development effort should yield lessons that can be applied to asset

management within the domain. Engineers' experiences with browsing and querying the library may result in recommendations for refining or correcting aspects of the library taxonomy or asset descriptions. Experiences with the tools used to facilitate asset understanding, tailoring, integration, and generation may yield recommendations for additional tools or improvements to the existing tools. Problems with assets that were thought to be well-qualified may reveal inadequacies in the asset qualification process. Some systems may require assets in multiple domains, and the libraries housing assets in some of the domains may be located remotely; lack of adequate access to the remote libraries may result in recommendations for improved library connectivity or interoperability.

5 Integrating Views of the Framework

Section 4 describes the STARS Reuse Process Framework, identifies the process families within the framework, and identifies and discusses in some detail the individual processes within each family. The primary focus of section 4 is on individual elements of the framework, with relatively little emphasis on how the elements interrelate, and even less emphasis on how the framework can be used to address organization or project needs.

This section provides three additional views of the framework that serve to integrate some of the individual concepts introduced in section 4:

- The **Seamless Library Interoperability** view illustrates that asset libraries can be managed and integrated to establish a distributed network of seamlessly interoperating libraries that asset utilizers can access transparently.
- The **Reuse-based Software Life Cycle Models** view identifies a variety of reuse-based life cycle models and discusses how processes within the framework can be applied to each.
- The **Technology Support for Reuse Processes** view discusses how the technology available and in use within an organization can impact the specific reuse processes that the organization employs.

5.1 Seamless Library Interoperability

STARS envisions that reuse in the future will occur in the context of a distributed network of heterogeneous domain-specific libraries. Each library will likely focus narrowly on one or a small set of vertical or horizontal domains, since libraries emphasizing relatively narrow domains are more likely to yield high impact reuse through greater depth of focus and better control of variability. However, this proliferation of domain-specific libraries will promote library heterogeneity, since the libraries will utilize distinct data models designed specifically to capture the characteristics of their respective domains. This heterogeneity, if unmanaged, may potentially inhibit reuse by forcing users to understand the structure and terminology of many different library data models. To further compound the difficulties library users may face, the libraries will operate on a variety of hardware and operating system platforms, and each library may potentially reside on a different host in a local or wide area network.

In this distributed, heterogeneous library context, one of the key challenges will be the establishment of mechanisms to allow users at a given host to locate, inspect, and reuse assets within the entire library network. Capabilities will be needed to enable users to find and retrieve assets that are of interest to them, regardless of the libraries in which those assets reside. Such capabilities will require some global knowledge of the contents of the networked libraries. The heterogeneity of the libraries makes the representation of this knowledge particularly challenging, since such knowledge, if represented in any depth, must reflect the structure of the individual library data models to some degree.

5.1.1 Levels of Library Interoperability

The challenges of library interoperability can be met in a number of ways and to varying degrees. Following are characterizations of several potential levels of interoperability.

Seamless

The goal for distributed, heterogeneous libraries is to create a truly "seamless" environment for library users, in which the boundaries between libraries are transparent to the user and a convincing illusion of a centralized library (or perhaps library of libraries) is created. The entire asset information space appears to have a uniform and consistent structure, in the sense that it all appears to be derived from a single meta-data model. The individual library data models also appear to have significant consistency, within the limits of the natural variability inherent in the domains being modeled.

In such a library environment, the user has a common set of operations to apply to assets in all the libraries. In particular, those operations are the operations provided by the user's native software engineering environment (SEE). Other users, in other SEEs, will similarly be able to employ the operations available to them. The key point here is that two users, in two different SEEs, will be able to access the same seamless asset information space, but their views of that space, and the ways in which they interact with it, may be substantially different (yet both entirely valid), depending on the characteristics of their respective SEEs.

Biggest Seams Show

In a somewhat less seamless heterogeneous library environment, the largest seams are evident to a user. The user is now aware that there are different libraries on different host systems, interconnected via a network. Nevertheless, the meta-data models of the various libraries appear consistent, and the operations that the user applies to all the assets are still basically the same as in the more seamless environment. However, those operations may need to be tailored somewhat to overcome the now apparent physical separation of libraries.

Whereas in the more seamless environment, there may be substantial interrelationships between physically distinct libraries, thus further blurring the boundaries between them, in this environment there are fewer such relationships expressed (that is, a somewhat reduced level of global knowledge), and the relationships that do exist may require the user to actively switch library context to follow them. At this level there will be a greater need to formally interchange assets (transport assets and data models) between libraries to replicate and localize knowledge about certain assets within the network, reflecting the somewhat greater difficulty involved in navigating the total asset information space.

Smaller Seams Show

At a lower level of library interoperability, the physical separation of libraries is highly apparent, and the user needs to actively and explicitly cross library boundaries to move from one library context to another. However, there is still significant global knowledge of the asset information space available at this level, possibly in the form of library and asset directory services (sometimes referred to as yellow pages services) to provide coarse guidance about where to go to find (or at least to look for) certain assets or classes of assets.

Differences in library structure and user interfaces are apparent at this level, but there may be significant documentation or on-line assistance to help the user operate in different libraries within this environment. There is a strong need (and sufficient automated support) for formal asset interchange between libraries, due to the greater difficulty in finding and retrieving assets than in the higher two levels and the concomitant desire to localize key assets to minimize future search. In addition, there may be significant global capabilities for retrieving remote assets via centralized storehouses of raw asset data, which are referenced by asset descriptions within individual libraries in the network.

All Seams Show

The lowest level of library interoperability, where all or nearly all the seams show, approximates the current state of the practice. In such an environment, the user may be aware of a set of libraries that are accessible by various means either locally or remotely, but little if any global knowledge of library contents is available, thus requiring the user to visit each library to obtain such knowledge and to inspect the assets contained within. Each library is likely to have a unique structure and user interface that must be learned by each user, often without the benefit of adequate documentation or on-line assistance, and there are likely to be few automated capabilities for retrieving remote assets and their associated domain context to facilitate either asset utilization by users or asset interchange by library administrators.

5.1.2 Interoperability between Libraries and SEEs

In addition to library-library interoperability, another important aspect of reuse in the future will be interoperability between asset libraries and the SEEs employed by users. Asset libraries and their associated tools, like any software engineering capabilities, can be integrated into a SEE with or without careful regard for how they will work together with other elements of the SEE to help solve users' problems. Libraries that are not well integrated may substantially inhibit efficient reuse by erecting artificial barriers to the management, understanding, and utilization of assets. In contrast, libraries that are very closely integrated with their SEEs, and can thus readily apply to assets many of the SEE capabilities that are applicable to ordinary SEE objects (e.g., communications, versioning, configuration management, access control, measurement and understanding), while also interoperating smoothly with the tools employed to reuse the assets, will not merely remove barriers to reuse, but actually encourage it as a natural element of day-to-day user activity within the SEE.

5.2 Reuse-based Software Life Cycle Models

Life cycles are usually modeled as a time-ordered series of major activities. A waterfall model treats the major activities as phases. A spiral model treats major activities as quadrants of an increasing spiral. As discussed briefly in Section 4, the reuse process framework is to be used to guide composition and instantiation of reuse-based software life cycle models by selecting compatible processes from among its process families. The processes selected should be compatible among themselves, with organizational goals, strategies, and strengths, with project requirements and constraints, and with characteristics of the domain.

We can derive at least three reuse-based life cycle models from the STARS vision based on different organizational goals:

- reuse-based domain development and evolution,
- reuse-based system integration, and
- reuse-based system evolution.

Reuse-based domain development has the goal of producing and evolving software components, software architectures, or application generators that may be used in many different software-intensive systems in the same domain. Domain development primarily *produces and evolves* software assets. Reuse-based system integration constructs new, complex software-intensive systems that are integrations from multiple (sub)domains. System integration primarily *reuses* software assets. Reuse-based system evolution has the goal of maintaining software-intensive systems while their underlying requirements, constraints, and supporting technologies evolve. System evolution primarily *reuses* software assets also. The basic difference between integration and evolution is that evolution begins when integration delivers a complete system. Note that these descriptions do not imply how tightly or loosely related are the organizations handing off product from one life cycle or project to another.

Domain development reflects an emerging DoD market niche to construct and evolve software-related products for individual domains. System integration complements domain development by constructing new systems from products of the development. System evolution reflects (1) an accommodation to changing DoD budgetary constraints emphasizing fewer new procurements and more long-lived systems and (2) a recognition that maintenance often consumes 65-80% of system life cycle costs. Furthermore, system evolution, as it should incorporate domain modeling processes, may be able to take advantage of products of evolving domain developments.

If we reorganize the processes of the reuse process framework into phases where activities in one phase precede or create products used by activities in another phase, the result is a phase for domain analysis and modeling processes followed by a phase for software asset creation processes followed by a phase for asset utilization processes. These phases are depicted in Figure 2.

We have grouped the processes of the *asset creation* family into two separate phases so that we can highlight our belief that all reuse-based life cycle models should include domain analysis and modeling processes. Note also that the processes of the *asset management* family are used in all three phases.

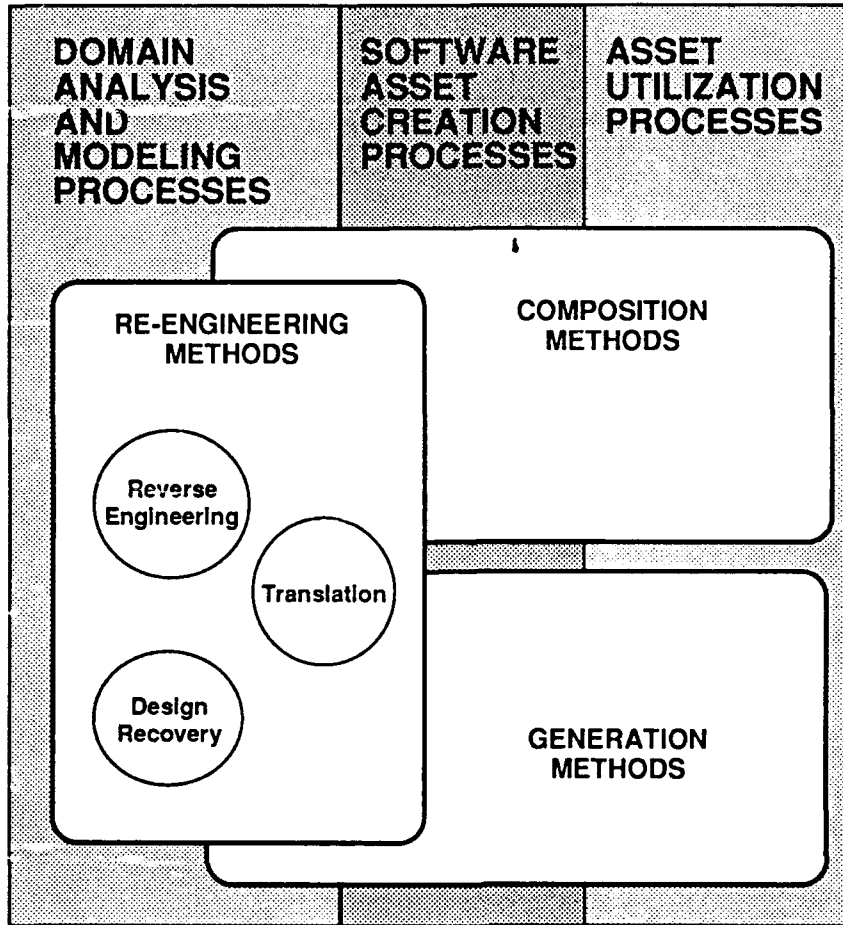


Figure 2: Methods Supporting Reuse

Different life cycle models may or may not use asset utilization or asset creation processes. For instance, it is plausible to define a domain development life cycle model that only iterates through domain analysis and modeling processes and software asset creation processes. It is also plausible to define a system integration life cycle model that uses domain analysis and modeling processes to adapt a more abstract domain model for a particular system that is to be built and then uses asset utilization processes to assemble the system, skipping software asset creation processes because the assets were previously created to be consistent with the original, more abstract domain model [STA90]. Further, it is plausible to define a system evolution model that, once a domain model has been created, primarily concentrates on asset utilization processes and uses the feedback processes to asset creation to refine domain knowledge and guide asset evolution. What is important to note about these life cycle models is that all use domain analysis and modeling to drive asset creation or utilization and that all should strive to ensure that the reuse investment remains viable by keeping domain knowledge current and incorporating feedback from asset utilization activities.

We can also use this Figure 2 to relate the reuse process families to current reuse and reengineering research activities and literature. These are often described in terms of domain engineering or forward engineering phases [Ree91]. The activities described as domain engineering, e.g., domain analysis and modeling and software asset creation, map to processes of the *asset creation* family of

the reuse process framework. The activities described as forward engineering, e.g., those that result in software products such as architectures, components, generators, or systems, map to processes of the *asset creation* and *utilization* families of the reuse process framework.

5.3 Technology Support for Reuse Processes

Figure 2, which depicts major activities or phases of reuse-based life cycle models, provides a context for considering the usual categorization of techniques, methods and tools as composition-, generation-, or reengineering-based. The figure shows reengineering technology supporting domain analysis and software asset creation; and shows composition and generation technology supporting domain analysis, software asset creation, and asset utilization.

Selection of specific tools and methods to be used on a project should follow the organization's reuse strategy and specific process selections. That is, the tools and methods depend upon the life cycle model to be used, the maturity of the domain for which the project is planned, the strengths and weaknesses of the organization and its members, and management objectives to maintain consistency and minimize effort expended to learn to use tools and methods. To put it plainly, a life cycle model and processes to support that model should be defined before methods, techniques, and tools are chosen.

The remainder of this section discusses the roles of construction (composition and generation) and reengineering technologies.

5.3.1 Reengineering Technology

The use of reengineering technology is a major reuse thrust for many organizations who find they have millions of lines of source code based on obsolete hardware and development approaches. The goal of applying reengineering technology is to analyze and rework existing systems in order to reuse expertise already encoded in them. As Figure 2 shows, reengineering methods include reverse engineering, design recovery, and source code translation.

We believe reengineering technology can be used to support a shift in focus for software maintenance, post-deployment support or reengineering organizations from a reactive life cycle model to a reuse-based life cycle model of system evolution. Reengineering technology, through support of domain-focused asset creation processes and domain analysis and modeling processes, can extract and make human understandable valuable and costly information that has been obscured by the passage of time or turnover in project personnel. Besides ameliorating inaccessibility of information, the products of reengineering can be used to (re)structure systems to take advantage of current and emerging standards and solution technologies, to better predict the impact of particular changes to a system, and to guide reapplication of a system or parts of a system to solve a similar problem.

Although numerous commercial reengineering tools have recently been announced, few successful applications have been widely publicized. Several factors are hindering the wide-spread use of commercial-off-the-shelf standalone or integrated reengineering tools. Reengineering tools are often based on an intermediate abstract representation of code (e.g., DIANA, IRIS, IDL, REFINE, etc.) such as compilers use, but there has been little cooperation between compiler and reengineering

tool vendors. Secondly, there is no official abstract representation standard that allows translation from one programming language to another. Thirdly, DIANA is an almost standard abstract representation for Ada but most legacy systems are not coded in Ada. Finally, design recovery depends on extracting semantic as well as syntactic information, which often requires the application of knowledge-based techniques that are just now being slowly adopted by software tool vendors.

However, there are some standardization efforts with respect to Ada compiler technology that can be used to support reengineering of Ada programs. These efforts are working to standardize interfaces to support extraction of semantic information from Ada source code. In particular, STARS has sponsored the development of the Ada Semantics Interface Specification (ASIS), a draft Ada interface binding to Ada compilers' databases [BB91], and has also supported proof-of-concept implementations of selected aspects of the draft ASIS bindings by some Ada compiler vendors. Widespread vendor support for this emerging standard should enable Ada reengineering tools to readily access information that is critical to sophisticated reverse engineering and design recovery techniques.

5.3.2 Construction Technologies

As discussed in section 4.4, *composition* and *generation* are the two major approaches to constructing software systems. The basic theme of *composition* is to assemble the desired system from software components, where the components may be newly created or reusable. The basic theme of *generation* is to transform specifications and constraints into the desired system.

Tools and methods supporting system construction often mix both approaches, making categorization of their implementations difficult. For instance, generation tools may actually assemble their output guided by the input specifications and constraints rather than apply a series of textual translations to the input. Tools supporting composition may produce individual software components for later assembly by filling in parameters (e.g., Ada generics).

Exact categorization of system construction technology is not important to evaluating whether to apply it in support of a particular life cycle model or project. What should be considered is the type of asset on which the technology operates. Composition technology operates on software components or subsystems; generation technology operates directly on system specifications or constraints. Thus, use of a generation approach is more common and appropriate in mature domains with well-understood requirements and where the impact of specific constraints on resulting systems is known. Composition is more appropriate for new or immature domains where specifications are difficult to write or to complete.

Technology supporting composition includes software component library systems, domain model and analysis product browsers, and software structure/design browsers. Technology supporting generation include application generators, meta-generators, and cross-compilers.

The choice of technology to support reuse-based system evolution or system integration life cycles will be dictated by the match between an organization's development environment, the domain expertise available to the organization, availability of assets, and the match between the supporting composition or generation technology for those assets.

The choice of technology to support a reuse-based domain development life cycle is complicated by

consideration of both the developing and using (customer) organizations with respect to development environment, domain expertise availability, and supporting composition or generation technology. For example, a developing organization may be able to construct a very flexible, complete set of components for a particular domain but may choose to construct a more limited application generator because potential customer organizations do not have sufficient domain expertise or experience to be able to take advantage of the more extensive software components library or do not wish to incur the extra cost or effort needed to use the flexibility available. The former case of limited customer experience is consistent with domain immaturity; the latter case is consistent with tactics followed by customer organizations wishing to limit development costs for use of a mature, enabling technology such as relational databases.

A Glossary

abstract representation; *noun:* An expression of the syntax and semantics of a program or program fragment in a form that abstracts away the concrete syntax of a programming language. For example, a parse tree or a DIANA expression of an Ada package.

application generator; *noun:* A software tool that accepts as input the requirements or design for a computer program [or component] and produces source code that implements the requirements or design. Also referred to as source code generator.

asset; *noun:* Any unit of information of current or future value to a software-intensive systems development and/or PDSS enterprise. Assets may be characterized in many ways including as software-related work products, software subsystems, software components, contact lists for experts, architectures, domain analyses, designs, documents, case studies, lessons learned, research results, seminal software engineering concepts and presentations, etc.

asset certification; *verb:* The process of confirming that an asset correctly implements its stated function(s), adheres to quality and reuse standards, and, possibly, is formally proven correct.

asset evaluation; *verb:* The process of determining whether a particular asset fits requirements and constraints of a particular software application, architecture, or domain model.

asset library; *noun:* A collection of software assets controlled by an asset library system. Typically, asset libraries are implemented using an asset library system, which is a computer-based system designed to facilitate the reuse and sharing of software assets. Asset library systems provide a set of services that support qualifying, reusing, and managing software assets. See asset library open framework for a discussion of these services.

asset library interoperability; *noun:* The ability of two or more distinct, heterogeneous asset libraries to dynamically provide access to the other's assets, asset descriptions, and data models.

asset understanding; *verb:* The process of thoroughly analyzing an asset and its description in order to grasp the functionality being provided as well as the constraints and limitations on its use.

collaborative development; *verb:* A development process characterized as a cooperative, team effort that may cross geographic or organizational boundaries. For instance, the DoD Prototech project is a collaborative development involving mixed academic and industrial teams.

component; *noun:* One of the parts that make up a software-intensive system. A component may be hardware or software and may be subdivided into other components. A complete software component includes both the object code and all related information that is needed to use it. This related information includes parameterization information, source code if not proprietary, test information, design information, evaluation results, and other descriptive information.

composition; *verb:* The reuse methodology or approach that combines software components, subsystems, etc. into a single application system.

constraint; *noun:* A functional or operational requirement for a software system that limits the possible solution space.

data model; noun: The organizing principles and concepts underlying a database.

design; verb: The process of defining the software structure, components, modules, interfaces, and data for an application.

design rationale; noun: The reasons for decisions and design elements that underlie a particular design.

design recovery; verb: The process of analyzing the results of reverse engineering to identify design elements, their interrelationships and interactions, and design principles, requirements, and constraints. See reverse engineering.

distributed, heterogeneous asset library; noun: An asset library that is implemented across distributed, heterogeneous computer platforms and contains heterogeneous asset data models.

domain; noun: An area of activity or knowledge. Domains have been characterized as application, horizontal, or vertical, technology, computer science, execution, execution models, etc.. Figure 3 graphically depicts relationships among some characterizations of domains, while the text below elaborates on those characterizations.

application domain; noun: The knowledge and concepts that pertain to a particular computer application area. Examples include battle management, avionics, C³I, nuclear physics. Each application domain can be decomposed into a tree or family of more specialized (sub)domains where the decomposition is guided by the purpose or aim of the domain. For example, C³I may be decomposed into C³I for land operations, for sea operations, for air operations, etc.

horizontal domain; noun: The knowledge and concepts that pertain to a particular functionality of a set of software components that can be utilized across more than one application domain. Examples include user interfaces, database systems, and statistics. Most horizontal domains can be decomposed into a tree or family of more specialized (sub)domains where the decomposition is guided by characteristics of the solution software. Distinguishing characteristics may be software decomposition style (functional, object-oriented, data-oriented, control-oriented, declarative, etc.), conceptual underpinning (relational, hierarchical data models), and/or required hardware. One example is subdividing user interfaces into ANSI terminal supporting versus bit-mapped, mouse input supporting solutions.

vertical domain; noun: The essential functionality of a restricted set of systems that pertain to a particular member of an application (sub)domain. This functionality can be organized as a hierarchy of functions. Also, a particular solution identified as implementing one horizontal (sub)domain may be recognized as a good fit to requirements as described/modeled for a specific vertical (sub)domain.

domain analysis; verb: The process of identifying, collecting, organizing, analyzing, and representing a domain model and software architecture from the study of existing systems, underlying theory, emerging technology, and development histories within the domain of interest.

domain engineering; verb: The construction of components, methods, and tools and their supporting documentation to solve the problems of system/subsystem development by the application of the knowledge in the domain model and software architectures.

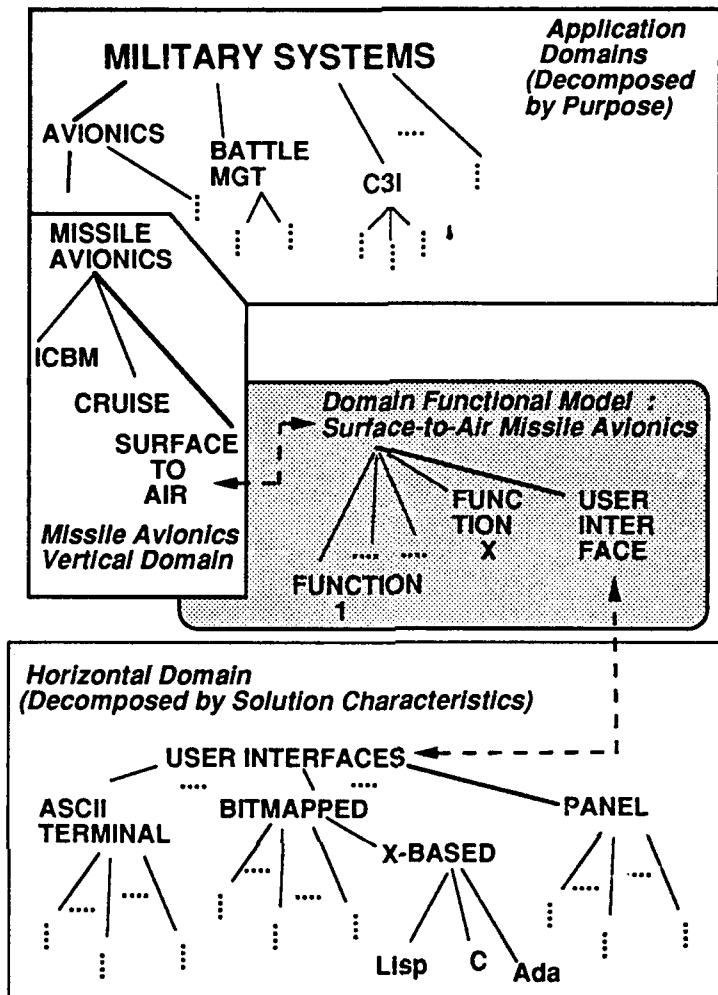


Figure 3: Types of Domains

domain model; noun: A definition of the functions, objects, data, requirements, relationships, and variations in a particular domain.

domain functional model; noun: A decomposition of representative systems of the domain that gives the functional capabilities for them and variability of those capabilities. Note that the decomposition does not imply a particular system architecture or set of subsystems.

domain-specific language; noun: A machine-processable language whose terms are derived from the domain model and that is used for the definition of components or software architectures supporting that domain.

framework; noun: A skeletal structure to support or enclose something. The skeletal structure in these reuse documents is a conceptual structure that delimits the concepts being discussed; supports understanding and technical transition; and promotes evolution.

reuse process framework; noun: The conceptual structure that categorizes and interrelates reuse processes by their purposes, goals, and activity characterizations.

asset library open architecture framework (ALOAF); noun: The conceptual structure that supports seamless interchange and interoperability among networked, distributed, heterogeneous asset libraries by defining a service model; protocols supporting that model; Ada package specifications for the protocols; and a specification for an asset exchange common data model, semantics and formats.

generation; verb: The reuse approach or methodology that constructs a software (sub)system from non-procedural user specifications of desired functionality. See composition.

library mechanism; noun: A software (sub)system that provides a framework for a logical capability. A library mechanism requires tailoring and, possibly, extension to become a library system instantiation.

life cycle; noun: All the activities a software or software-related product is subjected to from its inception until it is no longer useful. Note that this definition shifts the usual definition of life cycle, which is based on life of a *system*, to a more general concept covering the lifetime of a software *product*.

life cycle model; noun: A framework containing the processes, activities, and tasks involved in the development and maintenance of software and software related products, spanning the life of the product(s) from the definition of requirements to the termination of usage. This document shifts modeling of life cycles from phases to compositions of processes and characterizes different life cycles by their individual goals. Life cycles discussed in this document are described immediately following this paragraph.

reuse-based domain development life cycle model; noun: A reuse-based life cycle model whose goal is to produce architectures, domain models, software components, and application generators that provide a family of solutions for a particular domain.

reuse-based system integration life cycle model; noun: A reuse-based life cycle model that constructs new, complex software-intensive systems that integrate software-related assets from multiple domain developments.

reuse-based system evolution life cycle model; noun: A reuse-based life cycle model whose goal is maintaining a software-intensive system while its requirements, constraints, and supporting technologies evolve.

life cycle phase; noun: One element in a model of a life cycle that treats a life cycle as a series of major activities or product stages.

process; noun: A series of steps, actions, or activities to bring about a desired result.

process building block; noun: A precise definition of a process that can be composed with other process building blocks to build up broader process contexts or life cycles.

process definition; noun: A rigorous description of a process including defined outputs and results, possibly formal representations, well-defined beginning and end points, and testable start and stop criteria.

query; noun: A request for identification of a set of assets or library data model elements, expressed in terms of a set of criteria which the identified items must satisfy.

software-intensive; adjective: A characteristic of a system that suggests that its software components provide the majority of the system's functionality and capability.

reengineering; verb: The process of examining, altering, and re-implementing of an existing computer system to reconstitute it in a new form.

requirement; noun: A capability or characteristic that must be provided or met. Requirements can be functional, i.e., provide capability, or can be non-functional, i.e., meet important characteristics such as can be levied as criteria on dynamic performance for data access or retrieval.

reverse engineering; verb: The process of analyzing a computer system's software to identify components and their interrelationships. See design recovery.

reuse; verb: The transfer of expertise. In software engineering, reuse often refers to the transfer of expertise encoded in software related work products. The simplest form of reuse from software work products is the use of subroutine/subprogram libraries for string manipulations or mathematic calculations. The simplest form of reuse of expertise not represented in software work products is the employment of a human experienced in the desired endeavor.

reuse strategy; noun: An instantiation and tailoring of a reuse-based life cycle model as a strategy for a particular project or organization. The strategy describes the processes to be used.

reuse-based development; verb: The application of a disciplined, systematic, quantifiable approach to the development, operation and maintenance of software with reuse as a primary consideration in the approach.

software development plan (SDP); noun: The controlling document for managing a particular software development project.

software architecture; noun: The high level design for software system or subsystem. Includes the description of each software component's functionality (or result), name, parameters and their types and a description of the components' interrelationships. Note that this definition describes software architecture from a system point of view rather than a domain point of view. Many different definitions of software architecture are currently in use, often in the same sentence depending upon qualifiers such as 'generic' or 'domain-specific'. The next release of this document will bring some clarification to definition and usage of this term by STARS.

software engineering environment (SEE); noun: The computer hardware, operating system, tools, computer-hosted capabilities, and rules that an individual software engineer works within to develop a software system.

specification; noun: A document or formal representation that prescribes, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a software-intensive system or software component.

tailoring; verb: The process of adapting requirements, designs, architectures, components, tools, or processes for implementation in actual systems or development environments.

technology; noun: The methods and tools used in the application of a scientific or engineering discipline.

traceability; noun: The characteristic of software systems or designs or architectures or domain models that identifies and documents the derivation path (upward) and allocation/flowdown path (downward) of requirements and constraints.

translation; verb: A reengineering method that transforms a program fragment written in one programming language or language version into another.

validation; verb: The process of determining or approving a software-related product or asset.

variation; noun: The extent, degree, or description of how a single domain characteristic, requirement, constraint, functional, or architectural element may vary.