FUTURE OF COMPUTING

Components lie at the very heart of the future vision of computing. Corporations expect that they soon will be running their businesses using Web-enabled, enterprise business applications composed of predefined, replaceable components that are distributed over networks. Although part of the application may run on a client, part on the middle tier, and another part on a back-end database server, its comprising components, written in different languages and supplied from multiple sources, will be able to work together to perform the application’s services.

Such component-based applications offer the advantages of being both easily customized to meet current business needs, and easily modified to meet changing business needs in the future. Also, they leverage a corporation’s investment in its legacy systems by incorporating valuable existing functionality that has been wrapped into reusable components. These applications are typically composed of an interacting mixture of predeveloped components that preserve core functionality of the business, and new components that take advantage of the newest technologies, such as the Internet.

COMPONENT DEFINITION

A component can be thought of as an independent module that provides information about what it does and how to use it through a public interface, while hiding its inner workings. The interface identifies the compo-
component, as well as its behaviors and interaction mechanisms. Examples of components include small-grained, desktop-bound visual components (e.g., GUI widgets), as well as large-grained components that capture a complete business function (e.g., a shipping component that includes order entry and shipment functionality).

The idea of components is not new. Rather, earlier software engineering concepts are fundamental to the component concept, such as program modularization, structured programming, and information hiding, which were introduced in the 1960s by Edgar Dijkstra, David Parnas, and others.

COMPONENT-BASED DEVELOPMENT
The software development approach used to build a component-based application is called component-based development (CBD). This reuse-based, architecture-centric form of software development uses an assembly approach, whereby software products are constructed by assembling components designed to fit into a predefined architecture.

Although CBD may appear to be a simple, obvious approach to software development, it is complicated by its need for standards. To work in practice, CBD demands standardization of the components themselves, as well as the software life cycles by which they are developed, and in which they are used to develop component-based applications. Standards are the only known way to ensure the quality of the components and of the life cycle processes. In particular, process standards that capture proven software practices are considered the best means available to ensure quality.

Recently, the IEEE Computer Society published a process standard applicable to CBD. The IEEE Std. 1517 — Standard for Information Technology — Software Life Cycle Processes — Reuse Processes defines the elements of reuse-based software life cycle processes that can be used to define an organization’s CBD approach. This standard defines the elements of a software life cycle for developing and maintaining component-based applications, as well as a separate life cycle for identifying, selecting, developing, and maintaining components.

IEEE STANDARD 1517 — REUSE PROCESSES
As the software life cycle process standard for CBD, IEEE Std. 1517 provides the requirements specification for practicing reuse and CBD on an enterprisewide basis. It not only identifies the processes involved in practicing software reuse, but describes at a high level how the processes operate and interact during the software life cycle. As shown in Exhibit 1, this standard organizes the software life cycle processes into a framework that consists of four process categories: primary, supporting, cross-project, and organizational.
The cross-project category is a new category of life cycle process introduced by this standard to define the special life cycle requirements for selecting or building appropriate, high-quality components. As software, components are similar in many ways to software applications. However, a component is different in that it is by its very nature a part, whose primary function is its use as a building block in multiple software applications. This multi-use characteristic of components places new requirements on the software life cycle in activities such as planning, analysis, design, and testing. To meet these requirements, the IEEE Std. 1517 defines a software life cycle relevant to components.

As defined in the standard, domain engineering describes the cross-project software life cycle for components. This article describes the elements of domain engineering as specified by the standard.

DOMAIN ENGINEERING
Domain engineering is the process of supplying components for use in a particular domain. As the life cycle for components, it is concerned with the analysis, design, selection, development, and maintenance of components belonging to a particular domain. The notion of domain narrows CBD down to a size that is conceptually easier to understand, technically easier to implement, and organizationally easier to manage.

The IEEE Std. 1517 defines a domain as “a problem space,” which typically represents one organizational segment that has potential for CBD. The organization determines how to segment itself for CBD, be it by product lines, by business functions, or by technology platforms. According to the standard, a domain should be defined broadly enough to encompass components that may be applied to multiple software applications over a period of time. Since organizations generally practice CBD in multiple domains, the domain engineering process will usually be applied multiple times within an organization.
Supplying Components

Since these components are intended for use by many software projects, the domain engineering process generally takes place higher than the project level. To meet multiple project requirements, components must possess common properties that can be shared and reused by the component-based applications produced. It is because the domain engineering process exists beyond the boundaries and duration of one project that IEEE Std. 1517 categorizes domain engineering as a cross-project life cycle process.

In the IEEE Std. 1517, domain engineering is defined as follows:

A reuse-based approach to defining the scope (i.e., domain definition), specifying the structure (i.e., domain architecture), and building components (e.g., requirements, designs, software code, documentation) for a class of systems, subsystems, or applications. Domain engineering may include the following activities: domain definition, domain analysis, developing the domain architecture, and domain implementation.

The standard names the domain engineer as the party responsible for performing domain engineering activities, which may be an individual or a group of individuals. In most organizations, a project team will perform the domain engineering function. It will be assembled at the beginning of the project and remain throughout the domain engineering life cycle.

The IEEE Std. 1517 defines the domain engineering process as the set of activities listed in Exhibit 2.

1. Process Implementation

The purpose of the process implementation activity is to formally prepare for domain engineering by creating a plan. This defines the format

<table>
<thead>
<tr>
<th>Activity Name</th>
<th>Activity Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process implementation</td>
<td>• Create, document, and execute domain engineering plan</td>
</tr>
<tr>
<td></td>
<td>• Select representation forms for the domain models and domain architectures</td>
</tr>
<tr>
<td>Domain analysis</td>
<td>• Define domain boundaries and relationships with other domains</td>
</tr>
<tr>
<td></td>
<td>• Identify needs of domain software developers</td>
</tr>
<tr>
<td></td>
<td>• Build the domain models</td>
</tr>
<tr>
<td></td>
<td>• Construct the domain vocabulary</td>
</tr>
<tr>
<td>Domain design</td>
<td>• Create the domain architectures</td>
</tr>
<tr>
<td>Asset provision</td>
<td>• Develop the domain components</td>
</tr>
<tr>
<td>Asset maintenance</td>
<td>• Maintain the domain components</td>
</tr>
</tbody>
</table>
to be used in representing domain engineering outputs, as well as the technical and management procedures that will take place.

**Creating a Domain Engineering Plan.** Because the plan should treat domain engineering as a project to be properly managed, it should address such project constraints as budget, schedule, and resources. It should include both technical and nontechnical resources needed to perform domain engineering. On the technical side, it should identify tools, methods, and standards. On the nontechnical side, it must define activities, assignments, and responsibilities.

A domain engineering team will have responsibility for the project. The team members should provide the roles and skills listed in Exhibit 3. Because domain experts are a vital source of information about the domain, they are a critical part of the team. IEEE Std. 1517 defines a domain expert as “an individual who is intimately familiar with the domain and can provide detailed information to the domain engineers.” Domain experts can include knowledgeable end users and software professionals. The latter should be software system developers and maintainers who are experienced with software products that are similar to those being built for this domain. They should also know which properties are important for future software applications planned for the domain.

The domain engineering plan should also identify the tools to be used in supporting the domain engineering effort. Different types of tools will support analysis, design, and implementation activities. Some tools that are useful for performing domain engineering support the building and analysis of strategic planning models and information architectures, such as entity relationship, data flow, object modeling diagramming tools, and dictionary tools. Others include data synthesis tools, data and process reverse engineering tools, program code analyzers, flow graphing tools, complexity metrics tools, and process logic and data rationalization restructuring tools.

The specific tools that are appropriate for a particular project depend on the types of models and code to be analyzed, and the representation

---

**EXHIBIT 3 — Domain Engineering Team Responsibilities**

<table>
<thead>
<tr>
<th>Team Member</th>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systems/business analyst</td>
<td>Expert in analysis modeling and data synthesis</td>
</tr>
<tr>
<td>Data administrator</td>
<td>Responsible for corporate data dictionary and naming standards</td>
</tr>
<tr>
<td>Information architect</td>
<td>Knowledgeable about the information/enterprise architecture for the enterprise or business unit to which this domain belongs</td>
</tr>
<tr>
<td>Domain expert</td>
<td>Expert-level knowledge and understanding of the domain</td>
</tr>
<tr>
<td>End users and software developers</td>
<td>Knowledgeable about their current and future system needs</td>
</tr>
<tr>
<td>Reuse facilitator</td>
<td>Experienced in performing domain analysis</td>
</tr>
</tbody>
</table>
forms chosen for the domain models and architectures. In addition, repositories, browsers, cataloging tools, and configuration management tools are needed to store and manage the domain model, the domain architectures, and other types of domain components.

**Defining the Domain Model and Domain Architecture Formats.**

The next major task in the process implementation activity of a domain engineering project is to define representation forms for the domain models and architectures. In the IEEE Std. 1517, the definitions for a domain model and domain architecture are as follows:

Domain model: a product of domain analysis that provides a representation of the requirements of the domain. The domain model identifies and describes the structure of data, flow of information, functions, constraints, and controls within the domain that are included in software systems in the domain. The domain model describes the commonalities and variabilities among requirements for software systems in the domain.

Domain architecture: a generic, organized structure or design for software systems in a domain. The domain architecture contains the designs that are intended to satisfy requirements specified in the domain model. The domain architecture documents design, whereas the domain model documents requirements. A domain architecture: 1) can be adapted to create designs for software systems within a domain, and 2) provides a framework for configuring components with individual software systems.

Thus, the domain model is a generic analysis model, while the domain architecture is a high-level design model. Together, they provide an excellent starting point and guide for building components and component-based-applications for the domain.

The representation forms used for the domain models and architectures affect the choice of methods for supplying components, as well as the development tools to be used for CBD projects. Therefore, the representation forms selected should fit the available domain analysis and design approaches and tools. For example, the Features-Oriented Domain Analysis (FODA) approach uses one set of models to represent the domain model, including the entity relationship model, data flow diagram, and state transition diagram and structure diagram. Another set of models represents the domain architecture, including the process interaction model and module structure chart.

In addition, the representation forms should also fit the CBD methodologies and corresponding analysis and design models that will be used to develop component-based applications within this domain. When an organization uses the same kinds of representation forms in both places, there is a reduced learning curve, less need to convert from one representation form to another, and less need to acquire different tools to support domain engineering and CBD.
Organizations should consider a common modeling language such as the Unified Modeling Language (UML) from the Object Management Group for both the model and the architecture representation forms. The UML provides a common notation and semantics to model frameworks, distributed systems, and component-based applications, and to facilitate model interchange across teams and tools.

2. Domain Analysis Activity

Like the traditional software system life cycle, the domain engineering life cycle covers analysis, design, implementation, and maintenance activities. In this case, however, the cycle is applied to domain models, domain architectures, and other types of software parts that are used to assemble component-based systems.

Domain analysis is the analysis phase in the component life cycle. The IEEE Std. 1517 defines domain analysis as follows:

(A) The analysis of systems within a domain to discover commonalities and differences among them. (B) The process by which information used in developing software systems is identified, captured, and organized so that it can be reused to create new systems, within a domain. (C) The result of the process in (A) and (B).

Domain analysis analyzes, abstracts, and models the characteristics of existing and envisioned component-based applications within a domain to determine what they have in common (their commonality) and how they differ (their diversity). This information is captured in a set of domain models during domain analysis.

The purposes of the domain models are to:

1. aid in the understanding of the domain’s essential common elements and the relationships that exist between these elements
2. define a domain vocabulary to create a common understanding of the domain
3. capture the essential common and differentiating features, capabilities, concepts, and functions in the domain.

Defining Domain Boundaries. The IEEE Std. 1517 requires that the domain analysis activity define the boundaries of the domain in which the domain engineering process is being performed, in terms of which functions, features, properties, and capabilities are included in and excluded from the domain. Relationships should also be established, such as when one domain is a subset of another. This “domain definition” is needed to create and, if necessary, redefine the domain models.

The domain boundaries should be iteratively refined by comparison with other domains. Domain boundaries can also be determined from
market analysis information, customer requirements, software developers, and domain experts, as well as the enterprise architectures.

A context model, data flow model, or object model can be used to show the boundaries of a domain. A data flow model or structure chart can be used to show relationships between domains. The models showing the domain boundaries should be the same representation forms selected for the domain models and architectures.

**Identifying Developers’ Needs.** The IEEE Std. 1517 also requires that an organization identify the needs of the software developers who assemble the component-based applications, since they will likely be the primary users of the domain engineering outputs. Due to their experience, these developers can best identify the most useful components in performing development. Not only will components that they used in developing previous applications be needed in future versions and implementations of these applications, but these components can be used in new applications with similar features or capabilities. In addition, the components that developers find useful in building current and future component-based applications for the domain can help to define or refine the domain boundaries.

Therefore, the organization should interview the developers of component-based applications for the domain to identify the components that they believe would be most useful in their software projects. Also, the organization should evaluate the reuse experience and expertise of the domain software developers to determine if they have the skills for using components in their work. Based on this information, the organization may need to adjust the domain boundaries to ensure that a particular domain engineering project will identify components that the domain software developers will use in their work.

**Building Domain Models.** The primary task of domain analysis is building the domain models. Domain analysis is typically performed iteratively, as a combined top-down/bottom-up analysis activity. During the former, the team studies existing systems and models to identify common structures, grouping them for further study. The team identifies both the common and variable aspects of each group in order to create a generic structure that represents the group’s properties.

During bottom-up domain analysis, the team identifies common components in the systems and models studied. It also identifies relationships between common components, such as generalization (“is a”) and aggregation (“consists of”). The team then maps common components to the generic structure.

**Constructing the Domain Vocabulary.** The domain vocabulary, constructed during domain analysis, is the foundation for recognizing com-
monalities between domain components. This vocabulary enables domain engineers and software developers to speak the same language, so they can more easily recognize and understand which components will be the common building blocks in constructing the domain’s component-based applications.

The domain vocabulary can be created through discussion with the domain experts. It can also use concepts, keywords, nouns, and verbs contained in the existing software system documentation, analysis, and design models; and in the enterprise and business models that pertain to this domain. The domain vocabulary should be refined during the iterative process in which the domain models are created.

3. Domain Design Activity
The purpose of domain design is to create the domain architecture and the design specifications for domain components.

Building Domain Architectures. The domain architecture offers a common, generalized framework for assembling component-based applications from components. It provides:

1. the general structure for assembling components into a component-based application
2. the impetus to use existing components
3. a guide to selecting or creating new components
4. an aid in understanding the domain’s essential common components and the relationships that exist between components

Because domain components are designed to fit with the domain architecture, the latter acts as the “glue” to integrate components into a working application. However, the organization may need to develop more than one domain architecture for the domain if different target environments are required for its component-based applications. Thus, distributed versus host-based applications require different architectures.

To design the domain architecture, the organization should use the knowledge of architecture experts and software developers with experience in building this kind of software architecture. In addition, the domain model should be considered an important input in domain design. The generic structures of the design models are the basis for creating the domain architecture. They will either become the domain architecture, or a subsystem within the domain architecture.

A domain architecture must be generalized, standardized, and documented so it can be used in building multiple software products. Ways to generalize the domain architecture are listed in Exhibit 4.
Ways for standardizing the domain architecture are listed in Exhibit 5. The organization should validate the usability of the domain architecture, determining if it could have been used as a starter design for this product. It should compare the domain architecture with the design of at least one existing software application that belongs to this domain but was not studied during the domain engineering analysis activity. In addition, a domain expert who is not a member of the domain engineering team should review the domain architecture.

**Developing Component Design Specifications.** According to the IEEE Std. 1517, another important domain design task is to create design specifications for the domain components. Since a subset of all possible components for use by CBD projects is generally sufficient, organizations would be wise to use a selective reuse strategy to focus on the most appropriate domain components. Selective reuse singles out for development or acquisition those components that have the highest reuse potential in the domain, because they:

1. can be used the most frequently in the domain CBD projects
2. provide the greatest benefits to the organization (e.g., cost savings, time savings, reducing the risk for CBD project failure, enforcing standards)

**EXHIBIT 5 — Ways to Standardize a Domain Architecture**

- standardize the interfaces between components (e.g., standard interface between the application system and the database management system; standard protocol between the application system and the communications software)
- focus on subsystems (e.g., the communications subsystem) and their interactions
- use a standardized modeling language such as UML
3. can be used to assemble component-based applications that are of the greatest strategic importance to the organization
4. have been requested by software developers and maintainers

For ideas on which particular component design specifications to create, organizations should study commonalities (e.g., common features or services) in the domain’s software applications that are currently being built or reengineered. They also should ask software developers for their suggestions and contributions, and review components that vendors are currently offering in the marketplace.

Exhibit 6 lists several ways to identify which components have the highest reuse potential in CBD projects.

Exhibit 7 shows the design specification information that should be created when a component is to be used in CBD projects. The design specification is then used to build the component or to help in selecting an existing component.

EXHIBIT 6 — Ways to Identify Highly Reusable Components

1. the potential number of times the component can be used in building software products in the domain
2. the strategic importance of each component-based application in which the component can be used
3. the similarities and differences expected in these component-based applications
4. the impact of these differences on the reuse potential of the component and the reuse benefits of the component
5. the ability to create the component in a way to accommodate expected differences and capture expected similarities over its reuses
6. the ease of certifying the reusability and overall quality of the component
7. the cost to create/reengineer/acquire the component
8. the life expectancy of the component compared against the time to produce/supply the component; and the cost to manage and maintain the component over its lifetime
9. the number of times the component must be used to recover its lifetime cost
10. the business benefits that use of the component may provide (e.g., faster time to market)
11. the ease of fitting the component into the domain architecture

EXHIBIT 7 — Component Design Specification Information

1. the function performed by the component
2. what the component expects from its client components
3. what the component produces for its client components
4. performance characteristics of the component
5. the extent of commonality and variability required by the component
6. assumptions about the target environments of the component
7. limitations on the use of the component
4. Component Acquisition and Development Activity

Components that have been selected as potential CBD building blocks are acquired or developed in this part of the domain engineering process. The general requirements and standard software engineering practices followed when acquiring or developing software also apply to components.

Building Components. Exhibit 8 shows the activities in the IEEE Std. 1517 development process, while Exhibit 9 shows the activities in the standard's acquisition process. An organization should use these activities as a requirements specification to create or select an appropriate CBD approach that can aid in the development of high-quality component-based applications.

Likewise, when creating a component, the organization should follow the IEEE Std. 1517 development process; and, when acquiring a component, it should follow the standard's acquisition process. However, because it may be overkill to apply all of the standard's process activities when creating or acquiring a component, the organization should tailor the processes by eliminating some of the activities. In this way, it can better meet the scope, magnitude, complexity, and criticality of a component-based project, as opposed to an ordinary software application.

Building and Acquiring Reusable Components. Because of its multiple-use capability, a component should be viewed as a special kind of software that has properties over and above those normally expected in a software product such as an application system. For example, since any software product is expected to be of high quality, emphasizing that each component is thoroughly specified, documented, efficient, and tested will help ensure the component's general quality. However, to be reusable, a component must exhibit additional characteristics, such as portability, interoperability, understandability, and maintainability.

If the component is to be developed, then the domain engineer should build into it general quality and reusability characteristics. If the component is to be acquired, then the domain engineer should require general quality and reusability selection criteria.

The IEEE Std. 1517 defines reusability as follows:

The degree to which a component can be used in more than one software system, or in building other components. In a reuse library, those characteristics of a component that make it easy to use in different contexts, software systems, or in building components.

Generalizing a Component. Two important characteristics that enable reusability are generalization and standardization. When developing a component, the domain engineer should generalize the component to enable its use in multiple software products. A generalized component
<table>
<thead>
<tr>
<th>Primary Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquisition</td>
</tr>
<tr>
<td>Supply</td>
</tr>
<tr>
<td>Development</td>
</tr>
<tr>
<td>Operation</td>
</tr>
<tr>
<td>Maintenance</td>
</tr>
</tbody>
</table>

- Process Implementation Activity
- System Requirements Analysis Activity
- System Architectural Design Activity
- Software Requirements Analysis Activity
- Software Architectural Design Activity
- Software Detailed Design Activity
- Software Coding and Testing Activity
- Software Integration Activity
- Software Qualification Testing Activity
- System Integration Activity
- System Qualification Testing Activity
- Software Installation
- Software Acceptance Support Activity
EXHIBIT 9 — Activities Specified in the IEEE Std. 1517 Acquisition Process

Primary Processes

- Acquisition
  - Initiation
  - Request-for-Proposal Preparation
  - Contract Preparation and Update
  - Supplier Monitoring
  - Acceptance and Completion

- Supply
- Development
- Operation
- Maintenance
Standardizing a Component. Standardization makes CBD easier. Opportunities for reuse are created by standardizing such software features as menus, GUIs, help functions, and error handling. When components implement features in accordance with the standards, they can be used to enforce the standards. In addition, if a component complies with the organization’s documentation, interface design, and testing standards, its reusability is increased because of its better quality and general usefulness.

Exhibit 10 provides some general reusability criteria to follow when selecting a component for acquisition, or developing a new component.

5. Component Maintenance Activity

Like any software product, a component must be maintained over its lifetime. However, because of its multiple-use capability, any change made to a component has broader implications than changes made to a single-
use software part. The impact of the change must be considered not only in the context of software products for which the request was made, but to all other current and future software products that use or might use the component.

For example, a change made to a component may compromise its conformance with the domain models and architecture. It may then be difficult to use the component in building future component-based applications in the domain because the modified component no longer fits easily into the domain architecture. In addition, the reusability of a component may be greatly diminished if a change to it adversely affects its generality or adaptability.

An organization should develop component acceptance and certification procedures, using them as the basis for analyzing the appropriateness of a component modification request, and choosing how to satisfy the request. For example, a component modification request should be rejected if this is the only way to preserve the component's reusability, and to protect the integrity of the component-based applications in which the component has been used.

According to the IEEE Std. 1517, the domain engineer should use the domain engineering life cycle (i.e., the IEEE Std. 1517 domain engineering process) to maintain a component. However, only a subset of the domain engineering activities will be needed to modify an existing component. Therefore, those performing component maintenance should do so by selecting the appropriate domain engineering activities and tasks for a "mini" version of domain engineering.

SUMMARY
A collection of high-quality, reusable components is a prerequisite for practicing CBD. An organization may create a component inventory by developing and acquiring the components. Since components are intended to be building blocks in developing multiple component-based applications, their life cycle requirements differ from single-use software parts.

A new standard from the IEEE Computer Society has been developed to specify the life cycle requirements for CBD approaches and components. To meet the CBD requirements, the standard modified the specification for the traditional software life cycle by adding activities, tasks, and deliverables to guide the assembly of a software application from predefined components, and adding a new life cycle called domain engineering. Exhibit 11 lists the activities that comprise the IEEE Std. 1517 domain engineering life cycle.
References


Dr. Carma McClure is vice president of research at Extended Intelligence, a Chicago-based company offering reuse methodology products, process management, and reuse training and consulting services. Dr. McClure has lectured about and consulted on software technologies such as maintenance, reengineering, CASE, repositories, and reuse for major corporations, several governments, and military organizations around the world.