

The EDOC Component Collaboration Architecture

(Extracted from the EDOC specification)

The Component Collaboration Architecture (CCA) details how the UML concepts of classes, collaborations and activity graphs can be used to model, at varying and mixed levels of granularity, the structure and behavior of the components that comprise a system.

1.1 Rationale

1.1.1 Problems to be solved

The information system has become the backbone of the modern enterprise. Within the enterprise, business processes are instrumented with applications, workflow systems, web portals and productivity tools that are necessary for the business to function.

While the enterprise has become more dependent on the information system the rate of change in business has increased, making it imperative that the information system keeps pace with and facilitates the changing needs of the enterprise.

Enterprise information systems are, by their very nature, large and complex. Many of these systems have evolved over years in such a way that they are not well understood, do not integrate and are fragile. The result is that the business may become dependent on an information infrastructure that cannot evolve at the pace required to support business goals.

The way in which to design, build, integrate and maintain information systems that are flexible, reusable, resilient and scalable is now becoming well understood but not well supported. The CCA is one of a number of the elements required to address these needs by supporting a scalable and resilient architecture.

The following subsections detail some of the specific problems addressed by CCA.

1.1.1.1 Recursive decomposition and assembly

Information systems are, by their very nature, complex. The only viable way to manage and isolate this complexity is to decompose these systems into simpler parts that work together in well-defined ways and may evolve independently over time. These parts can then be separately managed and understood. We must also avoid re-inventing parts that have already been produced, by reusing knowledge and functionality whenever practical.

The requirements to decompose and reuse are two aspects of the same problem. A complex system may be decomposed “top down”, revealing the underlying parts. However, systems will also be assembled from existing or bought-in parts – building up from parts to larger systems.

Virtually every project involves both top-down decomposition in specification and “bottom up” assembly of existing parts. Bringing together top-down specification and bottom-up assembly is the challenge of information system engineering.

This pattern of combining decomposition in specification and assembly of parts in implementation is repeated at many levels. The composition of parts at one level is the part at the next level up. In today’s web-integrated world this pattern repeats up to the global information system that is the Internet and extends down into the technology components that make up a system infrastructure – such as operating systems, communications, DBMS systems and desktop tools.

Having a rigorous and consistent way to understand and deal with this hierarchy of parts and compositions, how they work and interact at each level and how one level relates to the next, is absolutely necessary for achieve the business goals of a flexible and scalable information systems.

1.1.1.2 Traceability

The development process not only extends “up and down” as described above, but also evolves over time and at different levels of abstraction. The artifacts of the development process at the beginning of a project may be general and “fuzzy” requirements that, as the project progresses, become precisely defined either in terms of formal requirements or the parts of the resulting system. Requirements at various stages of the project result in designs, implementations and running systems (at least when everything goes well!). Since parts evolve over time at multiple levels and at differing rates it can become almost impossible to keep track of what happened and why.

Old approaches to this problem required locking-down each level of the process in a “waterfall”. Such approaches would work in environments where everything is known, well understood and stable. Unfortunately such environments seldom, if ever, occur in reality. In most cases the system becomes understood as it evolves, the technology changes, and new business requirements are introduced for good and valid reasons. Change is reality.

Dealing with this dynamic environment while maintaining control requires that the parts of the system and the artifacts of the development process be traceable both in terms of cause-effect and of changes over time. Moreover, this traceability must take into account the fact that changes happen at different rates with different parts of the system, further complicating the relationships among them. The tools and techniques of the development process must maintain and support this traceability.

1.1.1.3 Automating the development process

In the early days of any complex and specialized new technology, there are “gurus” able to cope with it. However, as a technology progresses the ways to use it for common needs becomes better understood and better supported. Eventually those things that required the gurus can be done by “normal people” or at least as part of repeatable “factory” processes. As the technology progresses, the gurus are needed to solve new and harder problems – but not those already solved.

Software technology is undergoing this evolution. The initial advances in automated software production came from compilers and languages, leading to DBMS systems, spreadsheets, word processors, workflow systems and a host of other tools. The end-user today is able to accomplish some things that would have challenged the gurus of 30 years ago.

This evolution in automation has not gone far enough. It is still common to re-invent infrastructures, techniques and capabilities every time a new application is produced. This is not only expensive, it makes the resulting solutions very specialized, and hard to integrate and evolve.

Automation depends on the ability to abstract away from common features, services, patterns and technology bindings so that application developers can focus on application problems. In this way the ability to automate is coupled with the ability to define abstract viewpoints of a system – some of which may be constant across the entire system.

The challenge today is to take the advances in high-level modeling, design and specification and use them to produce factory-like automation of enterprise systems. We can use techniques that have been successful in the past, both in software and other disciplines to automate the steps of going from design to deployment of enterprise scale systems. Automating the development process at this level will embrace two central concepts; reusable parts, and model-based development. It will allow tools to apply pre-established implementation patterns to known modeling patterns. CCA defines one such modeling pattern.

1.1.1.4 Loose coupling

Systems that are constructed from parts and must survive over time, and survive reuse in multiple environments, present some special requirements. The way in which the parts interact must be precisely understood so that they can work together, yet they must also be loosely coupled so that each may evolve independently. These seemingly contradictory goals depend on being able to describe what is important about how parts interact while specifically not coupling that description to things that will change or how the parts carry out their responsibility.

Software parts interact within the context of some agreement or contract – there must be some common basis for communication. The richer the basis of communication the richer the potential for interaction and collaboration. The technology of interaction is generally taken care of by communications and middleware while the semantics of interaction are better described by UML and the CCA.

So while the contract for interaction is required, factors such as implementation, location and technology should be separately specified. This allows the contract of interaction to survive the inevitable changes in requirements, technologies and systems.

Loose coupling is necessarily achieved by the capability of the systems to provide “late binding” of interactions to implementation.

1.1.1.5 Technology Independence

A factor in loose coupling is technology independence i.e. the ability to separate the high-level design of a part or a composition of parts from the technology choices that realize it. Since technology is so transient and variations so prevalent it is common for the same “logical” part to use different technologies over time and interact with different technologies at the same time. Thus a key ingredient is the separation high-level design from the technology that implements it. This separation is also key to the goal of automated development.

1.1.1.6 Enabling a business component Marketplace

The demand to rapidly deploy and evolve large scale applications on the internet has made brute force methods of producing applications a threat to the enterprise. Only by being able to provision solutions quickly and integrate those solutions with existing legacy applications can the enterprise hope to achieve new business initiatives in the timeframe required to compete.

Component technologies have already been a success in desktop systems and user interfaces. But this does not solve the enterprise problem. Recently the methods and technologies for enterprise scale components have started to become available. These include the “alphabet soup” of middleware such as XML, CORBA, Soap, Java, ebXml, EJB & .net., What has not emerged is the way to bring these technologies together into a coherent enterprise solution and component marketplace.

Our vision is one of a **simple** drag and drop environment for the **assembly of enterprise components** that is integrated with and leverages **a component marketplace**. This will make buying and using a software component as natural as buying a battery for a flashlight.

1.1.1.7 Simplicity

A solution that encompasses all the other requirements but is too complex will not be used. Thus our final requirement is one of simplicity. A CCA model must make sense without too much theory or special knowledge, and must be tractable for those who understand the domain, rather than the technology. It must support the construction of simple tools and techniques that assist the developer by providing a simple yet powerful paradigm. Simplicity needs to be defined in terms of the problem – how simply can the paradigm solve my business problems. Simplistic infrastructure and tools that make it hard to solve real problems are not viable.