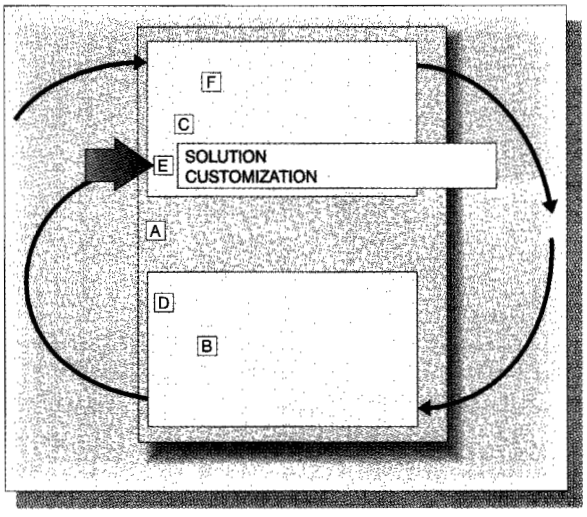
Solution customization

by D. A. Leishman



Customization involves* fit *and* alterability *and is based on understanding the commonality and variability (c/v) across industries, geographies, customers, and systems. This paper argues for an emphasis on c/v through a* customization life cycle, *from engineering customizable assets, components, and solutions to supporting their effective deployment. Examples of systems that focus on customization through c/v are given. These examples are described using the customization life cycle and show what mechanisms are most useful in each phase.

Today enterprises are facing many forces that compel them to take a larger view of their systems. These forces include globalization, "buyouts," regulatory changes, commerce, cost, multiple customer-access channels, product development cycles, changing business processes, etc. This larger systems view is creating a need to break down the application "silos"[1] that exist today in a cost-effective and low-risk manner. Companies are asking for help from their own internal information systems (IS) organizations as well as from external services consultants, product developers, and packaged solutions vendors.

External services consultants (Andersen Consulting, Electronic Data Systems, IBM) can bring many resources to help in creating customer solutions, from intellectual-capital assets, to hardware and software products and components, to packaged solutions, and finally to newly emerging component-based packaged solutions. For effective and profitable use, these items must be customizable to fit customers' requirements and environments. Developers of these customizable resources need guidelines: what is customization and how can it be implemented successfully?

Customization involves *fit* and *alterability* and is based on understanding the commonality and variability (c/v) across industries, geographies, customers, and systems. This paper introduces a *customization life cycle*, and through examples shows what mechanisms are most useful in each phase of the life cycle. A subsequent section of the paper focuses on the mechanisms themselves.

Successful, profitable creation and deployment of solutions across multiple customers, industries, and geographies is dependent on observing this customization life cycle. Focus on a customization life cycle and c/v is already beginning to permeate the customizable solutions developed in IBM, and this work will continue.

0018-8670/99/$5.00 © 1999 IBM

What is customization?

This section gives a definition of customization, describes the notion of scope, and shows that customization can apply at multiple levels. A "value chain" view of customization is presented and shows the interaction between defining, developing, deploying, and maintaining assets, components, and customizable solutions. These ideas form the backdrop in which to think about creating systems that are easily customizable and maintainable.

Definitions. The dictionary definition[2] of "customize" is "to build, fit, or alter according to individual specifications." For the customer solutions we develop, this implies (1) building "from scratch," (2) fitting an existing solution into a customer's environment, (3) altering a solution to fit the customer's requirements, or any combination of the three. The key is to fit to the customer's specification through either "green field" (new) development—by making sure that a packaged solution fits the customer very well and needs only minimal customization—or being able to easily configure a solution from existing customizable assets and components.

Services consulting businesses today are proficient in building custom solutions through green field development and systems integration. Services consultants that deploy ERP (enterprise resource planning) packaged solutions, such as SAP** or PeopleSoft**, often do extensive customization. Services and solutions businesses of the future will be focused on fitting and altering existing assets, components, and solutions to meet customer requirements.

To provide fit and alterability requires, for one thing, having the right parts to allow *configurable* systems. Examples of these parts include common services, data models, components, and objects. To fit the customer, a system must also be extensible and able to interoperate with existing systems. Thus customizability in the fullest extent means being configurable, extensible, and open, and supporting *interoperability* with legacy and other systems through messaging and architectures such as Microsoft's Distributed Component Object Model (DCOM**) or the Object Management Group's Common Object Request Broker Architecture** (CORBA**). We describe in a later section what mechanisms are used to give SAP and component-based systems these characteristics.

As used in this paper, *assets* refer to intellectual capital (proposals, contracts, presentations, etc.) or development life-cycle models (technical reference architectures, business models, design models, etc.) that could be reused on their own. *Components* refer to hardware and software components and the models that describe them. Software components can be products, Java** "beans," or components that adhere to a component architecture such as Enterprise JavaBeans**, DCOM, or CORBA. Software products include database management systems, workflow systems, etc. Customizable *solutions* can either (1) have a limited set of common components across the applications within them, as many of the current ERP systems do, or (2) contain many common underlying components (business objects, frameworks), as exemplified by the San Francisco* project, described later in this paper.

Keys to success in the solutions business are: analyzing the market to develop solutions that fit, selling solutions that fit the customer, and developing solutions, components, and assets that can be altered to fit the customer's environment and requirements, at both technical and business levels. When common components are sought, success depends on incorporating fit and alterability into the right parts or components. Success occurs when these reusable items are easily customizable by services deployment teams. Ultimate success is judged by the customer and requires a solution to be customized to fit the customer's requirements in a timely manner, with ongoing support and maintenance.

Scope of customization. To better understand how to specify fit and alteration we need to consider the scope of solutions being developed. It should be noted that the solutions described here are to be customized by other developers into final customer solutions. Therefore, the requirements to be considered are not just end-user functionality (although that is necessary), but also details of the business processes, functional and nonfunctional requirements for many different end users, and, more importantly, the variances among them. The most critical variances must be represented in the system so that developers of the customer solutions can configure and instantiate prespecified *variation points*. Where these variances come from is the subject of this section.

A dictionary definition[2] of "scope" is "space or opportunity for unhampered motion, activity, or thought." In this section, *scope* is used to indicate the opportunity for variances within a solution or set of solutions. Scope levels include:

1. An application or solution meant for one or a few customers (services)
2. An application or solution meant to be customized for many customers (packaged solutions)
3. Development of many similar applications or solutions to be customized for many customers (packaged and component-based solutions)
4. Development of applications or solutions to be customized for many customers in different industries and geographies (packaged and component-based solutions)
5. Multiple versions of assets and solutions being developed, customized, and deployed over time

At Level 1 are fully customized green field or integration engagements, where only one, or very few, customers are considered during development. Fit and alteration are not important design points. The solution does not need to fit or be altered for another customer. As well, customers will typically not pay for their solution to be able to fit or be altered to fit another customer's requirements.

At Level 2, variation must be provided to fit the solution to the customer and to allow for some alteration to fit differences of individual customers. The challenges are to produce the right solution for the market (fit) and to allow easy alteration of important customer differences. This could be through explicit variance points or through extensibility.

Level 3 design must fit the solution to the customer and allow easy alteration to fit differences of individual customers, but an additional design point is now added. The new design point includes support for production of several solutions or applications from some common set of parts. This design point can be added for several different reasons, but all point back to the need to utilize common parts when developing multiple applications or solutions. Possible goals leading to this design point include:

- The need for a common "look and feel" across a set of applications or solutions (e.g., Lotus SmartSuite**, Microsoft Office**)
- The need for a consistent set of interfaces across multiple access points for an application or solution (automated teller machine, call center, etc.)
- The need for common data models across enterprise systems and consistent management of persistent data
- The need to save development costs by reducing redundancy across applications or solutions
- The need for common legacy systems access from several applications or solutions

The challenge is to design multiple applications or solutions to meet all three design points: fit, alterability, and common parts, while maintaining project schedules for the individual solutions and while managing the dependencies set up between development of common parts and the applications and solutions that utilize them.

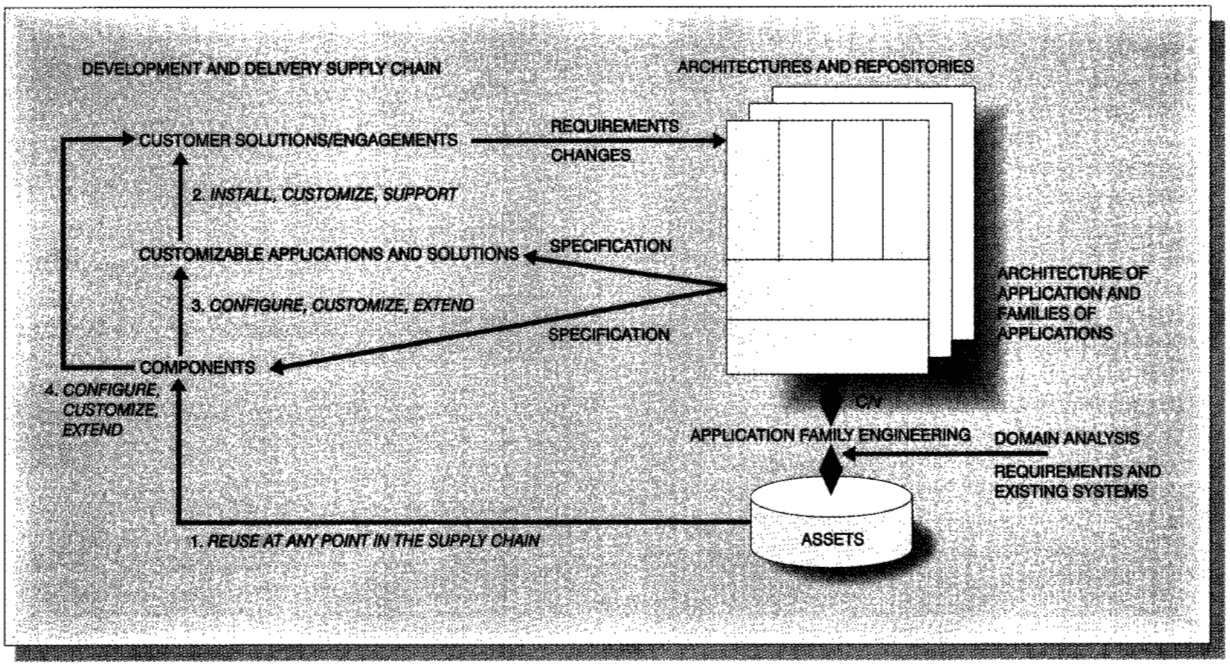
Applications or solutions at Level 4 will include the design points of fit and alterability as in Level 3, but the scope of alterability now must include the requirements of customers from not only one industry or geographic region, but many. If the solution or asset either does not fit, or is not easily alterable for this larger range of customers, it will be difficult to deploy and use. Additional complexity is added if the assets or solutions developed for similar industries are meant to be used as part of other solutions in other industries. Here we have the three design points of Level 3, with dependencies increased and project scheduling becoming more critical. An example of this would be development of a call center asset that needs to be specialized and used by many different industries.

Geographic differences in cultural norms, government regulations, etc., are also a large source of variance in Level 4 and must be explicitly represented and understood for deploying solutions in new regions. The decision to enter new regions should be driven by marketing goals and will require some level of geographic support and sales. The challenge is to pick those assets and solutions for which the complexity incurred is offset by expense reduction and better market strategies.

Finally, scope at Level 5 allows new versions of solutions to be easily installed at existing customer sites. This adds an additional design point for multiple versions. The challenge here is to integrate versioning into alterable solutions such that, once they have been altered to fit the specific requirements of a customer, a customer can easily migrate to the new version in a cost-effective way.

The scope of variances in solutions (over many customers, industries, and geographies) is important to understand, because the differences in customizability lead to different design points, architectural constructs, and levels of development management. Attempting to develop and deploy the range of so-

Figure 1 A value chain of components and solutions



lutions, components, and assets described above without a clear knowledge of the reasons for the choices (preferably using a cost/benefit analysis), and the means to manage the development and deployment, can lead to disappointing results. As well, it must be clearly understood that the systems described here are for creating customizable assets, components, and solutions, with deployment teams doing the customization that results in the final customer solution. For development of customizable solutions, we need requirements that show enough detail to allow us to find the critical variances and build them into the customizable solutions while supporting their inevitable evolution.

A solution-customization value chain. It is not enough for the asset, component, and solution developers to implement end-customer functionality in a general sense. There is a level of indirection that must be recognized and designed for. In a sense, we are developing tools to support the development of end-customer solutions. The full value chain is shown in Figure 1.

The right-hand side of Figure 1 shows the architectures that specify and repositories that store reusable assets. Multiple scopes are possible; as described earlier and discussed in the next section, commonality/variability analysis is a critical aspect in developing architectures, components, and solutions that incorporate the design points of fit and alterability. The left-hand side of Figure 1 shows a value chain of the full range of assets, components, and solutions to be developed, stored, deployed, and maintained by solutions and services groups. These can be used in multiple ways:

1. Services deployment teams use assets directly for customer solutions.
2. Solutions are created to fit a marketplace and the architectural specification allows for only minimal customization. Services deployment teams then customize a solution for the customer.
3. Customizable and configurable components are developed according to a generic architectural specification and used to develop multiple customizable applications and solutions, which are then customized and installed by services deployment teams.
4. Services deployment teams use the customizable components directly for customer solutions.

The customization life cycle

This section emphasizes the importance of understanding commonality and variability as the essence of customizability. With this understanding, a customization life cycle is introduced. This life cycle can be implemented in the methods and processes of developers using various mechanisms. It is this life cycle and its implementation that leads to customizable solutions and thus success in the solutions business.

Commonality and variability analysis. To be able to manage development and deployment of the scope described earlier, we must understand *what* will allow us to meet the design points of fit and alterability and thus customizability. We must understand the similarities and differences among the business processes, requirements, and existing solutions across the customer set for which the solutions are intended. The requirements are both functional and nonfunctional and exist at the technical as well as the application or business level.

If the requirements for the common parts of solutions for multiple customers are not understood, there will not be a good fit. If the variations of multiple customers across multiple geographies and multiple industries are not understood, it will be very difficult to ensure that the solutions are designed to be alterable where necessary. This does not mean that all variations will need to be maintained separately in a solution; often they can be generalized. However, if the solutions are not designed to be alterable when necessary, it will be very difficult for the deployment teams to alter them. Examples of required variability include differences in standards or business rules across customers and geographies, such as tax calculations or telecommunications protocols.

The requirement for customizable asset, component, and solution development is to explicitly represent this commonality and variability (c/v). This representation must support evolution of the assets and components. This also means developing the solutions so that the commonality and variability is factored out properly. We will examine several mechanisms for customization in a later section.

Steps in the customization life cycle. A customization life cycle includes managing the c/v across customers, incorporating the c/v into the assets, components, and solutions, providing interfaces to the customization points for deployment teams, and supporting customer migration to new versions while maintaining existing customizations. For success in the solutions business, this life cycle should be reflected in the methods and processes used by development and deployment teams. The life cycle contains five steps. These are not intended to be executed in a waterfall manner, but rather would find their way into methods such as iterative development. The five steps are:

1. *Analyzing and representing c/v.* Commonalities and variabilities of customer environments and requirements are analyzed and represented explicitly. The representation supports evolution as results are used and evaluated.
2. *Implementing c/v.* The commonalities and variabilities are incorporated into generic architectural descriptions and customizable components.
3. *Interfaces.* An interface and possibly tools are provided to aid in configuring components into solutions and in their customization.
4. *Customization.* The interface is used to configure and customize the components to fit the end customer's environment and requirements. Part of this customization is to fit the components and solutions to the customer as closely as possible.
5. *Versioning.* Customers migrate to new versions of the components and solutions.

In the first phase of the life cycle, the commonalities, variabilities, and invariants of customer environments and requirements should be analyzed and represented. This can be done in several ways—using domain experts, existing industry business models, existing systems, and requirements gathering. Invariants refer to those parts of a system that never change across customers and geographies; these become part of the core system. The c/v aspects are considered so that the common services, functions, objects, rules, etc., of a system can be identified, with the variations among them also identified. Some mechanisms for representing this c/v exist today, such as "use cases,"[3] where "uses" captures commonality and "extends" captures variation in requirements. "Hot spots" in framework development are also used to identify variation points.[4]

Other mechanisms for representing c/v in existing systems are included in domain analysis techniques such as FODA[5] and ODM.[6] All requirements during the life of the assets and solutions should be managed in the same way, giving rise to the need for a good requirements management system that can ac-

commodate c/v. Today, explicit analysis of c/v is often not done when creating customizable systems. Instead, developers rely on intuition and good design practices. In framework development, large numbers of iterations are used to get to the c/v (SEMATECH, described in a later section, iterated for six years). This is ineffective for the shorter lead times and longer lifetimes of current systems. It is also ineffective for systems at scope levels 3, 4, and 5, described in the previous section. These systems need a more systematic approach that includes both explicit representation of c/v and a good understanding of the mechanisms for applying it in customizable systems.

In the second phase of the life cycle, the c/v and invariants are implemented. This can be done through many different mechanisms and techniques, as described in a later section. At the core of this work is the need for good architectural techniques and representation. Implementation consists of moving the invariant parts to a mandatory common core, clustering the common parts for configurability, generalizing variable parts where possible, developing generic designs, and providing well-defined points of customization where variability is needed for different industries, customers, and geographies. A good example of applying c/v is the use of design patterns.[7] For example, the Strategy pattern allows algorithms to be varied within a common context. This could be used to implement different business rules across countries for taxation, for example. For component-based systems where the scope is closer to an enterprise level, the key notion is the development of generic architectures, common components, and rules that allow multiple applications within a family to be configured and customized.

When incorporating customizability, development teams can apply the techniques and mechanisms (described later) at three different times: when the asset or solution is being developed, at compile time, and at run time. This means that areas meant for customization are architected and designed into the asset or solution, but when compiling a particular version of an asset or solution, different parts, or versions of parts, can be combined. As well, deployment teams and customers may be able to do some customization at run time, such as changing screen color or window positions. This adds another dimension to the mechanisms and techniques.

The third phase of the customization life cycle provides an interface around the assets, components, or solutions for deployment teams and other users. Although it may be optional, an interface is often an indication of maturity. An example interface is the ABAP/4 (Advanced Business Application Programming/4) language used with the SAP assets. An interface can be used for several purposes: to explicitly identify variability and alterability points, to hide the source code behind application programming interfaces, to aid in the understandability of the assets and solutions for easier deployment, to view and adapt models of the system, and to aid in the configuration and extension of the assets and solutions. Examples of interfaces include scripting languages to "glue" components together, and support for configuring process and data models. Specialized tools, such as visual modeling and code generation tools, may be built that support the use of these interfaces as well.

The fourth phase of the life cycle is the use of the interface or other descriptions of the assets, components, and solutions by those higher in the value chain to fit and alter (i.e., customize) them to fit the customer's environment and requirements. As well, it may be necessary to extend them, and to ensure that they interoperate with other systems, such as legacy systems. Part of the deployment is to ensure that sales organizations consider the fit of the assets, components, and solutions to the customer's environment. Yet another part is configuration of the assets, components, and solutions as part of the effort to fit into the customer's environment.

An important part of the customization process is to instantiate the variability points. If these points have not been explicitly identified, teams will have to adapt or alter the assets in ways that may not have been anticipated by the development team. This may lead to problems with the overall fit of the asset or solution, but it may be the only way to satisfy the customer's requirements. Examples of customization include: application installation, configuration of solutions from parts, gluing the parts together with scripting languages, instantiating (configuring) variability points, and adapting models that describe the assets and solutions.

The final phase of the customization life cycle is the migration of customers to new versions of the assets, components, and solutions. This migration should be done in a way that minimally affects the customer solutions already in place and still allows easy reimplementation of the customizations previously done. New versions involve release manage-

ment and configuration management of the assets and solutions by the development teams. Much of the versioning effort is in managing the changes made by each customer. Versioning is made easier by minimizing the alterations that are possible, making good use of interface definitions, and allowing only extensions to components or objects.

Examples of customization

This section contains six examples of customizable systems. Each is described in terms of the customization life cycle discussed earlier. The first three descriptions represent general methods, mechanisms, and techniques, the fourth describes a successful ERP system, and the last two represent systems being developed within IBM. These examples are provided to show that successful systems being developed today incorporate the customization life cycle. This section also shows that different mechanisms can be, and are, used in each of the life-cycle phases.

Jacobson, Griss, and Jonsson. The book entitled *Software Reuse: Architecture, Process and Organization for Business Success*,[8] by Jacobson, Griss, and Jonsson, describes how to develop and deploy a family of applications, described by one generic architecture, with reuse of a common set of underlying components. This book also describes transformation processes and organizational structures that need to be put in place for any organization embarking on development, deployment, and management of such application families. This is much different than development of single, or multiple loosely related, applications. Here, a single architecture describes the components (objects and related work products), component systems (sets of related components, e.g., subsystems, frameworks), and applications (sets of component systems, configuration scripts, related documentation, etc.) that are developed from these components. Multiple versions of the applications for different geographies could be developed, and each application is part of a family of applications, such as those related to telephony switching systems or financial and banking systems.

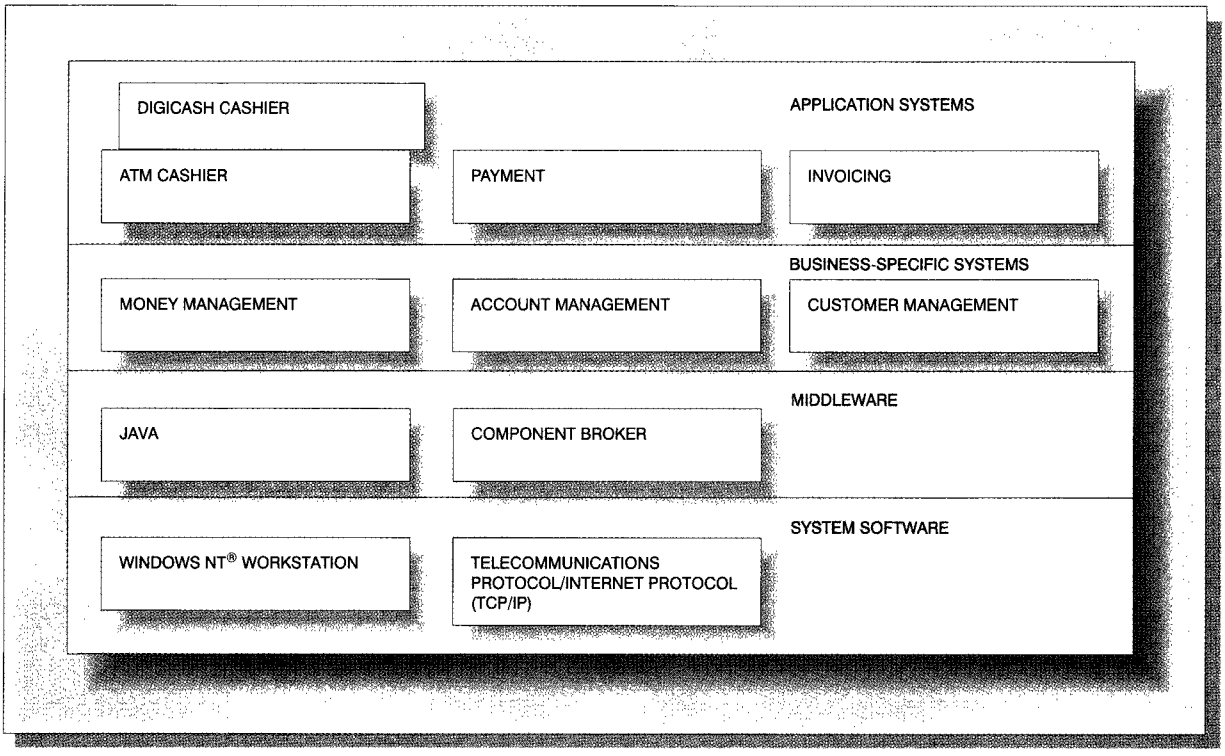
Figure 2 shows an architectural view of layers of technical and business components, with applications at the top developed from the components. The applications at the top are typically designed to be highly interoperable and possibly built within distribution architectures, such as CORBA. It should be clearly understood that the layered generic architecture specifies the common components, their interfaces, and the interactions required to develop *all* applications in a family, but any one application will use only some of the components (which may be available in different versions). Configuration support is needed when developing applications and solutions from the components.

A key theme throughout the book is the idea of variation points and variants. Variation points are represented in all models from use case requirements through component code, and variants implement (specialize) the variation points. The authors also stress the need for traceability of the variation points through the models in order to support full reuse of the component systems by application developers. These variation points and variants correspond to the emphasis on variability discussed throughout this paper.

Analyzing and representing c/v. Commonality and variability is analyzed and represented first using the use case method.[3] The key aspects of "uses" and "extends" support representation of commonality and variability in the use cases. "Uses" can be used to show common portions of use cases, while "extends" can be used to show optional or variable portions. Use cases are defined at the highest level of detail, essentially the business processes supported by the highest level applications. They are also used at the level of components. Use cases for each of the related applications are obtained and analyzed together to find commonality and variability. Not all possible use cases need to be gathered, but generally a good representative set should be obtained for the initial architecture of components. Analysis of the common parts of the use cases for the top-level applications leads to the architectural definition of the necessary component systems. Use cases at the component level, and further c/v analysis, lead to further representation of c/v at the component level.

This c/v at the use case level is then traced through subsequent analysis and design models. The first level of analysis is done to achieve the overall architecture and to detail the requirements for the component systems, which are developed separately from the applications. An important aspect of this approach is the focus on separate component and application development teams as well as separate deployment teams. Processes and methods are described for organizing around this type of architecture and development.

Figure 2 A banking family of applications*



* From I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success.* Copyright 1997 Addison-Wesley Publishing Company. Reprinted with permission.

Implementing c/v. Several mechanisms are used to implement the commonality and variability that is represented in the use cases and other models. The first is use of a *layered architecture* to separate technical parts from more business-related parts and to define the common *components*. Little discussion of the technical layers is included in the book. Most of the examples shown relate to the business and application layers. As well as showing the c/v in the analysis and design models, *facades* are described to separate public and private views of the components, particularly as they relate to the *variation points*. *Imports* relations define the use of components by other applications. The authors also describe the use of *design patterns* to implement the variation points and help specify where variants can be configured into component systems and applications. In addition, the authors define several other variation mechanisms that can be used, including inheritance, parameterization, and extensions.

Interfaces. Interfaces, as described earlier, are discussed here, such as wizards, scripting, and templates, but are not defined or shown in detail. The emphasis is on reuse of the component models by application developers, with a brief discussion of the possible need to add customization points, package up the applications further, and add installation scripts and documentation for other deployment personnel.

Through the customization life cycle, this paper makes a clear distinction among mechanisms used to implement c/v, the definition of interfaces into the c/v, and customization of the c/v. Jacobson, Griss, and Jonsson discuss variation points and variants and mechanisms to implement them but do not make the same important distinctions.

Customization. Applications are developed by different groups of practitioners than the component

system developers, and the models are reused, tracked, specialized (at the specified variability points), and extended during application development. Applications themselves may have further customization mechanisms built in to allow for customization on customer engagements, as described in the value chain of Figure 1.

Versioning. Facades are used to hide the internals of components and to support configuration management. The emphasis is on hiding internals and exposing only the variation points to the users. The authors push for a more "black-box" approach, but recognize that this may not be possible in the early stages of component use by application developers, who through early use of the components put more variability requirements on them. Hiding the source code behind well-defined interfaces allows updates in later versions without affecting users. With variation clearly specified, the changes can be more easily managed.

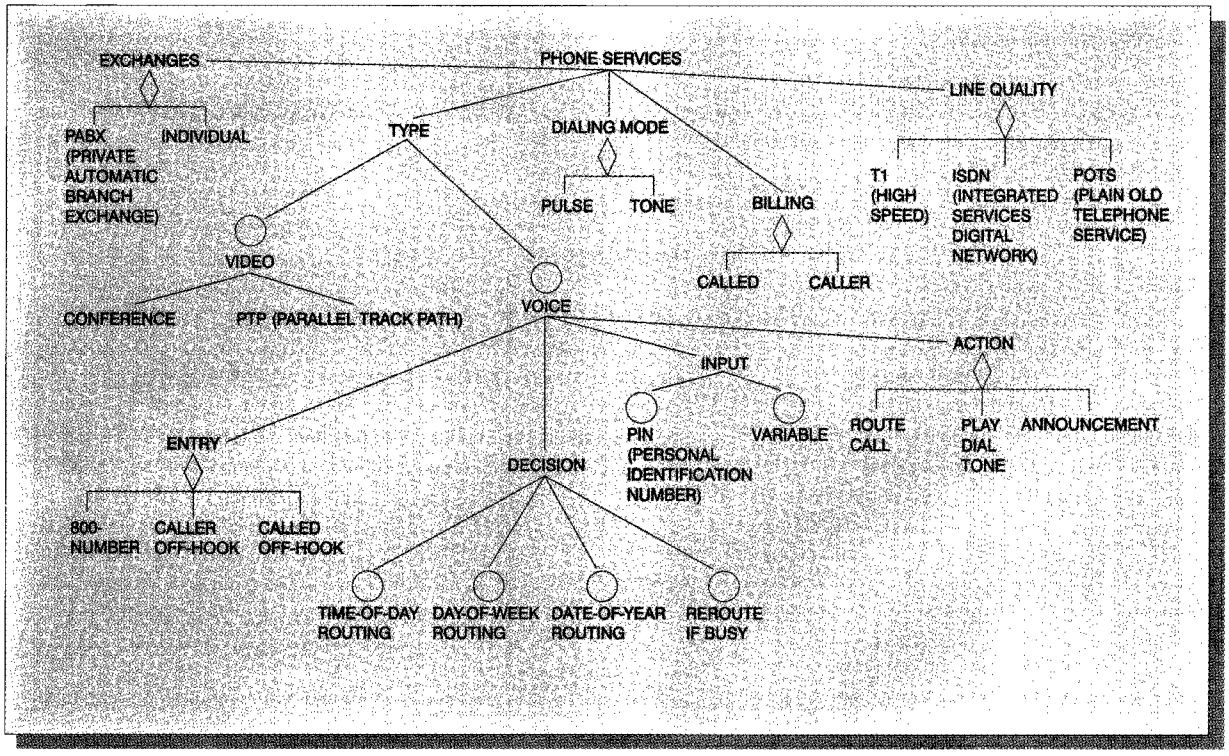
FODA (feature-oriented domain analysis). "Domain engineering" refers to the development of reusable architectures and components and their subsequent use in developing families of applications and systems. The first part of domain engineering is domain analysis for reuse, where generic architectures and their contained components are specified at analysis and design levels through models, followed by implementation of the design. The second part of domain engineering is the use of the generic architectures and components for application and systems development. Example domains might be business functions, such as marketing or human resources, or an entire enterprise or a product line, such as ink-jet printers or telephony switching systems. The analysis in domain engineering is usually example-driven, where previous applications developed in the domain are the main input, supplemented with domain expert knowledge and possibly future system requirements.

The most important aspect of domain engineering is its emphasis on analyzing multiple existing systems within some scope and doing a commonality and variability analysis across processes, functions, technical platforms, operational contexts, data, objects, tasks, etc. There are many levels where this type of analysis is useful, including an entire enterprise or some portion of it for one or typically several customers. The models used to develop the generic architectures and components are supplemented with explicit representations of the commonality and variability. This analysis is done in order to reduce redundant development, provide a common "look and feel" across applications in the scope, increase the quality and maintainability of applications, and to support better interoperability among applications within the scope. This type of analysis results in components that can be configured together in multiple ways to develop applications in a family of systems.

The emphasis on commonality and variability analysis develops because the components need to be reusable by many different application and system developers. This customizability needs to be built into the components so they can be produced and maintained in a systematic fashion. If the applications and systems themselves also need to be customizable, then that must also be built into the architecture and component definitions. Layered architectures are the norm, and although all components that will be used by the applications are defined within the generic architecture, not all applications will use all components. This means that the generic architectures must also maintain configuration views of the systems to be built. The generic architecture specifies all possible components and how they would work together, but not all applications need or can use certain combinations of the components.

FODA, or feature-oriented domain analysis, is one of several methods for doing domain analysis and has to date been one of the most highly used, especially by telecommunications companies such as Lucent Technologies and MCI WorldCom. Several models are described at analysis and design levels and are supplemented to show c/v information, but the most characteristic model is the *feature model.* Features are user-visible aspects or characteristics of a domain, and are used to define a domain in terms of the *mandatory*, *optional*, or *alternative* characteristics of related systems within it. Mandatory features must be included in all applications within the domain. Optional features will be in some applications and not in others. Alternative features specify specific variations and typically define a specialization or abstraction hierarchy. As stated earlier, applications within a domain share many common capabilities. These capabilities, from the point of view of the end user, are called features. Features include the services or functionality provided by the applications. They also include hardware platform requirements, performance requirements, and cost characteristics. The feature model in FODA is developed at analysis time and is used to generalize and parameterize the other models, including object

Figure 3 A telecom feature model for services*



* From M. Griss, J. Favaro, and M. D'Alessandro, "Developing Architecture Through Reuse."

models, functional models, process interaction models, and component models. The feature model also defines configuration rules that, for example, might specify that if air conditioning (feature) is chosen in a car, then a certain size engine (feature) must also be chosen.

FODA was first described in 1990[5] and comes from the Carnegie Mellon Software Engineering Institute. It predates many of the object analysis methods used today, including use case modeling. In a recent article,[9] Griss, Favaro, and D'Alessandro describe the use of the feature model to enhance the methods proposed by Jacobson, Griss, and Jonsson[8] and describe how the feature model works with the use case model to define a "reuser" oriented view of architectures and components for families of systems. An example of a feature model is shown in Figure 3 and describes the essential feature choices to be made when developing new services (call waiting, call forwarding, etc.) for telecommunications systems. Straight lines in the figure indicate "composed-of" relationships, circles above features indicate optional features, and diamonds indicate alternative features.

The feature model provides a catalog of features and gives a configuration road map of what can be selected, combined, and further customized in a system. This provides a view of more than just the functionality that a reuser is trying to implement. Developers of applications and systems select features from the catalog and use it to make initial configuration choices. Further detail of the functionality to be developed is given by the traceable use case and object models that point to detailed customization choices that must be made and implemented.

Analyzing and representing c/v. FODA uses several models at analysis, architecture, design, and implementation time to explicitly represent commonality and variability. The key model is the feature model,

which is used to generalize and parameterize the other models.

Implementing c/v. The c/v is represented in the generic architectures, which specify all components needed to implement the family of systems in the chosen scope. The feature model specifies all features that are available in the family of systems and is used to generalize and parameterize other models, where further detail is specified using several mechanisms described later in this paper.

Interfaces. The interfaces provided to a reuser are the models and generic architecture description. The feature model is provided to support configuration decisions and to select the mandatory, optional, and alternative aspects of the final system. Choice of these features then leads to the appropriate models, where details can be further specified and refined.

Customization. The first part of customization by a user of the generic architecture and components is analysis of the features needed by the end customer. This analysis is driven by the features available as specified in the feature model and the configuration rules. Once these choices are made, further customization will be needed in the various models and finally in the code that implements the feature model. In a value-chain view, there may or may not be a separate installer at the customer site, in addition to an application or system developer who may work away from the site.

Versioning. FODA itself makes no distinctions about versioning of components, but this is clearly described as part of the packaging and ongoing maintenance of component-based systems. Here several mechanisms, such as generators and separate interface definitions, are important.

Frameworks. Frameworks can be defined as "a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact."[10] A chief aspect of frameworks as we are defining them here is that a flow of control, which shows how the objects interact, is part of the framework description. Frameworks may be developed when several instances of the design are expected to be needed in the future. Frameworks can be *calling*, where the framework maintains control, or *callable*, where applications call a framework to provide a service. Many people refer to whole architectures as frameworks; for example, SEMATECH and CORBA are sometimes called frameworks. In this paper, frameworks have actual object representations.

Analyzing and representing c/v. Frameworks are usually developed by analyzing several examples of similar systems, or parts of systems. After analysis of many examples, generalization is the primary mechanism used to develop the core framework structure. The primary mechanism for analyzing and representing variability in frameworks today is the use of hot spots.[4] Hot spots define the abstract classes that can be implemented in different ways by different users of the framework. Typically, initial versions of frameworks are first defined and hot spots are later identified, based on analysis and user feedback.

Implementing c/v. In addition to generalization and hot spots, design patterns[7] are a key mechanism for developing frameworks. As described elsewhere,[11] an important aspect of design patterns is that they give good designs for areas where various types of variability are needed. Hot spots can be used to identify needed variability and a design pattern can be applied that incorporates that variability and has been shown to work well in the past.

Customization. Techniques for customization of frameworks include "white box" and "black box." White-box customization is done by subclassing and overriding abstract and concrete classes. Black-box reuse is done through composition and delegation. In black-box customization, the classes that implement the different variations exist and the user customizes by selecting the appropriate classes to configure into the system.

Versioning. Black-box reuse is preferred over white-box reuse because it avoids the difficulties associated with versioning, configuration management, and release management. But black-box frameworks are much harder to develop and often evolve after multiple uses of a white-box implementation.

SAP. SAP AG, an international company based in Germany, develops integrated packaged applications in the ERP (enterprise resource planning) domain. The structure of the current system, R/3, is shown in Figure 4.[12] It consists of a basis layer, an application layer, a development workbench, and a business engineering workbench.

The basis layer contains the middleware of the R/3 system. This middleware makes the applications in-

dependent of the system interfaces of the operating system, database system, and communication system used and ensures optimal handling of business transactions. On the basis layer sits the application layer, which implements the business functions and processes of the R/3 system. The basis layer is written in C and C++, while the application layer is written in the fourth-generation language ABAP/4.

Individual program modules in the basis layer provide the following services:

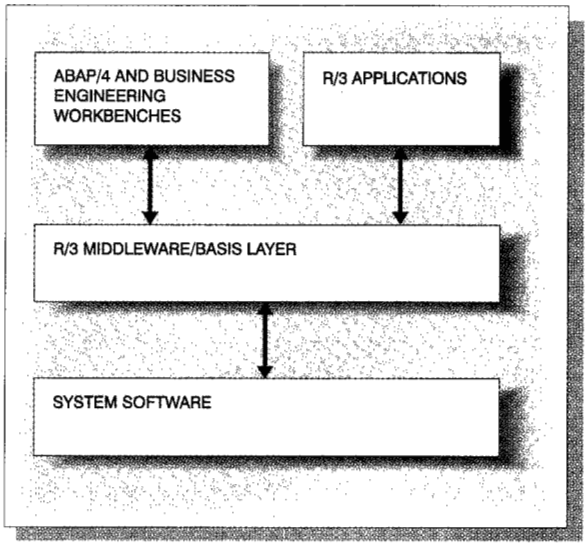
1. Presentation services for implementation of the graphical user interface
2. Application services for handling of the application logic and units of work
3. Database services for storage and recovery of business data

R/3 presentation services include modules for the representation of various document and graphic types as well as the required communication services. The applications of the R/3 system work in a transaction-oriented fashion. A SAP transaction is a sequence of logically linked dialog steps consistent with business practices. SAP also supports cross-application transactions and database updates using logical units of work. These transactions can happen within or across processes and in different computers. The logical connection of dialog steps belonging to a transaction is guaranteed by the SAP system. For the definition and manipulation of data, the R/3 system exclusively uses SQL (system query language) commands. The architecture of the system is laid out in such a way that differences in the syntax and semantics of the SQL implementations of different database manufacturers are isolated in special R/3 modules. Therefore, in principle, all relational database systems in the market can be supported.

The applications of the R/3 system are based on an overall business model that makes possible a uniform view of all data and business processes in the enterprise. The overall model covers financial accounting, controlling, asset management, materials management, production planning, sales and distribution, quality management, plant maintenance, project management, service management, human resources, office communication, workflow functions, industry solutions, and open information warehouse.

SAP also supports workflow management. It coordinates the sequence of work steps and the activities of the people involved, and it provides the software functions necessary for business processes. This is supported by workflow definition tools and a runtime environment that controls the workflow. The workflow architecture contains an organizational model, a process model, and an object model. Tasks performed by members of an organization link the processes depicted in a workflow. The workflow steps of a business process are described in the process model, and reference the tasks in the organizational model. The workflow steps are usually methods of a business object and are defined in a business object repository. Business objects are assigned a data model that describes the object from a data point of view, with additional areas for constraints, business rules, methods, attributes, and input and output events.

Figure 4 The SAP architecture*



* From R. Busk-Emden and J. Galimow, *SAP R/3 System: A Client/Server Technology*. Copyright 1996 Addison-Wesley Publishing Co. Reprinted with permission.

Analyzing and representing c/v. SAP AG has worked with several process standards groups to define the common business processes used in the system. This has helped to ensure that the processes and related functions provided in the SAP system will be close to what many companies will need. Although not much is known about how they explicitly represent and manage the variability in the system, Busk-Emden and Galimow state that "during implementation of the process chains, the different characteristics of the standard solutions needed for different branches

of industry and company types, as well as multilingualism and national particularities were taken into consideration."[12] Requests for future customer exits and changes to the system are handled through a requirements change process, which is used for future updates to the system and represents required variability. This requirements process is often initiated through customer interest groups.

Implementing c/v. Common business processes, functions, workflows, screens, basis (technical infrastructure layer), and data models are part of the SAP system and common default parameter settings are available upon first installation. Variability of the system is allowed through customer exits (calls to customer-specific application modules that have been anticipated by SAP) for differences within functions. Variability is also built into the system by allowing several key system models to be configured, altered, and extended in different ways. These include the organization model (organizations and the tasks performed by roles within the organization), the process model (all processes in the system), the function model (all functions in the model that are related to the processes), the data model and subsequent table settings, the distribution model of how applications and services are distributed across the computational tiers, and the user interface of screens and screen flows.

As well, the system can be expanded to allow new functionality to be added, and interoperability with other programs is possible through the ALE (application linking and embedding) mechanism and support for CORBA and OLE (object linking and embedding). Variability is also built into SAP through different versions of the system. These include country-specific versions as well as recently available industry-specific versions. The interpretative nature of R/3 also gives possible variability at run time.

Interfaces. Two interfaces into the SAP R/3 system are available. The first is the ABAP/4 Development Workbench, which is the programming environment for development of enterprise-wide client/server solutions. It supports the entire software development life cycle with tools for modeling, programming in ABAP/4, definition of data and table structures, and design of user interfaces. Support for testing, tuning, maintenance, and large development teams is also available. As a supplement to the development tools, the business and software components of the SAP system can be incorporated through this interface. ABAP/4 supports functional modules, and the components of SAP are based on these. Functional modules have a clearly defined calling interface, and import, export, and table parameters are defined there. Functional modules can also be called across system boundaries using remote function calls (RFCs). The ABAP/4 Development Workbench can be used to extend default system parameters provided with SAP. Most recently, SAP has added components and business objects. These components are open and accessible to other vendors through BAPIs (business application programming interfaces).

The second type of interface provided by the SAP system is the Business Engineering Workbench. This workbench contains all of the functions and information for process-oriented support of initial implementation projects, follow-up projects, and release-change projects. This interface is extensive and contains a default methodology for projects, implementation guides with experience-based details of the methodology, project management support, default documentation, the repository of all functions and processes for installation and customization, transactions for support in customizing the system, and a reference model that describes the entire SAP default system.

The reference model contains a function model (all function modules in the system), process model (all processes in the system, both workflow/event-based and input/output-based), information model (showing inputs and outputs to functions), communication model (communication between organizational units), organization model (organizational structure and task relationships), distribution model (distribution scenarios possible for R/3 and support for ALE), and the data model (entity-relationship model). Not listed here but also part of this interface is access to default screen layouts.

Customization. The two interfaces just described are used to install, customize, and extend the SAP R/3 system. First a default, industry-neutral version of the system is installed that contains a simple organizational structure, consistently set parameters for all applications, country-specific charts of accounts, standard settings for account assignment, configurations for control of standard processes, and standard settings for processes like "dunning and payment," "planning and forecasting," "pricing," and for printing and form layout, authorization privileges, and so on.

This default system is then used for understanding and selection of those parts of R/3 that fit the customer, providing a foundation for requirements analysis in the first phase of the methodology. The second phase of the methodology is focused on detailing and implementation by:

- Changing global system settings
- Configuring processes, functions, and data models
- Mapping the company organization
- Choosing the appropriate distribution model
- Implementing interfaces
- Implementing customer exits
- Deleting unneeded portions of user interfaces, applications, and data
- Extending the system using the ABAP/4 interface. Extensions include processes, functions, data model, database tables, and use of ABAP/4 aids in integration and distribution with other parts of R/3.

The complexity of the second phase is in the interrelationships that exist within the system. The R/3 system consists of several thousand custom-setting possibilities, and the need for consistent settings can make implementation of an R/3 system a very complex task. The implementation guides help with consistent settings and proper use of the interfaces. Implementation and customization can also be difficult due to the inflexibility of the business processes provided. Typically, a company must change its business and organizational structure to conform to the SAP software.

The third phase of the methodology is preparing to "go live" and includes developing custom documentation by adapting existing documentation, training users on the system, transferring data, and setting up the environment. The final phase of the methodology is putting the system into operation.

Versioning. The customer changes are maintained, and when a new version of SAP is installed, the customizations are again added. Only additions to the existing system are easily allowed: no source code changes should be made unless the source extends the system. If source code changes are not made, SAP can be fairly sure that new versions will not cause major changes to a company's installation. Bug fixes are delivered in separate small releases of the system.

Recently, SAP has been maintaining separate versions for different geographies and different industry variations of the software.

San Francisco. The San Francisco project in IBM is developing a set of frameworks that are very close in structure to the layered architecture of common components described by Jacobson, Griss, and Jonsson.[8] The base layer, as shown in Figure 5, consists of a foundation and utilities layer with a common business objects layer above it.

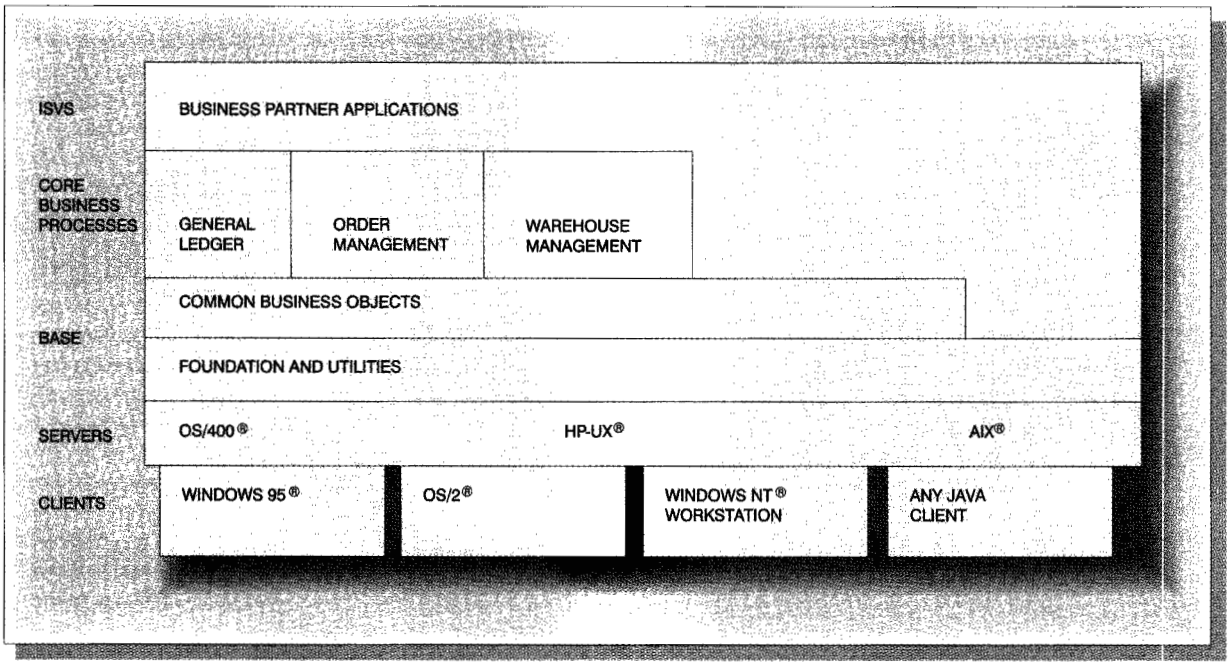
Currently, San Francisco supports application areas such as general ledger, accounts receivable, accounts payable, warehouse management, and order management. It provides the business and technical infrastructure on which business partner ISVs (independent software vendors) can develop their own applications. The common business objects are those that are used across the domains. Above the base layer are the business frameworks that support common business processes in the application domains. The application developers define their applications using the underlying technical and business infrastructure. Each ISV defines unique applications through differences in user interface, business rules, industry, geographic, or other competitive features.

The San Francisco project has three main objectives that support the customization life cycle presented in this paper.[13] The first objective is to offer easy entry into object-oriented (OO) development. That objective led to customizable frameworks that provide about 40 percent of the application code and allow extension and customization. Developers start with existing San Francisco system models and code and add variations.

The second objective is to support ISVs' applications to make their companies more competitive. This objective led to common business objects, customizable application frameworks, and a flexible technical infrastructure for San Francisco. This infrastructure provides common services such as transaction management, persistence management, security, and systems management. The infrastructure also supports multiple client and server operating systems and multiple architectures, ranging from fat client to Internet and thin client topologies.

The third objective is to provide an open solution that will allow trade-offs in cost, performance, and skill requirements. Developers can choose which parts of the frameworks to use and are able to use

Figure 5 The San Francisco architecture



one infrastructure in multiple ways for multiple markets, depending on cost and performance requirements. Developers can use the frameworks in various ways, depending on their skill level: without change, with changes, or with extensions. Developers can even build their own frameworks that make use of the San Francisco infrastructure.

San Francisco provides a good example of the customization life cycle, using the advanced techniques of frameworks.

Analyzing and representing c/v. Commonality and variability was analyzed and explicitly represented at multiple levels. Business processes for the application domains were decomposed into business tasks, modeled with use cases as described previously. Common tasks were identified, as were abstract and extendible tasks. The use case modeling concepts were modified to add an "inherits" notion in order to more fully capture the abstract tasks. These tasks were also classified as high, medium, or low volatility with respect to company or country requirements. Tasks were further elaborated at design time into one or more specific scenarios showing business logic detail. This detail was also analyzed for optional and mandatory inputs and outputs. The framework requirements documents explicitly represented commonality and differences in business rules, industry-specific differences, and country-specific differences. Variability in interfaces on the client side is also supported.

Implementing c/v. Several mechanisms are used in San Francisco to implement the c/v. First, analysis models are augmented with design patterns,[7] where variability in business processes were identified. Second, design-level classes that need to be extended are named with a specific prefix to help users identify them. Third, configurability and extendibility of classes is supported through a mechanism that allows dynamic addition of class relationships at run time through the use of *properties*. One way that properties support configurability is that Java packages can be maintained with only one-way dependencies, thus allowing the purchase and use of them more independently. Another way that properties support configurability is that code-level changes are not required.

Design patterns solve basic problems and provide classes to support variations on a solution to a prob-

lem. Several of the commonly known design patterns are used in San Francisco, and several new ones were added. In all, about a dozen design patterns are used, making it a very flexible set of frameworks and business objects. Some of the patterns support variation at compile time, such as the Policy pattern, while others support variation at run time, such as the Dynamic Identifiers pattern. Others, such as the Factory pattern, even allow dynamic change to a persistence server location. This aids in partitioning objects across servers and in mapping objects to different legacy databases. The use of "command" objects as business tasks also supports variation in the partitioning of work load across servers and in transaction management. The command objects can be executed as independent transactions or as part of a larger transaction.

Eight patterns are used in San Francisco as extension points for customization:

1. *Properties*—add attributes or relationships at run time.
2. *Policies*—replace or modify business rules.
3. *Encapsulated chain of responsibility*—allows use of different policies for different objects or processes; for example, first look up the discount policy for the customer; if none exists look at the product; otherwise use company-level policy.
4. *Dynamic identifiers*—support new user-defined categorizations, such as a new account code, transaction type, etc.
5. *Class factories*—can be customized to map objects to existing database tables or partition objects of the same type across different servers.
6. *Extensible items*—can have behavior added or removed dynamically to move execution up or down the class hierarchy.
7. *Life cycles*—allow redefinition of complex business processes to add or remove steps, conditions, and behavior.
8. *Keyables* and *cached balances*—define complex keys from multiple attributes to compute "on the fly" or cache summary data.

Commonality, and to some extent variability, is implemented in San Francisco using layered architecture techniques. A very important layer is the common infrastructure and its programming model. The programming model allows developers to add in common services, provided in the infrastructure, as needed. These services include transaction, persistence, and notification services. Common business objects, in another layer, provide objects common to multiple domains and include common business tasks and common application services such as interoperability. Unique structure and behavior, particular to a specific domain, is implemented as part of an application framework. A "business partner" is an example of a common business object, while a "warehouse" object is particular to a distribution or logistics domain.

Four patterns are used specifically for commonality and to ease maintenance and understanding:

1. *Aggregating and hiding controllers*—group objects and attach groups to different levels of a company hierarchy. They provide views at a single company level, aggregate higher levels, or hide certain groups.
2. *Shared/sharing controllers*—support objects that have some attributes that vary at a controller group level.
3. *Atomic update*—supports update validation and rollback involving multiple objects.
4. *Tightly coupled creation*—establishes ownership of an object when it is created. This allows creation of an object that is subject to policies and validation by its owning object.

Interfaces. Interfaces into the San Francisco frameworks and infrastructure are provided in multiple ways, including programming models, wizards, tools, extension guides, and code generation. Programming models provide an interface to the design model and to the services provided by the infrastructure. A program model for business object developers documents which methods must be overridden, which methods may be optionally overridden, and what new methods must be defined. A client programming model supports definition of transaction scope and choices for other services, such as locking models and persistence and execution locations. "Wizards" that work in conjunction with the programming models are also available. They make the programming models easier to use by, for example, guiding a developer to places in a framework where modifications are necessary.

Code generators are also provided as an interface. These generators use the object design model and the programming model, which specifies how services and object relationships are to be handled, and produce code. Other interfaces are provided to developers through various tools that support object modeling and link to the code generation tools.

Customization. The San Francisco frameworks today are examples of white-box frameworks and thus allow changes to the design models and code contained within them. These changes must be made in a way that preserves the contracts and interfaces specified in the frameworks. This is supported and enforced by the interface mechanisms just described. Basically, customization occurs in four steps. First, new requirements in the form of processes, tasks, and use cases are analyzed and variations on existing functions and business tasks are specified. Second, the object design models are customized and extended as necessary using the interface tools, programming models, wizards, and extension guides. Third, the design model and programming models are used by the code generator to produce code. Fourth, the class-level code is completed, tested, and integrated into a business environment.

Functional customization can be done in several ways:

1. By using a factory object to manage create, delete, and update access to a framework business object
2. By creating new domain classes from the base
3. By extending existing domain classes and methods by subclassing and overriding to add, for example, new attributes, or new logic in a method
4. By instantiating extension points in the patterns
5. By chaining policies and domain objects to support use of the right policy (business rule), based on the domain object involved
6. By using properties or dynamic identifiers for runtime customization

Customization must also be considered from architectural and nonfunctional perspectives. This includes choices concerning hardware topology, performance, transactions, persistence, object placement and execution, legacy integration, client interface choices, and others. The Factory pattern supports mapping objects and data to interoperate with legacy data and applications, for example.

Versioning. Versioning is supported in San Francisco through Java packages. These packages have been designed to support one-way dependencies that allow configurability and easier maintenance. Elimination of circular dependencies between the packages will allow easier maintenance of new packages upon new releases of the software. There is also versioning support for the interfaces and implementations supported and, to some extent, for object contents (instance data). Much of the work involved in creating new releases will be manual, aided by versioning tools.

SEMATECH and Super Poseidon. The Semiconductor Manufacturing Technology consortium (SEMATECH) has been developing an industry standard for a software framework for computer integrated manufacturing (CIM). The CIM framework defines a component-based architecture that forms the basis for a next generation of manufacturing execution systems (MESs).[14] The architecture is shown in Figure 6 and includes specification of classes to the method and attribute level, common components consisting of related classes, and characterization of functional groups of applications. Super Poseidon represents the IBM initiative to develop according to the SEMATECH specification.

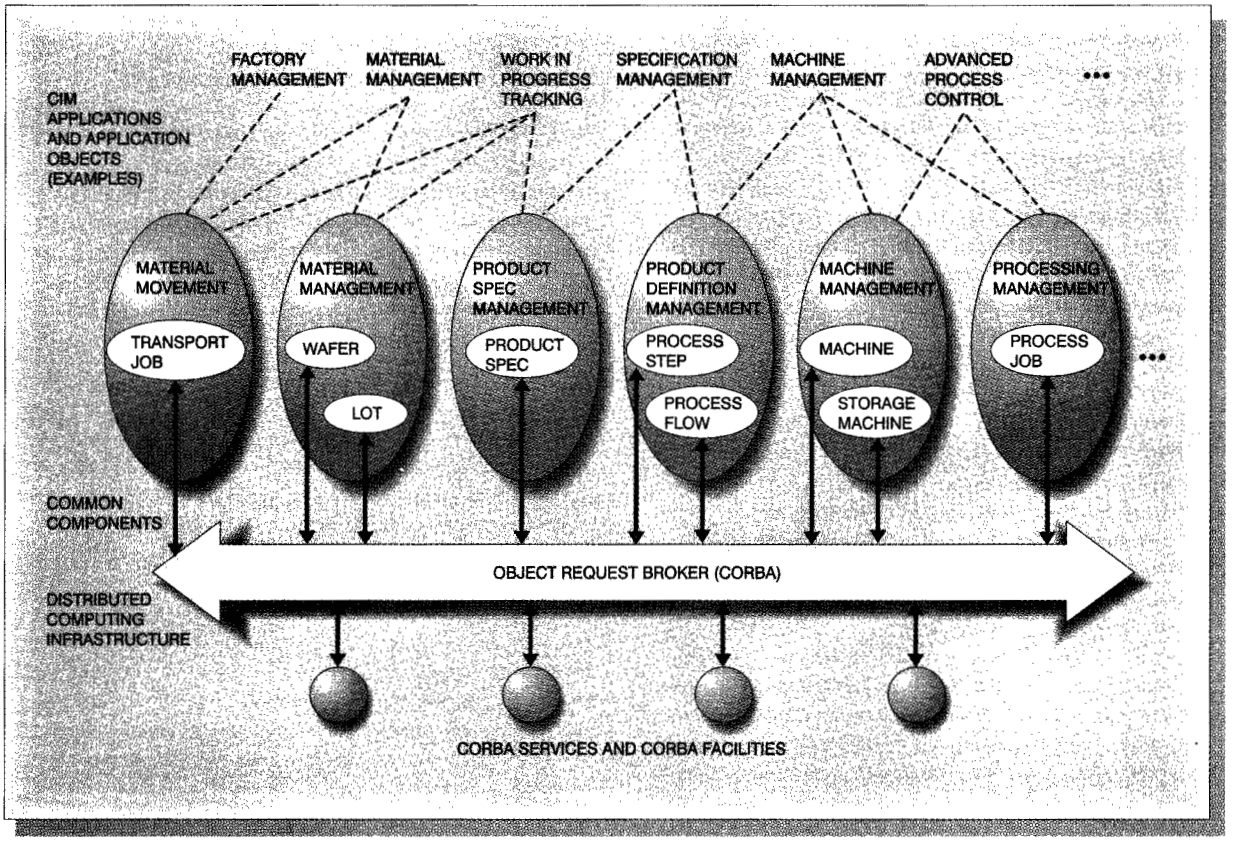
Suppliers of semiconductor MESs and users of these systems have collaborated to specify the standard partitioning and capabilities for a marketplace of commercial MES solutions. Development of the architectural specification consisted of an initial version in 1991 with several subsequent iterations of using the framework over a seven-year period. A robust change methodology was used to handle the changes requested from users of the initial specification. This is an example of framework definition that is not explicit about c/v. Commonality and variability are discovered, through iteration, while using the framework. Although iteration is needed, relying on it exclusively can lead to long development periods.

Implementation of the c/v is done by architecting common components and common services and by using design patterns with components for variability points. Interfaces to the classes and components are well specified and include contracts between the components specifying the services exchanged, the events recognized, the data passed, and the constraints enforced.

Mention of interfaces or special tools to facilitate customization of the components is not made and is not really appropriate in this instance. There is documentation of the architecture specification for suppliers of the components and applications. Developers are meant to follow these interface specifications.

Implementors use the specification of the CIM framework to develop their own versions of the components and applications and rely on inheritance and

Figure 6 The SEMATECH architecture*



* From D. Doscher and R. Hodges, "SEMATECH's Experiences with the CIM Framework."
 Communications of the ACM **40**, No. 10 (October 1997).

delegation of responsibilities to lower-level components to extend and add their own variations on the specification.

Suppliers of the components may add to, but cannot delete from the specification. This will ensure interoperability between components from different suppliers and will allow for upgrades on later versions. However it will not ensure that versions with extensions to the specifications will work well together.

Mechanisms for customization

This section provides a more generalized description of the mechanisms to be used in the customization life cycle. It is intended to support instruction on, and methods to develop customizable assets, applications, and solutions. While there is a cost associated with performing this level of analysis and representation, the benefits in terms of flexible, easily customizable systems can far outweigh the cost in the longer term.

Methods for analyzing c/v. The need to explicitly analyze and represent commonality and variability in enterprise systems and families of solutions is only beginning to be understood today. The goal is creation of customizable systems that are meant to be customized by other developers. Without an explicit analysis of the commonality and variability points required in our systems, it will be difficult to ensure their presence. Three types of analysis techniques exist: example-driven, requirements-driven, and architecturally focused. Many of these methods can be used individually or in concert.

Example-driven techniques. Two types of example-driven techniques exist:

- Domain analysis for reuse techniques. These include FODA (feature-oriented domain analysis)[4] and ODM (organization domain modeling).[5]
- Hot spot definition for representing variability points of frameworks[3]

These systems rely heavily on analysis of existing systems and use of domain experts. Existing systems are analyzed and their commonality and variability are explicitly represented in models at the design and implementation levels.

Requirements-driven techniques. Two different mechanisms exist that rely heavily on requirements and domain experts:

- Use cases with "uses" and "extends" to represent commonality, variability, and optionality[3,8]
- Definition of common business processes and associated common functions and data models, with variation points noted, as in SAP[12]

These mechanisms focus on analysis of requirements and business processes and explicitly represent commonality and variability at this level. This representation is then followed by further elaboration of this c/v at design and implementation levels, as well as in architectural models.

Architecture-focused technique. Another way to explicitly represent c/v is through an architecture that describes the common parts and the rules for how and when these parts can be configured together and which parts are optional (configuration rules).[9] These configurations are driven by understanding the features of a family of solutions, with points of variability at an architectural level.

Mechanisms for implementing c/v. Several different mechanisms for incorporating customizability into assets, components, and solutions have been discussed briefly in the examples. This section of the paper collects these mechanisms into one place. Some of these mechanisms can be used at design time, while others are applicable at compile or run time.

Layered architectures support separation of concerns at an architectural level.[8,13] This separation may be only conceptual or logical, or it may actually be at a physical level. This physical representation would exist as an application programming interface or programming model definition that abstracts the functional specifics of the layer away from its users. This mechanism is useful for partitioning commonality.

Components, subsystems, and packages are mechanisms for partitioning commonality and providing modularity in systems.[8] This modularity supports subsequent related variation points and configuration and version control. These structuring mechanisms should preserve the encapsulation of highly coupled portions of a system as well as the separation between them.

Design, analysis, business, and architecture patterns can be thought of as common ways to solve commonly encountered problems. These patterns are particularly useful because they also allow for needed variances. The paper by Lloyd and Galambos[15] in this issue defines technical reference architectures as an example of architectural patterns. The paper by McDavid[16] in this issue defines business patterns and describes how they can be specialized as well as mapped into requirements. Analysis patterns have been described by Fowler[17] and by Coad et al.[18] Design patterns[4,7] were discussed earlier. All of these patterns are most useful when considered as mechanisms to implement the required commonality and variability of a system.

Multiple versions allow variation. This mechanism can be used to produce multiple versions of a class, component, or even a solution. For example, different versions of a solution for different geographies or different industries could be developed. This would be an alternative to producing one common base, which could then be customized. The choice should be made using a cost/benefit analysis of the situation, including analyzing the cost and difficulty of maintaining multiple versions.[12]

Elaboration points in analysis and architectural models are discussed by Lloyd and Galambos. The technical reference architectures can be modeled at a logical and physical level, with commonality shown at the logical level and specialization or variation shown in the physical elaboration of the logical level model. Consider, for example, that every customer has a different physical information technology environment as a result of history. Some will be MVS (multiple virtual storage) and CICS*; some will be UNIX**; etc. Each will have a different affinity for individual vendors. Some customers will be more prepared to

plunge into new technology than others. So there is variability here.

At the same time, many companies need to do transactional processing, and most have call centers. It is possible to define logical technical architectures that are standard patterns, valid across a wide customer set, to represent such domains as transactional processing. That represents commonality. Variability can be accommodated by mapping capabilities, and by defining architectures and designs at the physical level (for example, "transactional processing on MVS," "transactional processing on AIX*") that share a common logical architecture.

Abstract classes allow behavior and data to be defined abstractly, then instantiated at the time of customization.[3]

Subclassing and inheritance allow common data and behavior to be defined concretely in a class, which is then subclassed to specialize each variation needed.[3,9]

Parameters defined for methods within a class allow for variances in behavior at run time. Functions and procedures can also be parameterized to allow for variability.[9,12]

Templates can be used to describe the generic structure of a class, component, or data structure. These might include points of variability as parameters. Templates are often used in conjunction with generators and wizards to help users instantiate the templates for their particular variances.[8,13]

DLLs (dynamic link libraries) support run-time linking of different libraries. These allow variability in the configuration of the software of a system at run time.[8]

Customer exits specify points in a function or procedure where it is expected that customers will need to vary its behavior. This technique could support, for example, variations in business rules for different industries or geographies. This provides not only variability but also supports the explicit tracking of changes made during the customization of a solution.[12]

Stored procedures can be attached to databases. This technique allows variation in the procedures to be triggered in association with state changes in database systems. It also provides performance enhancements in systems with high transaction rates.

Parameter tables are used in many packaged solutions to support customization of data and function parameters. This technique collects the parameters and groups of parameters that can be customized into one area.[12]

User profiles define potential users of a system. This technique supports installation of different portions of a system, depending on the user profiles selected.

Install scripts define what components or modules are to be present in a system. This technique allows different components to be installed for different systems.[12]

Configuration supports the choice of alternative functions and implementations. These can be in the form of classes, components, functions, procedures, packages, etc. This technique requires a modularized system that can be configured in different (varying) ways.[9]

Configuration rules define which parts of a system rely on which other parts. Such rules ensure that installed functionality selected by an end customer will have all required supporting components of the system installed as well. They also specify required and optional parts of a system.[9]

Facades and component interface extensions allow commonality across multiple interfaces on a component. Facades are particularly important when the components contain variability points within them, and where the variability points are associated with particular interface definitions.[8]

Properties, property sheets, and customizers are the mechanisms used in JavaBeans** to support definition and customization of bean attributes.[13,19]

Import/export mechanisms allow reusable entities such as classes, components, models, etc., to be exported and made available for subsequent importing into a particular system or solution under development.[8] The entities can be exported because they have been defined in relation to an overall generic architecture of a family of solutions. This not only allows the definition of reusable entities; it also ensures that they will interoperate in a syntactic as well as a semantic manner to solve a family of related business problems.

A *registry* defines and maintains the objects and components contained in a system. It allows variance in the objects that are registered.

Adapters and connectors support the connection to multiple legacy and database systems. These adapters form a generic interface.

Iteration refers to finding c/v through several iterations or versions of a component or solution. Iteration is used to determine the required variability points for different industries, countries, customers, etc. This mechanism is typically recommended as the way to build frameworks, for example.[4]

Interface definition mechanisms. There are several mechanisms that can be used for defining interfaces into customizable assets, components, applications, and solutions.

Programming models document, and provide a definition of, how portions of a system are to be used from an interface perspective. This can include the actual application programming interface for part of a system, or define the rules for using or extending some part of a system. Examples include the programming models of the San Francisco project in IBM.[13,19]

Wizards support developers in accessing and using various programming models, templates, and component definitions.[8,13]

Generators produce code "skeletons" and "build" files from input templates, scripts, instantiated parameters, and rules for specialization.

Component interface definition supports separation of interfaces from implementation. Examples include CORBA's IDL (Interface Definition Language), Microsoft's COM (Common Object Model), JavaBeans containers, and the function module definition in SAP's ABAP/4.[12,13]

Tool support for interface definition mechanisms includes:

1. Development workbenches, such as ABAP/4, which provide support for development of new functional modules and components.
2. Scripting languages, such as those in ABAP/4, Java, or Lotus Notes**, that support "gluing together" existing components.

Customization mechanisms. Given that customizability has been implemented in the assets, components, and solutions, we can describe several general ways in which these assets, components, and solutions are then customized, by developers and deployment teams, to develop end-user customer solutions. Examples include:

1. Assembly and configuration of parts, components, and functional modules
2. Scripting, or gluing together parts, components, and functional modules
3. Framework completion in either a white-box fashion, where the code is adapted at variation points, or in a black-box fashion, where existing objects and components (variants) are configured into the defined extension (variation) points[10]
4. Parameter setting for methods, functions, procedures, tables, templates, etc.
5. Architecture topology determined to be fat client, thin client, Internet based, etc. Other technical architecture decisions must also be made, including install vs execute points, persistence mechanisms, etc.[12,13,19]
6. Extension of the system through subclassing, overriding methods, extending component interfaces, adding new functions or components, etc.
7. Integration with other applications, and linkages made to customer business processes, including workflow and transaction management systems

Versioning mechanisms. We can consider versioning mechanisms as ways to ensure that it will be possible and easy to fix bugs and for customers to upgrade to new versions. This requires easy evolution of the system, based on the changes made to it in the field and on new variations or changes requested by customers. All versioning mechanisms require some level of change management, but this gets much harder if code is actually changed. Use and extension of the system should trigger the capture and feedback of these extensions to the core development group for consideration in new versions. Example mechanisms include:

1. Usage of defined interfaces only—preventing changes to implementation code at other than specified points of variability will ease version management
2. Extensions to the interfaces allowed—no deletion of any interface elements, interfaces, and implementations
3. White-box frameworks—will be more difficult to update, code changes will need to be tracked and managed
4. Black-box frameworks—variations are handled

through configuration and thus will make versioning much easier than white-box frameworks
5. Packages—versions of packages can be supported
6. Versioning of data and legacy mapping
7. Versioning tools—support change management across versions

Summary

The major consideration in this paper is the creation of assets, components, and solutions that are customizable, based on defining and developing common, yet variable components. For enterprise-level systems, this requires viewing the system to be built as being not just for the end user, but for other system developers whose job it is to fit, assemble, configure, customize, and alter the system into a final customer solution, possibly also developing additional customizable applications in the middle of a "value chain."

Development of customizable systems requires specification of a generic architecture for multiple integrated applications in the enterprise, with specifications for the common components used to build these applications and configuration rules for developing the final applications and solutions. Development of the generic architecture requires different analysis and representation techniques and work products from those meant for only a small number of applications. Development of these types of systems is usually most effective if the component development is separate from the application and final customer solution development. At the center of development and deployment of customizable systems is the notion of commonality and variability analysis and a customization life-cycle method.

Systems in the future will require more emphasis on customization, and use of the customization life cycle defined here will be necessary for success in the solutions business.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of SAP AG, PeopleSoft, Inc., Microsoft Corporation, Object Management Group, Sun Microsystems, Inc., Lotus Development Corporation, or The Open Group.

Cited references and note

1. A "silo" is a part of an IT solution that is not integrated with other parts. The name comes from the visualization of such a solution—the different applications stand apart from one another, like farm silos in silhouette against a prairie sky.
2. *Webster's Ninth New Collegiate Dictionary.*
3. I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley Publishing Co., Reading, MA (1992).
4. W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley Publishing Co., Reading, MA (1995).
5. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, *Feature-Oriented Domain Analysis Feasibility Study: Interim Report*, CMU/SEI-90-TR-21 Technical Report (August 1990).
6. M. Simos, "Organization Domain Modeling (ODM)," *Proceedings of the ACM-SIGSOFT Symposium on Software Reusability*, Seattle, WA (April 1995).
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading, MA (1995).
8. I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Addison-Wesley Publishing Co., Reading, MA (1997).
9. M. Griss, J. Favaro, and M. D'Alessandro, "Developing Architecture Through Reuse," *Object Magazine* **7,** No. 7, 35–41 (September 1997).
10. R. E. Johnson, "Frameworks = (Components + Patterns)," *Communications of the ACM* **40,** No. 10 (October 1997).
11. D. Leishman and S. Fraser, International Conference on Software Engineering, 1995.
12. R. Busk-Emden and J. Galimow, *SAP R/3 System: A Client/Server Technology*, Addison-Wesley Publishing Co., Reading, MA (1996).
13. K. Bohrer, "Architecture of the San Francisco Frameworks," *IBM Systems Journal* **37,** No. 2, 156–169 (1998).
14. D. Doscher and R. Hodges, "SEMATECH's Experiences with the CIM Framework," *Communications of the ACM* **40,** No. 10 (October 1997).
15. P. T. L. Lloyd and G. M. Galambos, "Technical Reference Architectures," *IBM Systems Journal* **38,** No. 1, 51–75 (1999, this issue).
16. D. W. McDavid, "A Standard for Business Architecture Description," *IBM Systems Journal* **38,** No. 1, 12–31 (1999, this issue).
17. M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley Publishing Co., Reading, MA (1996).
18. P. Coad, M. Mayfield, and D. North, *Object Models: Strategies, Patterns and Applications*, Yourdon Press, Englewood Cliffs, NJ (1996).
19. V. D. Arnold, R. J. Bosch, E. F. Dumstorff, P. J. Helfrich, T. C. Hung, V. M. Johnson, R. F. Persik, and P. D. Whidden, "IBM Business Frameworks: San Francisco Project Technical Overview," *IBM Systems Journal* **36,** No. 3, 437–445 (1997).

Accepted for publication November 5, 1998.

Deborah A. Leishman *IBM Global Industries, 3039 Cornwallis Road, Research Triangle Park, North Carolina 27709 (electronic mail: leishman@us.ibm.com).* Dr. Leishman joined IBM in 1995 and is an IBM Senior Technical Staff Member. She currently leads the ESS development team for IBM Global Industries. Dr. Leishman received a Ph.D. degree in computer science in 1994. She has worked on object-oriented systems and focused on reuse for several years, and prior to joining IBM she worked as reuse manager on a large architecture-driven framework development project. Dr. Leishman has also worked as a development manager for spatial data products and as a researcher for knowledge-based systems.

Reprint Order No. G321-5698.