Business Application Components

Tom Digre

Senior Member Technical Staff

Business Application Components
Information Technology Group
Texas Instruments, Inc.
6620 Chase Oaks Blvd. MS: 8417
Plano, Texas 75023
E-mail:digre@ti.com Fax: 214-575-2866

ABSTRACT. *Information Technology is being driven by the need for rapid provisioning of business solutions within an environment of increasing complexity. Component-based architecture principles and complexity-hiding model-based development techniques will empower users to dynamically change business processes, workflows, rules, policies, presentation, and other aspects of their environment.*

KEY WORDS: *component, development, architecture, semantics, complexity.*

1. Introduction

Global business competition and a shift from commodity to custom products has created an environment of continuous business structure change. In order to effectively compete, businesses are constantly revisiting products, processes, suppliers, and customer care-abouts. As shown in "Figure 1 Business Drivers", a primary business driver for Information Technology includes the profit-oriented objective to decrease time-to-market for products and services **[WIRTHMAN 95]** within an environment of increasing business complexity **[MARTIN 91]** resulting from accelerating changes to products, processes, customers, partners, and Information Technology. A successful information strategy will accommodate these business drivers by provisioning business solutions at a rate commensurate with the increasing rate of business structural change.

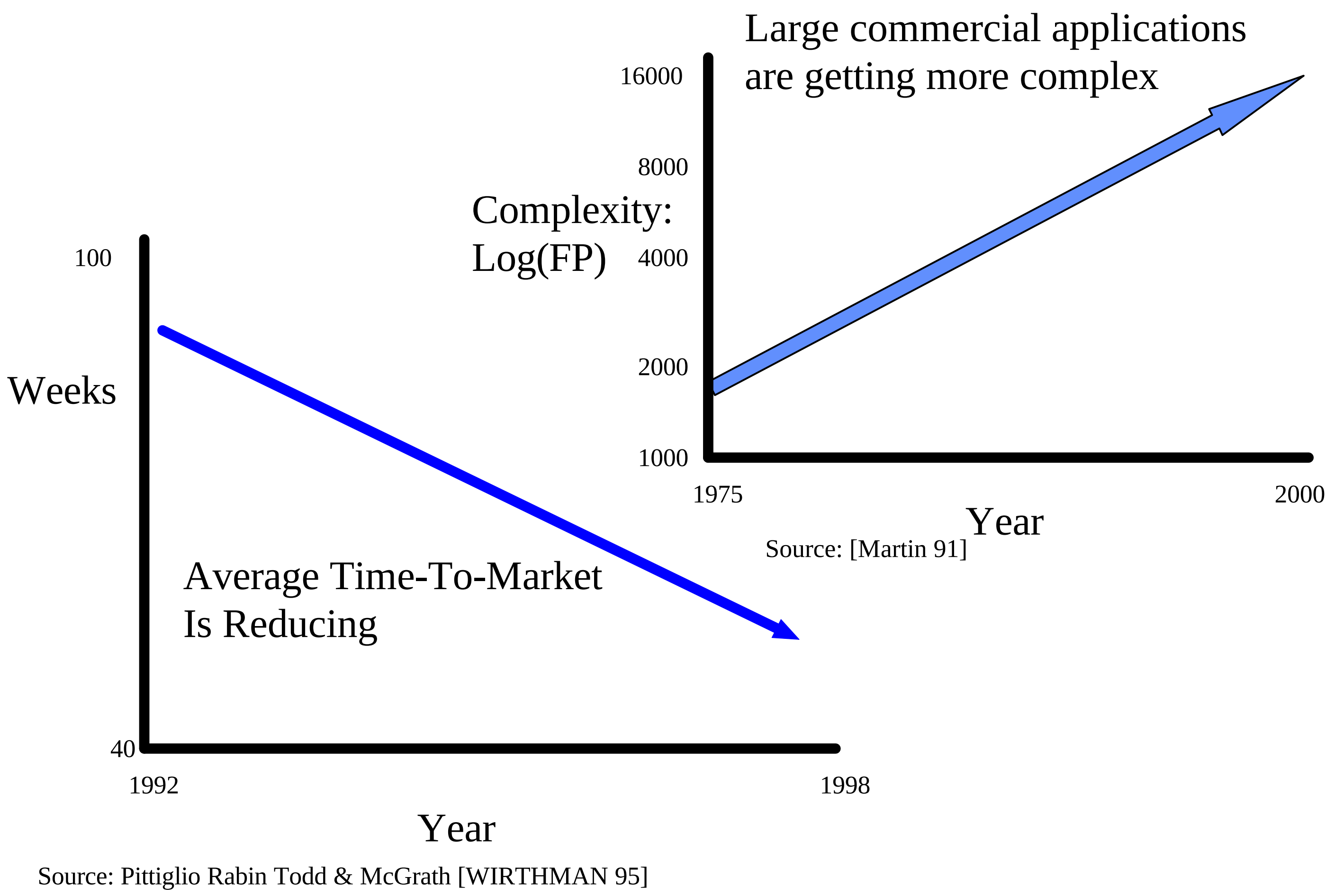


Figure 1 Business Drivers

Rapid solution delivery in response to continuous business process changes requires direct involvement of empowered users to dynamically change their business processes, workflows, rules, policies, presentation, and other aspects of their environment. Impediments to achieving these goals have been technological and organizational barriers between business and the enterprise's information technology. Enterprise IT organizations are on the critical path to achieving the IT productivity, performance, and cycle time gains necessary to support business change.

2. Coping with Complexity

IT organizations are severely challenged to meet the solution provisioning cycle time requirements imposed by business. These organizations are facing the dilemma of minimal software productivity improvements, particularly when using third generation languages such as COBOL, C++, and Smalltalk. A fundamental problem with software solutions implemented using 3GL is the inability to cope with increasing business complexity. As depicted in “Figure 2

Software Complexity", as complexity increases there is an adverse impact on productivity, quality, delivery cycle time, and cost **[MARTIN 91]**.

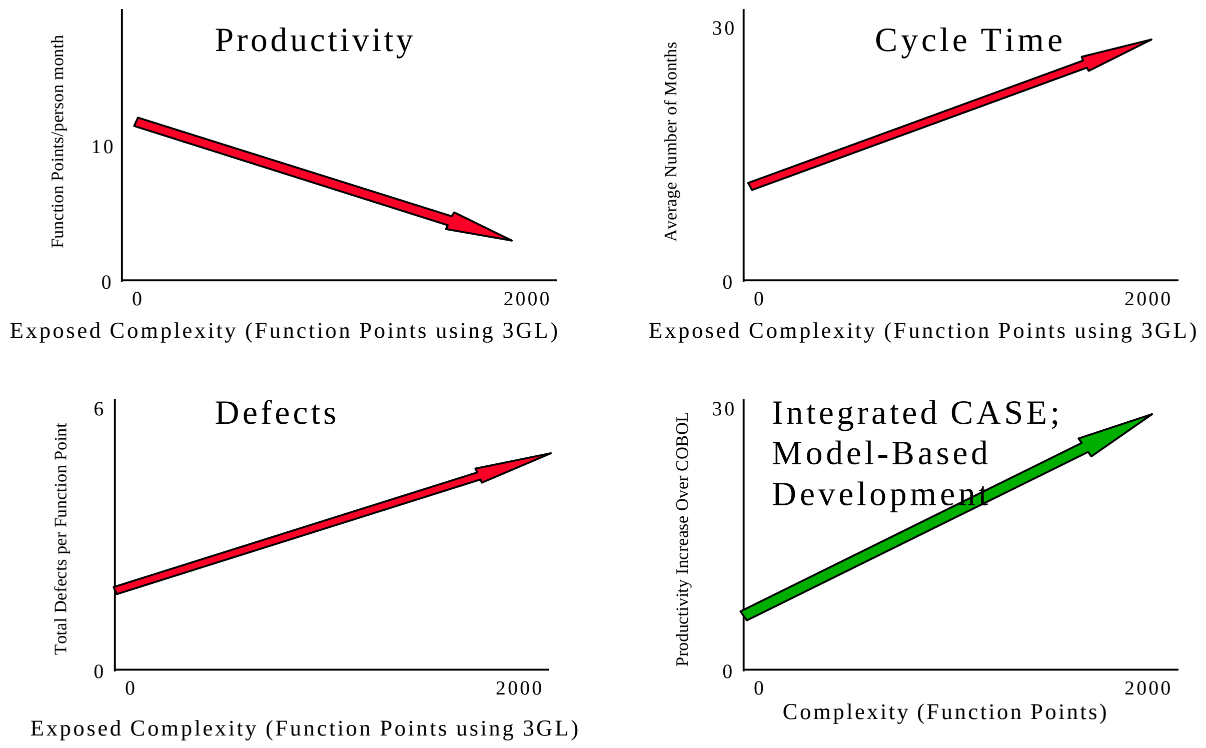


Figure 2 Software Complexity

Software productivity depends not only on sheer bulk but also on "surface area", the number of things that must be understood and properly dealt with in order to successfully enable interoperability between software components **[COX 87]**. Factors influencing surface area include:

- Amount of visible information. Surface area increases with the number of names that are exposed through the software component interface, including data element names, data types, and function names.
- Sequence dependencies. Surface area increases with each requirement that the software component user must perform operations in a particular order.
- Environment and responsibility scope dependencies. Surface area increases whenever the software component user is responsible for managing lifecycle, persistence, location, or environmental aspects of software components within a more global application context.
- Technology dependencies. Surface area increases with exposure to each technical domain and form of interface, including middleware communications, data storage.
- Concurrency. Surface area increases when concurrency issues are exposed to the software component user.

Use of third generation languages typically increase exposed “surface area” proportional to increases in underlying complexity. This is the fundamental problem: the complexity of the entire IT solution space, including the underlying IT infrastructure, is exposed to the solution provider. It is the exposed “surface area”, not the underlying complexity, that impedes progress. Contrast software productivity with gains achieved in hardware componentization, as shown in “Figure 3 Need for Componentization Concepts”. Hardware complexity has increased by nearly 2^{30} in the last 30 years, and it has been accomplished with minimal increase of surface area. While semiconductor (and downstream hardware) technology has consistently sustained an annual doubling of productivity and performance gains, significant software productivity gains have not materialized. Business application software continues to be characterized by surface area increases commensurate with complexity increases.

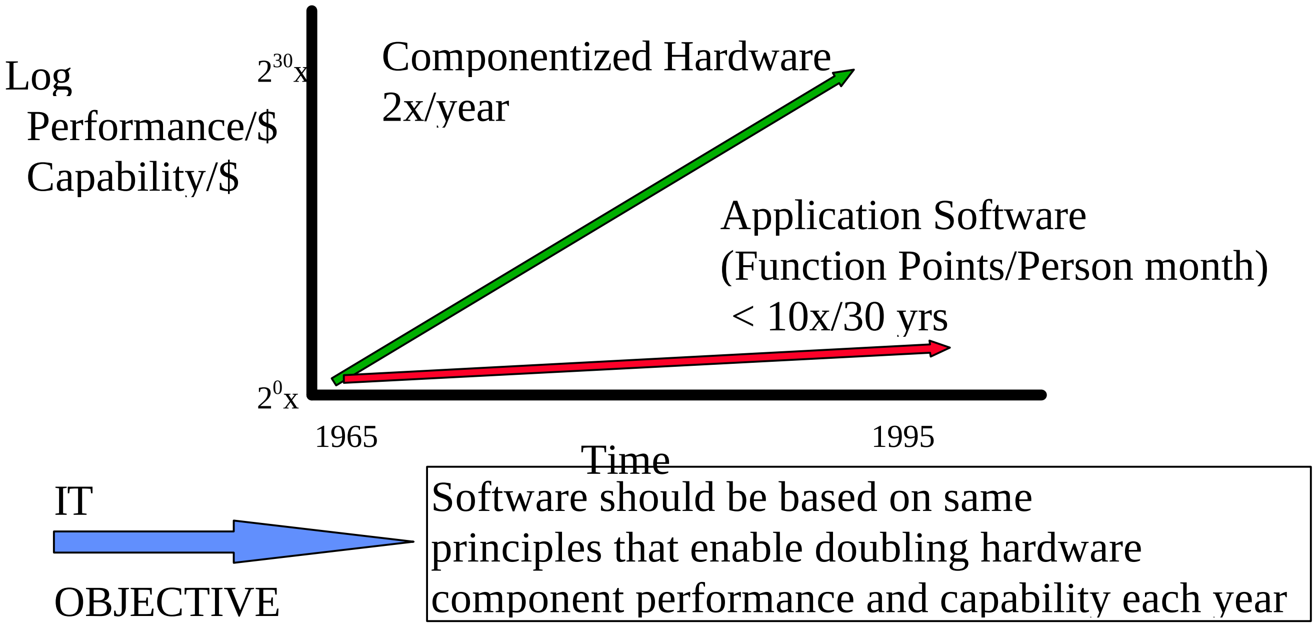


Figure 3 Need for Componentization Concepts

The success of the semiconductor industry to successfully manage complexity inspired Brad Cox to reflect on the potential for software componentization (as paraphrased from **[COX 87]**):

Gordon Moore, the chairman of Intel Corporation, once predicted that the number of ***components*** *on a silicon integrated chip would continue to double yearly. The prediction, now known as Moore’s law, has held up remarkably well during the twenty years I’ve been in this business. One of its many implications is that during the same period that my productivity has been growing arithmetically with each improvement in programming language technology, the productivity of my friends in the hardware industry has been growing geometrically as the capability of the building blocks that they work with doubled each year for twenty years. My productivity certainly improved in moving from assembly language to FORTRAN to C to Lisp. But it certainly did not improve by the million-fold increase implied by Moore’s law:* $2^{20} = 1{,}048{,}576$.

Board designers routinely reuse the work of circuit designers, who reuse the efforts of workers at even lower levels - wafer manufacturers, mask fabricators, printing shops. Communication channels (trade magazines) allow suppliers to communicate with potential consumers, and consumers have catalogs, filing systems, performance tests, price comparisons for selecting among multiple suppliers. The impressive vitality of the marketplace in reusable hardware ***components*** *is legendary.*

The appeal of all this is the possibility that the software industry might obtain some of the benefits that the silicon chip brought to the hardware industry; the ability of a supplier to deliver a tightly encapsulated unit of functionality that is specialized for its intended function, yet independent of any particular application. The silicon chip is the unit of hardware reusability that has most conspicuously contributed to the hardware productivity boom. Might the Software-IC concept do the same for software?

Hardware IC componentization is an iterative process of encapsulating function, then using that function as the semantic basis for the next iteration of function delivery. Each iteration is starting at a higher conceptual level. The geometric function increase did not occur by going back to NAND gates, it occurred by raising the conceptual foundation for each iteration. The component user "assembles" the solution from increasingly sophisticated components to satisfy his unique engineering requirement. He assembles logical components directly into Application Specific Integrated Circuits. The resulting aggregate component has its function integrity maintained across the continual changes of infrastructure resulting from manufacturing process technology improvements. To protect both the consumer and the manufacturer, component reuse occurs at the design level and is critically dependent upon semantic clarity of exposed functionality.

Exploitation of these component concepts within the software domain will be aided by open standards for specification of component syntax and semantics; the existence of a marketplace (suppliers and consumers) for reusable software components; and tools for the construction of components and consequent assembly of components into solutions. Together, the component concept and supporting tools will enable the business user to:

- Directly provision his business solution.
- Express his specification in a form consistent with the problem domain.
- Focus "surface area" exposure on the business problem domain. All underlying technological and implementation complexity will be hidden.

Software is often viewed as creative technological "art" derived from solitary, mental, abstract activity akin to mathematics or novel writing. However, there is increasing consumer demand to establish manufacturing and engineering discipline in the production of software, driven by the need to build and use commercially robust repositories of trusted, stable components whose properties can be understood and tabulated in standard catalogs, like the handbooks of other mature engineering domains **[COX 90]:**

The denizens of the software domain, from the tiniest expression to the largest application, are as intangible as any ghost. And because we invent them all from first principles, everything we encounter there is unique and unfamiliar, composed of components that have never been seen *before and will never be seen again, and that obey laws that don't generalize to future encounters. Software is a place where dreams are planted and nightmares harvested, an abstract, mystical swamp where terrible demons compete with magical panaceas, a world of werewolves and silver bullets. As long as all we can know for certain is the code we ourselves wrote during the last week or so, mystical belief will reign over quantifiable reason. Terms like `computer science' and `software engineering' will remain oxymorons -- at best, content-free twaddle spawned of wishful thinking and, at worst, a cruel and selfish fraud on the consumers who pay our salaries.*

The `software industrial revolution' means ... transforming programming from a solitary cut-to-fit craft into an organizational enterprise like manufacturing. This means allowing consumers at every level of an organization to solve their own software problems as home owners solve plumbing problems: by assembling their own solutions from a robust commercial market in off-the-shelf subcomponents, which are in turn supplied by multiple lower level echelons of producers.

3. Component-based Solution Provisioning

A component-based IT architecture is founded on well-defined information components that can be specialized and/or assembled into applications. The architecture recognizes that components, including tools and services, can be purchased from external vendors (e.g., desktop tools); specialized and assembled by users (e.g., user-developed financial analysis); or developed by the IT provisioner. The component-based architecture defines the platform, or board, on which these reusable components can fit together.

Cycle time reduction goals require active participation by the business user. Ultimate cycle time reduction will occur when the business user is able to express his problem in a form consistent with his problem domain and have that specification automatically and instantly implemented. Building on component concepts and the need for solution provisioning directly by business users, an open component-based architecture will feature plug and play components and end-user empowerment. It is recognized that the goal of end-user solution provisioning will require cultural changes within the enterprise work force leading to a sharing of responsibility between the business user and the Information Technology provisioner.

Within the component-based architecture scenario, the business user will assume the responsibility for solution provisioning, as shown in “Figure 4 Component-based Solution Provisioning”. The role of solutions provisioning will migrate from the enterprise’s Information Services organization to the business. The business user will be empowered to:

- Administrate business rules, work flows, and presentation.
- Specialize and assemble components into solutions using Enterprise modeling tools.

The Information Technology provisioner will provision reusable components that can be combined to form solutions by the business user. The IT provisioner will:

- Provision components through purchase, construction, or specialization and assembly of finer-grained components.

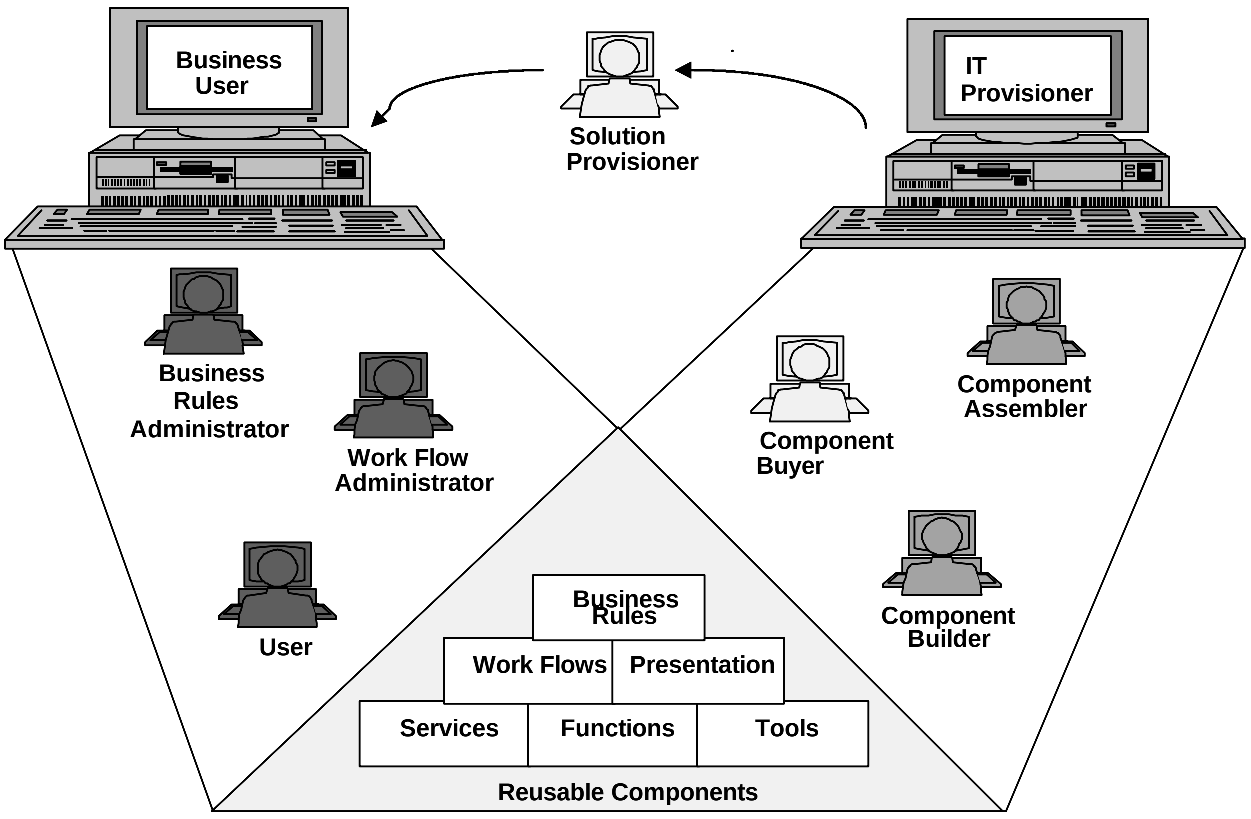


Figure 4 Component-based Solution Provisioning

- Encapsulate purchased and legacy systems so they can be utilized as reusable, inter-operable components.
- Share Components with business users. The shared components include business rules, work flows, presentations, services, functions, and tools.

All provisioning roles are inextricably tied to components. All roles specialize and assemble components. In most cases, the product of assembly is itself a reusable component. Cycle time reduction and business semantic integrity is ultimately dependent upon end-user assembly of these components.

When a business condition changes, the resulting modification to software is envisioned to occur only in those components directly associated with the changing business specification. Components will be loosely coupled, the impact of changes will be isolated to those components associated with the business semantic.

4. Component-based Architecture

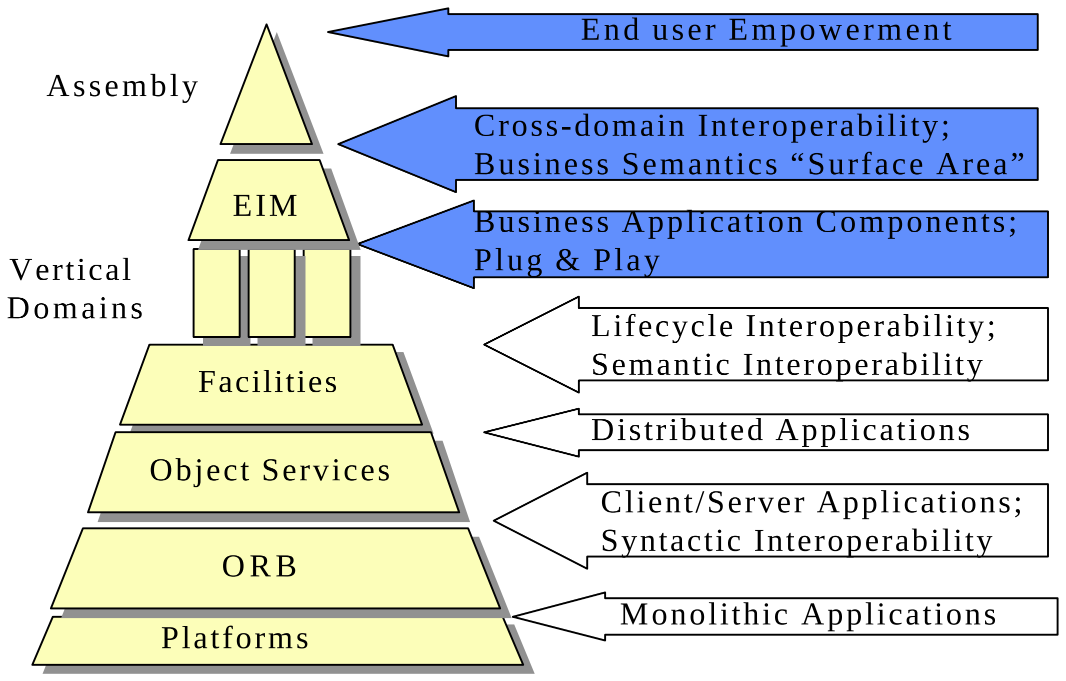
A component-based architecture will help achieve the vision of business user solution provisioning. The component-based architecture shown in “Figure 5 Component-based Architecture” is built upon layers of technology successively enabling capability on the path towards user-empowered solution assembly. The architectural structure is motivated by OMG-defined architectures **[OMG 95.01.02, OMG 93.12.29, OMG 95.01.12]** augmented with the concept of an **Enterprise Integration Model** that exposes relevant, domain-specific semantics of business application components to a suite of user empowerment tools.

The technology layers of the architecture include:

- **Platforms**, operating systems, database management systems, networks, and other fundamental software infrastructure components. This layer of technology currently enables rapid and transparent installation/replacement of tailorable infrastructure components within a broad range of reliability, performance, capacity, and price characteristics. These infrastructure components are no longer on the critical path to achieving objectives such as business solution cycle time reduction and end-user empowerment. Applications built directly on this layer are typically characterized as monolithic and centric.
- **ORB**(Object Request Broker). Using any of a variety of middleware communications mechanisms, this layer of technology typically enables construction of client/server applications founded on syntactic interoperability. CORBA is an example implementation **[OMG 93.12.29]**. The following features characterize components using this technology layer:
    - Implementation and location transparency. Clients are unaware of the location of server software components or their implementation details. Component implementation and location can change dynamically, without any modification to the client.
    - Universally accessible on federated enterprise ORBs.
    - Physical black-box encapsulation of related services.
    - Programming language independent. All component public interfaces are defined in IDL(Interface Definition Language).
    - Extensible, reusable specifications. Interface specifications can be inherited.
- **Object Services**. A set of fundamental services necessary to implement distributed applications. The services encompass distributed concepts of object relationships, concurrency, persistence, transactions, etc. **[OMG 95.01.12]**
- **Facilities**. Industrial adoption of a business component interoperability standard is on the critical path to definition and implementation of Business Application Components. A business component interoperability standard would enable specification of Business Application Components having the following characteristics:
    - Rigorous Semantics which enable plug-and-play, extensibility integrity, comprehension and control by business user.
    - Fully traceable life cycle, from business process engineering through deployment and continuing to final disposition.
- **Vertical Domains**. For each business domain, a set of plug and play Business Application Components compliant with industry-standard specifications. Industry standards are expected to evolve from activities of OMG Special Interest Groups and industrial consortia such as SEMATECH and OAG.
- **Enterprise Integration Model**. Provides for tailoring and integration of vertical domain components within an enterprise. Characterized by:
    - Context sensitivity, which enables domain-specific specification while maintaining cross-domain integrity and integration within the enterprise.

- Complexity-hiding views which aid the process of empowering business users to directly provision their solutions.
- Reuse through concepts of abstraction context inheritance, model-based specifications, and synthesis.
- Versionable configurations of components.

- **Business User Empowerment**. A suite of tools interoperating with the **Enterprise Integration Model,** including business rule tools, work flow tools, presentation tools, software assembly tools, data access tools, and integrated CASE tools.



Assembly
EIM
Vertical Domains
Facilities
Object Services
ORB
Platforms
End user Empowerment
Cross-domain Interoperability; Business Semantics "Surface Area"
Business Application Components; Plug & Play
Lifecycle Interoperability; Semantic Interoperability
Distributed Applications
Client/Server Applications; Syntactic Interoperability
Monolithic Applications

Figure 5 Component-based Architecture

5. Model-based Development

The goals of the component-based architecture include:

- Business Semantic Integrity
- Rapid provisioning of business solutions;
- Reduction of "surface area".

These goals are coupled with an environment of increasing complexity in the areas of:

- Business.
- Technology.

- Architecture. The component-based architecture which has been outlined has, perhaps, orders of magnitude more implementation complexity than contemporary application architectures.

As discussed in the “Coping with Complexity” section, there is a need to identify principles which would enable geometric productivity gains in software provisioning. Such gains are plausible using a model-based development approach. Such an approach would handle increasing complexity with minimal gain of surface area. The basic principle is to iteratively utilize generic constructs formulated at one level of abstraction as the “language” and semantic for the next level of specialization.

The complexities of using Object Services directly from 3GL code will be prohibitive for most application development organizations. Consequently, model-based development will practically become a necessity above the “ORB” layer of the architecture, and will likely encompass all architectural layers. Although model-based development is applicable to all architectural layers, the “Enterprise Integration Model” encompasses the most demanding aspects of model based development.

The (conceptual) Enterprise Integration Model (EIM) requires a set of tightly coupled concepts to support Business Application Component Engineering. Some of the concepts of EIM are reflected in conventional OO methodologies and toolsets. However, even for those few concepts which are conventionally available, the current implementations are disjoint, incomplete, and lack automation, rigor, consistency and integrity. Model-based development at the Enterprise Integration Model layer include the following, largely orthogonal, concepts:

- Life cycle
- Behavior
- Relations
- Specialization
- Context
- Genericity
- Version
- View

5.1. Life cycle

The concept of life cycle encompasses a set of inter-related activities and artifacts related to Business Application Components. The life cycle is generally partitioned into the following “phases”, but this should not necessarily imply that a “waterfall” lifecycle is being advocated:

- Business Process Engineering
- Analysis
- Design
- Implementation
- Construction
- Deployment
- Transition

Business Application Component Provisioning will generally be performed using the following life cycle strategies **[OMG 92.10.01]**:

- Non-staged, knowledge-based development. This approach does not place any management controls over the sequence of steps and their placement within stages. The steps are performed in any sequence, but the work is always constrained by the knowledge-base (rules of the method). For example the rules may prevent the recording of an operation unless it can be attached to an object type; or may define what constitutes a complete and correct object definition.
- Additive progression. Each stage adds more objects and more details to the model or design produced in the previous stage. This enables concurrent stage development, maintains traceability, avoids artificial "brick walls" between stages, and minimizes re-work when changes are needed in earlier stages. This approach avoids transformational strategies (including manual transformations). Note that a transformation may still be needed to "generate" the code and database designs of the production system. Enactable specifications are used during development to enable direct and immediate execution of designs.
- Incremental implementation. The underlying infrastructure, in conjunction with the EIM, enable incremental implementation of business application components and semantics. For example, adding new relationships (and enforced referential integrity) between components should be implementable without modification to any of the participant components.

Consistency must be automatically maintained between artifacts within all lifecycle stages. In particular, changes must be constrained by or propagated to, as appropriate, dependent artifacts in other stages.

Activities of the lifecycle, particularly during later stages, should be automated to the extent possible. Techniques include automated code generation of "construction" deliverables and knowledge-based synthesis of design artifacts from analysis.

5.2. Behavior

A model of objects should rigorously specify behavioral and static semantics. The model should rigorously and declaratively specify concepts of states, events, triggers, operations, attributes, constraints, invariants, pre-conditions, post-conditions, interactions, business rules, and transactional integrity.

5.3. Relationships

Some forms of business semantics can be expressed using the inter-object relationship concepts of a relationship model.

The relationship model within EIM enables specification of complexity-reduction concepts such as **virtual attributes** and **virtual functions [BAPAT 94]**. These concepts simplify usage of the EIM by enabling attributes or functions to "appear" as if they belonged in one object type, but are actually executed in a related object type (possibly via multi-staged relationship traversal). Similarly, **virtual relationships**, based on semantics of role-pairs, can be used to capture indirect relationships within a single specification element. Virtual relationships can be used to clarify or refine business semantics when specializing objects.

Aggregates are complex object types consisting of simpler object types called components. Components may themselves be aggregates. The concept of aggregation is a special case within the General Relationship Model. There is no semantic differentiation between aggregation and other forms of relations. Aggregation should be used within EIM to aid in hierarchically structuring solutions based on one or more perspectives of a problem domain. The EIM should allow specification of several independent aggregation hierarchies. Components may appear in one or more independent aggregation hierarchies. Within each aggregation hierarchy:

- A component can appear in one or more aggregations.
- Both concrete and abstract components can appear in an aggregation.
- A component in an aggregation can itself be an aggregation.
- Recursive aggregation is permitted (e.g., a component in the aggregation can be the aggregation itself).
- An aggregate has an independently specified cardinality for each of its components.

An aggregate may be subclassed. The new subclass may extend the aggregation definition by adding new components or subtyping existing components, providing it does not violate any rules of the aggregation superclass, including any cardinality constraints. Aggregates, as a form of relationship, utilize complexity reduction concepts of virtual attributes, functions, and relationships.

5.4. Specialization

EIM specialization/generalization concepts facilitate extensions to the behavioral model through inheritance mechanisms. Constraints on inheritance are well defined from **specialization theory [BAPAT 94]** and apply to permissible specializations of operations, attributes, states, pre-conditions, post-conditions, etc. Additionally, specialization theory interprets, defines permissible specializations of, and places restrictions on, relationships inherited by objects. These inheritance implications require a level of syntactic and semantic clarity in the object model which is often missing in conventional methodologies and toolsets.

5.5. Context

The roles and responsibilities of enterprise elements can best be expressed within domain-specific contexts **[DSOUZA 94]**. Domain-specific contexts reduce exposed complexity and enable visualization of semantically related elements. Contexts can be considered a set of overlapping views, or filters, of the Enterprise Integration Model. A context explicitly restricts exposure of enterprise element details, but the semantic constraints of the underlying model are implicitly maintained. Modal contexts may be used to provide controlled exposure (e.g., visual rendering) of model fragments such as nested aggregations.

5.6. Genericity

Genericity is the staged refinement of entire models (see “Figure 6 Genericity: Inheriting Industrial Models”) **[ESPRIT 93]**. The concept uses stepwise instantiation to go from aggregations of generic components, through increasing specializations of business domains, to enterprise, facility, and work area implementations. Genericity is a controlled process which utilizes principles of specialization, inheritance, relationships, and contexts to leverage reusable industrial models and rapidly provision tailored business solutions.

The concept of genericity should be used to specialize the "language" and semantics appropriate for each business domain. The business language should be defined in terms of higher level generic constructs. Subdomain specialization will in turn be defined in terms of domain constructs.

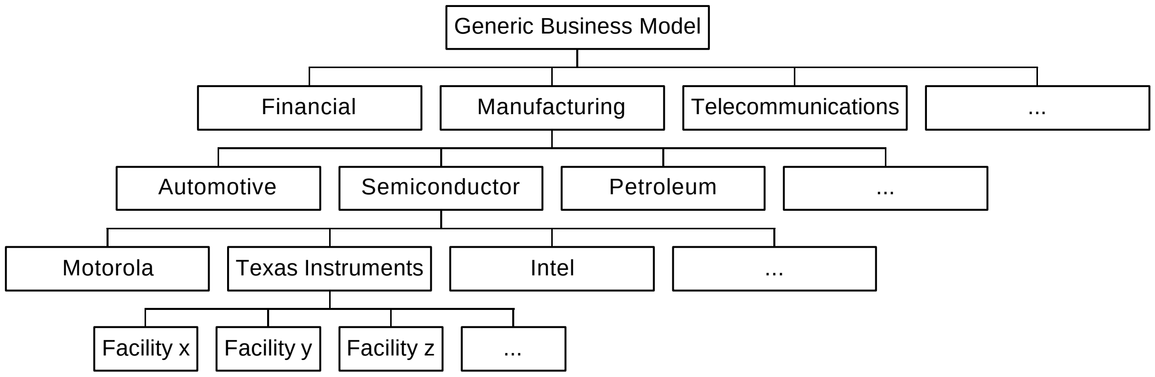


Figure 6 Genericity: Inheriting Industrial Models

5.7. Version

All elements within the EIM are continually evolving **[WJ 93]**. Versioning keeps track of the evolution of each EIM element. Closely associated with versioning is configuration management, which manages a set of individual versions of model elements.

5.8. View

View concepts provide the final reduction in exposed surface area and empower the end user to directly interact with the EIM. The view concept is based on **virtual object types [BAPAT 94]**, whose members are completely determined by certain criteria satisfied by members of another object type. The primary mechanisms which form the basis of virtual object types are selection (using arbitrary set-partitioning selection criteria), projection (e.g., list of attributes and functions to be exposed), and conjunction (related object types). Virtual object types are often coupled with security mechanisms to constrain attribute and operational access by end users.

Virtual object types, used in conjunction with other virtualizing concepts can significantly enhance semantic recognition by the end user while reducing or eliminating complexity clutter. The following forms of exposed surface area reduction (complexity hiding) are enabled by these concepts:

- Hiding attributes and operations.
- Hiding relationships by:
    - Compressing multiple object types into single virtual objects.
    - Compressing multiple relationships into single virtual relationships.
- Hiding objects (through selection criteria).

6. Example

As an example of how principles of componentization and the Enterprise Integration Model might work in practice, consider the business needs associated with manufacturing equipment. This single physical component within the enterprise is often viewed from many perspectives, including:

- Manufacturing process. Process specification, material delivery, monitor and control, quality, scheduling, capacity planning, etc.
- Facilities. Spatial requirements, safety, consumable delivery, liquid/gas/power/communication supplies, emergency procedures, installation, etc.
- Financial. Asset inventory, purchase orders, supplier contracts, depreciation, product manufacturing cost, etc.
- Operations. Operator training requirements, skills, scheduling, charge-outs, operating procedures, reprocessing, alignment, handling, etc.
- Maintenance. Parts bill of material, diagnostic procedures, periodic maintenance requirements, equipment performance, etc..

It is not unusual to find these perspectives implemented by completely independent systems which have little or no interoperability. A new business requirement crossing these boundaries may take considerable time to implement and would suffer semantic entropy during the process of communicating business needs to IT provisioners.

In an EIM scenario, each perspective could have been independently evolved, during different time frames, with incremental non-obtrusive implementation strategies. Employing the concepts of specialization and genericity, each perspective may have been independently derived from standard domain models, specialized down to the specific requirements of the work area. Contexts would be used to reduce exposed complexity within each domain while ensuring underlying enterprise model integrity. A new business requirement, which typically involves business rule, workflow, or GUI modification, would have a solution directly implemented by the business user. The business user would implement his solution utilizing EIM-aware tools in conjunction with a combination of EIM views exposing only the relevant business semantics (see “Figure 7 Example: Using the Enterprise Integration Model”).

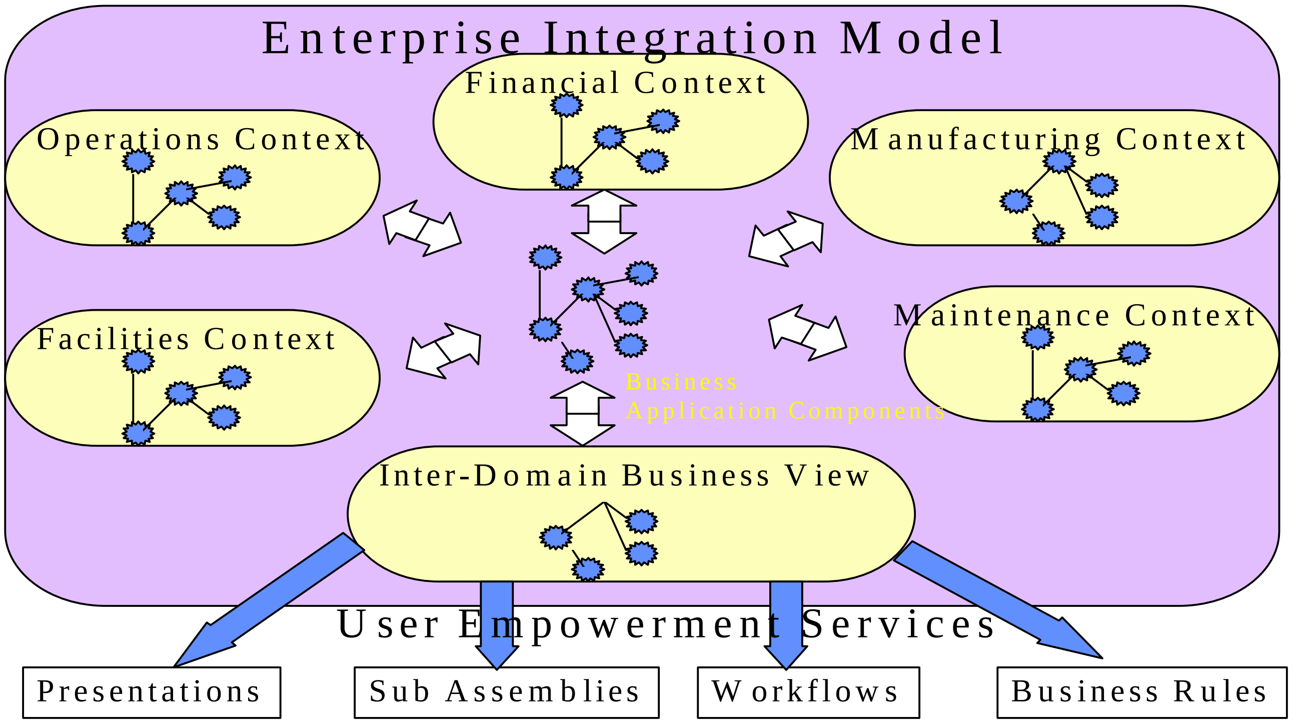


Figure 7 Example: Using the Enterprise Integration Model

7. Semantic Extensions to OMG Standards

OMG's central mission is to establish an architecture and set of specifications to enable distributed integrated applications. Primary goals are the reusability, portability and interoperability of object-based software components in distributed heterogenous environments. Much of the effort to date has been to establish an enabling infrastructure based on open and standard interface definitions. While the enabling infrastructure will have positive impact on the enterprise, orders of magnitude higher impact will be achieved through rapid delivery of interoperable business application components.

Business application components do not exist in isolation. Rigorous and concise semantics are required to ensure enterprise integrity, particularly as empowered users directly provision their business solutions using some form of component assembly paradigm. The Enterprise Integration Model (EIM) embodies a set of tightly coupled concepts necessary to maintain semantic integrity of business application components across the enterprise.

A natural extension of the OMG Architecture would be to formally define business domain facilities in terms of the concepts specified within EIM. Such an architectural extension would:

- Help resolve issues related to semantic integrity of business application components.
- Lay a foundation for an OMG common business infrastructure.
- Create a true plug-and-play business component market.
- Enable implementation of viable end-user solution assembly tools.

An opportunity to incorporate EIM concepts into the OMG Architecture may exist in the form of a BOMSIG initiative. This initiative is soliciting proposals for an OMG standard facility which would serve as the underpinning technology for interoperable business components and many of the concepts referenced in this paper.

8. Conclusion

For many industries, the most compelling business driver for IT is the need for rapid provisioning of business solutions within an environment of increasing complexity. IT organizations are having difficulty meeting these decreasing cycle time requirements, particularly when increased business complexity has an adverse impact on software productivity, quality, cost, and cycle time. The software industry could address these issues by applying concepts of componentization. The benefits of this approach are exemplified by the integrated circuit industry's ability to consistently double producitivity and performance each year. Cycle time reduction goals also require active participation by the business user. Ultimate cycle time reduction will occur when the business user is able to express his problem in a form consistent with his problem domain and have that specification automatically and instantly implemented. Building on component concepts and the need for solution provisioning directly by business users, an open component-based architecture will feature plug and play components and end-user empowerment. This architecture is inherently more complex than contemporary application architectures. The architecture will need to be implemented using a complexity-hiding model-based development approach. Such an approach would iteratively utilize generic constructs forumulated at one level of abstraction as the language and semantic for the next level of specialization. Empowered business users will express their problem in their business language; rapidly provision their solutions; assemble, specialize, and customize business application components. Technology complexity will be hidden, business semantics will be rigorously enforced, and an Enterprise Integration Model will capture the enterprise-specific semantics which allow business application components from multiple business domains to interoperate.

Abbreviations

BOMSIG	Business Object Management Special Interest Group (OMG)
EIM	Enterprise Integration Model
IT	Information Technology
OAG	Open Applications Group
OMG	Object Management Group

References

[BAPAT 94] Subodh Bapat, *Object-Oriented Networks*, Prentice Hall, 1994.

[COX 87] Brad J. Cox, *Object Oriented Programming An Evolutionary Approach*, . Addison-Wesley Publishing Company, April 1987.

[COX 90]	Brad J. Cox, *Software Technologies of the 1990's,* IEEE Software Magazine, IEEE, November 1990.
[DSOUZA 94]	Desmond D'Souza, *Object-Oriented Analysis, Modeling, and Conceptual Design*, ICON Computing, Inc, Jan 1994.
[ESPRIT 93]	ESPRIT Consortium AMICE, *CIMOSA: Open System Architecture for CIM*, Springer-Verlag, 1993.
[MARTIN 91]	James Martin, *Rapid Application Development*, Macmillan Publishing Company, 1991.
[OMG 92.10.01]	Object Management Group, *Object Analysis and Design*, Draft 7.0, 1 October 1992.
[OMG 93.12.29]	Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 1.2, 29 December 1993.
[OMG 95.01.02]	Object Management Group, Common Facilities Architecture, Revision 4.0, 3 January 1995.
[OMG 95.01.12]	Object Management Group, *Object Services Architecture*, Revision 8.1, 12 January 1995.
[WIRTHMAN 95]	Lisa Wirthman, *Time is ticking away*, PC WEEK, July 24, 1995.
[WJ 93]	Wakeman & Jowett, *PCTE The Standard for Open Repositories*, Prentice Hall, 1993.