New Paradigms for Software Application Development: 
Software architectures and component-based development

Sergio David Villarreal, Guillermo Villasana, Juan Carlos Lavariega 
ITESM, Monterrey, Nuevo León, México

Abstract In recent years several technologies and programming languages have appeared for developing software systems. Each one provides advantages and specific implementations to business applications. However, traditional approaches are still in use in many software projects, and therefore the software code was not initially developed with reuse in mind. This situation leads to delays in delivery times, production costs above budget, and possible generation of incomplete/defective software products. The architectures and the component-based development emerge as alternatives for traditional software development. The challenge is software with and for reuse, and interoperability between different technologies, platforms and applications. This paper provides a description of the component-based development and the uses of software architectures. Also, examples of programmed technologies that implement these concepts are given such as .NET, Web Services, OSGi, ICE, and SCA.

Keywords
Software architecture, component-based development, CBSD, .NET, Web Services, OSGi, ICE, and SCA.

1. Introduction
Software industry is moving away from the giant, monolithic, and hard development code-based practices. Software developers now have more variety of tools and methodologies to choose for building software products. But, despite the advances and alternatives available, during the design and development phase the software application may have several mishaps that can compromise the success of a software project: incomplete requirement analysis, use of technologies not suited to the type of problem to be solving, poor designs, and delays in delivery times, production costs above budget, or defective end products. Furthermore, the use of traditional approaches prior to object oriented programming (OOP) do not take into account concepts such as modularity, low coupling, information hiding, or encapsulation. These leads to develop new software projects from scratch. To overcome the aforementioned problems, the experts in the software industry have adopted the use of component-based solutions [19]. The component-oriented programming approach proposes a paradigm change in the process of software construction. The software needs to be conceived for and with reuse in mind to allowing previously developed components to be assembled and utilized in new projects. On the other hand, software architectures are useful to try to meet the necessities and requirements of customers. Roughly speaking, software architecture is a system design that includes high-level structures and sets the properties of interest that the system must meet.

Some actual technologies, like Microsoft .NET, Web Services, Open Services Gateway initiative (OSGi), Internet Communications Engine (ICE), and Service Component Architecture (SCA) are examples of technologies which implement the concepts above mentioned. SCA is an outstanding platform that incorporates and extends many of the features seen in the other technologies, and brings an easy interoperability implementation among different platforms/languages.

The use of software architectures and the component-oriented development approach seeks to provide a comprehensive solution that favors successful construction of software applications, with lower development costs and time, and a higher overall quality.

The organization of this work is as follows: Section 2 is a description of the concept of software architecture and its main features, such as high-level structures of a system and the relationships between them and their properties of interest (performance, reliability, security, and maintenance). Section 3 talks about Component-Based Software Development (CBSD) and its advantages to modify, extend, reuse and make language independence code. Section 4 gives a general description of how .NET, Web Services, OSGi, ICE, and SCA implement the concepts related to component-oriented development and software architectures. Finally, Section 5 presents a developed software project using SCA and software architecture.

2. Software Architecture
The software architecture comprises high-level structures of a system, the relationships between them and their environment [8]. An architectural design aims to facilitate the development of software applications, verifies the correct evolution of the system, aid in the detection of errors, contribute to the maintenance actions and help to reduce the associate costs. A well-designed architecture allows reasoning about satisfaction of customer’s key requirements and to make agreements on engineering principles and the properties of interest, such as performance, reliability, security, and maintenance. Also provides a clear allocation of functions to components establishes conceptual integrity principles and pursues to minimize rework applications during the life of the system [5].
A complex software system may comprise several structures of interest: modules, runtime entities, development teams, physical devices, and networks, to name a few. Therefore appropriate architectural design must be described in terms of different views. Each view represents an architectural perspective of the system. Each perspective shows certain structures and characteristics of the system to deal with a particular set of problems.

When designing software architectures is important to take into account the following aspects:

**Requirements.** - Software architectures are generated from the functional and non-functional requirements. A functional requirement specifies the actions that a system must meet, while a non-functional requirement establishes general limitations in existing solutions, like performance or design’s constraints [12].

**Complexity.** - One of the design activities of software architecture is the decomposition of the system into subsystems (components). The purpose of decomposition is to reduce complexity into smaller and more manageable parts. While the complexity cannot be entirely eliminated, any reduction facilitates the development of the system [1].

**Anticipation of changes.** - Changes are very common during the software development. A single change may lead to new requirements or consider re-evaluating existing ones. The architecture should be flexible enough and reusable to adapt positively to the changes [12].

**Performance and scalability.** - The performance of a system is the largest and greatest risk to the success of a software project. Performance issues are caused generally by deficient architectures derived from poor design choices in the initial stages of the software life cycle [3].

Garlan & Schmerl (2006) suggest the use of execution structures, or views (or graphs) of component and connector (C & C) to deal with software architectures. The C&C views express more directly the critical features related to dependence, such as reliability, safety, performance. The C&C allows the employing of traditional lines and diagrams of boxes to represent software architectures. Also, C&C has a correspondence with the primitive building blocks of most architectural description languages (ADLs)[5].

## 3. Component-Based Software Development (CBSD)

The component-based software development (CBSD) tries to provide an effective approach for the construction of software products. Splitting up systems in its binary components it is possible to achieve a higher level of reusability, extensibility, and system maintenance compared to traditional object-oriented approach. An application in CBSD consists of a collection of one or more components in conjunction with the link calls to interact between them. The functionality of each component contributes to implement and execute the business logic of the application [14].

A component is an auto deployment self-contained entity that implements a business function and provides a high level of software reuse. Some components may be general purpose while other components may be highly specialized and/or built specifically for the application [21].

Some advantages of CBSD are:

- **Modifiability.** Components can be added or removed according to the requests of each application.
- **Extensibility.** When a new requirement needs to be implemented, if it can be incorporated using new components it is not necessary to modify existing ones that are not related to the implementation.
- **The changes, if necessary, will apply only to the involved components.** The components can be updated even while a client application is running, provided that the components are not being used.
- **Improvements and arrangements made to a component can be immediately available to all applications that use the component.**
- **Component-oriented programming allows customer applications and components to be developed and evolve separately.**
- **The language independence promotes the exchange of components, their adoption and reuse.** Developers using component-oriented development can focus on the decomposition of interfaces. The interfaces will be used as contracts between clients and services provided by the components.

Component-oriented programming (COP) takes the good methodologies of the OOP as it base, but it has the components as its basic programming elements [14]. Some of the principal characteristics in COP are: It is based in interfaces. It is a distribution technology and component packaging. It supports high-level reuse. COP, in principle, can be written in any language. They are loosely coupled components. Components have long granularity. Support for multiple interfaces, and a design-oriented interfaces. Have mechanisms that enhance the integration of third-party compositions. It supports multiple ways to link and dynamic discovery. Provides support higher order services like security and transactions.

In COP the basic application unit is an interface. An interface is a logical grouping of method definitions that act as a contract between the customer and the service provider. Each provider is free to give their own interpretation of the interface. The interfaces are implemented by a black box component that completely encapsulates the interior. To use the service offered by a component, it is not necessary to know how an interface is internally implemented. The client only needs to know the definition of the interface [14].

## 4. Component-Based Development Technologies

The current challenge of new technologies is to promote software reuse and component-based development. Interoperability and reusability not only represents a long-term challenge - because software is constantly evolving -,
but also a great opportunity for improvement in terms of time and quality software development. In the following subsections we introduce various technologies, such as .NET, Web Services, OSGi, ICE, and SCA. These technologies, at different levels, implement the concept of component-based development and promoting software reuse.

4.1 Microsoft .NET

.NET is a technology designed to simplify the development and implementation of components, while providing interoperability between various programming languages such as Visual Basic, Visual C++, C#, among others [14]. The .NET framework is made up of two main parts [21]: Common Language Runtime (CLR), and a set of unified libraries like ASP.NET Web Forms, Windows Form and ADO.NET.

The CLR provides a common context in which all .NET components are running, regardless of the language they are written [14]. The CLR consists of: Common Type System (CTS), Intermediate Language (IL) code, Just-In-Time (JIT) compiler, an execution unit, and some other management services. Figure 4.1 shows how the CLR works.

In .NET all compilers generate code in agreement to the common type system (CTS). Any .NET component is transformed to an intermediate common language infrastructure (CLI), also calling Microsoft Intermediate Language (MSIL), instead of a processor’s specific object or platform. The MSIL instruction set is platform independent, and can be run in any environment that supports CLI. The use of MSIL helps to eliminate the necessity to distribute different executable for different platforms and CPU types.

The basic unit of packaging in .NET is the assembly. An assembly gets together multiple physical files into a single logical unit. The assembly can be a class library (DLL) or a standalone application (EXE) [14]. An assembly consists essentially of: MSIL code modules, a manifest, metadata modules, and several resources. A metadata is a comprehensive, standard, mandatory, and complete way to describe the content of the assembly. A manifest describes the assembly itself; provide the logical attributes shared by all modules and components within the assembly [14]. In .NET the component composition can be implemented in two ways: by aggregation (external exposure of the interface), or by containment (the process is performed internally and transparent to the user). .NET allows the use of code contracts. A code contract sets the preconditions, post conditions and invariant program objects codes. Contracts act as documentation for internal and external APIs, and are used to improve testing via runtime revisions. The composition of components, property inheritance, and methods of classes - written in different languages - allow reuse of components.

4.2 Web Services

A Web Service is a software system designed to support interoperability and interaction machine-to-machine over a network that has an interface described in a machine-processable format [16]. Laws et al. (2011) said Web Service is a term generally used to describe an interface provided by a software application that can be called via network. More recently, the term has been used to describe the services provided in a network using SOAP over HTTP protocol. By using SOAP a XML format is described for message passing from the client to server and server to the client. To describe the interfaces provided by a Web Service a Web Services Definition Language (WSDL) is used. WSDL is a XML language that defines the functionality of the interfaces in terms of the providing operations, and the physical details about where the Web Service is hosted.

The basic actions performed by a Web Service are [7]: publish a Web Service, and consume a Web Service. On the other hand, an application that consumes a Web Service has two components: a proxy object to interact with the Web
Service, and a client application to consume – by invoking methods on the proxy object - the Web Service.

The Web Service communication can be functionally made between two completely different environments by using standard protocols. The calls in Web Services are translated into a language and standard protocol that both computers can understand. Generally, a XML format is used. The XML language is a text-based format commonly understood among different applications. Also, the Web Services allows the calling of remote applications by remote procedure calls (RPC) [10].

Some characteristics of Web Services that promote software reuse design [11] are: open infrastructure (usage of widely documented and accepted protocols like HTTP and XML), transparency of language (interoperability between clients written in different programming languages), and modular design (aggregation services through integration and layering). These features distinguish Web Services from other distributed software systems.

There are several methods of Web Services. Some of them are: Simple Object Access Protocol (SOAP), Universal Description, Discovery and Integration (UDDI), Web Services Description Language (WSDL), Representational State Transfer (REST), and Action Message Format (AMF).

There are several methods of Web Services. Some of them are: Simple Object Access Protocol (SOAP), Universal Description, Discovery and Integration (UDDI), Web Services Description Language (WSDL), Representational State Transfer (REST), and Action Message Format (AMF). Languages like C / C++, C#, Java, Perl, Python, and Ruby provide libraries, utilities, and even frameworks that support Web Services.

Figure 4.2 shows a Web Service architecture with three elements: a client, a provider Web Service and UDDI Registry. The Web Service provider registers/publish its services in the UDDI register. The services are globally available to the customers who require them.

4.3 Open Services Gateway initiative (OSGi)

The alliance Open Services Gateway initiative (OSGi) emerged in 1999. Its purpose was to provide Java embedded technology to network gateways in households [6]. Currently, OSGi framework is a component specification that provides modularity to the Java platform. OSGi allows the creation of highly cohesive and loosely coupled modules which can be integrated into larger applications. Each module can even be independently developed, tested, implemented, updated, and managed with zero or minimal impact with respect to the other modules [20].

OSGi is built over the Java platform, and is made of several layers: module definition, lifecycle modules, service registration, services, and security layers. The OSGi framework and the Java platform are illustrated in Figure 4.3.

The module part defines a deployment model based on Java. In OSGi the implementation unit is the bundle. The OSGi bundles are very similar to JAR files, except that its META-INF/MANIFEST.MF file contains specific OSGi metadata, including a final name, version, dependencies, and some other implementation details. A bundle can be compared to a Web ARChive (WAR) in the context of a web container, or an Enterprise ARChive (EAR) in the context of Java Enterprise Platform[6].

A bundle can be installed, initialized, stopped or uninstalled from the framework according to the life cycle prescribed by the OSGi specification. The OSGi framework provides a service registry, in which bundles can be publish and/or consume services. However, unlike some interpretations of service oriented architecture (SOA) using Web Services, OSGi services are published and consumed within the same Java virtual machine. OSGi is also described as a "SOA JVM". OSGi, as well, defines an optional security layer to authenticate bundles to be deployed in a safe manner.

OSGi provides the following additional modular features to Java [20]: hiding content, service record (the services are known by their interfaces), parallel versions of bundles, dynamic modularity, strong naming (the bundles are identified by a symbolic name and version number).

4.4 Internet Communications Engine (ICE)

ICE is an object-oriented middleware that allows developers to build client-server applications in a distributed fashion with minimal effort. Similar in concept to CORBA, ICE provides a simpler and more powerful object model. ICE includes improvements such: user datagram protocol (UDP) support, sending asynchronous mode, security, automatic object persistence, and interface aggregation. The object model is a set of definitions about computational entities properties, like available types and their semantics, rules for type compatibility, and behavior in case of error [9].

ICE has tools, APIs, and support libraries to build client-server object oriented application. ICE can be used in heterogeneous environments, as clients and servers can be written in different programming languages. ICE can run on different operating systems and architectures, using a variety of network technologies to communicate [17]. ICE currently supports C++, Java, C #, Objective-C, Python, Ruby, and PHP languages, on Linux, Mac OS, Windows, Android and Solaris platforms.

ICE operation is based on RPC using TCP or UDP to invoke remote objects as if they were local. The objects are called
ICE objects, which are a local or remote entities responsible for responding to customer requests. It is necessary to have a client-side proxy to establish communication with remote objects hosted on remote servers. The local client needs a Servant to know the implementations and methods which a remote object has. The Servant will be responsible for explaining the behavior of operations.

Slice is the property of ICE that can transform objects written in different languages supported by the middleware in ICE objects. Each ICE object has an interface with a specific number of operations. The Slice language defines the interfaces, operations, and data types that are exchanged between the client and the server. Slice allows establishing the entity contracts between the client and server independently of the programming language.

The ICE architecture provides several benefits to software developers [18]: object-oriented semantics, asynchronous, synchronous messages, multithreading and multiple interfaces support; machine architecture, implementation, operating system, and transport independence; location and server transparency, and security.

4.5 Service Component Architecture (SCA)

SCA was originally created by a group of companies such as IBM, Oracle, SAP, and BEA. SCA is a programming model that abstracts standard business functions into software components [15]. The basic building block in SCA is the component. The components are then used as building blocks. The implementation of an SCA component can be performed in any technology, like Ruby, BPEL, Java, or even frameworks like Spring, Java EE and OSGi.

A SCA components consist of [2]: services (interfaces), references (also called interfaces, they are the required dependences to perform its task), properties (configuration), and intention policies (component’s behavior). The SCA component’s parts are illustrated in Figure 4.4.

The unit of deployment in SCA is the composition. A composition is an aggregation of one or more components. A composition can provide externally the services and references provided by its internal components through promotions[4]. Applications can be built using one or more compositions. The components within a composition can use the same technology or be implemented in different technologies. This feature promotes the reuse of components [4].

![Figure 4.4. - SCA component’s elements](image)

An SCA composition is described in a configuration file with a .composite extension. The .composite file is build using a Service Component Definition Language (SCDL) based on XML. The SCDL describes the existing components within the composition and the relationships between them. An item package that is part of the business solution is known as contribution. A contribution is a unit of deployment, and may contain compositions, Java classes, and XSD or WSDL files.

One of the most important concepts in SCA is the bindings. A binding specifies the communication methods that a client can use to access a service, and the methods that a service can use to access other services, either within the same SCA domain or outside it. Services can be configured to use different types of bindings without have to change the component’s code. Therefore multiple bindings can be associated for the same service. For example, one software solution should have JMS bindings, Web Services bindings, Atom bindings, and Corba bindings. Through the use of bindings it is possible to focus on the business logic of the components, instead of the problems associated with communications and management protocols. This feature allows SCA compositions to be flexible, and grow and adapt without code changes.

5. Software Project

A software project was developed to illustrate how software architectures and component-based development contribute to build and reuse the software. The SCA was selected due is a component architecture platform that extends many of the features seen in the other technologies mentioned above.

5.1 Project description

A system with a Web application was built to provide available billboard information from several movie theaters within certain particular region. The application offers an online catalog service where users can query and evaluate the results according to their preferences.

5.2 Objectives

The main objectives of the project were: collection and processing of information in different formats, allow the use of different technologies to communicate components and services, using software architectural designs according to the project specifications, development of component-based software to implement system functionality, and software development with and for reuse.

5.3 Architecture

The selected pattern was a layered client-server approach. The server is always active and waiting for connections and queries from customers. The architecture of the project, seen from a general point of view, has five main components: Clients, Cinemas, Intelligent Agents, MovieCatalog, and Data Access. The five components and their respective relationships are illustrated in Figure 5.1.
The description of each component of Fig 5.1 is as follows:

**Customers.** - Represent users who will use the service MoviesSCA.

**Cinemas.** – The movie theaters that will be recorded in the system. Each movie theaters will provide movie data from their billboards.

**Intelligent Agents.** - Perform search operations over Internet to collect billboard information. This mechanism is proposed to automate the data collection process periodically.

**MovieCatalog.** – This component offers the query and movie research services.

**Data Access Layer.** – This component directly communicates with the database engine to perform query and update operations. The data access layer component brings independence to the database from the other components.

Figure 5.2 shows a class diagram from an architectural perspective. The architectural view adds the interfaces and classes that compose each one of the components.

### 5.4 Implementation

For the exchange of information between components a scheme based on XML was created. The framework Tuscany was chosen to build the project. Tuscany is a lightweight infrastructure that implements the following technologies: Service Component Architecture (SCA), Service Data Objects (SDO), and Data Access Service (DAS). The My Structured Query Language (MySQL) was selected to implement the database of the software project. The MySQL is an open source relational database management system (RDBMS) widely used in software projects.

### 5.5 Results

A well-designed architecture provides system decomposition into several subsystems, specifies the role of each component, and helps to meet the key project requirements.

The SCA technology offers an excellent component based solution for software development with reuse in mind. The SCA diagrams make easier to understand the communication, dependency, and interaction between components within a system. The use of bindings allows interoperability among different technologies outside SCA in a transparently manner. To add a new communication protocol only is needed to aggregate a few lines in the composite file, leaving the component’s code without any modification.

Finally, with SCA the reuse of software is feasible due the SCA components can be ported to other environments and can run without major complications, although the rest of the application was developed in a different language.

### 6. Conclusions

Nowadays, there are several platforms, technologies, and programming languages to build software systems. Each one offers different solutions and implementation to client’s business requirements. Unfortunately, traditional approaches prevent the reuse of previously developed software -mainly because of bad software practices -, deriving in the necessity to build new software from scratch every time.

The architectural designs and the CBSD are some of the latest tools available for software developers. The use of architectural designs facilitates the development of software applications. It provides system decomposition into several subsystems, specifies the assigned roles to each component, gives conceptual integrity principles, and tries to minimize rework applications during the lifetime of the system. Also, a well-designed architecture allows the verification of the correct evolution of a system, the satisfaction of key project requirements, helps to detect system errors, reduce associated costs, and contribute to the maintenance actions.

The component-oriented programming promotes the software development with and for reuse. The component-oriented technologies discussed in this paper bring different features and capabilities for software reuse. Some are easier to implement, as the case of Web Services, while others have a higher learning curve, such as OSGi. .NET is a good option if you work mostly on Microsoft’s platforms and programming languages. ICE offers many improvements over its predecessor, Corba, but it is still under development, so many of its features are not available. Finally, SCA is a component architecture that incorporates and extends many of the features seen in other technologies, thus SCA brings a more comprehensive and complete software solution.

The SCA’s advantages found with this research are listed below:

- It is a component-based architecture.
- Using SCA diagrams it is easy to design the component architecture for the software project. During the implementation phase, a very close correspondence exists between the code and the different architectural elements.
- In SCA services and references can be developed and connected in a distributed fashion, using a variety of technologies and languages.
- SCA is a flexible and versatile platform that can interact with other applications outside SCA through the use of bindings.
• The bindings enable SCA interoperability with applications developed in other languages or using different communication protocols.
• To add a new binding to the SCA service it is not necessary to make changes to the component’s code. Only a few lines in the .composite file are required to implement the communication service.
• The compositions and contributions promote the programming with and for software reuse.
• Using the SCDL simplifies the description of the components within a composition.

In general, the software architectures and the component-based development represents a long-term challenge - because the software is constantly evolving -, but also represents a great opportunity for design, time, cost, and software quality improvements.

7. References