Chapter 16 The essentials of component-based software engineering: 1. Independent components that are completely specified by their interfaces. 2. Component standards that define interfaces and so facilitate the integration of components. 3. Middleware that provides software support for component integration. 4. A development process that is geared to component-based software engineering.

Underlying CBSE are sound design principles that support the construction of understandable and maintainable software: 1. Components are independent, so they do not interfere with each other's 3. Component infrastructures offer a range of standard services that can be used in application systems.

define a component as: A software element that conforms to a standard component model and can be independently deployed and composed without modification according to a composition standard.†

Viewing a component as a service provider emphasizes two critical characteristics of a reusable component: 1. The component is an independent executable entity that is defined by its interfaces. 2. The services offered by a component are made available through an interface, and all interactions are through that interface.

These interfaces reflect the services that the component provides and the services that the component requires to operate correctly: 1. The "provides" interface defines the services provided by the component. 2. The "requires" interface specifies the services that other components in the system must provide if a component is to operate correctly.

In a UML component diagram, the "provides" interface for a component is indicated by a circle at the end of a line from the component icon. 2. The "requires" interface specifies the services that other components in the system must provide if a component is to operate correctly.

The services provided by a component model implementation fall into two categories: 1. Platform services, which enable components to communicate and interoperate in a distributed environment. 2. Support services, which are common services that many different components are likely to require.

presents an overview of the processes in CBSE. At the highest level, there are two types of CBSE processes: 1. Development for reuse This process is concerned with developing components or services that will be reused in other applications. 2. Development with reuse This process is the process of developing new applications using existing components and services.

basic processes of CBSE: 1. Component acquisition is the process of acquiring components for reuse or development into a reusable component. 1. Component acquisition is the process of acquiring components for reuse or development into a reusable component.

Changes that you may make to a component to make it more reusable include: ▪ removing application-specific methods; ▪ changing names to make them more general; ▪ adding methods to provide more complete functional coverage; ▪ making exception handling consistent for all methods; ▪ adding a "configuration" interface to allow the component to be adapted to different situations of use; ▪ integrating required components to increase independence.

However, the essential differences between CBSE with reuse and software processes for original software development are as follows: 1. The user requirements are initially developed in outline rather than in detail, and stakeholders are encouraged to be as flexible as possible in defining their requirements. 2. Requirements are refined and modified early in the process depending on the components available. 3. There is a further component search and design refinement activity after the system architecture has been designed.

to create a new component: 1. Sequential composition In a sequential composition, you create a new component from two existing components by calling the existing components in sequence. 2. Hierarchical composition This type of composition occurs when one component calls directly on the services provided by another component. 3. Additive composition This occurs when two or more components are put together (added) to create a new component, which combines their functionality.

Three types of incompatibility can occur: 1. Parameter incompatibility 2. Operation incompatibility

This description appears to explain what the component does, but consider the following questions: ▪ What happens if the photograph identifier is already associated with a photograph in the library? ▪ Is the photograph descriptor associated with the catalog entry as well as the photograph? That is, if you delete the photograph, do you also delete the catalog information?

This description appears to explain what the component does, but consider the following questions: ▪ What happens if the photograph identifier is already associated with a photograph in the library? ▪ Is the photograph descriptor associated with the catalog entry as well as the photograph? That is, if you delete the photograph, do you also delete the catalog information?

Chapter 17

distributed systems: 1. Resource sharing 2. Openness Distributed systems are normally open systems—systems designed around standard Internet protocols 3. Concurrency In a distributed system 4. Scalability

The three dimensions of scalability are size, distribution, and manageability.

A distributed system must defend itself against : 1. Interception 2. Interruption 3. Modification 4. Fabrication

Ideally, the QoS requirements should be specified in advance and the system designed and configured to deliver that QoS. Unfortunately, this is not always practicable for two reasons: 1. It may not be cost-effective to design and configure the system to deliver a high quality of service under peak load. 2. The quality-of-service parameters may be mutually contradictory.

In a distributed system, middleware provides two distinct types of support: 1. Interaction support 2. The provision of common services

application structured into four layers: 1. A presentation layer that is concerned with presenting information to the user and managing all user interaction. 2. A data-handling layer that manages the data that is passed to and from the client. 3. An application processing layer that is concerned with implementing the logic of the application and so providing the required functionality to end-users. 4. A database layer that stores the data and provides transaction management and query services.

In this section, I discuss five architectural styles: 1. Master-slave architecture 2. Two-tier client–server architecture 3. Multi-tier client–server architecture 4. Distributed component architecture 5. Peer-to-peer architecture

showing an application structured into four layers: 1. A presentation layer that is concerned with presenting information to the user and managing all user interaction. 2. A data-handling layer that manages the data that is passed to and from the client. 3. An application processing layer that is concerned with implementing the logic of the application 4. A database layer that stores the data and provides transaction management and query services.

which shows two forms of this architectural model: 1. A thin-client model, where the presentation layer is implemented on the client and all other layers (data handling, application processing, and database) are implemented on a server. 2. A fat-client model, where some or all of the application processing is carried out on the client.

The peer-to-peer architectural model may be the best model for a distributed system in two circumstances: 1. Where the system is computationally-intensive and it is possible to separate t 2. Where the system primarily involves the exchange of information between individual computers on a network and there is no need for this information to be centrally stored or managed.

The notion of software as a service and service-oriented architectures are not the same: 1. Software as a service is a way of providing functionality on a remote server with client access through a web browser. 2. Service-oriented architecture is an approach to structuring a software system as a set of separate, stateless services.

Three important factors have to be considered: 1. Configurability 2. Multi-tenancy 3. Scalability

You then use these preferences to adjust the behavior of the software dynamically as it is used. Configuration facilities may allow for: 1. Branding 2. Business rules and workflows 3. Database extensions 4. Access control

general guidelines for implementing scalable software: 1. Develop applications where each component is implemented as a simple stateless service that may be run on any server. 2. Design the system using asynchronous interaction so that the application does not 3. Manage resources, such as network and database connections, as a pool so that no single server is likely to run out of resources. 4. Design your database to allow fine-grain locking.5. Use a cloud PaaS platform

Configuration facilities may allow for: 1. Branding 2. Business rules and workflows 3. Database extensions 4. Access control