Process and Products for Software Reuse and Domain Analysis

Sholom Cohen
Software Engineering Institute
Sponsored by the U.S. Department of Defense

The adoption of reuse into software development will include the definition of new products and processes. On the product side, we must identify the form of deliverables that can support reuse; on the process side, the approach needed to develop and apply those products. In order to move from the current, *ad hoc* approach to reuse, to a systematic reuse process, we must be able to both abstract a problem domain and create reusable solutions. The solutions can then be used in the development of a range of systems in the domain.

Each of the next three sections will examine the processes and products that can lead to successful reuse. The first process, one widely used today, is to adapt an existing system to meet a new set of requirements. The second, a relatively new practice, identifies families of programs, providing support for parameterization of commonality, and customization for unique requirements. The third process is an abstract-based engineering approach to discover and exploit commonality in software systems as the basis for software development. Figure 1 compares these three approaches.

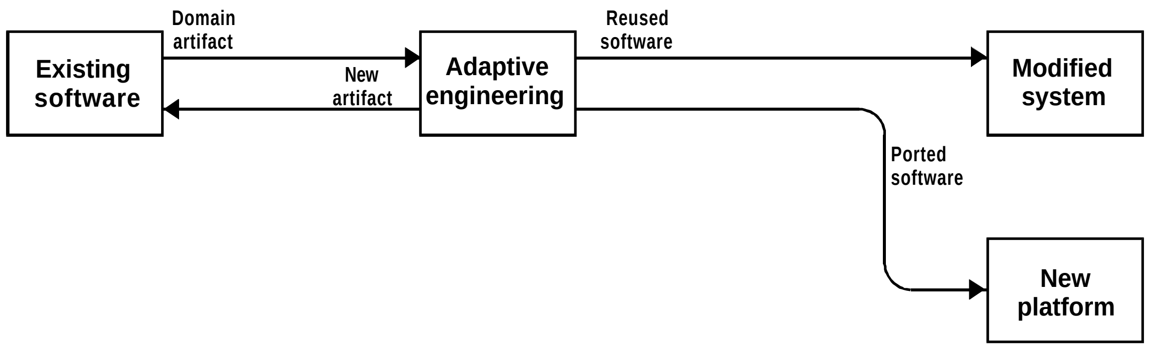
Each process offers its own set of benefits and risks. The *adaptive* approach (Figure 1-a) requires little new investment by an organization, and, can support new developments, provided they require only incremental changes from previous applications. However, applications that require major modifications and upgrades typical of most aerospace applications, will only achieve marginal benefits from adapting old software. The *parameterized* approach (Figure 1-b) establishes a framework for all new implementations, a major investment for an organization. There is a significant pay-off, provided the framework is stable. However, in areas with rapidly evolving technology, there is no stable framework. The investment in establishing standard products may be at risk if new requirements do not fit previously established standards. Like the parameterized approach, *engineered reuse* (Figure 1-c) requires a large investment. The domain resources must meet the requirements of a wide range of applications or this investment will also be at risk. If the resources are properly developed this approach offers a greater degree of flexibility than the parameterized method, and can adapt to changing requirements. The following discussion explores each approach in depth.

1 The Adaptive Approach to Software Reuse

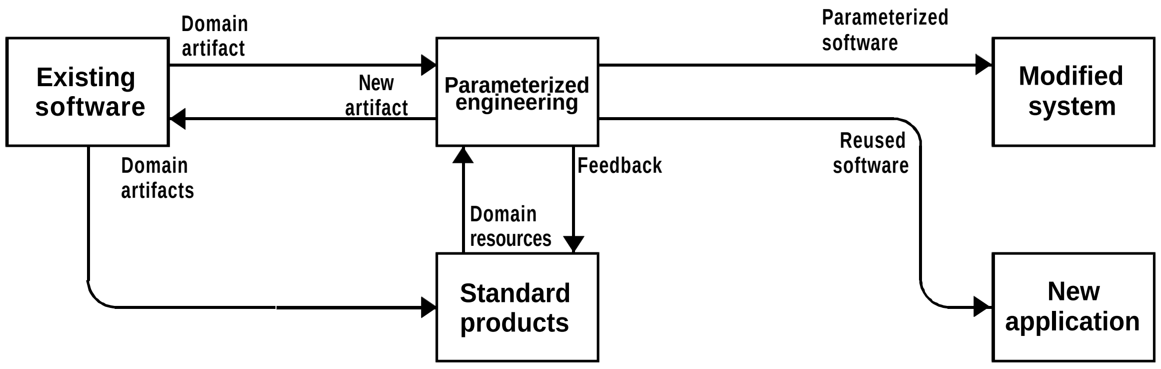
The most common form of reuse in practice is the sharing of design and code between similar projects. Most organizations performing software development build new applications using previous developments as a starting point. In some cases, such as prototyping or modification, this adaptive approach is planned, and reuse of software is a natural occurrence. In others, the ability to reuse software from one development to the next is assumed, but often not fully realized due to inconsistencies between current and previous projects.

In adapting previous designs to new requirements, developers must be aware that these new requirements will mean significant modification to the existing application. If the application was not built to support subsequent modification and reuse, the adaptation may be as costly as a complete redesign. This was the lesson learned from the P-7A project, and it applies in software development, as well.

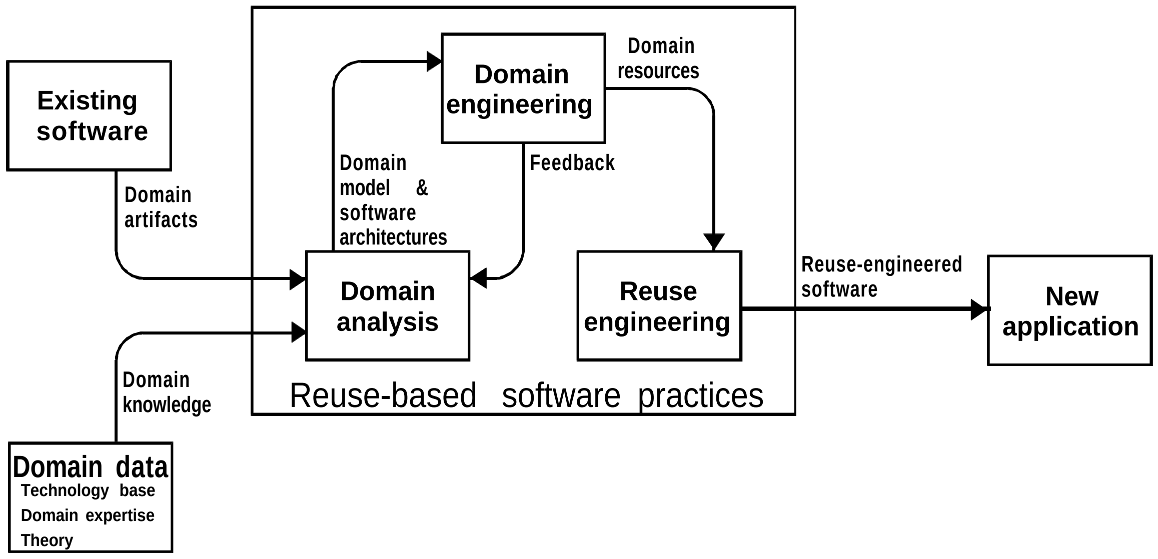
A process model for adaptive software reuse (Figure 1-a) shows different paths that this approach may support. For example, a developer may wish to modify an existing software system to incorporate new requirements. The software taken



(a) Adaptive Process



(b) Parameterized Process



(c) Engineered Process

Figure 1: Processes to support reuse

from the original system is "reused software," in the sense that it has come from a previous development. Similarly, when porting an existing system to a new platform, developers reuse the old software, making necessary changes to achieve compatibility. While the reuse of software not explicitly designed for reuse has been called "software scanvenging" such reuse is very common [Tracz 90].

The adaptive approach can be systematized for process improvement by:

- Applying standard software engineering techniques to build modular software to increase adaptability
- Classifying existing software by projects, and indexing that software to increase awareness and lead to more effective reuse
- Organizing test and evaluation software to support decisions about applicability
- Utilizing CASE tools and re-engineering to facilitate customization and integration of reused and new software

While the adaptive process can be ad hoc, these steps will improve the value of existing software and increase the ability to reuse that software.

2 Parameterized Reuse and Program Families

The identification of classes of program families can lead to significant levels of reuse. This approach has proved to be extremely successful in organizations that deliver large systems to a broad range of customers to meet similar sets of requirements. The successes in reuse attributed to the Japanese all stem from establishing a common architecture for a family of systems that can be applied through parameterization to a wide class of users.

This approach is illustrated with a particular program family, that of the Air Traffic Control (ATC) system. The reuse work on these systems done by Thomson-CSF shows the potential payoff of the parameterized form of reuse. The approach is illustrated in Figure 1-b. Under this process, standard products can be used to create new systems, through parameterization. A second use of these products is for more traditional reuse, following the adaptive approach.

2.1 Identifying the family

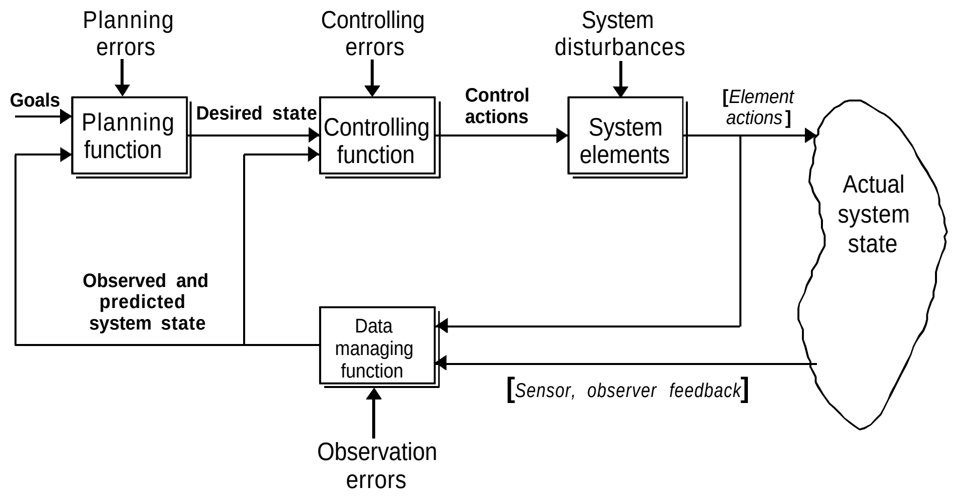
The successful identification of a family of related systems in the ATC domain is illustrated by by reuse technology developed by the Thomson-CSF company. The first step in applying the process to the ATC domain is to scope the domain according to its generic properties (i.e., what features do all systems in the domain have in common). The feedback diagram of Figure 2 illustrates the common features of most ATC systems.

The real world environment, labeled Actual System State, consists of all of the entities controlled or used by the ATC system. These entities include: aircraft, radars, flight planning operational input, weather reports, etc. The data managing function must handle these entities, organizing them into a coherent representation of the system state, and making them available to other system functions. The long-term planning function will use this data to establish the desired state for the area controlled by the ATC system over a long-term (on the order of hours) basis. The controlling function performs similar operations, over a duration measured in minutes. Commands from the controlling function to effect the desired system state go to system elements, such as aircraft or to the database to update the current state. These commands will, indirectly, update the actual system state, as well.

By illustrating the primary components of an ATC system, the feedback diagram provides support for scoping the domain for further analysis. In the work done by Thomson-CSF, the domain of interest covered three areas:

1. Radar operations: processing radar sensor return data to establish aircraft tracks and reports. This data is labeled as the sensor feedback from the actual system state in Figure 2 and is handled by the data management function.
2. Flight operations: flight plan, weather and other information forwarded to the ATC system. This includes the data labeled as observer feedback in Figure 2 and also covers output labeled as element actions on the figure, which is also fed back to the data managing function.

3. Operator interface: software facilitating the processing performed by ATC operators. This interface permits the operator to obtain current information about the system state, via the data managing function, and to perform controlling functions.



Goals	**smooth traffic flow, fuel-economic flight, etc.**
Planning	**formulates the desired state of the system based on a set of goals.**
Controlling	**estimates deviations between observed, predicted and desired states ... formulates ... actions ... to minimize ... deviations ... transmits to elements.**
Elements	**1) execute the control actions received subject to external disturbances; 2) aids to allow/ assist in execution of the required control actions.**
Data	**observes and records the system state and distributes this information to planning and controlling functions.**

Source: Hunt, V.R.; Zellweger, A. "The FAA's Advanced Automation System: Strategies for Future Air Traffic Control Systems." *Computer*, **February, 1987. pp. 26-27.**

Figure 2: Feedback Diagram Illustrates Components of Air Traffic Control Systems

2.2 Developing a parameterized architecture

The next step in the parameterization process is to establish common functionality and data flow within the domain areas defined within this scope. Figure 3 illustrates the need of all ATC systems for both flight plan and radar tracking data in order to provide operators with the information required to support air traffic control activities. The figure also provides information on the interfaces of components of the system and a context for each of the three domain areas.

Recognizing that this functionality is common across a range of systems built for ATC, Thomson-CSF has constructed a generic architecture of large scale Ada components to implement new systems. This architecture is represented in Figure 4, and currently accounts for 20% reuse among installed systems. The company hopes to achieve 50% reuse, the approximate figure also attained by Japanese firms, by supporting reuse within the operator interface subsystem.

The Thomson-CSF architecture uses three sets of reusable packages in constructing its ATC systems. These correspond to support packages for each of the three domain areas (radar, flight planning, operator interface) described above. The domain analysis approach also supports identification of commonality between the individual areas. This commonality is captured in a common utilities package that is a part of the architecture within each domain area.

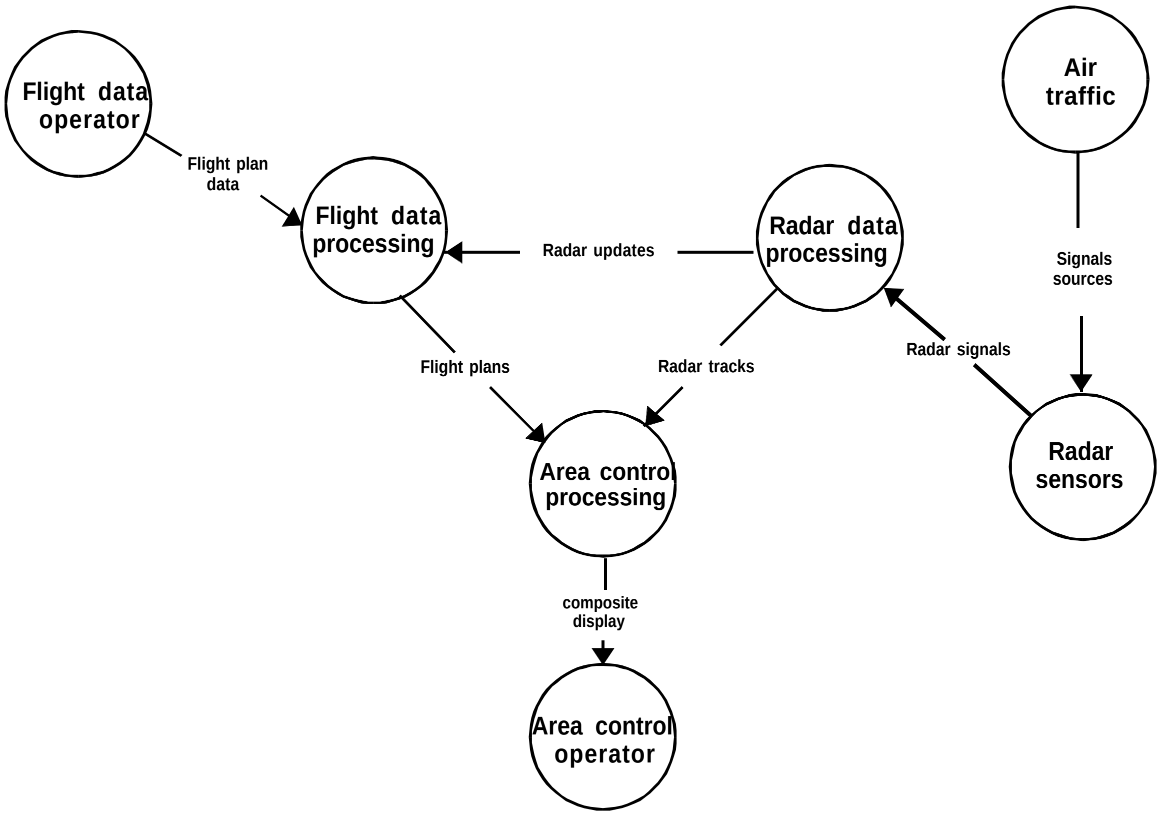


Figure 3: Context Diagram Illustrates Primary Data Flows of Air Traffic Control Systems

3 Engineered Reuse

While parameterized reuse is appropriate in mature domains, reuse in domains that rely on emerging technology cannot standardize at the same level of parameterization. Applications within such a domain, sometimes called "unprecedented applications," rely on new techniques such as domain analysis to establish commonality and enable software reuse. Implementation approaches, such as objected-oriented design and programming also play a significant role in the support of reuse. These techniques form the process of Engineered Reuse, as shown in Figure 1-c. Engineered reuse in terms of domain analysis is discussed below using the ATC problem as an example.

3.1 Domain Analysis

The systematic discovery and exploitation of commonality across related software systems is a fundamental technical requirement for achieving successful software reuse. Domain analysis is one technique that can be applied to meet this requirement. By examining related software systems and the underlying theory of the class of systems they represent, domain analysis can provide a generic description of the requirements of those systems. It can also propose a set of approaches for implementation of new systems in the class.

The development of complex aerospace software systems requires a clear understanding of desired system features and of the capabilities of those features. Software reuse, which has long promised improvements in this development process, will become feasible only when the features and capabilities within the domain of a particular system can be properly defined in advance of formal software development of the system. Thus, the need to develop domain analysis technology and apply it within specific software development programs is a factor that can promote software reuse for those programs.

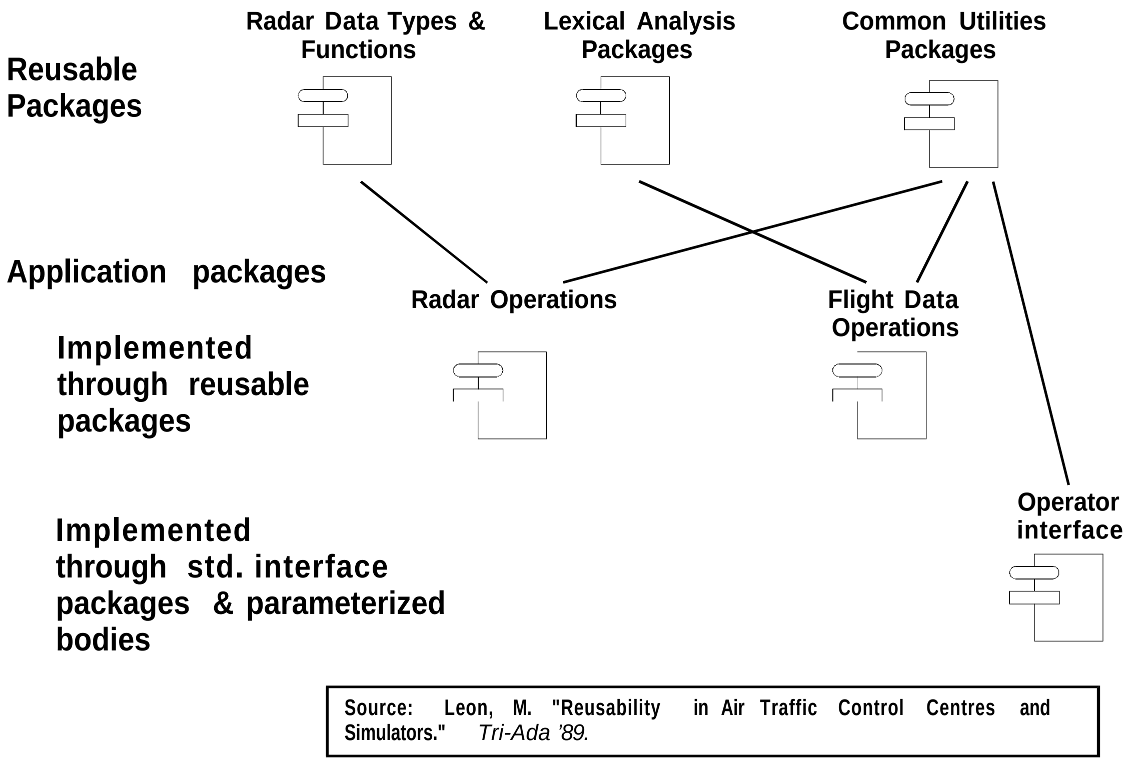

Radar Data Types & Functions
Lexical Analysis Packages
Common Utilities Packages
Reusable Packages
Application packages
Radar Operations
Flight Data Operations
Implemented through reusable packages
Operator interface
Implemented through std. interface packages & parameterized bodies
Source: Leon, M. "Reusability in Air Traffic Control Centres and Simulators." Tri-Ada '89.

Figure 4: Packaging Structure Illustrates Reusable Design Components

3.1.1 Domain Analysis Process

Domain analysis gathers and represents information on software systems that share a common set of capabilities and data. Three basic activities characterize this process:

- Scoping: defining a domain for analysis
- Domain modeling: providing a description of the requirements met by software within the domain
- Architectural modeling: creating the software architecture(s) that implements a solution to the problems in the domain

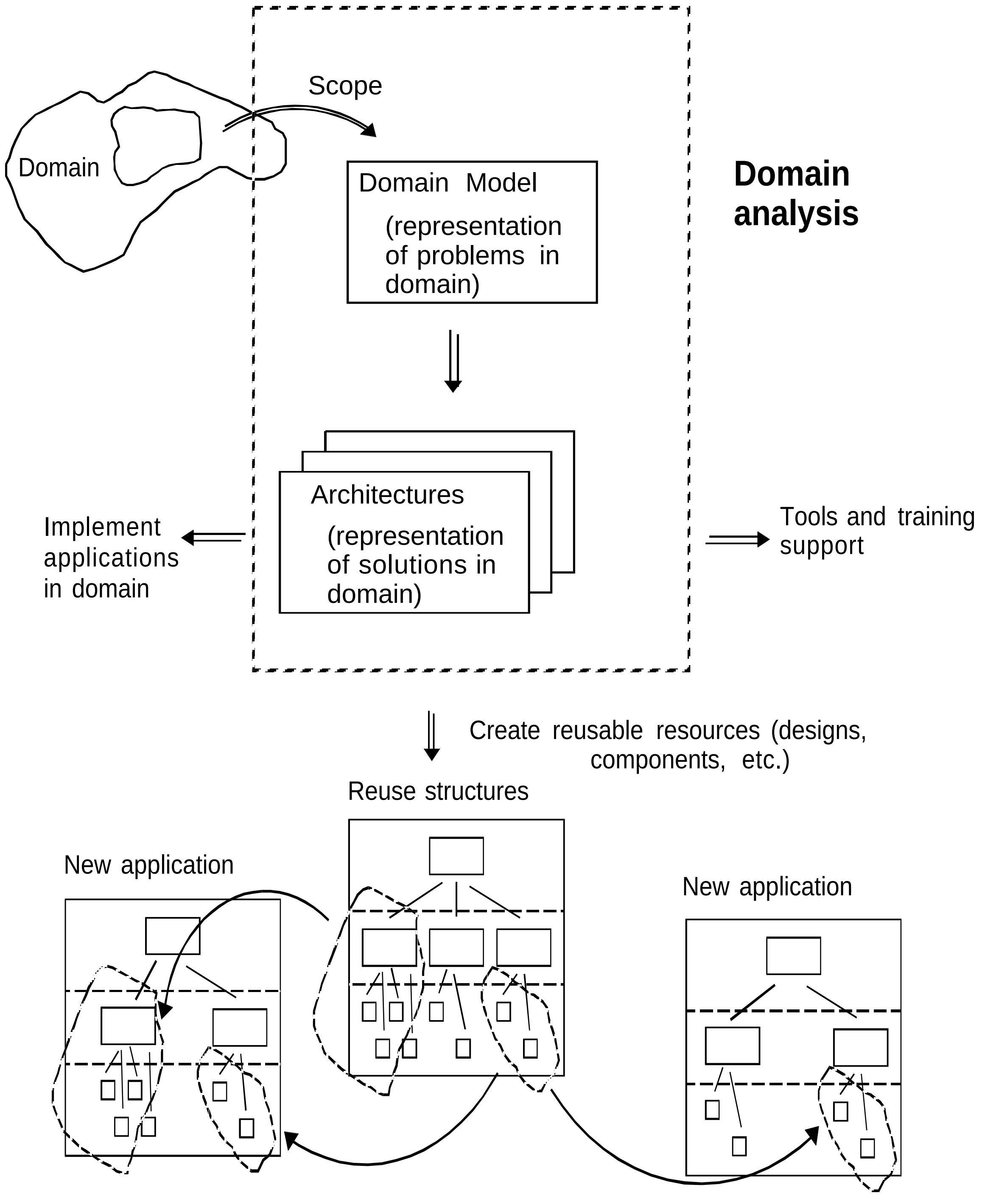
The domain analysis process must also be integrated into the more general process for software development. The domain analysis can then support implementation of applications in the domain, creation of reusable resources, and support for creation of domain tools and training. Figure 5 illustrates the support that domain analysis provides to the software development process.

3.1.2 Domain Analysis Products

The domain analysis method should provide specific representations to document the results of each of the domain analysis activities. These representations should describe the problems solved by software in the domain and architectures that can implement solutions.

For each of the three phases of the domain analysis process, there is a separate set of representations or views of the domain.

Scoping	The results of this phase should provide the context of the domain. This requires representing the primary inputs and outputs of software in the domain as well as identifying other software interfaces. Figure 2 illustrates the scoping of the ATC domain.



Scope
Domain
Domain Model
(representation of problems in domain)
Domain analysis
Architectures
(representation of solutions in domain)
Implement applications in domain
Tools and training support
Create reusable resources (designs, components, etc.)
Reuse structures
New application
New application

Figure 5: Domain Analysis Supports Software Development

Domain modeling

The products of this phase describe the problems addressed by software in the domain. The products provide the typical vocabulary of domain experts, document the entities embodied in software, and establish generic software requirements via control flow, data flow, and other specification techniques. Figure 3 supports domain modeling.

Architectural modeling

This phase establishes the structure of implementations of software in the domain. The purpose of these representations is to provide developers with a set of models for constructing applications. They can also guide the development of libraries of reusable components. Figure 4 provides the basis of a common architecture for ATC systems.

The next subsection describes a specific method for domain analysis that covers the three phases of domain analysis.

3.2 Feature-Oriented Domain Analysis

The Software Engineering Institute is currently refining a method for discovering and representing commonalities among related software systems [kang 90]. The *Feature-Oriented Domain Analysis* (FODA) method stresses the identification of prominent or distinctive *features* of a software system. These features are user-visible characteristics of the domain and include both common aspects of the domain as well as differences among related systems in the domain. Features are also used to define the domain in terms of the mandatory, optional, and alternative characteristics of these related systems, leading to the creation of a *feature model*. The feature model establishes rationale for selecting a feature, rules concerning the composition of two or more features, and the effect of *binding* time of a feature within an application.

3.2.1 The feature model concept

Figure 6 illustrates each of the concepts regarding features, their interrelationships and attributes:

- Horsepower and transmission are mandatory features. All cars must have a power supply and a way to turn that power into forward motion.
- Air conditioning is an optional feature.
- The selection of manual or automatic transmission represents a choice between alternative features.
- If the air conditioning feature is selected, it may require a more powerful engine, illustrating the concept of composition of features.
- The choice between transmission features could be based on issues such as fuel efficiency or ease of operation.
- The choice of transmission may be bound, i.e., installed, at the factory. Porsche, however, offers a special feature called the Tiptronic transmission, that offers binding of the transmission feature while the car is in operation. The driver may select automatic transmission for ease of operation in heavy traffic and can *shift* to manual for open road driving.

The next subsection describes the application of the feature method to domain analysis. A domain analysis of window manager software, fully documented in [kang 90], illustrates the use of the method.

3.2.2 Applying the feature method to domain analysis

The FODA method establishes a *context analysis* phase as the process for scoping a domain. There are two products of this phase:

1. Context diagram: data flow showing primary inputs and outputs of the domain. Figure 7 shows a context diagram for the window manager domain.
2. Structure chart: showing related software. For the window manager domain, the structure chart shows the relationship between window manager software, other applications, graphical user interface software (e.g., OpenLook, Motif), toolkits, and programming interfaces (e.g., Xlib), protocols, servers, and clients.

The products of the domain modelling phase define the operations, mission, and interfaces of the domain:

1. Entity relationship model: defining the entities (objects) of the domain. For the window manager, these include the screen, main window, icon, pointer and process. The model also defines relationships and constraints on these entities.

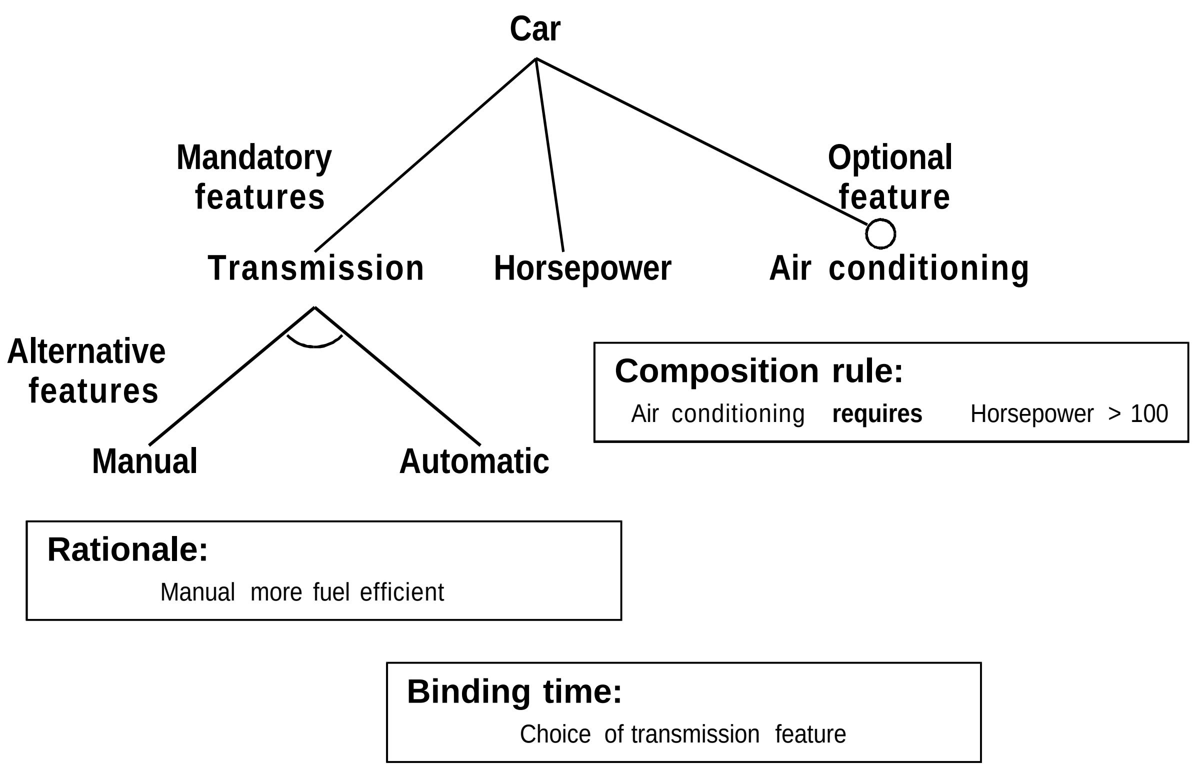


Figure 6: The Feature Model is Critical to the FODA Method

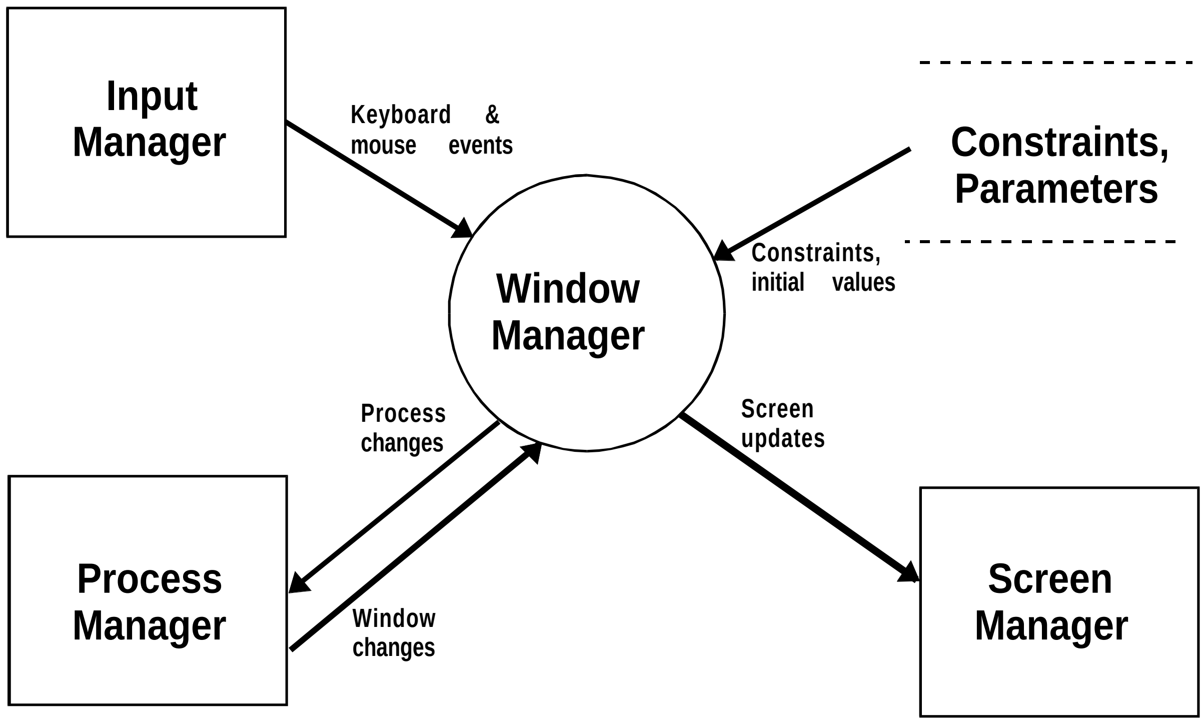


Figure 7: The Context Model Defines the Scope for Analysis

2. Feature model: shows the operations and mission of software in the domain. For the window manager the operations include: create, destroy, move, resize, iconify (optional), deiconify (optional), etc. This model also includes composition rules and textual documentation of feature definitions. For example, if a window manager has the optional iconify feature, it must also have the deiconify feature. Mission information includes information about user feedback, the ability to move windows off a screen, horizontal/vertical constraints on the move operation, etc. Figure 8 shows the model of the window manager domain for the move feature. The boxed in regions identify features of particular window managers. Note the selection of both alternatives under interactive feedback for the X10/uwm window manager; this illustrates a binding time attribute of the feature. A user may select either type of feedback before executing the move. In the other feature model, this binding to the feature occurs when the window manager is initiated; it cannot be changed during execution.
3. Functional model: control and data flow for each operation in the domain
4. Domain terminology: a dictionary of standard terms

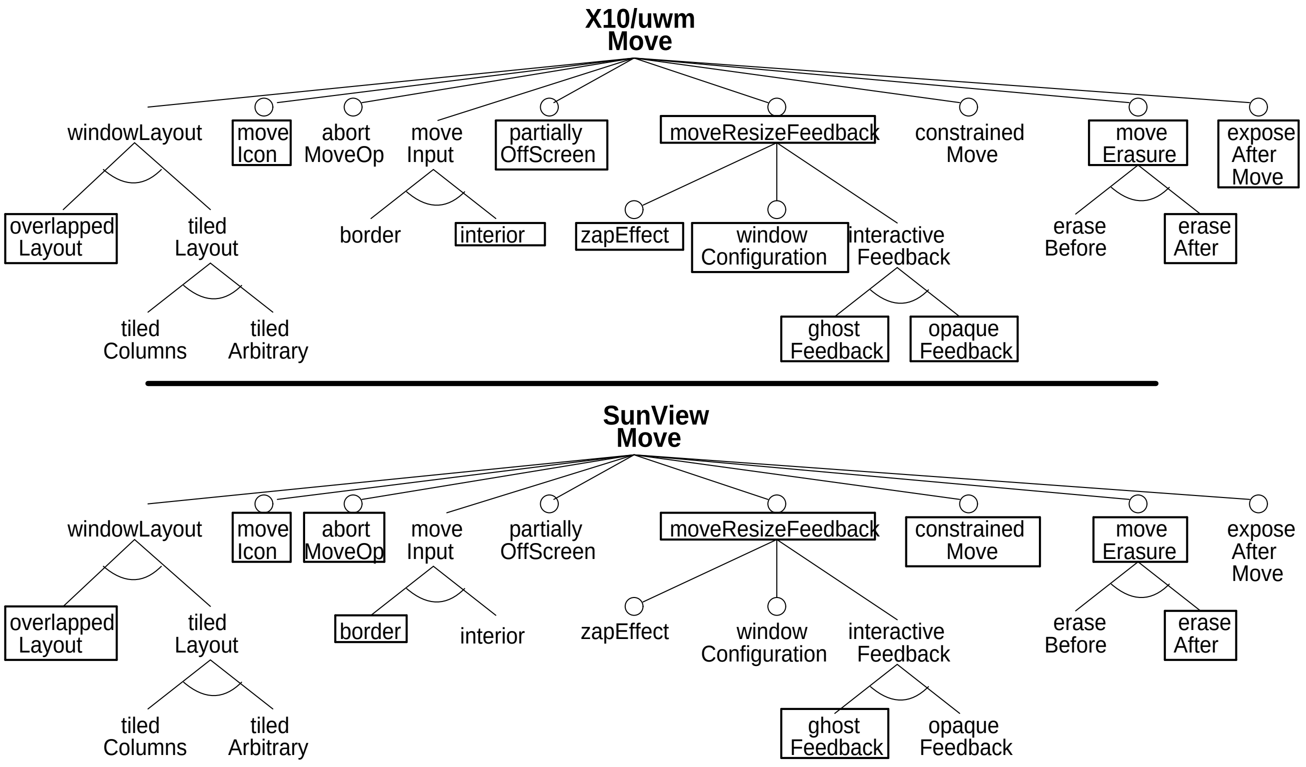


Figure 8: The Feature Model Defines the Requirements for Specific Systems

The FODA report presents a method for defining an architectural model of the domain. For the window manager domain, this architecture relies upon the Xlib and toolkit.

4 Lessons Learned

The methods and products of both the parameterized and engineered processes can lead to successful reuse. Under both approaches, the goal is to create a common structure for application development. For air traffic control systems, this structure was the basic architecture for a complete implementation; from this base level, up to half of a system can be built. Using these two approaches for guidance, reuse methods can succeed when:

1. The domain of applicability is appropriate, large enough to meet the needs of typical applications, yet tractable in scope;
2. The analysis attempts to abstract the requirements from the application level (ATC or window manager) to the problem level (control of system state); and,
3. The products provide documentation of the problem abstraction and guidance in tailoring the abstraction to meet specific requirements.

Nonetheless, software reuse does not come without some costs. The building of reusable resources, or of mechanisms to improve reuse of existing resources, requires a major investment. Furthermore, the reuse process must limit or eliminate inappropriate reuse. A failure in reuse engineering could result if: the wrong software is assumed capable of reuse in a new development, or reusable software is used incorrectly in a new development

There are several general goals that must be addressed to achieve a successful method for domain analysis, as well.

- Developing domain analysis products to support implementation of new applications. This goal will be met when domain analysis products are used in new implementations.
- Establishing domain analysis methods to produce these products. This goal will be met when domain analysis methods are incorporated into the software development process.

Given the results of domain analysis, the process of engineered reuse must now create specific resources--software components. The application of reuse design principles is covered in several other technical reports [Cohen 90, Palmer 90].

A model of software reuse must include both a process and products to ensure the availability of the correct software and support for using it correctly. The software engineering community must identify a standard process for developing and applying software for reuse to overcome technical barriers to the widespread adoption of reuse. The product development must promote a common understanding of: (1) the problem space of applications in a domain; and (2) the solution space for implementing new applications.

To establish an appropriate model for reuse, the software engineering community can take the following steps:

- identify major application areas appropriate for establishing a model of the problem domain and its context
- perform analyses of these problem domains (i.e., domain analysis)
- establish one or more generic architectures that can be used for implementation of applications in each domain (i.e., architecture modelling)
- establish and adopt a common approach to design for reuse in Ada
- build resources to support reuse (e.g., code generators, components, reference models) within and across domains
- provide guidance to the practitioner through handbooks that capture reuse knowledge and document the means of applying that knowledge [Traub 89].

These steps can help advance the state of the technology of reuse from its current *ad hoc* position to a managed and controlled state. Moreover, by building reuse into an overall model for software development, these steps will promote an approach to software engineering that is built upon reuse technology.

References

[Cohen 90] Sholom G. Cohen.
Ada Support for Software Reuse.
Special report CMU/SEI-90-SR-16, Software Engineering Institute (also available via ftp from ajpo.sei.cmu.edu), Pittsburgh, PA, October, 1990.

[kang 90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, A. Spencer Peterson.
Feature-Oriented Domain Analysis (FODA) Feasibility Study.
Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, PA, November, 1990.

[Palmer 90] Constance Palmer, et al.
Developing and using Ada parts in real-time embedded applications.
CAMP-3 Report Contract No. F08635-88-C-0002, McDonnell Douglas Astronautics Company, St. Louis, MO, April, 1990.

[Tracz 90] William Tracz.
Where does reuse start?
ACM SIGSOFT Software Engineering Notes 15(2):42-46, April, 1990.

[Traub 89] Joseph Traub.
Scaling up: a research agenda for software engineering.
National Research Council. National Academy Press, Washington, D.C., 1989.

Table of Contents

List of Figures